University of Pretoria etd – Joseph, G (2005)

DESIGN AND IMPLEMENTATION OF HIGH-SPEED ALGORITHMS FOR PUBLIC-KEY CRYPTOSYSTEMS

By

George Joseph

Submitted in partial fulfillment of the requirements for the degree

Master of Engineering (Electronics)

in the

Faculty of Engineering, Built Environment & Information Technology

UNIVERSITY OF PRETORIA

March 2005

University of Pretoria etd – Joseph, G (2005)

SUMMARY

DESIGN AND IMPLEMENTATION OF HIGH-SPEED ALGORITHMS FOR PUBLIC-KEY CRYPTOSYSTEMS

The aim of this dissertation is to improve computational efficiency of modular exponentiation-based public-key cryptosystems. The operational speed of these public-key cryptosystems is largely determined by the modular exponentiation operation of the form $A = g^e \bmod m$ where g is the base, e is the exponent and m is the modulus. The required modular exponentiation is computed by a series of modular multiplications.

Optimized algorithms are required for various platforms, especially for lower-end platforms. These require the algorithms to be efficient and consume as little resources as possible. In these dissertation algorithms for integer multiplication, modular reduction and modular exponentiation, was developed and implemented in software, as required for public-key cryptography. A detailed analysis of these algorithms is given, as well as exact measurement of the computational speed achieved by each algorithm.

This research shows that a total speed improvement of 13% can be achieved on existing modular exponentiation based public-key cryptosystems, in particular for the RSA cryptosystem. Three novel approaches are also presented for improving the decryption speed efficiency of the RSA algorithm. These methods focus on the selection of the decryption exponent by careful consideration of the difference between the two primes p and q. The resulting reduction of the decryption exponent improves the decryption speed by approximately 45% .

Keywords: Public-key cryptosystems, RSA, modular exponentiation, modular multiplication, modular reduction, RSA decryption, Montgomery reduction, Karatsuba-Ofman multiplication, addition chains, Chinese remainder theorem.

University of Pretoria etd – Joseph, G (2005)

OPSOMMING

DESIGN AND IMPLEMENTATION OF HIGH-SPEED ALGORITHMS FOR PUBLIC-KEY CRYPTOSYSTEMS

The doel van hierdie verhandeling is om die verwerkingseffektiwiteit van modulêre eksponensiëringsgebaseerde publieke-sleutel kriptostelsels te verbeter. Die operasionele spoed van sulke publieke-sleutel kriptostelsels word oorwegend bepaal deur die modulêre eksponensiërings operasie van die vorm $A = g^e \bmod m$ waar g die basis, e die eksponent en m die modulus is. Die vereiste modulêre eksponensiëring word bereken deur 'n reeks modulêre vermenigvuldigings.

Optimale algoritmes word vereis vir verskeie platforms, spesifiek lae-skaal platforms met beperkte rekenkundige vermoe. Die vereiste is dat algoritmes effektief moet wees en so min hulpbronne moontlik gebruik. In hierdie verhandeling is algoritmes vir heelgetal vermenigvuldiging, modulêre vereenvoudiging en modulêre eksponensiasie ontwerp en in sagteware geïmplementeer, soos vereis vir publieke-sleutel kriptografie. 'n Gedetaileerde analise van hierdie algoritmes word voorsien, asook presisie-metings van die verwerkingspoed wat behaal word vir elke algoritme. Hierdie navorsing toon dat 'n totale spoedverbetering van 13% verkry kan word teenoor huidige modulêre eksponensiasie-gebaseerde publieke-sleutel stelsels, spesifiek die RSA kriptostelsels. Drie nuwe benaderings om die spoedeffektiwiteit van die RSA dekriptering te verbeter, word ook voorgestel. Hierdie metodes fokus op die selektering van die dekripsie-eksponent na deeglike inagneming van die verskil tussen twee priemgetalle p en q. Hierdie vereenvoudiging van die dekripsie-eksponent verbeter die dekripsiespoed met ongeveer 45% .

Sleutelwoorde: Publieke-sleutel Kriptostelsels, Modulêre Eksponensiasie, Modulêre Vermenigvuldiging, Modulêre Vereenvoudiging, RSA dekripsie, Montgomery vereenvoudiging, Karatsuba-Ofman vermenigvuldiging, sommasie-skakels, Sjinese res-teorema.

University of Pretoria etd – Joseph, G (2005)

To Appa, Mama, Jikku and Jeff who supported me through my years of study and to God Almighty for giving me talents, and the opportunities to use them.

University of Pretoria etd – Joseph, G (2005)

ACKNOWLEDGEMENT

"Tell me what company thou keepst, and I'll tell thee what thou art."

MIGUEL DE CERVANTES, 1547-1616

First of all, I would like to express my gratitude to my promoter, Professor W.T. Penzhorn, for giving me the opportunity to begin a dissertation in an exciting and moving domain like cryptography, particularly in relation with public key cryptosystems. Without his guidance and astute advice, this dissertation would not be possible.

I appreciate the support of my employer Telkom SA Ltd who provided funding during the completion of this dissertation. I was deeply involved in a tremendous and successful Centre of Excellence program that allowed me to present my research to fellow researchers from various universities.

My deepest thanks go to my good friend Jacques van Wyk, who has always provided me with the guidance, friendship, inspiration, to complete my research here at UP.

It is my privilege to acknowledge the support of my colleague, Cobus Potgieter for collaborating with me on important aspects of this dissertation.

My close friend, Saurabh Sinha, must also be thanked. Saurabh, though you never directly impacted my dissertation, you are an ever reliant friend whose patience, help, and inspiration took me through my undergraduate study. I would never had this opportunity to partake in a postgraduate study without you.

Finally I would particularly like to thank my family whose love and support has taken me through the really difficult periods of my dissertation.

University of Pretoria etd – Joseph, G (2005)

CONTENTS

CHAPTER ONE - INTRODUCTION **1**

1.1 Cryptographic Background 1
1.2 Modular Exponentiation 2
1.3 Objectives 4
1.4 Research Contribution 4
1.5 Dissertation Outline 5

CHAPTER TWO - PUBLIC-KEY CRYPTOSYSTEMS **7**

2.1 Diffie-Hellman Key Exchange 8
2.1.1 The Algorithm 8
2.1.2 Security of the Algorithm 9
2.1.3 Applications of DH 10
2.2 The ElGamal Algorithm 11
2.2.1 The Algorithm 11
2.2.2 Security of the Algorithm 14
2.2.3 Applications of ElGamal 14
2.3 Digital Signature Standard (DSS) 14
2.3.1 The Algorithm 15
2.3.2 Security of the Algorithm 17
2.3.3 Applications of DSS 18
2.4 The RSA Algorithm 18
2.4.1 The Algorithm 19
2.4.2 Security of the Algorithm 23
2.4.3 Applications of RSA 24
2.5 Chapter Summary 24

University of Pretoria etd – Joseph, G (2005)

CHAPTER THREE - FAST MULTIPLICATION TECHNIQUES **26**
3.1 The Classical Method . . . 27
3.1.1 Application to Multiplication . . . 27
3.1.2 Application to Squaring . . . 28
3.2 The Comba Method . . . 29
3.2.1 Application to Multiplication . . . 29
3.2.2 Application to Squaring . . . 31
3.3 The Karatsuba-Ofman Method . . . 32
3.3.1 Application to Multiplication . . . 32
3.3.2 The Computational Complexity of the Algorithm . . . 34
3.3.3 Recursive Properties of the Algorithm . . . 35
3.3.4 The Optimum Break-point . . . 38
3.3.5 Application to Squaring . . . 39
3.4 Experimental Results . . . 41
3.5 Chapter Summary . . . 44

CHAPTER FOUR - FAST REDUCTION TECHNIQUES **45**
4.1 Classical Reduction . . . 46
4.1.1 Description . . . 46
4.1.2 The Algorithm . . . 47
4.1.3 Computational Improvements . . . 48
4.2 Barrett Reduction . . . 48
4.2.1 Description . . . 49
4.2.2 The Algorithm . . . 49
4.2.3 Computational Improvement . . . 51
4.3 Montgomery Reduction . . . 52
4.3.1 Description . . . 52
4.3.2 The Algorithm . . . 53
4.3.3 Computational Improvements . . . 56
4.4 Experimental Results . . . 58
4.5 Chapter Summary . . . 62

CHAPTER FIVE - FAST EXPONENTIATION TECHNIQUES **63**
5.1 The Classical Method . . . 64

University of Pretoria etd – Joseph, G (2005)

5.2 The Binary Method 64
5.2.1 The Algorithm 64
5.2.2 Computational Efficiency 65
5.3 The K-ary Method 66
5.3.1 The Algorithm 67
5.3.2 Computational Efficiency 67
5.4 Sliding Window Methods 69
5.4.1 The Algorithm 70
5.5 Constant Length Nonzero Windows 71
5.5.1 The Algorithm 71
5.5.2 Computational Efficiency 72
5.6 Variable Length Nonzero Windows 73
5.6.1 The Algorithm 74
5.6.2 Computational Efficiency 75
5.7 Addition Chains 77
5.7.1 Description 77
5.7.2 Addition Chain Heuristics 77
5.7.3 The Algorithm 78
5.7.4 Practical Enumeration 83
5.7.5 Discussion 86
5.8 Theoretical Limits 88
5.9 Experimental Results 89
5.10 Discussion 93
5.11 Chapter Summary 96

CHAPTER SIX - FAST EXPONENT TECHNIQUES **97**
6.1 RSA Decryption 98
6.2 Fast Decryption using CRT 98
6.2.1 The Chinese Remainder Theorem (CRT) 99
6.2.2 Computational Efficiency 101
6.3 Fast Decryption by choosing the Decryption Exponent (Method I) 103
6.4 Fast Decryption by choosing the Decryption Exponent (Method II) 105
6.5 Fast Decryption by choosing the Decryption Exponent (Method III) 106

University of Pretoria etd – Joseph, G (2005)

6.6 Prime Generation 108
6.6.1 Low Hamming Weight Prime Difference 108
6.6.2 Small Prime Difference 109
6.6.3 Low Hamming Weight Prime Sum 110
6.7 Experimental Results 112
6.8 Discussion 116
6.8.1 Performance Analysis 116
6.8.2 Simple Factoring Attack on the Modulus 117
6.8.3 Wiener's Attack on Short Decryption Exponents 117
6.8.4 Fermat and Lehman Attacks 118
6.8.5 Security Risk of Method II 118
6.9 Chapter Summary 119

CHAPTER SEVEN - CONCLUSION **121**
7.1 Assessment of study 121
7.2 Summary and further research 125

REFERENCES **126**

University of Pretoria etd – Joseph, G (2005)

CHAPTER ONE
INTRODUCTION

"The design and evaluation of public-key cryptographic functions is a special topic on its own, requiring advanced knowledge of combinatorial mathematics, number theory, abstract algebra, and theoretical computer science."

P.F. SYVERSON [1]

1.1 CRYPTOGRAPHIC BACKGROUND

The need for information security has grown steadily over the years, paralleling growth in the use and interconnectivity of computers. Users require protection of information from unauthorized access and alteration. System experts have drawn on the discipline of cryptography to meet the increasing needs for information security [2].

The word cryptography comes from the Greek words $\kappa\rho\upsilon\pi\tau o$ (hidden or secret) and $\gamma\rho\alpha\phi\eta$ (writing), hence cryptography is the art of secret writing [3]. More formally *cryptography* is the study of mathematical techniques related to the security services of information security.

The ITU-T X.800 [4] standard defines the security services provided by a system to give a specific kind of protection to system resources. X.800 divides security services into the following four categories:

- *Confidentiality*: This is a service to protect the content of information from all but those authorized to have it. Secrecy and privacy are synonymous with confidentiality.

University of Pretoria etd – Joseph, G (2005)

- *Data integrity*: This pertains to the unauthorized alteration of data. To ensure data integrity, it must be possible to detect data manipulation by unauthorized parties. Data manipulation includes operations such as insertion, deletion or substitution.

- *Authentication*: This service applies to the communicating parties as well as the information. Two parties involved in a communication should identify each other. Information delivered over a channel should be authenticated regarding the origin, date of origin, data content, time sent, etc. For these reasons this service is subdivided into two major classes: entity authentication and data origin authentication.

- *Non-repudiation*: This service prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary. A procedure involving a trusted third party is needed to resolve the dispute.

Cryptographic techniques are fundamental to the implementation of these security services and may be divided into two classes: symmetric-key and public-key cryptography.

Symmetric-key cryptography requires a single secret key that is used for both encryption and decryption. The exchange of this secret key forms part of the key management problem, that is concerned with the secure distribution of keys to the communicating parties.

A major advance in cryptography came in 1976 with the publication by Diffie and Hellman of the concept of public-key cryptography (PKC) [5]. The primary feature of PKC is that it removes the need to use a single key for encryption as well as decryption. With PKC, a pair of matched keys is used, termed ”public” and ”private” keys. The public part of the key pair can be distributed publicly without compromising the security of the private key, which must be kept secret by the receiver. A message encrypted with the public key can only be decrypted with the corresponding private key. The key management problem is greatly simplified by the use of public-key cryptosystems.

1.2 MODULAR EXPONENTIATION

Most public-key cryptosystems used today are based on the difficulty of factorizing large integers as well as the difficulty to compute the discrete logarithm of a large integer. The

implementation of these public-key cryptosystems requires modular exponentiations. In Chapter 2 a detailed overview of public-key cryptosystems based on modular exponentiation is given. These include the RSA algorithm [6], the Diffie-Hellman key exchange scheme [5], the ElGamal algorithm [7] and the Digital Signature Standard [8]. The operational speed of these public-key cryptosystems is largely determined by the speed of the modular exponentiation operation, which may be stated as follows:

$$A = g^e \bmod m \tag{1.1}$$

where g is the base, e is the exponent and n is the modulus. The required modular exponentiation is computed by a series of modular multiplications. This is performed in two steps: first an integer multiplication is done followed by a reduction by modulo m. The implementation of a public key cryptosystem can be modelled as a hierarchical structure that reflects the various mathematical operations that are required, as shown in Fig. 1.1.

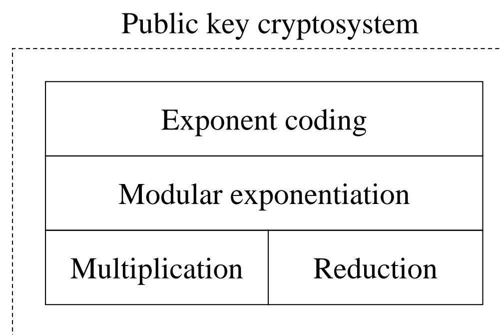


Figure 1.1: Graphical illustration of the PKC implementation

Fig. 1.1 depicts the following:

- *Modular multiplication layer*: The lowest layer consists of a integer multiplication and a modular reduction step. When combined they form a single modular multiplication.
- The *modular exponentiation layer* consists of a series of modular multiplications. The number of modular multiplications depends on the exponent's characteristics.
- *Exponent coding layer* involves the modification or manipulation of the exponent, which is applicable to fixed exponent public key cryptosystems such as the RSA algorithm.

University of Pretoria etd – Joseph, G (2005)

In this dissertation, we concentrate on the development of high-speed algorithms for public-key cryptosystems. In this framework a complete study of the basic components of a modular exponentiation, depicted in Fig. 1.1, which forms the core of the public-key cryptosystem is performed.

1.3 OBJECTIVES

> *"The computational cost of software cryptography is a function of the underlying algorithm and the quality of implementation of the algorithm ."*
>
> P. ROGAWAY AND D. COPPERSMITH [9]

Public key cryptographic functions require operations with elements of a large finite group, and need to be optimized for the chosen platform for high-speed implementation. For example, the RSA cryptosystem uses modular arithmetic algorithms with large integers, usually in the range of 1024 to 2048 bits. Arithmetic with such large integers is time consuming for most PKC applications [10]. Other public-key cryptosystems, described in Chapter 2, also require implementation of modular arithmetic algorithms with large integers.

Software implementations of these modular arithmetic algorithms are often desired because of their flexibility and cost effectiveness. The layers depicted in Fig. 1.1 can be implemented in software. These software implementations need to be efficiently designed to accommodate processing of large integer arithmetic efficiently. Hence the aim of this dissertation is to develop and implement integer-arithmetic algorithms that will enhance the speed of the public-key cryptosystem.

1.4 RESEARCH CONTRIBUTION

The contributions made by this dissertation:

- Various integer multiplication, modular reduction and modular exponentiation algorithms are developed and implemented in software. The dissertation provides a detailed analysis of these algorithms, as well as exact measurement of the computational speed achieved by each algorithm.

University of Pretoria etd – Joseph, G (2005)

- A thorough analysis of addition chains, and how each exponentiation method can be expressed in terms of addition chains is investigated. The simulations were conducted for various window sizes and various Hamming weight exponents.
- Three new methods for fast RSA decryption are proposed in Chapter 6. These techniques are implemented and exact simulation results are obtained. These methods allow the reduction of the size of the decryption exponent from the industry standard 1024-bits to 412-bits, taking the necessary security considerations into account.

The publications and reports either emanated or benefited during the completion of this dissertation are [11, 12, 13, 14, 15, 16, 17].

1.5 DISSERTATION OUTLINE

This dissertation consists of seven chapters in which the high-speed integer arithmetic aspects of public-key cryptosystems are discussed.

Chapter 2 provides a review of public key cryptosystems that are based on modular exponentiation. These include the RSA algorithm [6], the Diffie-Hellman key exchange scheme [5], the ElGamal algorithm [7] and the Digital Signature Standard [8]. A concise overview of the each cryptosystem's algorithm, its security and applications is given.

Integer multiplication forms one part of the modular multiplication step. Chapter 3 analyzes the different methods that implement this operation, namely the Classical [18], the Comba [2] and the Karatsuba-Ofman [19] methods. A significant portion of the modular exponentiation involves squarings. This chapter will adapt each of the above multiplication methods to perform squaring. This chapter provides simulation results to compute the computational speed of each one of the multiplication methods.

Considerable effort was invested in the design of efficient modular reduction methods. Reducing the computational complexity of these methods is addressed in Chapter 4. The chapter provides a detailed analysis and implementation of the Classical [18], Barrett [20] and Montgomery [21] algorithms. The chapter concludes with a comparison of the three methods and provides exact simulation results of each of method's performance in a modular

University of Pretoria etd – Joseph, G (2005)

exponentiation.

Chapter 3 and Chapter 4 provide algorithms to reduce the time required for a modular multiplication. Chapter 5 focuses on how to reduce the number of modular multiplications required for a modular exponentiation. This chapter investigates the Binary [18], K-ary [18], Sliding window [22] and Addition chain methods. It conducts an addition chain length analysis of the methods with various Hamming weights of the exponent. The chapter provides exact simulation results for each method.

Further speed enhancements can be made by modifying or manipulating the exponent of the modular exponentiation. However, this method only works on cryptosystems where the exponent is fixed, eg. the RSA cryptosystem. For the RSA cryptosystem the encryption is a fast operation, since the exponent is very short. However the decryption procedure is very slow, due to the fact that the decryption exponent is generally very large. This fact presents a problem in many applications of the RSA algorithm. Chapter 6, in addition to analyzing the use of the Chinese Remainder Theorem method for faster RSA decryption [23], proposes three novel methods for choosing the RSA decryption exponent. The chapter gives a complete analysis, the security risks of such selections and exact simulation results of each of the proposed methods.

Finally, Chapter 7 summarizes the research that has been done in the dissertation, and highlights the most outstanding results. Proposals for future research are made, based on the results and topics discussed in this dissertation.

University of Pretoria etd – Joseph, G (2005)

CHAPTER TWO

PUBLIC-KEY CRYPTOSYSTEMS

”The 1976 publication of New Directions in Cryptography was epochal in cryptographic history. Many regard it as the beginning of public-key cryptography, analogous to a first shot in what has become an ongoing battle over privacy, civil liberties, and the meaning of sovereignty in cyberspace.”

ALAN WESTROPE, 1998

The concept of public-key cryptography was introduced by Diffie and Hellman in 1976. Their contribution to cryptography was the notion that keys could come in pairs (an encryption key and a decryption key) and that one could not generate one key from the other. Since 1976, numerous public-key cryptosystems have been proposed. Many of these were very insecure. Of those still considered secure, many were impractical (the key was too large or the ciphertext was much larger than the plaintext). Only a few algorithms were both secure and practical [24].

The chapter will focus on the following public-key cryptosystems:

- The Diffie-Hellman key exhange (DH) [5],
- The ElGamal algorithm [7],
- The Digital Signature Standard (DSS) [8] and
- The Rivest Shamir Adleman algorithm (RSA) for both encryption and digital signatures [6].

University of Pretoria etd – Joseph, G (2005)

This chapter will give a complete analysis of the above listed public-key cryptosystems. The aim of this chapter is not to emphasize the security aspects of public-key cryptosystems, but to give a comprehensive summary of the cryptosystems with respect to their applicability to fast modular algorithms and their applications. For the sake of completeness, the security of each public-key cryptosystem will be briefly discussed.

2.1 DIFFIE-HELLMAN KEY EXCHANGE

Diffie-Hellman (DH) was the first public-key algorithm invented in 1976. It provided the first practical solution to the key management problem, allowing two parties, never having met in advance or shared keying material, to establish a shared secret by exchanging messages over an open channel [25].

DH is used in key distribution. However it cannot be used to encrypt and decrypt messages in its basic form, due to its incapability to provide entity authentication. Bellovin and Merritt [26] propose a modification to the basic DH algorithm that enables it to be used for encryption and decryption.

2.1.1 The Algorithm

The objective of the key-exchange algorithm is that the communicating parties can securely distribute the shared key k amongst themselves over an open channel. The following algorithm describes the key-exchange operation between parties A and B.

ALGORITHM: DIFFIE HELLMAN KEY EXCHANGE

1. *One-time setup.* An appropriate prime p and generator α is selected where $(2 \leq \alpha \leq p-2)$
2. *Protocol messages.*

   $A \rightarrow B : \alpha^x \bmod p \;\; (1)$

   $B \rightarrow A : \alpha^y \bmod p \;\; (2)$
3. *Protocol actions.* Perform the following steps each time a shared key is required.

   3.1 A chooses a random secret x, $1 \leq x \leq p-2$, and sends B message (1).

   3.2 B chooses a random secret y, $1 \leq y \leq p-2$, and sends A message (2).

   3.3 B receives α^x and computes the shared key as $k = (\alpha^x)^y \bmod p$.

   3.4 A receives α^y and computes the shared key as $k = (\alpha^y)^x \bmod p$.

University of Pretoria etd – Joseph, G (2005)

The algorithm is adapted from [25]. Parties A and B computed k independently, hence making DH suitable for creating keys over a public domain. Outsiders cannot compute k as only p, α, $\alpha^x \bmod p$ and $\alpha^y \bmod p$ are publicly known. In order to recover x, y and k, the attacker must compute a discrete logarithm. The algorithm can be visualized in Fig. 2.1.

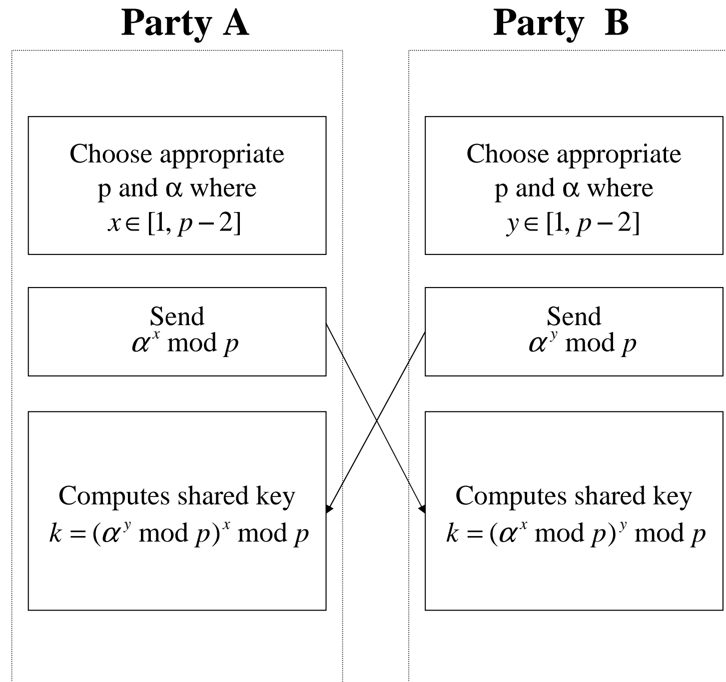


Figure 2.1: Diffie Hellman key exchange between party A and B

A variant [27] of the algorithm provides mutual key authentication: Fix $\alpha^x \bmod p$ and $\alpha^y \bmod p$ as long-term public keys of the respective parties, then distribute them with the use of signed certificates, thus fixing the long-term shared key for the user pair to $k = \alpha^{xy} \bmod p$ [25].

2.1.2 Security of the Algorithm

DH's security is based on the difficulty of calculating discrete logarithms in a finite field as opposed to computing an exponentiation in the same field. It is based on exponentiation modulo a large prime number p. Attacks to this cryptosystem are determined by a security parameter that relates directly to the key size.

The current safe[1] key-size of the length of the modulus p in bits is 1024 bits. Another

[1] Attacks based on Pollard's methods and the General Number Field Sieve

University of Pretoria etd – Joseph, G (2005)

security factor is the size of exponent used in the exponentiation. In DH the exponents are usually the same size as the modulus p, but they can be reduced to between 160 and 256 bits. This is safe in certain conditions shown in [28].

The basic version provides protection in the form of secrecy of the resulting key from passive adversaries, but not from active adversaries capable of intercepting, modifying, or injecting messages (man-in-the-middle attacks). Neither party has assurance of the source identity of the incoming message or the identity of the party that may know the resulting key [25].

DH can be extended to work in communitive rings, as shown in [29]. Shmuley [30] and McCurley [31] discuss a variant of the algorithm where the modulus is a composite number. Koblitz [32] extended this algorithm to elliptic curves. ElGamal [7] uses the basic idea of DH to develop an encryption and digital signature algorithm.

Extensions of the DH algorithm allows key-exchange with more than two parties. Hughes [33] proposes a variant to allow multiple parties. The advantage of this variant over basic DH is that shared key k can be computed before any interaction. Party A can encrypt a message using k prior to contacting Party B. Party A then sends it to a variety of people and interacts with them individually to exchange the key [24].

DH key-exchange is vulnerable to man-in-the-middle attacks. In order to prevent this problem, both parties must sign their messages before sending it to each other. This is referred to as STS (Station-to-Station Protocol) and is comprehensively explained in [34]. This protocol assumes that Party A has a certificate with Party B's public-key and vice versa. These certificates are signed by a trusted authority outside the protocol. Other variants of the DH algorithm that improve its security are shown in [35, 36, 37, 26].

2.1.3 Applications of DH

For online communications such as web-browsing, it is possible to encrypt the communications session with a key passed from one party to another. In the online case, it is possible to achieve a property called forward secrecy where if either of the keys

University of Pretoria etd – Joseph, G (2005)

are compromised then the past session key remains secure [38]. DH is suited for such applications that require active exchange.

2.2 THE ELGAMAL ALGORITHM

In 1985 ElGamal [7] proposed an alternative public-key cryptosystem. This algorithm, an extension of the DH algorithm, depends on the difficulty of computing discrete logarithms over finite fields. The ElGamal algorithm can be used for both encryption and digital signatures.

2.2.1 The Algorithm

The ElGamal algorithm requires an initial key generation step. Each entity needs to create a public key and a corresponding private key. Hence each party must do the following:

ALGORITHM: ELGAMAL KEY GENERATION

1. Generate a large random prime p and a generator α
2. Select a random integer a, $1 < a < p - 2$, and compute $\alpha^a \bmod p$.
3. The public key is (p, α, α^a) and the private key is (a).

The encryption and decryption algorithms are similar to DH, differing in that it requires two computed parameters γ and δ to be sent to the other party. The encryption and decryption procedures for Party A (who encrypts a message m) to Party B (who performs the decryption on the ciphertext to obtain m) is as follows:

ALGORITHM: ELGAMAL PUBLIC-KEY ENCRYPTION

1. *Encryption*. Party A must do the following:
   1.1 Obtain B's authentic public key (p, α, α^a).
   1.2 Represent the message as an integer m in the range $(0, 1, ..., p - 1)$.
   1.3 Select a random integer k such that $1 < k < p - 2$.
   1.4 Compute $\gamma = \alpha^k \bmod p$ and $\delta = m \cdot (\alpha^a)^k \bmod p$.
   1.5 Send the ciphertext $c = (\gamma, \delta)$ to A.
2. *Decryption*. To recover plaintext m from c, B must do the following:
   2.1 Use the private key a to compute $\gamma^{p-1-a} \bmod p$ note that $\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$.
   2.2 Recover m by computing $(\gamma^{-a}) \cdot \delta \bmod p$.

University of Pretoria etd – Joseph, G (2005)

The algorithm is adapted from [25] and can be visualized in Fig. 2.2.

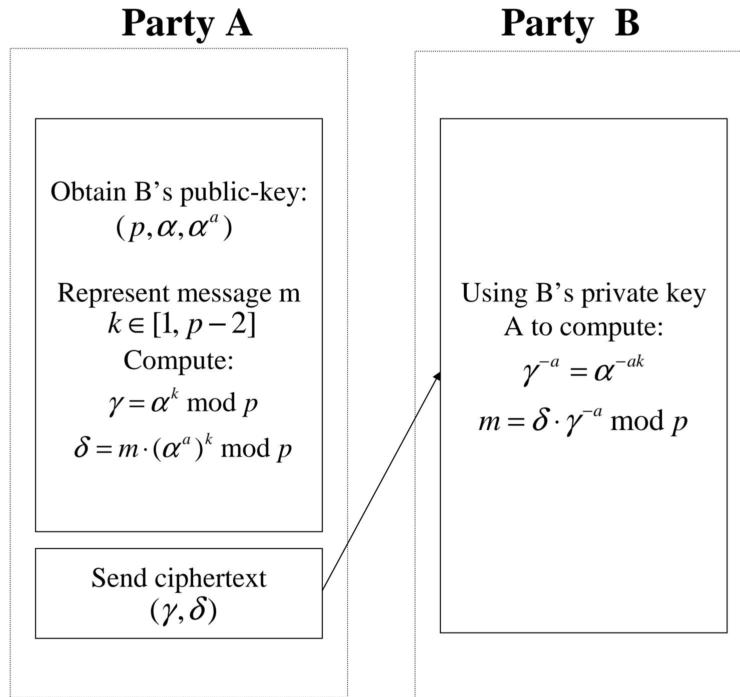


Figure 2.2: ElGamal public-key encryption

The ElGamal signature scheme is a randomized signature mechanism. Party A signs a binary message m of arbitrary length. Any Party B can verify this signature by using A's public key (in this case $y = \alpha^a \bmod p$ and $h(m)$ is the hash function).

ALGORITHM: ELGAMAL DIGITAL SIGNATURE GENERATION AND VERIFICATION

1. *Signature generation*. Party A must do the following:
   1.1 Select a random secret integer k, $1 < k < p-2$, with $\gcd(k, p-1) = 1$.
   1.2 Compute $r = \alpha^k \bmod p$.
   1.3 Compute $k^{-1} \bmod (p-1)$.
   1.4 Compute $s = k^{-1}\{h(m) - ar\} \bmod (p-1)$.
   1.5 As signature for m is the pair (r, s).
2. *Verification*. To verify Party A's signature (r, s) on m, Party B must do the following:
   2.1 Obtain A's public key (p, α, y).
   2.2 Verify that $1 < r < p-1$; if not, then reject the signature.
   2.3 Compute $v_1 = y^r r^s \bmod p$.
   2.4 Compute $h(m)$ and $v_2 = \alpha^{h(m)} \bmod p$.
   2.5 Accept the signature if and only if $v_1 = v_2$.

University of Pretoria etd – Joseph, G (2005)

The algorithm generates digital signatures with appendix on binary messages of arbitrary length, and requires a hash function $h : \{0,1\}^*$ [25]. The DSA algorithm is a variant of the ElGamal signature mechanism.

Signature generation by ElGamal is relatively fast. It requires one modular exponentiation ($\alpha^k \bmod p$), the extended Euclidean algorithm ($k^{-1} \bmod (p-1)$), and two modular multiplications. The exponentiation and application of the extended Euclidean algorithm can be done independently, in which case the signature generation (in instances where precomputation is possible) requires only two modular multiplications [25]. The signature generation algorithm can be visualized in the left-hand side of Fig. 2.3.

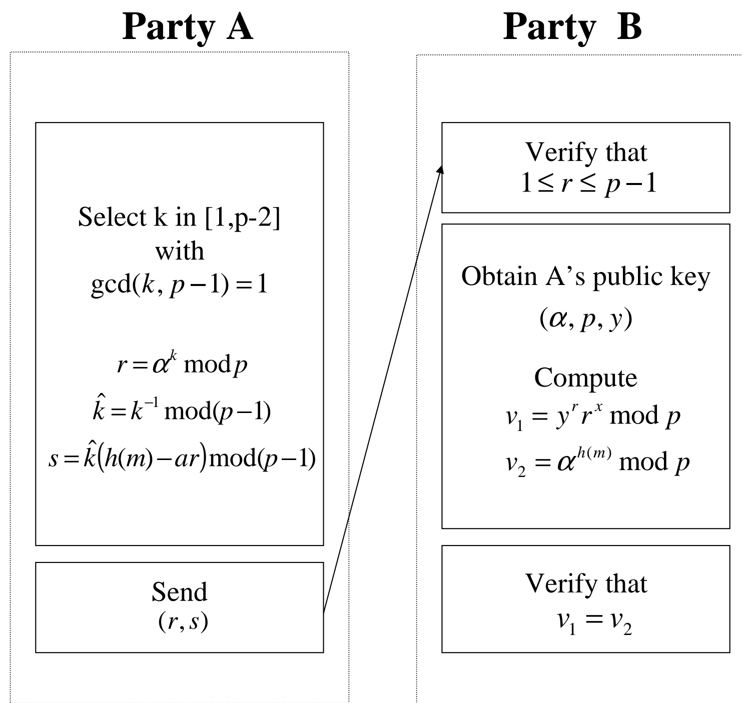


Figure 2.3: ElGamal signature generation and verification

Signature verification is more costly, requiring three exponentiations. Signature verification calculations are all performed modulo p, while signature generation calculations are done modulo p and modulo $(p-1)$ [25]. The verification algorithm can be visualized in the right-hand side of Fig. 2.3.

University of Pretoria etd – Joseph, G (2005)

2.2.2 Security of the Algorithm

The ElGamal encryption comprises of a DH key-exchange to determine the session key α^{ak}. Hence the problem of recovering m given p, α, α^a, δ and γ is similar to the the DH problem given in Section 2.1.2. For this reason, the security of the ElGamal encryption algorithm is also based on the discrete logarithm problem [25].

A variant of ElGamal for signatures is provided in [39]. Beth [40] proposes a variant that enables ElGamal to be used for proofs of identity. There are also variants for password authentication [41] and for key-exchange [42]. ElGamal can also be modified to implement encrypted key-exchange, as shown in [26].

Parameter selection is critical for the security of the ElGamal signature algorithm. Incorrect parameter selection results in index-calculus and Pohlig-Hellman attacks. According to the latest progress of the discrete logarithm problem provided in [25], a 512-bit modulus provides marginal security from concerted attack. As of 1996 a modulus p of at least 768 bits is recommended. For long term security a 1024-bit or larger modulus must be used.

2.2.3 Applications of ElGamal

"In God we trust. Everybody else we verify using PGP."

TIM NEWSOME, 1999

Pretty good privacy (PGP), the workhorse known throughout the world for encrypting and signing e-mail and documents, uses the ElGamal procedure for its key management. DSS uses the ElGamal algorithm as its basis for its signature scheme.

2.3 DIGITAL SIGNATURE STANDARD (DSS)

NIST, the U.S. National Institute of Standards and Technology, had proposed an algorithm for digital signatures. The algorithm is known as Digital Signature Algorithm (DSA). As a proposed standard it is known as the Digital Signature Standard (DSS). The DSA algorithm is due to Kravitz [43] and was proposed as a Federal Information Processing Standard in

University of Pretoria etd – Joseph, G (2005)

August 1991 by NIST. It became the Digital Signature Standard (DSS) in May 1994, as specified in FIPS 186 [8].

2.3.1 The Algorithm

The signature mechanism requires a hash function $h : \{0,1\}^* \rightarrow Z_q$ for an integer q, more explicitly it requires use of the Secure Hash Algorithm (SHA-1 [44]). The following algorithms are adapted from [25] and [24].

For the generation of DSA primes p and q in the algorithm below one must select the prime q first and then try to find a prime p such that q divides $(p-1)$. Each party creates a public key and corresponding private key. Each party must do the following:

ALGORITHM: DSA KEY GENERATION

1. Select a prime number q such that $2^{159} < q < 2^{160}$.
2. Choose t so that $0 < t < 8$, and select a prime number p where $2^{511+64t} < p < 2^{512+64t}$, with the property that q divides $(p-1)$.
3. Select a generator α of the unique cyclic group of order q in Z_p.
   3.1 Select an element $g \in Z_p^*$ and compute $\alpha = g^{(p-1)/q} \bmod p$.
   3.2 If $\alpha = 1$ then go to above step.
4. Select a random integer a such that $1 < a < q-1$.
5. Compute $y = \alpha^a \bmod p$.
6. The public key is (p, q, α, y); and the private key is (a).

The above algorithm is an ElGamal extended digital signature scheme with appendix deployed by Schnorr [45]. Performance is the main differences between DSA and ElGamal algorithms. ElGamal computes all exponentiations in modulo p (where p is 512 to 1024 bit prime), whereas DSA computes certain exponentiations in modulo q (where q is a 160-bit prime). This makes DSA much faster than ElGamal. DSA is also slower than ElGamal in certain aspects, in particular the extra inverse calculation required by both the signer and verifier using DSA [3].

Party A can generate a signature on a binary message m of arbitrary length and any party B can verify this signature by using A's public key. It proceeds as follows:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: DSA SIGNATURE GENERATION AND VERIFICATION

1. *Signature generation.* Party A must do the following:
   1.1 Select a random secret integer k, $0 < k < q$.
   1.2 Compute $r = (\alpha^k \bmod p) \bmod q$.
   1.3 Compute $k^{-1} \bmod q$.
   1.4 Compute $s = k^{-1}(h(m) + ar) \bmod q$.
   1.5 Party A's signature for m is the pair (r, s).
2. *Verification.* To verify Party A's signature (r, s) on m, Party B must do the following:
   2.1 Obtain A's public key (p, q, α, y).
   2.2 Verify that $0 < r < q$ and $0 < s < q$; if not, then reject the signature.
   2.3 Compute $w = s^{-1} \bmod q$ and $h(m)$.
   2.4 Compute $u_1 = w.h(m) \bmod q$ and $u_2 = rw \bmod q$.
   2.5 Compute $v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$.
   2.6 Accept the signature if and only if $v = r$.

The visualization of the algorithm is shown in Fig. 2.4.

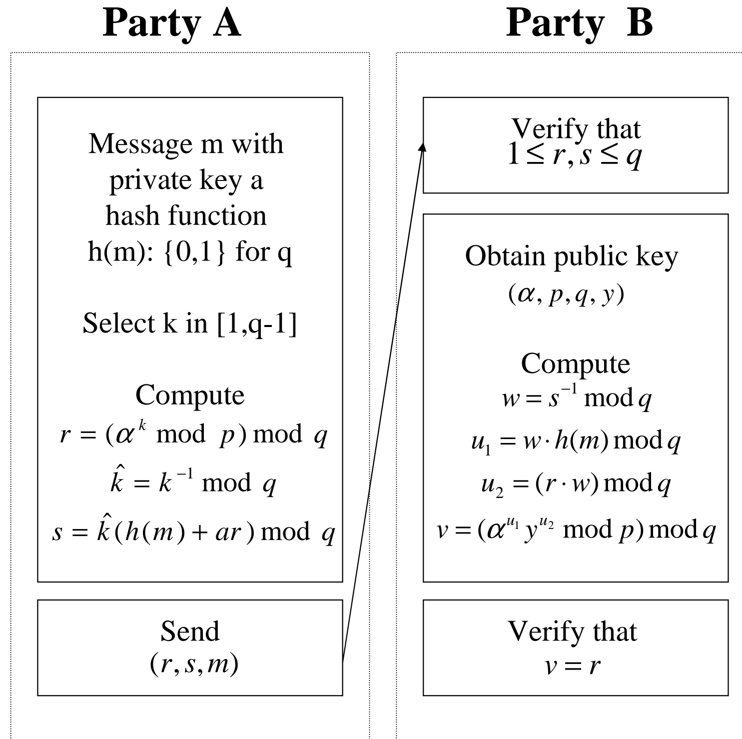


Figure 2.4: DSA signature generation and verification

University of Pretoria etd – Joseph, G (2005)

A detailed proof of the signature verification is shown in [8]. Signature generation requires one modular exponentiation, one modular inverse (using a 160-bit modulus), two 160-bit modular multiplications, and one addition. The 160-bit operations are relatively small compared to the exponentiation operation. DSA has the advantage that the exponentiation can be precomputed [25].

Naccache *et al.* [46] discusses techniques for improving the efficiency of the DSA algorithm. They propose that the computation of $k^{-1} \bmod q$ in step 1.3 of the signature algorithm be replaced by the generation of an integer b. The computation would hence change to $u = bk \bmod q$ and $s = b \cdot h(m) + ar \bmod q$. The resulting signature then contains (r, s, u). The verifier can then compute $u^{-1} \bmod q$ and $u^{-1}s \bmod q = \tilde{s}$. This type of signature generation is beneficial in computationally constricted environments.

2.3.2 Security of the Algorithm

The security of the DSA has two distinct but related discrete logarithm problems. Firstly the logarithm problem in Z_p where the powerful index-calculus methods apply, and secondly the logarithm problem in the cyclic subgroup of order q. A complete analysis of the security of the DSA algorithm is shown in [24]. The size of q is fixed at 160-bits, while p can be any multiple of 64 between 512 and 1024-bits inclusive. A 512-bit modulus p provides marginal security against a concerted attack. However for long term security a modulus of 1024-bits is recommended.

Yen [47] and McCurley [48] propose extensions to the DSA algorithm that improves the computation speed of the verification procedure. The extension works as follows: To sign a message m, party A generates a random number k less than q. The signature is then computed as $r = (\alpha^k \bmod p) \bmod q$ and $s = k \cdot (h(m) + xr^{-1})^{-1} \bmod q$. Party B verifies the signature by computing $u_1 = (h(m) \cdot s) \bmod q$ and $u_2 = (sr) \bmod q$. Now if $r = ((\alpha^{u_1} \cdot y^{u_2}) \bmod p) \bmod q$, the signature is verified. Lim and Lee [49] proposes another variant that allows batch verification where party B can verify signatures in batches. For additional information of this variant, refer to [46].

Naccache *et al.* [46] also proposed the idea of ”use-and-throw” coupons which eliminate

University of Pretoria etd – Joseph, G (2005)

the computation of $r = (\alpha^k \bmod p) \bmod q$. Since this exponentiation is the most computationally intensive portion of DSA signature generation, ”use-and-throw” coupons can greatly its efficiency. Coupons require storage, and only one signature can be created for each coupon. Since there is limited storage, only a fixed number of DSA signatures can be created with this method [25].

2.3.3 Applications of DSS

There is one important application which will benefit from the DSA algorithm: smart card signature generation. A smart card generally has a low performance processor. It will require to perform a signature before the user can successfully login into a network. The time of the signing operation is critical to the user. If DSA inverse operation is precomputed before the signing operation is performed, the time required for signature generation will be greatly decreased.

2.4 THE RSA ALGORITHM

The RSA cryptosystem [6], named after its inventors Rivest, Shamir, and Adleman, is the most widely used public-key cryptosystem. It may be used to provide both encryption and digital signatures. Of all the public-key algorithms proposed thus far, RSA is by far the easiest to understand and implement. Its security, unlike the public-key cryptosystems presented before, is based on the difficulty of the integer factorization (recovering the plaintext from the public key and the ciphertext is equivalent to factoring the product of two primes) [25].

The algorithm can be briefly described as follows: Let p and q be two distinct large random prime integers. The modulus n is the product of these two primes ($n = pq$). Hence Euler's totient function of n, $\phi(n)$, is computed as

$$\phi(n) = (p-1)(q-1) \tag{2.1}$$

Now select the encryption exponent e such that

$$\gcd(e, \phi(n)) = 1 \tag{2.2}$$

University of Pretoria etd – Joseph, G (2005)

The decryption exponent d can be computed using the extended Euclidean algorithm [18] as:

$$d = e^{-1} \bmod (\phi(n)) \tag{2.3}$$

It can be proven that d and n are also relatively prime, the proof is shown in [25]. The public key is (e, n) and the private key is (d). The decryption exponent d and the two primes p and q must be kept secret.

Encryption is performed on a message m such that $0 \leq m \leq n$.

$$c = m^e \bmod n \tag{2.4}$$

If the message is larger than the modulus, it can be broken into smaller pieces and encrypted piece by piece. Usually one selects a small public exponent for e, $e = 2^{16} + 1$ is a popular choice [50, 25, 24]. c is the ciphertext produced by Eq. 2.4.

The decryption is computed as follows:

$$m = c^d \bmod n \tag{2.5}$$

The correctness of Eq. 6.3 can be proven by Euler's theorem. A detailed explanation is given [50]. The RSA algorithm can be used for signing and verifying. More information on this is given in Section 2.4.1.2.

2.4.1 The Algorithm

The following algorithms are a generalization of what has been described already and are adapted from [25]. In order to implement the algorithms, each party creates an RSA public key and a corresponding private key. Each party must do the following:

ALGORITHM: RSA KEY GENERATION

1. Generate two large random and distinct primes p and q, of the same length.
2. Compute $n = pq$ and $\phi(n) = (p-1)(q-1)$.
3. Select a random integer e where $1 < e < \phi(n)$, such that $gcd(e; \phi(n)) = 1$.
4. Use the extended Euclidean algorithm to compute the decryption key d such that $1 < d < \phi(n)$, such that $ed \equiv 1 (\bmod \phi(n))$.
5. The party's public key is (n, e) and the private key is (d)

University of Pretoria etd – Joseph, G (2005)

2.4.1.1 The RSA Encryption Algorithm

The RSA encryption algorithm allows party A to encrypt a message m for B, which only B can decrypt. This procedure, using the key generation described in Section 2.4.1, is done as follows:

ALGORITHM: RSA ENCRYPTION

1. *Encryption*. Party A must do the following:
   1.1 Obtain party B's public key (n, e).
   1.2 Represent the message as an integer m in the interval $[0, n-1]$.
   1.3 Compute ciphertext $c = m^e \bmod n$.
   1.4 Send the ciphertext c to party B.
2. *Decryption*. To recover plaintext m from the ciphertext c, party B should do the following:
   2.1 Use the decryption exponent d to recover $m = c^d \bmod n$.

The RSA algorithm requires a fixed exponentiation that is essential to its security and speed. The exponent e is fixed and arbitrary choices of the base m are allowed. Encryption can be sped up by selecting e to be small or to have a low Hamming weight. The decryption can be sped up by using the Chinese Remainder Theorem, as shown in [51].
The encryption exponent $e = 65537$ is often used in practice [25, 50]. This integer has only two 1's in its binary representation, hence the encryption operation requires 15 modular squarings and 1 modular multiplication[2]. This results in a very fast encryption operation.

2.4.1.2 The RSA Signature Algorithm

> *"Please, your Majesty," said the Knave, "I didn't write it, and they can't prove that I did: there is no name signed at the end."*
>
> LEWIS CARROLL, *Alice's Adventures in Wonderland*

The message m and ciphertext c for the RSA public-key encryption occurs in $Z_n = 0, 1, 2, ..., n-1$ where $n = pq$ is the product of two randomly chosen distinct prime numbers. Digital signatures can be created by reversing the roles of encryption and decryption. In essence Party A signs message m by creating:

$$\tilde{m} = m^d \bmod n \tag{2.6}$$

[2] Modular squarings and multiplications are done repeatedly in an modular exponentiation

University of Pretoria etd – Joseph, G (2005)

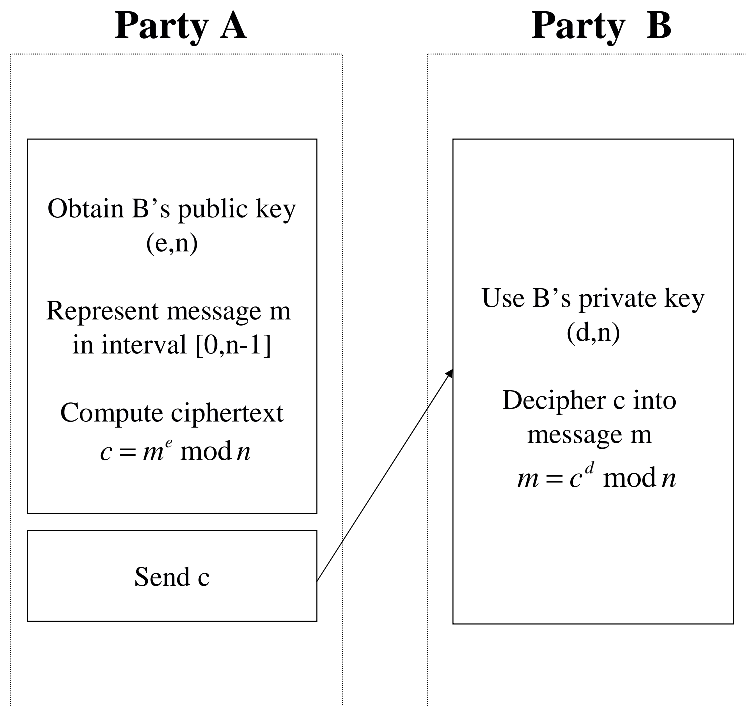


Figure 2.5: RSA encryption and decryption

Party A sends $(\tilde{m}, e)$, once Party B receives $(\tilde{m}, e)$, it computes

$$\tilde{m}_1 = \tilde{m}^e \bmod n \tag{2.7}$$

If $m = \tilde{m}_1$ then the signature has been successfully verified. However a more secure algorithm than what is discussed in the above paragraph can be implemented. The algorithm is as follows:

ALGORITHM: RSA SIGNATURE GENERATION AND VERIFICATION

1. *Signature generation*. Party A must do the following:
   1.1 Compute $\tilde{m} = R(m)$, an integer in the range $[0; n-1]$.
   1.2 Compute $s = \tilde{m}^d \bmod n$.
   1.3 A's signature for m is s.
2. *Verification*. To verify A's signature s and recover the message m, B must:
   2.1 Obtain A's authentic public key (n, e).
   2.2 Compute $\tilde{m} = s^e \bmod n$
   2.3 Verify that $\tilde{m} \in \tilde{M}$; if not, reject the signature.
   2.4 Recover $m = R^{-1}(\tilde{m})$.

University of Pretoria etd – Joseph, G (2005)

The RSA digital signature scheme was the first practical signature scheme based on public-key techniques. It is a deterministic digital signature with appendix[3] that can be modified to provide message recovery [25]. $\tilde{m}$ is the redundancy generated by A where $R(m)$ is the redundancy function. $R(m)$ maps arbitrary messages of m from a message space $\tilde{M}$ into the Z_n domain.

The redundancy function is a better alternative to breaking the message m into blocks. The blocks can be mixed up and counterfeiting of signatures can occur using the keys of one of the messages to duplicate the other messages.

Signature generation is a generalization of the RSA decryption procedure. The signature generation is shown in the left-hand side of Fig. 2.6.

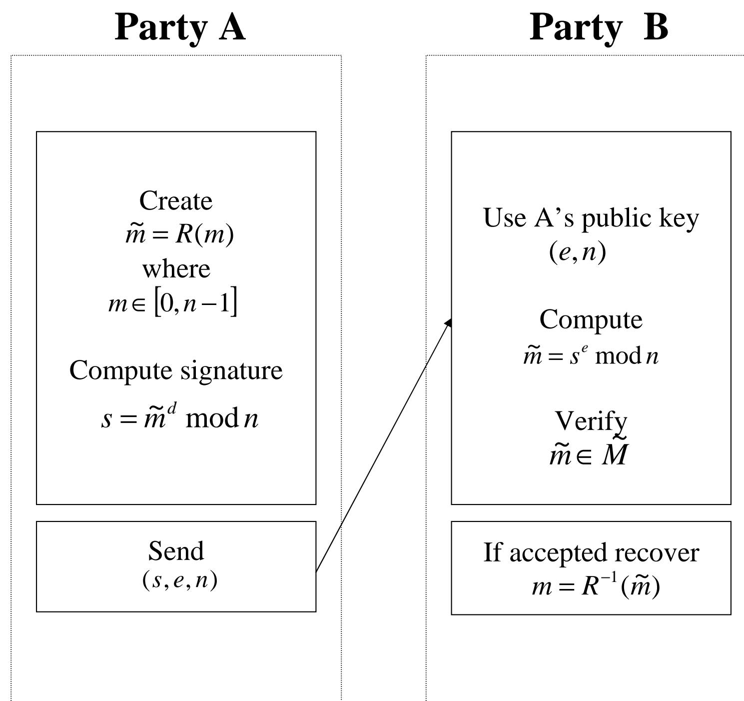


Figure 2.6: RSA signature generation and verification

Signature Verification can be much faster than signing if the public exponent is chosen to be a small number, i.e. $e = 65537$. The verification is shown in the right-hand side of Fig. 2.6.

The RSA algorithm can be sped up utilizing software and hardware implementations.

[3] Digital signatures that must be checked by a sperate transmission of message m is called digital signatures with appendix

University of Pretoria etd – Joseph, G (2005)

Good surveys of hardware implementations are described in [52, 53, 54, 55]. To speed up the algorithm in software, efficient number-theoretical methods are required. Surveys of these requirements are shown in [50, 56, 57].

2.4.2 Security of the Algorithm

The security of the RSA cryptosystem, as mentioned in Section 2.4, depends on the problem of factoring large numbers. The computational equivalence of computing the decryption exponent and factoring the modulus, shown by Rivest *et al.* [6], was based on earlier work done by Miller [58]. It has never been mathematically proven that in order to factor a large integer, one must have at least one factor. It is conceivable that there might be an entirely different way to break the RSA cryptosystem [24].

The RSA cryptosystem can be attacked using a forward search attack. If the message space is small or predictable, an adversary can decrypt the ciphertext by simply encrypting all possible plaintext messages. Håstad [59] discusses the attacks associated with choosing small encryption exponents. Recommendations to prevent attacks on RSA choosing a small decryption exponent are addressed by Wiener [60]. Kaliski *et al.* [61] provides an overview of the major attacks on RSA encryption and signatures, and the practical methods of counteracting these threats.

Further attacks and recommendations on the RSA algorithm are shown in [62, 63, 64, 65]. Rivest *et al.* [66] provides a set of recommendations to use strong primes in RSA key generation. Shamir [67] proposed a variant of the RSA encryption operation called ”unbalanced RSA” that makes it possible to enhance security by increasing the modulus size without any deterioration in performance.

Given the latest progress in algorithms[4] for factoring integers, a 512-bit modulus n offers only marginal security from concerted attack. For long term security, a 1024-bit or larger modulus must be used [25].

[4] The best attack known to RSA is the General Number Field Sieve (GNFS) which tries to factor the modulus into its original primes

University of Pretoria etd – Joseph, G (2005)

2.4.3 Applications of RSA

In practice, RSA encryption is most commonly used for the transport of symmetric keys. It is also used for the encryption of small data items.

The RSA signature operation is ideally suited to situations where signature verification is the predominant operation performed. For example, when a trusted third party creates a public-key certificate for party A, this requires only one signature generation. This signature may then be verified numerously by various other parties. ISO/IEC 9796 [68] provides criteria and examples based on the RSA signature operation. It became an international standard in October 1991. The ANSI X9.31 standard [69] defines a method for digital signature and verification of messages using the RSA algorithm. The standard provides criteria for generation of public and private keys required by the algorithm. The latest version of this standard was revised in 1998.

Certification Authority (CA) key pairs are used for signing and verifying the signatures on certificates and Certificate Revocation Lists (CRLs). The certificate is signed once but requires to be verified numerous times. Since the predominant method is signature verification, the RSA algorithm is best suited for this task. In order to send a secure email, the message needs to be signed and encrypted. The signature must then be verified by each recipient with the correct decryption key. Since RSA can be used for encryption and digital signatures, it provides suitable backbone for secure emails [38].

2.5 CHAPTER SUMMARY

Once a cryptosystem has set up the modulus, the private and public exponents are determined and the public components are published, the senders as well as the recipients perform a single operation for signing, verification, encryption, and decryption. The RSA algorithm in this respect is one of the simplest cryptosystems [50].

The operation most required is the computation of a modular exponentiation ($g^e \bmod n$). The modular exponentiation operation is a common operation for scrambling in each of the public key cryptosystems. However, the modular exponentiation in certain

University of Pretoria etd – Joseph, G (2005)

cryptosystems (i.e. DH, ElGamal and DSS) is based on the discrete logarithm problem. In these cryptosystems the base g and the modulus n are known in advance. This type of exponentiation is referred to as fixed base exponentiation. In the modular exponentiation of the RSA algorithm the exponent e and the modulus n are known in advance but not the base, hence RSA relies on fixed exponent exponentiation [50].

There is no ”best” public-key cryptosystem, as each cryptosystem is better suited for certain applications than the others. Comprehensive summaries of the discussed public-key cryptosystems are shown in [11, 38]. A comparison of practical public key cryptosystems, based on integer factorization and discrete logarithms, is given in [70].

This chapter gives a comprehensive outline of the popular types of cryptosystems used in industry. These cryptosystems consist of the same subsystems. The following chapters will review these subsystems and their effects on the performance of the cryptosystem.

University of Pretoria etd – Joseph, G (2005)

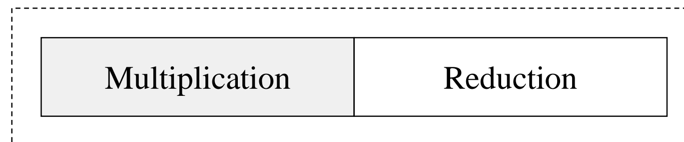
CHAPTER THREE

FAST MULTIPLICATION TECHNIQUES

"I don't know if we have any real chance. He can multiply and all we can do is add. He represents progress and I just drag my feet"

STEN NADOLNY, *God of Impertinence*

The modular multiplication operation is accomplished using two steps. It first computes a large-integer multiplication step followed by a modular reduction step. This chapter is concerned with the initial multiplication step, leaving the modular reduction step for the subsequent chapter.



The chapter will focus on the following multiplication algorithms:

- The Classical method [18],
- The Comba method [2], and
- The Karatsuba-Ofman method [19].

In public-key cryptography, a significant portion of the modular exponentiation operation involves squarings. This chapter will also show how to adapt each of the above multiplication algorithms to perform squaring. The chapter concludes with a comparison of the methods, giving exact numerical results obtained by means of simulation.

University of Pretoria etd – Joseph, G (2005)

3.1 THE CLASSICAL METHOD

3.1.1 Application to Multiplication

The product of two integers can be computed by means of the standard long-hand multiplication algorithm that is taught in grade school. The algorithm requires n^2 single-precision multiplications for two n-digit inputs denoted as a and b respectively. More specifically for a m and n digit input, $m \times n$ single precision multiplications are required.

A digit is defined as a B-bit integer. A single-precision multiplication is the multiplication of two base B digits, where B is the base and can be any positive integer. In computer implementations one base B digit is selected as 2^w, where w is the word-size of the processor. Typical examples in practice include $B = 8, 16, 32$ [50].

Knuth [18] refers to the Classical method as a multi-precision multiplication, as it computes $n \times m$ single-precision multiplications. Since multi-precision multiplication requires multi-precision integers, multiplicands a and b are defined as:

$$
\begin{aligned}
a &= a_{n-1}a_{n-2}...a_1 = \sum_{i=1}^{n-1} a_i B^i \\
b &= b_{m-1}b_{m-2}...b_1 = \sum_{i=1}^{m-1} b_i B^i
\end{aligned}
\tag{3.1}
$$

A 3×3 digit multi-precision multiplication, where $a = a_3a_2a_1$ and $b = b_3b_2b_1$, is depicted in Fig. 3.1.

			a_3	a_2	a_1
$\times$			b_3	b_2	b_1
			$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$
		$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	
	$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$		
t_6	t_5	t_4	t_3	t_2	t_1

Figure 3.1: (3×3) digit Classical multiplication

The Classical method multiplies each digit of b with the entire number a to obtain partial products t_{ij}. These partial products are then summed row-by-row to obtain the final product

University of Pretoria etd – Joseph, G (2005)

t which comprises of $(n+m)$-digits, where n and m are the respective digit sizes of a and b. The Classical multiplication algorithm, adapted from [18, 25], is given as follows:

ALGORITHM: CLASSICAL MULTIPLICATION

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$ and $b = (b_{m-1}b_{m-2}....b_1b_0)_B$

Output. $a \times b = t = t_{n+m-1}t_{n+m-2}....t_1t_0$ of base B

1. *Initialize*: For i from 0 to $(n+m-1)$ do: $t_i \rightarrow 0$.
2. *Zero multiply*: If $a = 0$ or $b = 0$ then return $t = 0$
3. *Multiply and add*: For i from 0 to $(n-1)$ do the following:
   3.1 Set $c \rightarrow 0$ (c is the carry)
   3.2 For j from 0 to $(m-1)$ do the following:
      Compute $(ul)_B = t_{i+j} + a_j \cdot b_i + c$, and set $l \rightarrow t_{i+j}$, $u \rightarrow c$.
   3.3 $u \rightarrow t_{i+m}$.
4. *Final result*: Return $(t_{n+m-1}...t_1t_0)$.

The computationally intensive part of the algorithm is step 3.2. Computing the inner product, $t_{i+j} + a_j \cdot b_i + c$, will require two base B digits to hold the carry and the remainder of the inner product.

3.1.2 Application to Squaring

The first description of a multiple-precision squaring was due to Tuckerman [71]. Squaring is a special case of multiplication where both multiplicands are equal. Fig. 3.2 is an adaption of Fig. 3.1 for the special case of squaring.

			a_3	a_2	a_1
$\times$			a_3	a_2	a_1
			$a_3 \cdot a_1$	$a_2 \cdot a_1$	$a_1 \cdot a_1$
		$a_3 \cdot a_2$	$a_2 \cdot a_2$	$a_1 \cdot a_2$	
	$a_3 \cdot a_3$	$a_2 \cdot a_3$	$a_1 \cdot a_3$		
t_6	t_5	t_4	t_3	t_2	t_1

Figure 3.2: (3×3) digit Classical squaring

Squaring an integer is more efficiently performed by using a specialized squaring algorithm than by using a multiplication algorithm. This is because in squaring there are many

University of Pretoria etd – Joseph, G (2005)

cross-product terms that need to be computed only once, whereas a multiplication algorithm would compute them twice [2].

It is can be seen from Fig. 3.2 that $t_{ij} = a_i \times a_j = t_{ji}$. Hence half of the single-precision multiplications can be avoided. Taking this characteristic into account, multi-precision squaring can be formulated as:

$$t = \sum_{i,j=0}^{n-1} a_i a_j B^{i+j} = 2 \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} a_i a_j B^{i+j} + \sum_{i=0}^{n-1} a_i^2 B^{2i} \tag{3.2}$$

The squaring algorithm, based on Eq. 3.2, is as follows:

ALGORITHM: CLASSICAL SQUARING

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$

Output. $a \times a = t = t_{2n-1}t_{2n-2}....t_1t_0$ of base B

1. *Initialize*: For i from 0 to $(2n-1)$ do: $t_i \rightarrow 0$.
2. *Multiply and add*: For i from 0 to $(n-1)$ do the following:
   - 2.1 $(ul)_B = t_{2i} + a_i \cdot a_i$, and set $l \rightarrow t_{2i}$, $u \rightarrow c$.
   - 2.2 For j from $(i+1)$ to $(n-1)$ do the following:
     Compute $(ul)_B = t_{i+j} + 2a_i \cdot a_j + c$, and set $l \rightarrow t_{i+j}$, $u \rightarrow c$.
   - 2.3 $u \rightarrow t_{i+n}$.
3. *Final result*: Return $(t_{2n-1}...t_1t_0)$.

The computationally intensive part of the algorithm is step 2. This step requires $(n^2+n)/2$ single-precision multiplications, discounting the multiplication by 2. This is significantly lower than the n^2 single-precision multiplications required by a Classical multiplication. The multiplication by 2, in step 2.2, can be computed using a simple left-shift.

3.2 THE COMBA METHOD

3.2.1 Application to Multiplication

The Classical method utilizes a double loop to compute each partial product and writes the lower B-digit product to the final result. Experiments in [2] identified that the loops requires

University of Pretoria etd – Joseph, G (2005)

a small portion of the execution time that can be avoided and the Classical method requires three memory accesses to the intermediate results during each loop. These inefficiencies were addressed Comba [2]. He described a way of reducing the number of memory accesses and removing the looping required by the Classical method.

To fully understand the Comba method, convert Fig. 3.1 to a pyramid of partial products as shown below:

			a_3b_1		
		a_3b_2	a_2b_2	a_2b_1	
	a_3b_3	a_2b_3	a_1b_3	a_1b_2	a_1b_1
t_6	t_5	t_4	t_3	t_2	t_1

Figure 3.3: Pyramid of partial products for 3×3 digit multiplication [72]

Comba unravelled both loops of the Classical method and computed the columns in Fig. 3.3 directly. Care must be taken with respect to the precision of each t_i where the column answer is kept. Looking at column that computes t_3 in Fig. 3.3, it is highly possible that the entire sum of that column can exceed two base B digits. Hence an extra digit is required to avoid an overflow.

The Comba algorithm is an in-line program, dependent on the number of base B digits contained by each multiplicand. The algorithm is given as follows:

ALGORITHM: COMBA MULTIPLICATION

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$ and $b = (b_{n-1}b_{n-2}....b_1b_0)_B$

Output. $a \times b = t = t_{2n-1}t_{2n-2}....t_1t_0$ of base B

1. *Compute with inline coding*. For i from 0 to $(2n-1)$ without looping:

   1.1 *Compute the column directly*:

   1.1.1 $t_i = \sum_{i=0}^{i+j=2n-1} a_jb_i$

   1.2 *Compute the carries and product* t_i:

   1.2.1 $t_{i+1} = t_{i+1} + \left\lfloor \frac{t_i}{B} \right\rfloor$

   1.2.2 $t_i = t_i \bmod B$

2. Return $t = t_{2n-1}...t_1t_0$

University of Pretoria etd – Joseph, G (2005)

To compute step 1 sequentially $2n$ times, three registers are needed. The first register will store the partial product, the second register will store the carry of the partial product. If there is an overflow in the carry register, a third register will be incremented so as not to lose the precision of the column. After each column has been computed, the result is written to t_i. If there is an overflow in the overflow register, a fourth register can be assigned and so forth.

Fig. 3.3 and Fig. 3.4 show that the partial products of the column are equivalent to the column number (i.e column 3 has 3 partial products). Since the column lengths vary, first increasing and then decreasing, the required coding can become complicated and time-consuming for larger integers.

3.2.2 Application to Squaring

Comba squaring is computed similarly to Classical squaring, as shown in Section 3.1.2. To understand this fully, adapt Fig. 3.3 for squaring as shown below

$$
\begin{array}{cccccc}
 & & & a_3a_1 & & \\
 & & a_3a_2 & a_2a_2 & a_2a_1 & \\
 & a_3a_3 & a_2a_3 & a_1a_3 & a_1a_2 & a_1a_1 \\
\hline
t_6 & t_5 & t_4 & t_3 & t_2 & t_1
\end{array}
$$

Figure 3.4: Pyramid of partial products for 3×3 digit squaring

Apart from the cross-product procedure, the optimization techniques that are applicable to multiplication are essentially the same for squaring. For this reason, the discussions that apply to Comba multiplication will also apply for Comba squaring [2].

From Fig. 3.4 it follows that $t_{ij} = a_i \times a_j = t_{ji}$, which is identical to Classical squaring. The Comba squaring algorithm can thus be formulated in accordance with Eq. 3.2:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: COMBA SQUARING

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$

Output. $a \times a = t = t_{2n-1}t_{2n-2}....t_1t_0$ of base B

1. *Compute with inline coding*. For i from 0 to $(2n-1)$ without looping:
   1.1 *Compute the column directly*:
      1.1.1 $t_i = 2\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} a_ia_j + \sum_{i=0}^{n-1} a_i^2$
   1.2 *Compute the carries and product* t_i:
      1.2.1 $t_{i+1} = t_{i+1} + \lfloor \frac{t_i}{B} \rfloor$
      1.2.2 $t_i = t_i \bmod B$
2. Return $t = t_{2n-1}...t_1t_0$

The explanation for the Comba multiplication algorithm, given in Section 3.2.1, also applies to the above Comba squaring algorithm.

3.3 THE KARATSUBA-OFMAN METHOD

3.3.1 Application to Multiplication

This method was introduced by Russian mathematicians Karatsuba and Ofman [19] in 1962. This recursive method was the first method that computed a multiplication less than $O(n^2)$ operations.

To explain the method, one must first decompose the n-bit multiplicands a and b into two separate and equal parts.

$$\begin{aligned} a &= 2^{n/2}a_1 + a_0 \\ b &= 2^{n/2}b_1 + b_0 \end{aligned} \tag{3.3}$$

a_1 and a_0 are the higher and lower $n/2$ bits of a respectively, assuming n is even. Therefore from Eq. 3.3 the classical form of the product is

$$\begin{aligned} t &= a \times b \\ &= (2^{n/2}a_1 + a_o)(2^{n/2}b_1 + b_0) \\ &= 2^n(a_1 \cdot b_1) + 2^{n/2}(a_1 \cdot b_0 + a_0 \cdot b_1) + a_0 \cdot b_0 \end{aligned} \tag{3.4}$$

University of Pretoria etd – Joseph, G (2005)

Eq. 3.4 computes the multiplication of two n-bit integers by four $n/2$-bit multiplications and an extra addition. This is illustrated in Fig. 3.5.

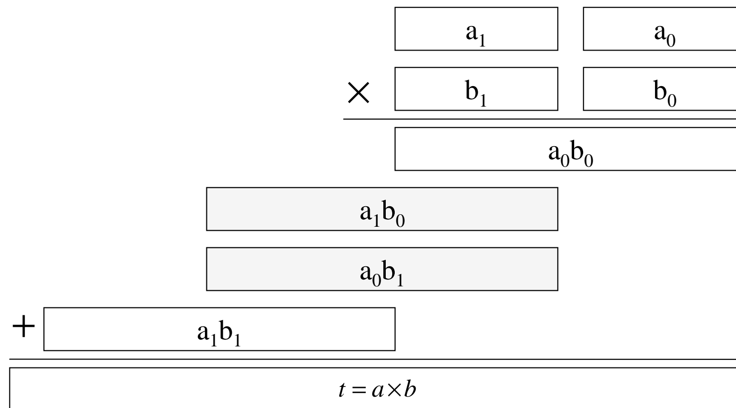


Figure 3.5: Classical multiplication computed as a product of $n/2$-bit multiplicands [72]

Karatsuba and Ofman modified Eq. 3.4, using simple algebra, to formulate the following equation

$$t = 2^n(a_1 \cdot b_1) + 2^{n/2}((a_1 + a_0)(b_1 + b_0) - a_1 \cdot b_1 - a_0 \cdot b_0) + a_0 \cdot b_0 \tag{3.5}$$

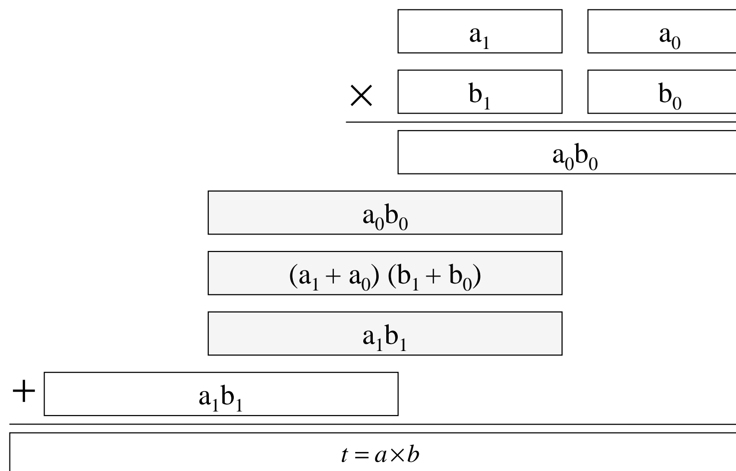


Figure 3.6: Karatsuba-Ofman multiplication computed as a product of $n/2$-bit multiplicands [72]

Although Eq. 3.5 appears more complicated than Eq. 3.4, Fig. 3.6 shows that t can be computed using 4 additions/subtractions but only three multiplications.

University of Pretoria etd – Joseph, G (2005)

3.3.2 The Computational Complexity of the Algorithm

The computational complexity of the Classical method can be determined by using recurrence relations. Eq. 3.4 shows that multiplying two n-digit integers is accomplished by performing four $n/2$-digit multiplications and one addition. Thus $C(n)$, the cost of multiplying two n-digit integers, can be formulated as

$$C(n) = 4 \times C(n/2) + \alpha \cdot n \tag{3.6}$$

where α denotes the number of bit operations required to compute the addition and shift operations in Eq. 3.4 (α is constant) [73, 50]. For $n = 2^m$ and using the initial condition that $C(1) = 1$, the computational complexity of the Classical method is calculated as follows

$$\begin{aligned} C(n) &= C(2^m) = 4(4C(2^{m-2}) + \alpha \cdot 2^{m-1}) + \alpha \cdot 2^m \\ &= 4^2 \cdot C(2^{m-2}) + \alpha \cdot 2^m(1+2) \\ &= 4^m \cdot C(1) + \alpha \cdot 2^m(1 + 2 + ... + 2^{m-1}) \\ &= \hat{\alpha}(2^m)^2 = \hat{\alpha} \cdot n^2 \end{aligned} \tag{3.7}$$

The above derivation is adapted from [73] and approximated by [50]. It is a simple proof to show that the computational complexity of the Classical method is $O(n^2)$ operations. Now in case of the Karatsuba-Ofman method, using Eq. 3.5, the recurrence relation is

$$C(n) = 3 \times C(n/2) + \beta \times n \tag{3.8}$$

where β denotes the number of bit operations required to compute the addition and shift operations in Eq. 3.5 (β is constant) [73, 50]. Again, for $n = 2^m$ with the initial condition $C(1) = 1$, the computational complexity is calculated as follows

$$\begin{aligned} C(n) &= C(2^m) = 3(3T(2^{m-2}) + \beta \cdot 2^{m-1}) + \beta \cdot 2^m \\ &= 3^2 \cdot C(2^{m-2}) + \beta \cdot 2^m(1 + \tfrac{3}{2}) \\ &= 3^m \cdot C(1) + \beta \cdot 2^m(1 + \tfrac{3}{2} + ... + \tfrac{3}{2}^{m-1}) \\ &= 3^m + \beta \cdot 2^m \left(\frac{\frac{3}{2}^m - 1}{\frac{3}{2} - 1} \right) \\ &= \hat{\beta} \cdot 3^m = \hat{\beta} \cdot 2^{m \cdot \log_2 3} = \hat{\beta} \cdot n^{\log_2 3} \end{aligned} \tag{3.9}$$

University of Pretoria etd – Joseph, G (2005)

The above derivation, adapted from [73], shows that the computational complexity of a Karatsuba-Ofman multiplication takes $O(n^{1.58})$ operations to multiply two n-digit numbers.

3.3.3 Recursive Properties of the Algorithm

Many researchers [73, 50, 19, 72, 74] avoid describing the recursive nature of the Karatsuba-Ofman algorithm in great detail. Knuth [18] suggests that the recursive algorithm can be implemented for larger n-bit integers. Koç [50] explains that with current implementations of the algorithm, it only starts to pay off once $n > 250$ bits.

A break-point is described by Scott [72], however it is not well reported. Geddes *et al.* [73] provide a recursive algorithm but do not comment on the break-point of the algorithm. Welschenbach [74] explains the first recursion level and concludes that the Karatsuba-Ofman algorithm has no real significance for his cryptographic applications. This subsection will provide a detailed analysis and implementation of Karatsuba-Ofman recursion.

Many complex problems are easier to solve if they are defined as simpler versions of themselves. A recursive function, conceptually depicted in Fig. 3.7, calls on itself in order to reduce the amount of code involved and to simplify the problem. It also requires a step to terminate the recursion process i.e. a break-point.

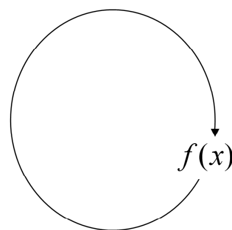


Figure 3.7: A recursive function

From Fig. 3.6 the Karatsuba-Ofman algorithm requires 3 $n/2$-bit products for two n-bit integers. Each $n/2$-bit product can be decomposed further into three more $n/4$-bit products. Each of these $n/4$-bit products can again be decomposed into $n/8$-bit products and so forth. This recursion evolves into a tree-type structure, each branch decomposing into three additional branches. Fig. 3.8 shows one branch of the tree-like recursion implemented by the Karatsuba-Ofman algorithm.

University of Pretoria etd – Joseph, G (2005)

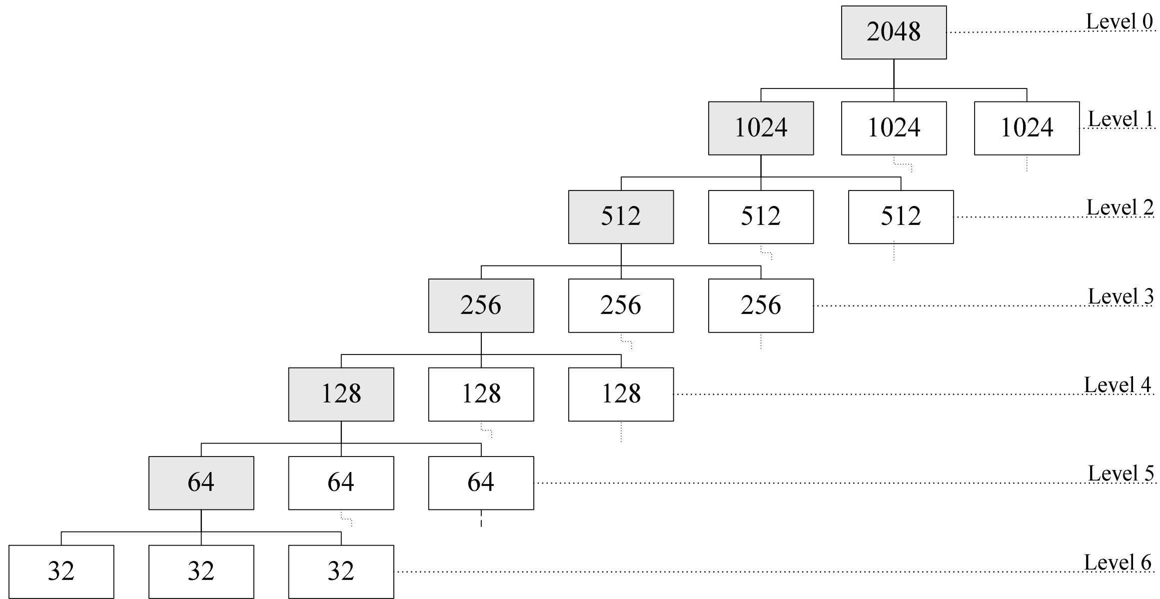


Figure 3.8: The Karatsuba-Ofman algorithm implemented for 6 levels of recursion

From Fig. 3.8, the recursion levels are decomposed to the processor word-size of 32-bits. The value in the box is the resulting bit-size of the multiplicands decomposed by the algorithm. Each of the boxes can be expanded into a similar tree. The recursion level is depicted on the right-hand side of the above figure.

The evolving tree requires a break-point to terminate its growth. This break-point [50, 72] is denoted by the recursion level. At this level the Karatsuba-Ofman algorithm then applies a conventional multiplication algorithm, using either the Classical or Comba methods, to complete the multiplication.

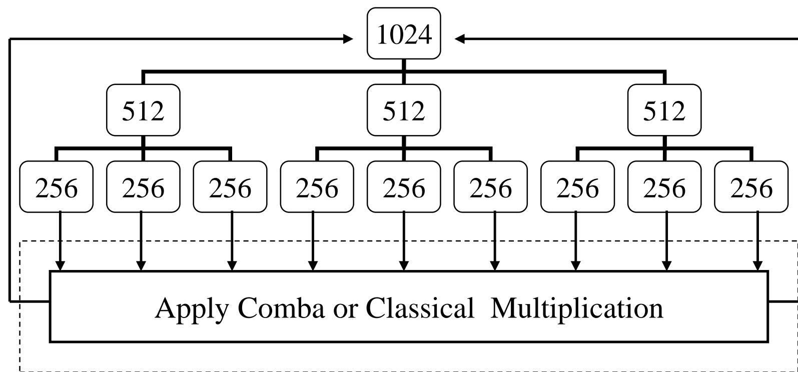


Figure 3.9: Two level recursive Karatsuba-Ofman algorithm

Fig. 3.9 is a visualization of 1024-bit Karatsuba-Ofman multiplication using 2 levels of

University of Pretoria etd – Joseph, G (2005)

recursion, each branch terminated with a Classical or Comba multiplication. The algorithm is initially applied recursively to the 1024-bit multiplicands decomposing them into half its original size. After two recursion levels (where each decomposed multiplicand's size is 256-bits), the decomposed multiplicands are multiplied using either the Classical or Comba methods.

The Karatsuba-Ofman algorithm, adapted from [73], is given as follows

ALGORITHM: KARATSUBA-OFMAN MULTIPLICATION

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$ and $b = (b_{n-1}b_{n-2}....b_1b_0)_B$, and break-point $\tilde{p}$.

Output. $a \times b = t = t_{2n-1}t_{2n-2}....t_1t_0$ of base B.

Function Karatsuba-Ofman(a, b, n)

1. *Break point:* If $n = \tilde{p}$

   $(t_{n-1}...t_1t_0) = (a_{n-1}...a_1a_0) \times (b_{n-1}...b_1b_0)$ [use a suitable multiplication]

2. *Break a into two:* Set $x_1 = a_{n-1}....a_{n/2}$ and $x_0 = a_{n/2-1}....a_0$
3. *Break b into two:* Set $y_1 = b_{n-1}....b_{n/2}$ and $y_0 = b_{n/2-1}....b_0$
4. *Calculate* $(n/2)$*-bit multiplications*

   4.1 $m_0 = x_0y_0$ = *Karatsuba-Ofman*$(x_0, y_0, n/2)$

   4.2 $m_1 = x_1y_1$ = *Karatsuba-Ofman*$(x_1, y_1, n/2)$

   4.3 $m_2 = (x_1 + x_0) \cdot (y_1 - y_0)$ = *Karatsuba-Ofman*$((x_1 + x_0), (y_1 + y_0), n/2)$

5. *Final result*: Return $(t = m_1B^n + (m_2 - m_1 - m_0)B^{n/2} + m_0)$.

The break-point $\tilde{p}$ is the bit-size of the multiplicand at the terminating recursion level. This algorithm, as developed for these simulations, applies the Karatsuba-Ofman recursion down to the break-point, and then uses either the Classical or Comba multiplication.

When applying the Karatsuba-Ofman algorithm recursively on multiplicands a and b of n-bit length, generally n must be even. More specifically n's length must be equal to an integer factor of the processor word size i.e. if w = 32-bits then $n \in (32, 64, 128, ..., 32p)$ where p is a positive integer. Weimerskirch *et al.* [75] provides more efficient methods of splitting up the multiplicands to be used in a Karatsuba-Ofman method.

University of Pretoria etd – Joseph, G (2005)

3.3.4 The Optimum Break-point

The advantage of the Karatsuba-Ofman method is that only 3 multiplications are needed instead of the 4 multiplications required by the Classical method, as shown in Fig. 3.5 and Fig. 3.6. However, this reduction in multiplications leads to more additions. The key question is to determine the break-point at which it is no longer worth applying Karatsuba recursively and it is faster to proceed with a multiplication [72].

In order to find this optimum break-point, a simulation of random 2048-bit multiplicands must be conducted for different recursion levels. Fig. 3.10 depicts the optimal break-point for the Karatsuba-Ofman method for the criteria shown in Section 3.4.

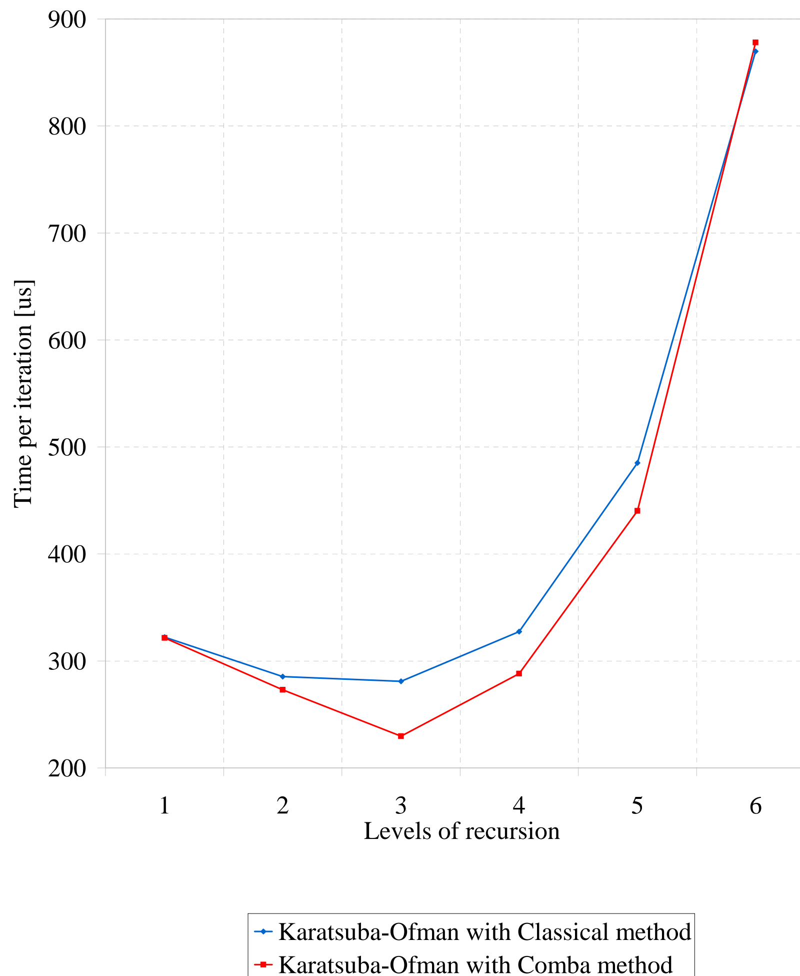


Figure 3.10: Optimum break point for a 2048-bit Karatsuba-Ofman multiplication algorithm

The optimum break-point, shown in Fig. 3.10, occurs at level 3 ($n = 256$-bits). Similar

University of Pretoria etd – Joseph, G (2005)

simulations were conducted for $128, 256, 512$ and 1024 bit multiplicands. These timing results utilizing its specific optimum break-points are depicted in Fig. 3.12.

The break-point depends on the processor used, specifically on the relative speed of the MUL and ADD instructions of the processor. The slower the former with respect to the latter, the deeper should the recursion proceed [72]. The results obtained in recursion levels 5 and 6, in Fig. 3.10, shows the time taken to compute the Karatsuba-Ofman multiplication is larger than applying Classical multiplication (applying the Classical method is equivalent to performing a 0 level Karatsuba-Ofman algorithm).

3.3.5 Application to Squaring

The Karatsuba-Ofman multiplication algorithm can also be used to implement a more efficient squaring algorithm. In squaring the multiplicands are equal, hence Eq. 3.5 can be modified as follows

$$t = a \times a = a_1^2 B^n - \left((a_1 + a_0)^2 - a_1^2 - a_0^2\right) B^{n/2} + a_0^2 \tag{3.10}$$

Inspection of Eq. 3.10 shows that only three $n/2$-size squarings are required. As with Karatsuba-Ofman multiplication, this algorithm can be applied recursively. The algorithm is as follows

ALGORITHM: KARATSUBA-OFMAN SQUARING

Input. $a = (a_{n-1}a_{n-2}....a_1a_0)_B$, break-point $\tilde{p}$

Output. $a \times a = t = t_{2n-1}t_{2n-2}....t_1t_0$ of base B

Function Karatsuba-Ofman(a, b, n)

1. *Break point:* If $n = \tilde{p}$

   $(t_{n-1}...t_1t_0) = (a_{n-1}...a_1a_0) \times (a_{n-1}...a_1a_0)$ [use a suitable squaring algorithm]

2. *Break a into two:* Set $x_1 = a_{n-1}....a_{n/2}$ and $x_0 = a_{n/2-1}....a_0$

3. *Calculate $(n/2)$-bit multiplications*

   3.1 $m_0 = x_0^2 =$ *Karatsuba-Ofman*$(x_0, x_0, n/2)$

   3.2 $m_1 = x_1^2 =$ *Karatsuba-Ofman*$(x_1, x_1, n/2)$

   3.3 $m_2 = (x_1 + x_0)^2 =$ *Karatsuba-Ofman*$((x_1 + x_0), (x_1 + x_0), n/2)$

4. *Final result*: Return $(t = m_1B^n + (m_2 - m_1 - m_0)B^{n/2} + m_0)$.

University of Pretoria etd – Joseph, G (2005)

The algorithm at break point, executes a Classical or Comba squaring to make the squaring operation more efficient.

Fig. 3.11 depicts the optimal break-point for a 2048-bit Karatsuba-Ofman squaring, under the conditions set in Section 3.4 for different recursion levels.

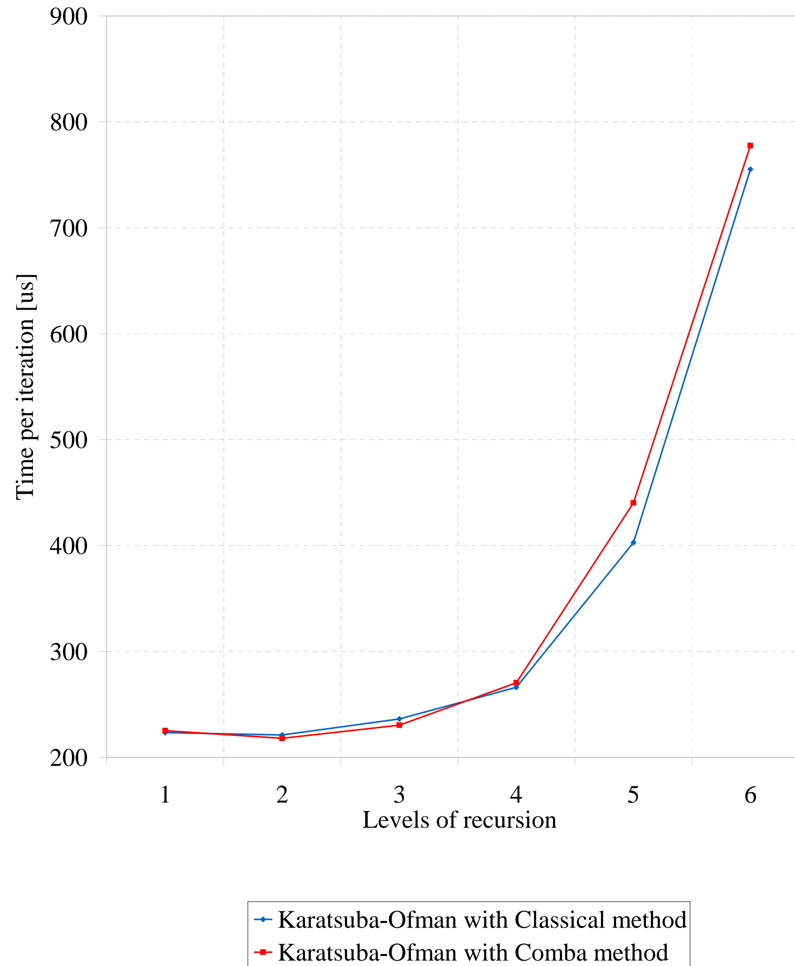


Figure 3.11: Optimum break point for a 2048-bit Karatsuba-Ofman squaring algorithm

The optimal break-point occurs at 2 levels of recursion. Similar simulations were conducted for $128, 256, 512$ and 1024 bit multiplicands. These timing results of its specific optimum break-points are depicted in Fig. 3.13. Note that a squaring is almost twice as fast as a multiplication, thus the cutoff point is higher. Hence, the Karatsuba-Ofman squaring algorithm requires fewer recursion levels to obtain its optimal speed.

University of Pretoria etd – Joseph, G (2005)

3.4 EXPERIMENTAL RESULTS

In this section, timing analyzes of the multiplication methods discussed in the chapter are given. In order to obtain exact numerical results for the methods, simulations were done on a Pentium III processor running at 550 MHz with 256 Mbyte main memory under Windows XP Home Edition platform using a Borland C Builder 6.0 compiler. The simulations were performed under the following conditions:

Algorithms tested:

- The multiplication algorithms that were tested were the Classical method, the Comba Method, The Karatsuba-Ofman method combined with either the Classical or the Comba multiplications.
- The optimal break-point of the Karatsuba-Ofman methods was determined in advance, using the simulations shown in Section 3.3.4.

Programming conditions:

- Each multiplication algorithm was implemented using standard ANSI C coding.
- The multiplicands were randomly generated, using MIRACL's pseudo random number generator, for bit sizes $128, 256, 512, 1024$ and 2048 bits.
- Though a lot of effort has been done to remove the overhead generated by the compiler, the test is still subjected to a little overhead generated by the platform and compiler.

Timing analysis parameters:

- One iteration consisted of a 1000 runs of each multiplication algorithm.
- The total time period of each test was 20 seconds.
- The iterations were incremented until the total time period had elapsed.
- The total number of runs was the product of the number of runs (1000) and the number of iterations.
- The average time was calculated as a function of the total time divided by the total number of runs.

University of Pretoria etd – Joseph, G (2005)

Fig. 3.12 and Fig. 3.13 provide the time results of the various multiplication and squaring methods. They depict the average time it takes each method to compute a multiplication or a squaring over different bit sizes of the multiplicands.

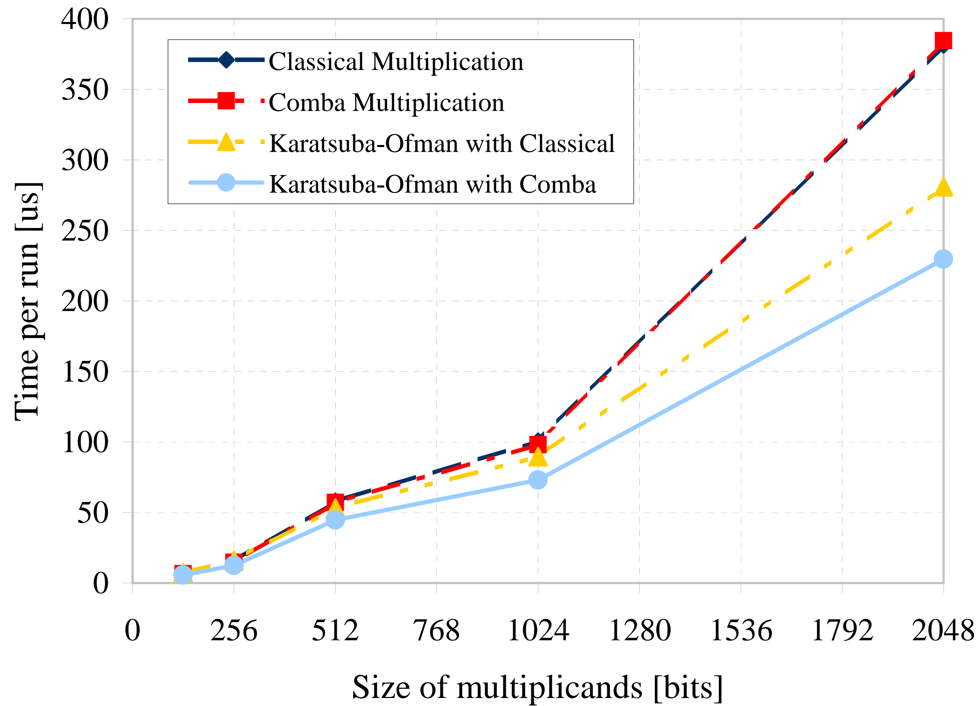


Figure 3.12: Comparison of multiplication methods

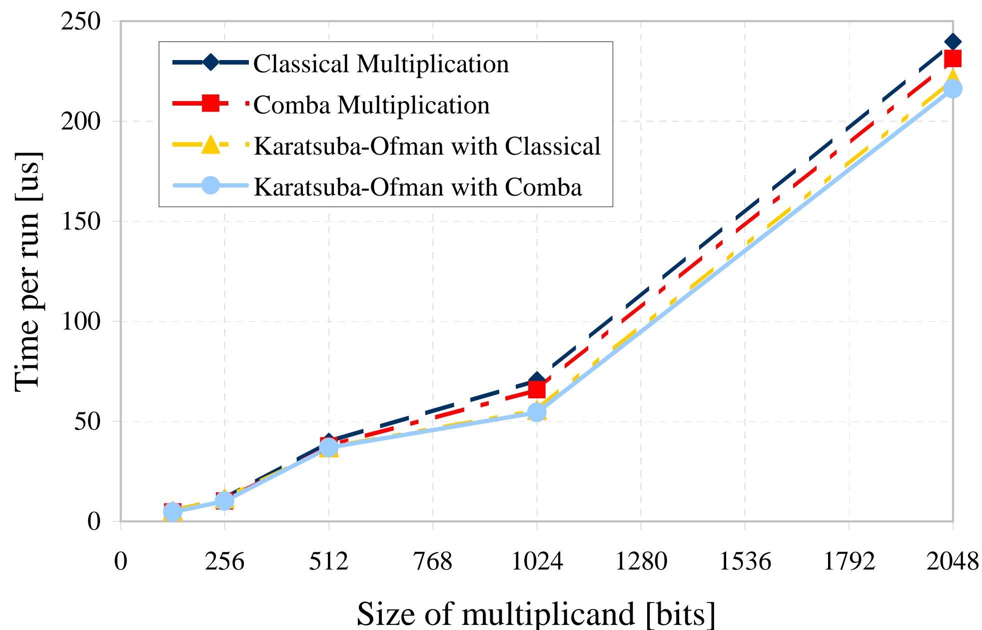


Figure 3.13: Comparison of squaring methods

University of Pretoria etd – Joseph, G (2005)

From Fig. 3.12 and Fig. 3.13 the Karatsuba-Ofman method gives the best results for larger numbers, 512-bits and above. However for multiplicands less than 512-bits, the Comba method provides the best results.

Mathematically the Comba method is identical to the Classical method as they both require $O(n^2)$ operations. It improves on the Classical method by employing clever programming optimizations. It unravels the loop to repeat the code in-line a number of times and computes the partial products directly to reduce the number of memory writes required. Hence difference in speed between the Comba and Classical methods is dependent on the processor, specifically on the time the processor takes to execute a memory write and a loop function.

For larger integers, the Comba method generates a large amount of in-line code, as it requires additional overflow registers to keep each column's precision, shown in Fig. 3.3, from overflowing and also requires additional control overhead to manage these registers. This large amount of in-line coding can become impracticable for processors that have insufficient internal memory.

In terms of $n \times n$ multiplication, the Karatsuba-Ofman algorithm, which requires $O(n^{1.58})$ bit operations, is asymptotically faster than the Classical algorithm which requires $O(n^2)$ bit operations [73]. However, in practice the algorithm requires a number of intermediate results that must be stored which adds unavoidable control overhead that detracts from the algorithms efficiency for relatively small integers.

Fig. 3.13 shows that the squaring methods are essentially comparable in speed. This is due to the fact that a squaring requires only $(n^2 + n)/2$ operations compared to the multiplication's n^2 operations, hence the improvements of each method is applied to fewer operations.

Menez *et al.* [25] states that squaring a positive integer a (i.e., computing a^2) can at best be no more than twice as fast as multiplying distinct integers a and b. To prove this, they consider the identity $ab = ((a + b)^2 - (a - b)^2/4)$ which shows that $a \times b$ can be computed with two squarings. Practically the difference between a squaring and a multiplication is

University of Pretoria etd – Joseph, G (2005)

due to the difference in the number of operations and the control overhead required by the operations. Hence from Fig. 3.12 and Fig. 3.13, a squaring algorithm computes a squaring in 65% of the time that is required by a multiplication to compute the same squaring.

3.5 CHAPTER SUMMARY

"How fast can we multiply"

DONALD E. KNUTH, [18]

The above quote describes the objective for this chapter. Three methods were discussed, implemented using simulations, and modified to perform more efficient squarings. The goal was to determine which is the fastest integer multiplication that one could use for a modular exponentiation.

Each of the methods, except the Classical method, is optimal under certain circumstances. Though the Classical method is asymptotically slower than the Karatsuba-Ofman method and the Comba method, it is simpler to implement and for small numbers, gives better performance than the Karatsuba-Ofman methods.

The Comba method is most suitable for integers less than 256-bits. However for multiplication of large numbers, especially 512-bits and higher, Karatsuba-Ofman with Comba method should be used.

There exists other methods (i.e. FFT and convolutional methods) that perform the multiplication step. Though these methods are mathematically elegant, their improvement in speed only starts paying off for multiplicands larger than 8196 bits [18, 25, 76]. Thus, these methods are not applicable for the size of integers of practical importance to public key cryptosystems.

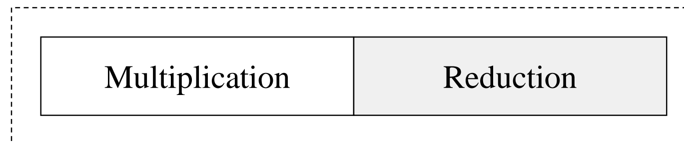
University of Pretoria etd – Joseph, G (2005)

CHAPTER FOUR

FAST REDUCTION TECHNIQUES

"Your life is the sum of a remainder of an unbalanced equation inherent to the programming of the matrix. You are the eventuality of an anomaly, which despite my sincerest efforts I have been unable to eliminate from what is otherwise a harmony of mathematical precision. While it remains a burden to sedulously avoid it, it is not unexpected, and thus not beyond a measure of control."

THE ARCHITECT, *Matrix Reloaded*



The modular reduction operation, $a \bmod m$, is conventionally accomplished by dividing a by m to obtain the remainder. The steps of the division algorithm can be modified in order to speed up the process. Reducing the time and memory complexities of this operation is a challenging problem on which relies the practical feasibility of the cryptosystem's signature and encryption methods [77].

Three modular reduction methods are discussed in this chapter:

- The Classical method [18]. The simplest implementations of large integer modular reduction are computed utilizing this algorithm.

University of Pretoria etd – Joseph, G (2005)

- The Barrett method [20]. The Barrett method was the first approach to perform modular reduction that utilized precomputation to remove the time-consuming division step.
- The Montgomery method [21]. Montgomery's method is a ingenious technique that performs efficient modular reduction utilizing simple shift operations which can implemented on general processors.

The purpose of this chapter is to investigate the above mentioned methods for speeding up modular reduction in various ways. This chapter will first describe these three algorithms with their respective modifications developed to improve their speed. It will then present implementation results of these algorithms to see their relative speed performance in a modular exponentiation.

4.1 CLASSICAL REDUCTION

"And marriage and death and division, make barren our lives"

ALGERNON CHARLES SWINBURNE, *Dolores*

The easiest method for performing modular reduction is to compute the remainder r by division using the modulus m as the divisor. A standard division computes the quotient and the remainder. However the quotient is of little concern, as one only needs the remainder. Therefore, the steps of the standard division algorithm can be simplified to enhance the speed of the reduction.

4.1.1 Description

Classical reduction is a formalization of the sequential division algorithm. Division is the most complex of the four basic arithmetic operations. First of all, it has two results: the quotient and the remainder. Given the dividend a and a divisor m, the quotient q and the remainder r are calculated using

$$a = q \cdot m + r \tag{4.1}$$

If a and m are positive, then the q and r will be positive. Classical reduction successively shifts and subtracts m from a until r, with the property $0 \leq r < m$, is found. However, if

University of Pretoria etd – Joseph, G (2005)

the subtraction of a yields a negative r, an addition of m is required to restore r as a positive integer [50].

The reduction method does not explicitly compute the quotient, but uses its estimate $\tilde{q}$ to calculate each digit of the remainder.

4.1.2 The Algorithm

The Classical reduction algorithm computes a remainder r by dividing a n-bit a by a t-bit m, where $n \leq t \leq 1$. The algorithm is as follows

ALGORITHM: CLASSICAL REDUCTION

Input. $a = (a_n...a_1a_0)_B$ and $m = (m_t...m_1m_0)_B$

Output. $r = (r_t...r_1r_0)_B$ where $a = \tilde{q}m + r$ $(0 \leq r < m)$

1. Copy a to r: $r \leftarrow a$
2. While $(r \geq mB^{n-t})$ do the following:
   $r \leftarrow r - mB^{n-t}$
3. For i from n down-to $(t+1)$ do the following:
   3.1 If $r_i = m_t$ then set $\tilde{q} \leftarrow B - 1$ else $\tilde{q} \leftarrow (r_iB + r_{i-1})/m_t$
   3.2 While $(\tilde{q}(m_tB + m_t - 1) > r_iB^2 + r_{i-1}B + r_{i-2})$ do: $\tilde{q} \leftarrow \tilde{q} - 1$
   3.3 $r \leftarrow r - \tilde{q} \cdot mB^{i-t-1}$
   3.4 If $x < 0$ then set $r \leftarrow r + m \cdot B^{i-t-1}$ and $\tilde{q} \leftarrow \tilde{q} - 1$
4. Return $r = (r_t...r_1r_0)_B$

The above algorithm, adapted from [77, 25], contains an integer division in its main loop. An integer division requires many more machine cycles to compute than an integer multiplication on a standard processor. Thus, the algorithm is computationally intensive.

The basis of the algorithm consists of estimating the quotient as accurately as possible and in doing so, reduce the number of steps required to calculate the reduction. Dividing the two most significant digits of a by m_t will result in the estimate $\tilde{q}$ never being too small, and if $m_t \geq \left[\frac{B}{2}\right]$, $\tilde{q}$ is at most two in error. Using an additional digit for a and m (i.e., using the three most significant digits of a and the two most significant digits of m as shown in step 3.2), $\tilde{q}$ can be at most one in error [77]. Furthermore, this error occurs with approximate

University of Pretoria etd – Joseph, G (2005)

probability $\frac{2}{b}$ [18].

The initial formalization of the algorithm is due to Knuth [18]. Koblitz [32] provides a comprehensive description for the use of this method in a modular multiplication. Two variations of the Classical reduction method with slightly different ways of quotient estimation are shown in [78, 79].

4.1.3 Computational Improvements

One can always guarantee that $m_t \geq \left[\frac{B}{2}\right]$ by replacing the integers (a, m) by $(\beta a, \beta m)$ for a suitable choice of β. The remainder is β times the remainder of a divided by b. Since the base B is a power of 2, then the choice of β should be a power of 2; multiplication by β is achieved by simply left-shifting the binary representations of a and m. Multiplying by a suitable choice of β, to ensure that $m_t \geq \left[\frac{B}{2}\right]$, is called *normalization*. The resulting normalized remainder requires a simple division by β to obtain the actual remainder [25].

Step 3.2 can be modified to $qm_{t-2} > (r_iB + r_{i-1} - qm_{t-1})B + r_{i-2}$. Since $r_iB + r_{i-1}$ $-qm_{t-1} < m_t$, this step can be reduced to two multiplications. Thus the algorithm requires $k(k+2)$ multiplications and k divisions for $2k$-bit dividend [80].

A more involved kind of normalization is described by Walter [81]. This normalization fixes the modulus' most significant digit in such a way that the most significant digit of a is used as a first estimate for q, resulting in a faster reduction. However, this increases the length of the modulus by at least one digit and all the intermediate results of a modular exponentiation. Hence what is saved during the modular reductions, is lost again by additional multiplications [77].

4.2 BARRETT REDUCTION

Barrett reduction [20] was inspired by fast division algorithms that multiply the reciprocal of the divisor to emulate division. This reduction technique is advantageous in a modular exponentiation where many reductions are performed with the same modulus. It was the first approach to perform reduction without explicitly using the division step in the loop.

University of Pretoria etd – Joseph, G (2005)

4.2.1 Description

Barrett introduced the idea of estimating the quotient $\lfloor a/m \rfloor$ with operations that either are less expensive in time than a division by m, or can be done as a precalculation for a given m (viz., $\mu = B^{2k}/m$ where μ is a scaled estimate of the $2k$-digit modulus' reciprocal) [77].

The estimate $\tilde{q}$ of $\frac{a}{m}$ is obtained by replacing the floating point divisions in $q = \left\lfloor (a/B^{2k-t})(B^{2k}/m)/B^t \right\rfloor$ by integer divisions:

$$\tilde{q} = \frac{\frac{a}{B^{2k-t}}\mu}{B^t} \tag{4.2}$$

The number of multiplications and the resulting error is more or less independent of t. The best choice for t, resulting in the least number of operations and the smallest maximal error, is $k+1$.

The estimate $\tilde{q}$ is at most two smaller than the correct q. This can be shown using the following inequality:

$$\begin{aligned}
\frac{a}{m} \geq \tilde{q} &> \frac{1}{B^{k+1}}\left(\frac{a}{B^{k-1}} - 1\right)\left(\frac{B^{2k}}{m} - 1\right) - 1 \\
&= \frac{a}{m} - \frac{a}{B^{2k}} - \frac{B^{k-1}}{m} + \frac{1}{B^{k+1}} - 1 \\
&\geq q - \left(\frac{a}{B^{2k}} + \frac{B^{k-1}}{m} - \frac{1}{B^{k+1}} + 1\right) \\
q \geq \tilde{q} &> q - 3
\end{aligned} \tag{4.3}$$

where the inequality $\frac{y}{x} - 1 < \left\lfloor \frac{y}{x} \right\rfloor < \frac{y}{x}$ was used. Therefore the remainder can be computed by subtracting $\tilde{q}m \bmod B^{k+1}$ from a and then adjusting the result with at most two subtractions of m [80].

Naccache *et al.* [82] provides mathematical correctness of the Barrett method and its possible optimizations.

4.2.2 The Algorithm

Given the inputs a, m and the precomputation $\mu = [B^{2k}/m]$, the method computes $r = a \bmod m$ using the following steps:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: BARRETT REDUCTION

Precomputation. $\mu = [b^{2k}/m]$

Input. $a = (a_{2k-1}...a_1a_0)_B$ and $m = (m_{k-1}...m_1m_0)_B$

Output. $r = (r_{k-1}...r_1r_0)_B$

1. *Compute the estimate quotient* $\tilde{q}$
   1.1 $q_1 \leftarrow [a/B^{k-1}]$
   1.2 $q_2 \leftarrow q_1 \times \mu$
   1.3 $\tilde{q} \leftarrow [q_2/B^{k+1}]$
2. *Compute the remainder* r
   2.1 $r_1 \leftarrow a \bmod B^{k+1} \rightarrow r_1$
   2.2 $r_2 \leftarrow (\tilde{q} \times m) \bmod B^{k+1}$
   2.3 $r \leftarrow r_1 - r_2$
3. *Fix the remainder*
   3.1 If $r < 0$ then $r + B^{k+1} \rightarrow r$
   3.2 While $r \geq m$ do the following: $r \leftarrow r - m$
4. Return $r = (r_{k-1}...r_1r_0)_B$

The algorithm, adapted from [25], requires 2 divisions by a power of the base B and a partial multiplication. All divisions performed, in base B representation, are simple right-shifts.

Step 1 finds the estimate quotient $\tilde{q}$ and step 2 computes the remainder. If the computed error does not fall in the limits $0 \leq r < m$, a simple addition/subtraction by m will be implemented to "fix" the remainder. Bosselaers *et al.* [77] state that for about 90% of the values of $a < m^2$ and modulus m, $\tilde{q}$ will be correct and 10% of the cases will it be two in error.

The precomputation of μ is based on a technique of emulating floating point data types with fixed precision integers. Computing $1/m$ would generally result in a fraction. Menez *et al.* [25] state that one can obtain the integer equivalent of $1/m$ using fixed point arithmetic. It follows on the concept that if one sets B^{2k} equivalent to one then B^{2k}/m is equivalent to $1/m$ using basic arithmetic. Hence the integer equivalent of $1/m$ is truncated to $k+1$ digits (B^{2k} is chosen instead of B^k as B^k/m will generate a 1-digit number).

University of Pretoria etd – Joseph, G (2005)

4.2.3 Computational Improvement

Looking at step 1 of the algorithm, the subsections of step 1 can be depicted as follows:

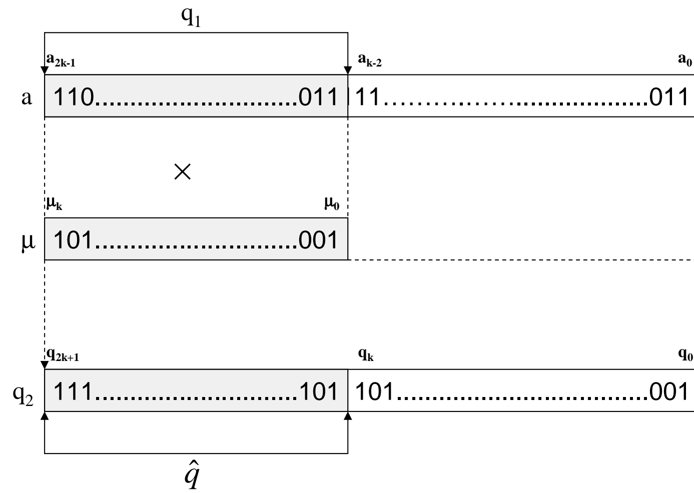


Figure 4.1: Computing $\tilde{q}$ with Barrett reduction

Since the $k+1$ least significant digits of q_2 are not needed to determine $\tilde{q}$, only a partial multiple-precision multiplication of $q_1 \times \mu$ is necessary. The only influence of the $k+1$ least significant digits have on the higher order digits is the carry from position $k+1$ to position $k+2$. Provided the base B is sufficiently large with respect to k, this carry can be accurately computed by only calculating the digits at positions k and $k+1$. Hence, the $k-1$ least significant digits of q_2 need not be computed. Since μ and q_1 have at most $k+1$ digits, determining $\tilde{q}$ requires at most $\frac{1}{2}\left(k^2+5k+2\right)$ single-precision multiplications [25].

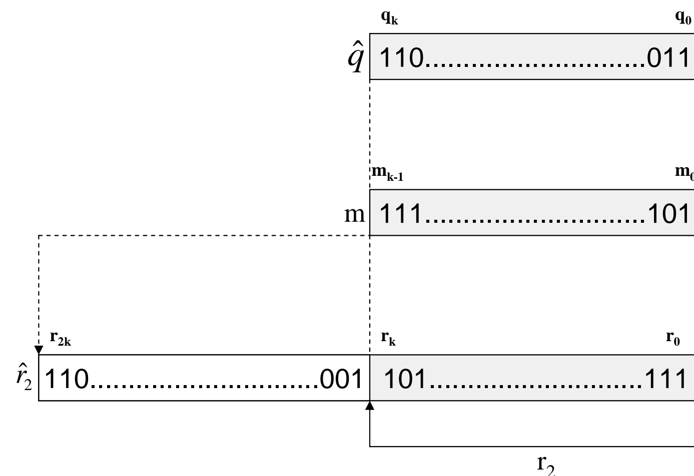


Figure 4.2: Computing the initial remainder with Barrett reduction

Step 2.2 can also be implemented using a similar type of modification implemented by step

University of Pretoria etd – Joseph, G (2005)

1, however it only needs the bottom $k+1$ bits. This can be depicted in Fig. 4.2. In Fig. 4.2 r_2 can also be computed by a partial multiple-precision multiplication which evaluates only the least significant $k+1$ digits of $q_3 \times m$. Since $\tilde{q}$ and m are k-digit integers, this computation can be done in at most $\frac{1}{2}(k^2+3k-2)$ single-precision multiplications. Therefore, the total number of single-precision multiplications required by the algorithm is at most $k(k+4)$ [25].

4.3 MONTGOMERY REDUCTION

In 1985, P. L. Montgomery introduced an efficient algorithm [21] for modular multiplication without explicitly carrying out the classical modular reduction step. This is done by transforming the original integer into an ingenious representation in the residue class of modulo m to speed up the reduction operation [80].

4.3.1 Description

Montgomery reduction is a generalization of a much older technique due to Hensel [83]. Hensels observation is the following: If m is an odd positive integer less than 2^k (k a positive integer) and T is some integer such that $2^k < T \leq 2^{2k}$,then $R_0 = (T + q_0 \cdot m)/2$, where $q_0 = T \bmod 2$ is an integer and $R_0 \equiv T2^{-1} \bmod m$. More generally, $R_i = (R_{i-1} + q_i m)/2$, where $q_i = R_{i-1} \bmod 2$ is an integer and $R_i \equiv N2^{-i+1} \bmod m$. Since $T < 2^{2k}$, it follows that $R_{k-1} < 2m$ [25].

Mathematically, Montgomery reduction can be described as follows: Assuming that the modulus m is a k-digit integer, i.e. $B^{k-1} \leq m \leq B^{2k}$, let $R = B^k$. The Montgomery reduction requires R and m to be relatively prime, that is

$$\gcd(R, m) = \gcd(B^k, m) = 1 \tag{4.4}$$

In order for Eq. 4.4 to be satisfied, given that base B in general processors is always a power of 2, m has to be odd. Montgomery [21] uses an ingenious transformation which converts the original integer into its m-residue form, before it can be utilized.

The m-residue with respect to R of an integer $a < n$ is denoted as $\tilde{a} = aR \bmod m$. Hence the set $\{a \cdot R \bmod m | 0 \leq a \leq n-1\}$ is a complete residue system. Thus, there is a

University of Pretoria etd – Joseph, G (2005)

one-to-one correspondence between the integers in $[0, n-1]$ and the integers in the set.

Montgomery exploits this property by introducing a routine which computes the m-residue product of two m-residue integers [50]. Given two m-residues, $\tilde{a}$ and $\tilde{b}$, the Montgomery product is defined as the m-residue

$$\tilde{R} = \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m \tag{4.5}$$

where R^{-1} is the inverse of $R \bmod m$, i.e. $R^{-1} \cdot R = 1 \bmod m$ since

$$\begin{aligned} \tilde{R} &= \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod n \\ &= a \cdot R \cdot b \cdot R \cdot R^{-1} \bmod n \\ &= a \cdot b \cdot R \bmod n \end{aligned} \tag{4.6}$$

In order to describe the Montgomery reduction algorithm, an additional quantity $\acute{m}$, with the property $R \cdot R^{-1} - m \cdot \acute{m} = 1$ is defined. The integers R^{-1} and $\acute{m}$ can be computed using the extended Euclidean algorithm (see [18] for more details).

The rationale behind the m-residue transformation is the ability to perform a Montgomery reduction $(a \times b) \cdot R^{-1} \bmod m$ for $0 \leq a \times b < Rm$ in almost the same time as a multi-precision multiplication [77]. This is based on the following theorem:

The Montgomery Reduction Theorem. Let $\tilde{m} = -m^{-1} \bmod R$. If $\gcd(m, R) = 1$, then for all integers T, $(T + Um)/R$ is an integer satisfying

$$\frac{T + Um}{R} \equiv TR^{-1} (\mathrm{mod} m) \tag{4.7}$$

where $U = T\tilde{m} \bmod R$ [77]. The justification, shown in [25], implies that the estimate $\tilde{T} = (T + Um)/R$ for $TR^{-1} \bmod m$ is never too small and the error is at most one. This means that a Montgomery reduction is not more expensive than two multi-precision multiplications.

4.3.2 The Algorithm

Montgomery reduction requires the mathematical steps (i.e. the modular multiplication and modular exponentiation steps shown in Fig. 1.1) to be modified to implement the reduction

University of Pretoria etd – Joseph, G (2005)

operation. This section will thus look at the following algorithms:

- *Montgomery reduction*. The core reduction that performs $a \cdot R^{-1} \bmod m$.
- *Montgomery product*. The crux procedure that computes $\tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m$.
- *Montgomery exponentiation*: A Montgomery modified modular exponentiation.

Montgomery reduction: The algorithm computes the Montgomery reduction of integer T, where $R = B^k$ and $T < mR$, and requires that $\gcd(m, R) = 1$. The algorithm makes implicit use of the Montgomery theorem by computing quantities which have similar properties to $U = T\tilde{m} \bmod R$ and $T + Um$. The algorithm is as follows:

ALGORITHM: MONTGOMERY REDUCTION

Precomputation. $\acute{m} = -m^{-1} \bmod R$
Input. $T = (t_{2k-1}...t_1t_0)_B$ and $m = (m_{k-1}...m_1m_0)_B$
Output. $\tilde{T} = (\tilde{t}_{k-1}...\tilde{t}_1\tilde{t}_0)_B$
1. *Copy* T *to* $\tilde{T}$. $\tilde{T} \leftarrow T$
2. *Perform the reduction*
2.1 $U \leftarrow \tilde{T} \cdot \acute{m} \cdot m \bmod R$
2.2 $\tilde{T} \leftarrow (T + U \cdot m)/R$
3. *Fix the remainder*. If $\tilde{T} \geq m$ then $\tilde{T} \leftarrow \tilde{T} - m$
4. Return $\tilde{T} = (\tilde{t}_{k-1}...\tilde{t}_1\tilde{t}_0)_B$

The most important feature of the Montgomery reduction algorithm is that the operations involved are multiplications modulo R and divisions by R, both of which are intrinsically fast operations on general processors since R is usually a power 2 [50].

It can be easily verified that $\frac{T+Um}{R}$ is an integer (substitute U into Eq. 4.7). At step 3 a subtraction of m is required which implies that $\tilde{T} < 2m$. From step 2.2 $\tilde{T} = T + Um$, but $Um < Rm$ and $T < Rm$; hence $\tilde{T} < 2m$ [25].

Montgomery product: The Montgomery product algorithm can be used to compute the product of $\tilde{a}$ and $\tilde{b}$ modulo n, where $\tilde{a}$ and $\tilde{b}$ are the m-residue transforms of a and b respectively. The algorithm is given below:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: MONTGOMERY PRODUCT

Precomputation. $\acute{m} = -m^{-1} \bmod R$
Input. $\tilde{a} = (\tilde{a}_{k-1}...\tilde{a}_1\tilde{a}_0)_B$, $\tilde{b} = (\tilde{b}_{k-1}...\tilde{b}_1\tilde{b}_0)_B$ and $m = (m_{k-1}...m_1m_0)_B$
Output. $\tilde{r} = (\tilde{r}_{k-1}...\tilde{r}_1\tilde{r}_0)_B$
1. *Multiply* $\tilde{a}$ *and* $\tilde{b}$: $\tilde{t} = \tilde{a} \times \tilde{b}$
2. *Perform the reduction*.
2.1 $u \leftarrow \tilde{t} \cdot \acute{m} \bmod R$
2.2 $\tilde{r} \leftarrow (\tilde{r} + u \cdot m)/R$
3. *Fix the remainder*. If $\tilde{r} \geq m$ then $\tilde{r} \leftarrow \tilde{r} - m$
4. Return $\tilde{r} = (\tilde{r}_{k-1}...\tilde{r}_1\tilde{r}_0)_B$

Since $\tilde{r}$ is the product of two m-residues, the result is the m-residue of the remainder, and the remainder itself is obtained by applying one additional Montgomery reduction [77]. Using the Montgomery reduction algorithm shown above as an additional step, the $\tilde{r}$ can be transformed into r. This is easily shown since $\tilde{r} = r \cdot R \bmod m$ which immediately implies that $\tilde{r} \cdot R^{-1} \bmod m = r \cdot R \cdot R^{-1} \bmod m = r \bmod m$ [50].

The initial transformation to the m-residue domain, the precomputation of $\acute{m}$, and the inverse transformation from the m-residue domain (using an additional reduction step) are fundamentally required, even for a one-digit reduction. Thus, the use of the Montgomery product algorithm will be slower than the Classical and Barrett reduction methods when a single modular multiplication has to be performed.

Montgomery exponentiation: The Montgomery product and reduction algorithms are more suitable when several modular multiplications with respect to the same modulus are needed, i.e. a modular exponentiation. In the following algorithm a summary of the modular exponentiation operation which makes use of the Montgomery product and reduction functions is given.

For algorithmic reference, the notation that is given to the Montgomery product function is MontProd$(\tilde{a}, \tilde{b})$ where $\tilde{a}$ and $\tilde{b}$ are m-residues. The Montgomery reduction function is MontRed$(\tilde{a})$ where $\tilde{a}$ is in m-residue format. The modular exponentiation technique used is the binary method (see Section 5.2).

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: MONTGOMERY EXPONENTIATION

Precomputation. $\acute{m} = -m^{-1} \bmod R$

Input. $g = (g_{k-1}...g_1g_0)_B$, $e = (e_{l-1}...e_1e_0)_2$ and $m = (m_{k-1}...m_1m_0)_B$

Output. $A = (a_{k-1}...a_1a_0)_B$

1. *Transform g into m-residues.* $\tilde{g} \leftarrow g \cdot R \bmod m$
2. *Represent the initial value of A as a m-residue.* $\tilde{A} \leftarrow 1 \cdot R \bmod m$
3. *Exponentiation* For i from $l-1$ down-to 0 do the following:
   3.1 *Modular squaring* $\tilde{A} \leftarrow \text{MontProd}(\tilde{x}, \tilde{x})$
   3.2 *Modular multiplication* if $e_i = 1$ then $\tilde{A} \leftarrow \text{MontProd}(\tilde{g}, \tilde{x})$
4. *Transform to normal form.* $A \leftarrow \text{MontRed}(\tilde{A})$
5. Return $A = (a_{k-1}...a_1a_0)_B$

The Montgomery algorithms can be modified similarly for the modular exponentiation techniques shown in Chapter 5, and is not specific to the binary method. The transformation of g and A into their m-residues can be computed using a classical reduction, as it will make a very small time difference (which can be neglected) when computing the modular exponentiation. However, once the transformations have been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations that perform only multiplications modulo R and divisions by R [50].

When the exponentiation method finishes, the m-residue of A remains. The ordinary residue number is obtained from the m-residue by executing the MontRed function. Notice that MontRed$(\tilde{A})$ is equivalent to MontProd$(\tilde{A}, 1)$. This is easily shown to be correct since $\tilde{A} = A \cdot R \bmod m$ that immediately implies that $A = \tilde{A} \cdot R^{-1} \bmod m = \tilde{A} \cdot 1 \cdot R^{-1} \bmod m =$ MontProd$(\tilde{A}, 1)$ [50].

The above described Montgomery algorithms can be refined and made more efficient, particularly when involved in multi-precision integer arithmetic. These improvements, due to Dussé and Kaliski [61], are described in the following section.

4.3.3 Computational Improvements

The Montgomery reduction, described the previous section, is not more expensive than two multi-precision multiplications. The following improvements due to Dussé and Kaliski [61]

University of Pretoria etd – Joseph, G (2005)

will almost be twice as fast. Hereto it is sufficient to observe that the basic idea of Montgomery's Theorem is to make $\tilde{r}$ a multiple of R by adding multiples of m. Instead of computing all of u at once, one can compute one digit u_i at a time, add $u_i m B^i$ to $\tilde{r}$ and repeat. This change allows one to compute $\acute{m}_0 = -m^{-1} \bmod B$ instead of $\acute{m}$ [77]. The resulting algorithm is as follows:

ALGORITHM: MODIFIED MONTGOMERY PRODUCT
Precomputation. $\acute{m}_0 = -m^{-1} \bmod B$
Input. $\tilde{a} = (a_{k-1}...a_1a_0)_B$, $\tilde{b} = (b_{k-1}...b_1b_0)_B$ and $m = (m_{k-1}...m_1m_0)_B$
Output. $\tilde{r} = (r_{k-1}...r_1r_0)_B$
1. *Multiply the multiplicands*. $\tilde{r} = \tilde{a} \times \tilde{b}$
2. *Perform the reduction*. For i from 0 to $(k-1)$ do the following:
2.1 $u_i \leftarrow r_i \cdot \acute{m}_0 \bmod B$
2.2 $\tilde{r} \leftarrow \tilde{r} + u_i \cdot m \cdot B^i$
3. *Calculate intermediate remainder*. $\tilde{r} \leftarrow \tilde{r}/B^k$
4. *Fix the remainder*. If $\tilde{r} \geq m$ then $\tilde{r} \leftarrow \tilde{r} - m$
5. Return $\tilde{r} = (r_{k-1}...r_1r_0)_B$

Thus, a greatly simplified Montgomery product routine is developed by avoiding the full computation of $\acute{m}$ and by using only single-precision multiplication to multiply u_i and $\acute{m}_0$ [50]. As seen from the above algorithm, the number of single-precision multiplications is reduced from $2k^2$ to $k(k+1)$.

In Section 4.3.1 it was noted that R and m had to be relatively prime, where $R = 2^k$ for general processors. Hence in order for one to implement the Montgomery reduction step, m has to be odd. Koc [50] describes a method to implement the Montgomery reduction for an even modulus by utilizing the Chinese remainder Theorem and operand scaling.

Bosselaers *et al.* [77] and Shand [84] discuss the generalization of Hensel's observation that formed the basis of the Montgomery reduction. Numerous methods of hardware implementations of the Montgomery reduction have been proposed [85, 86, 87, 54, 88]. A complete survey of how the Montgomery reduction can be applied to various cryptosystems is described by Naccache *et al.* [89].

University of Pretoria etd – Joseph, G (2005)

Koc *et al.* [90] provide an excellent reference for implementing multi-precision Montgomery multiplication algorithms. Koc's approach utilizes Montgomery reduction to provide different implementations of the algorithm. The conventional algorithm, denoted by Koc as the Separated Operand Scanning (SOS) method, multiplies the two multiplicands before reducing the product (which is similar to the modified Montgomery product algorithm). The cost of using the SOS method is the additional memory required to hold the $2k$-bit intermediate product. In addition, Koc also describes an interleaved multiply-and-reduce modular multiplication denoted as the Coarsely Integrated Operand Scanning (CIOS) method, which interleaves the multiplication in the modular reduction step. This technique eliminates the need of additional memory space, however it does not allow the possibility to utilize Karatsuba-Ofman multiplication and squaring optimizations.

4.4 EXPERIMENTAL RESULTS

In order to obtain practical times for the discussed reduction methods to be used in a public-key environment, i.e. a modular exponentiation $g^e \bmod m$, specific simulations must be performed. In order to obtain exact numerical results for the methods, simulations were done on a Pentium III processor running at 550 MHz with 256 Mbyte main memory under Windows XP Home Edition platform using a Borland C Builder 6.0 compiler. The simulations were performed under the following conditions:

Algorithms tested:

- The reduction algorithms that were tested were the Classical method, the Barrett Method and the Montgomery method.
- The respective modifications and improvements for each method were taken into account.
- The modular exponentiation algorithm, $g^e \bmod m$, implemented was the Binary method (for further details see Section 5.2).
- The multiplication method utilized in the simulation was the Classical method.

Programming conditions:

University of Pretoria etd – Joseph, G (2005)

- Each algorithm was implemented using standard ANSI C coding. The base B was chosen as 2^{32}, hence used basic operations on integers of **unsigned int** type.
- The base g and m were randomly generated 1024 bit integers utilizing the MIRACL pseudo random number generator.
- The simulations were conducted for randomly generated values for the exponent e for bit sizes $256, 512, 1024$ and 2048.
- Though a lot of effort has been done to remove the overhead generated by the compiler, the test is still subjected to a little overhead generated by the platform and compiler.

Timing analysis parameters:

- One iteration consisted of a single run of the exponentiation algorithm for each reduction algorithm. The total time period of each test was 20 seconds.
- Each simulation was run until the total time period had elapsed and the number of iterations exceeded 20. The average time was calculated as a function of the total time elapsed divided by the total number of iterations.
- The timing of the precomputations were not taken into account, however argument transformations and postcomputations were taken, as they were computed within the modular exponentiation.

Fig. 4.3 provides the time results of the three reduction methods implemented in a modular exponentiation. It depicts the average time it takes to compute a modular exponentiation over different bit sizes of the exponent.

The calculation of $g^e \bmod m$ in the simulation used the standard binary method in which various exponent bit sizes were used. Each of the three reduction algorithms are used in this implementation resulting in three modular exponentiation functions. The speed differences between the reduction functions are consequently reflected in the speed differences between the exponentiation functions, as shown in Fig. 4.3.

University of Pretoria etd – Joseph, G (2005)

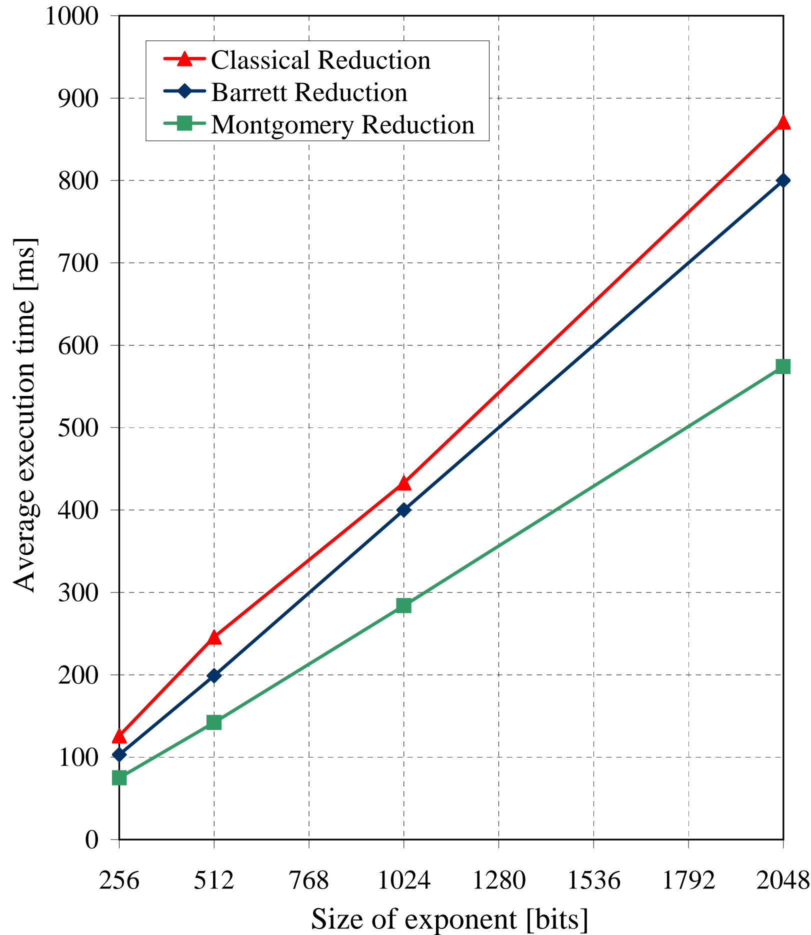


Figure 4.3: Comparison of the discussed reduction methods

For the various sizes of the exponent e for the modular exponentiation, the Montgomery based exponentiation is slightly faster than the Barrett based exponentiation, which in turn is slightly faster than the Classical technique.

The above observation can be explained with reference to Table 4.1. An indication of the performance of the different reduction methods can be given by the number of single-precision multiplications and divisions required to reduce an integer twice as long as the modulus. This approach is justified by the fact that a multiplication and a division are the most time consuming operations in the inner loops of all three methods with respect to which the others are negligible [77].

University of Pretoria etd – Joseph, G (2005)

Table 4.1: Complexity of the reduction methods in reducing a $2k$-digit integer [77]

Method	Classical	Barrett	Montgomery
Multiplications	$k(k+2)$	$k(k+4)$	$k(k+1)$
Divisions	k	0	0
Precomputations	Normalization	B^{2k}/m	$-m^{-1} \bmod B$
Argument transformation	None	None	m-residue
Postcomputations	Unnormalization	None	Reduction
Restrictions	None	$a < B^{2k}$	$a < mB^{2k}$

The number of multiplications and divisions in Table 4.1 are only for the core reduction operation (i.e. it does not include the multiplications and divisions of the precomputations, the argument transformations and the postcomputations). The reference operation is the multiplication of two k-digit integers which produces $2k$-digit a to be reduced by k-bit modulus m [77].

Table 4.1 indicates that if only the core reduction operation is considered, the Montgomery algorithm, in terms of single-precision multiplications, is clearly faster than both the Barrett and the Classical reduction and is almost as fast as a Classical multiplication. However, this is restricted to moduli m where $\gcd(m, B) = 1$ is satisfied. The Barrett reduction, although requires more single-precision multiplications than the Classical method, does not have the time-consuming division step. This provides its slight time advantage over the Classical method.

The precomputations, transformations and postcomputations introduce an overhead penalty for using Montgomery reduction. The impact of this overhead varies greatly depending on the application; in the case of modular exponentiation the overhead is subjected across thousands of modular multiplications, effectively eliminating it altogether from any sort of performance analysis. Whereas in the case of fewer modular multiplications, the overhead can effectively double the execution time of the algorithm, making the Montgomery method infeasible. Thus, it is better to use the Classical or Barrett method for such operations.

University of Pretoria etd – Joseph, G (2005)

4.5 CHAPTER SUMMARY

In general all the methods implemented in industry thus far, are variations of the Classical, Barrett, and Montgomery reduction methods [77, 80, 56]. A theoretical and practical comparison has been made of three methods for the reduction of large numbers. The classical reduction is the best choice for single modular multiplication. Modular exponentiation based on Barrett's reduction is superior to the others for a small number of modular multiplications. For general modular exponentiations the exponentiation based on the Montgomery method provides the best performance.

University of Pretoria etd – Joseph, G (2005)

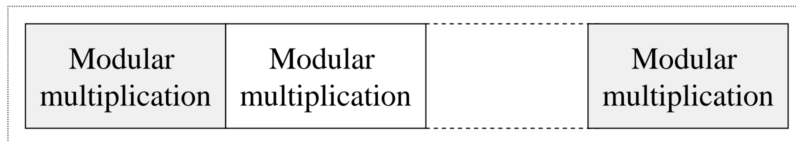
CHAPTER FIVE

FAST EXPONENTIATION TECHNIQUES

"It is insufficient to protect ourselves with laws; we need to protect ourselves with mathematics"

ANONYMOUS

In previous chapters, the focus was on reducing the time required to perform a modular multiplication utilized in a modular exponentiation. The focus of this chapter is to reduce the number of modular multiplications in a modular exponentiation, thus also reducing the time to perform a modular exponentiation.



This chapter gives algorithmic descriptions of currently implemented methods that perform the modular exponentiation operation i.e. $g^e \bmod m$. Hence the following algorithms will be evaluated:

- *The Binary method* [18].
- *The K-ary method* [18]. The K-ary method partitions the exponent e into words of equal length and then performs as many modular multiplications as there are nonzero words.
- *The Sliding window methods* [22]. Certain partitioning strategies to reduce the number of nonzero words, and thus reduce the number of modular multiplications. Modular

University of Pretoria etd – Joseph, G (2005)

exponentiation algorithms that use such partitioning strategies are termed sliding window methods.

- *Addition-chain heuristics* [91, 57, 92]. Addition-chains are closely related to modular exponentiation, since the optimal strategy to compute a modular exponentiation corresponds to some minimum length addition-chain.

This chapter will provide a comparison of the above methods with respect to the number of modular multiplications, the exponent's Hamming weight and its time complexities.

5.1 THE CLASSICAL METHOD

The Classical method is the simplest method, derived straight from basic arithmetic. The computation is simple for g^e : multiply the base g by itself e times, where e is the exponent [16].

In terms of modular exponentiation i.e. $A = g^e \bmod m$, first set $A = g$ then compute $A = A \cdot g \bmod m$ and keep repeating $A = A \cdot g \bmod m$ until $A = g^e \bmod m$. This would require $e - 1$ modular multiplications to compute the exponentiation. For example computing $g^{15} \bmod m$ would require computing all the powers of g until 15. That is:

$$g \rightarrow g^2 \rightarrow g^3 \rightarrow g^4 \rightarrow ... \rightarrow g^{15}$$

This method would require 14 multiplications [50]. As can be seen, this method is extremely inefficient. In the following sections we will describe more efficient methods.

5.2 THE BINARY METHOD

The Binary method is a substantial improvement on the Classical method. It dates back to antiquity and is also known as the square-and-multiply method [50].

5.2.1 The Algorithm

The left-to-right Binary method scans the bits of the exponent from the most significant bit (MSB) to the least significant bit(LSB). A squaring is performed after each bit scan, and

University of Pretoria etd – Joseph, G (2005)

depending on the scanned bit value, a subsequent multiplication is performed.

Let n be the number of bits in the exponent e, i.e. $n = \lfloor \log_2 e \rfloor$, and the binary expansion of e is given as:

$$e = e_{n-1}e_{n-2}...e_1e_0 = \sum_{i=0}^{i=n-1} e_i 2^i$$

Hence the Binary algorithm that computes $g^e \bmod m$ can be stated as follows:

ALGORITHM: BINARY EXPONENTIATION

Input. Base g, modulus m and exponent $e = (e_{n-1}e_{n-2}....e_1e_0)_2$

Output. $A = g^e \bmod m$

1. *Initialize* A Set $A \leftarrow g$
2. *Loop function* For i from $(n-2)$ down to 0 do the following:
   2.1 Set $A \leftarrow A^2 \bmod m$.
   2.2 If $e_i = 1$ then set $A \leftarrow A \cdot g \bmod m$.
3. *Final result*: Return A.

The above algorithm, adapted from [25], is the left-to-right Binary method. Knuth [18] provides a detailed description for the right-to-left version of the method. The right-to-left method requires one extra variable to store the powers of g, hence requires more memory.

5.2.2 Computational Efficiency

The total number of modular multiplications (T) is a summation of three components. Namely, precomputations before the algorithm (P), the squarings (S) and the multiplications (M) that occur in the algorithm loop.

For an arbitrary n-bit exponent e, the Binary method requires [50]:

- *Precomputations*: $P = 0$. The Binary method requires no precomputations.
- *Squarings*: $S = n - 1$. A squaring is computed for each bit of the exponent, except for the most significant bit.

University of Pretoria etd – Joseph, G (2005)

- *Multiplications* $M = \frac{1}{2}(n-1)$. A multiplication is performed each time $e_i = 1$. Hence the number of multiplications in the loop is equal to $H(e) - 1$, where $H(e)$ is the Hamming weight of the exponent e. Therefore, for an even distribution of ones, the number of multiplication is approximately equal to $\frac{1}{2}(n-1)$.

Thus, the total number of modular multiplications T is found as:

$$T = S + M = n - 1 + \frac{1}{2}(n-1) = \frac{3}{2}(n-1) \tag{5.1}$$

where it is assumed that $e_{n-1} = 1$.

Table 5.1 tabulates the total number of multiplications required by the Binary method and the Classical method for typical n-bit values of e.

Table 5.1: The computational efficiency of the Classical and Binary methods for a n-bit exponent

	Classical	Binary		
n	T	S	M	T
128	$2^{128}-1$	127	64	191
256	$2^{256}-1$	255	128	383
512	$2^{512}-1$	511	256	767
1024	$2^{1024}-1$	1023	512	1535
2048	$2^{2048}-1$	2047	1024	3071

From the above table one can see the practical efficiency of the Binary method over its Classical counterpart. Cohen [93] provides a more comprehensive treatment of the practicality of the Binary method.

5.3 THE K-ARY METHOD

e = 1234567 = | 100 | 101 | 101 | 011 | 010 | 000 | 111 |

window size = 3 bits

The K-ary method builds on the idea of the Binary method, but instead of breaking the exponent into single bits the K-ary methods breaks the exponent into k-bit windows, and then

University of Pretoria etd – Joseph, G (2005)

performs as many multiplications as there are nonzero windows. An illustrative example ($e = 1234567$) of the formalization of the k-bit windows, where $k = 3$, is shown on the previous page.

5.3.1 The Algorithm

The K-ary method first computes the values of $g^i \bmod m$ for $i = 2, 3, ..., 2^k - 1$. The method then partitions the binary expansion of the exponent, i.e. $e = (e_{n-1}e_{n-2}...e_1e_0)_2$, into s blocks of bit-length k (note that $s \times k = n$). The actual k-bit windows are then defined as:

$$f_i = (e_{ik+k-1}e_{ik+k-2}...e_{ik}) = \sum_{j=0}^{k-1} e_{ik+j}2^j \tag{5.2}$$

In a series of steps, the partial result is raised to the 2^k power and multiplied with $g_{f_i} \bmod m$ where f_i is the current nonzero window [50]. The algorithm is shown as follows:

ALGORITHM: K-ARY EXPONENTIATION

Precomputation. Compute and store $g_j \bmod m$ for $j = 2, 3, 4, ..., 2^k - 1$
Break e into f_i words of k-bit length for $i = 0, 1, 2, ...s - 1$

Input. Base g, modulus m and partitioned exponent e
Output. $A = g^e \bmod m$

1. *Initialize* A Set $A \leftarrow g_{f_{s-1}} \bmod m$
2. *Loop function* For i from $(s - 2)$ down to 0 do the following:
   2.1 Set $A \leftarrow A^{2^k} \bmod m$.
   2.2 If $f_i \neq 0$ then set $A \leftarrow A \cdot g_{f_i} \bmod m$.
3. *Final result*: Return A.

This algorithm, unlike the Binary method, contains a certain amount of precomputations which if used effectively, will reduce the total number of operations needed by the modular exponentiation. Knuth [18] explains the K-ary method in great detail. An analysis in also found in Koç [50].

5.3.2 Computational Efficiency

Since the K-ary method is the generalization of the Binary method, the exponent can be represented by more than two states. The drawback, however, is that it requires a certain

University of Pretoria etd – Joseph, G (2005)

degree of precomputation. Thus, for an arbitrary n-bit exponent e with $e = \sum_{i=0}^{s-1} f_i$, the K-ary method requires [50]:

- *Precomputations*: $P = 2^k - 2$.
- *Squarings*: $S = k(s-1)$. This is simplified to $(\frac{n}{k} - 1)k = n - k$, where s is the number of windows in e.
- *Multiplications*: $M = (\frac{n}{k} - 1)(1 - 2^{-k})$. A multiplication is performed if $f_i \neq 0$, since $(2^k - 1)$ out of 2^k values of f_i are nonzero, that is the probability of $f_i \neq 0$ is $1 - 2^{-k}$.

Therefore the total average number of modular multiplications is

$$T = 2^r - 2 + k - r + (\tfrac{n}{k} - 1)(1 - 2^{-k}) \tag{5.3}$$

There exists an optimum k (denoted k^*) for a given n-bit exponent length that will reduce T in Eq. 5.3 to be a minimum. These values can be calculated by enumeration [18].

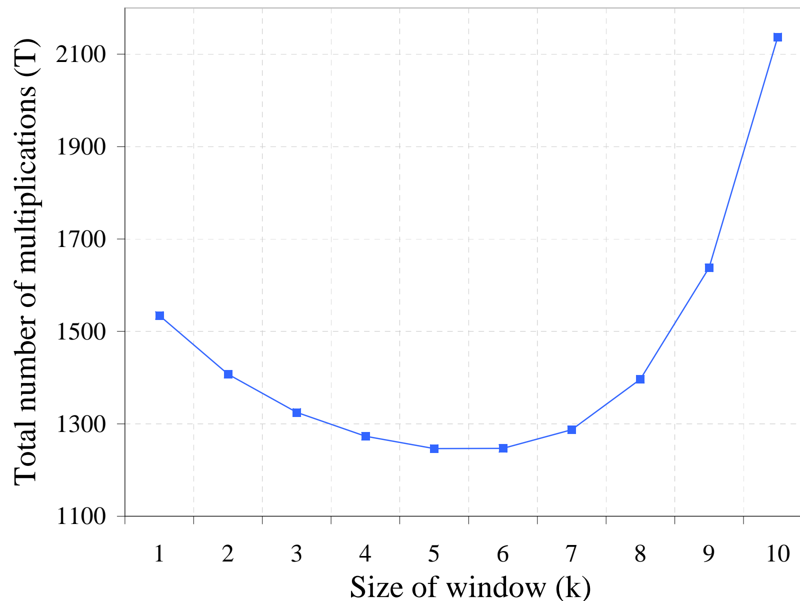


Figure 5.1: Enumeration graph for $n = 1024$ using the K-ary method

Utilizing Eq. 5.3 for 1024-bit exponent, the enumeration graph shown in Fig. 5.1 is established. From Fig. 5.1, one can see as k gets larger; T decreases until a specific window size. At that specific window size, i.e. the optimal window size k^*, T will be at minimum. From Fig. 5.1, the lowest number of modular multiplications (1246) is obtained at $k = 5$,

University of Pretoria etd – Joseph, G (2005)

hence $k^* = 5$. Similar enumeration graphs were computed for $n = 128, 256, 512$ and 2048, and the results are tabulated below.

Table 5.2: The computational efficiency of the Binary and K-ary methods

	Binary	K-ary				
n	T	k^*	P	S	M	T
128	191	3,4	6,14	125,124	36,29	167
256	383	4	14	252	59	325
512	767	5	30	507	98	635
1024	1535	5	30	1019	197	1246
2048	3071	6	62	2042	335	2439

Table 5.2 tabulates each of the components required to compute T for both the Binary and K-ary methods (using k^*) respectively for different n-bit values of e. The average number of modular multiplications can be found by substituting $k = 1$ into Eq. 5.3, which gives $\frac{3}{2}(n-1)$.

The use of the K-ary method over the Binary method results in an average saving of 13% to 21%, with respect to the bit-size of the exponent. Koç [50] shows an asymptotic value of savings offered by the K-ary method over the Binary method is 33% as the bit-length n tends to infinity.

5.4 SLIDING WINDOW METHODS

The sliding window methods are adaptive K-ary techniques which modify their structure according to the exponent e. These adaptive methods partition the exponent into a series of variable zero and nonzero windows in order to decrease the total number of nonzero windows [50]. The main aims of the sliding window methods are to reduce the number of nonzero windows and to reduce the number of precomputations.

In step 2.2 of the K-ary method, a loop multiplication is skipped if a zero window is encountered. Thus, the total number of modular multiplications is decreased by decreasing

University of Pretoria etd – Joseph, G (2005)

the number of nonzero windows. The sliding window methods also attempt to half the number of precomputations required by the K-ary by partitioning the exponent in such a way that only odd nonzero windows are created.

5.4.1 The Algorithm

A sliding window exponentiation algorithm first decomposes exponent e into zero windows (ZW) and nonzero windows (NZW) f_i of length $L(f_i)$. The number of windows s may not be necessarily equal to n/k, where n is the bit-length of e and k is the window size. In general, it is not required that the length of the windows be equal.

The decomposition of the exponent is structured such that the LSB of each NZW equals 1, i.e. the NZW is odd. Consequently, the number of precomputations is halved, since only odd powers of g needs to be precomputed [50]. The generic sliding window method is as follows:

ALGORITHM: SLIDING WINDOW EXPONENTIATION

Precomputation. Compute and store $g_j \bmod n$ for $j = 2, 3, 5, 7, ..., 2^r - 1$

Break e into f_i words of $L(f_i)$-bit length for $i = 0, 1, 2, ...s - 1$

Input. Base g, modulus m and partitioned exponent e

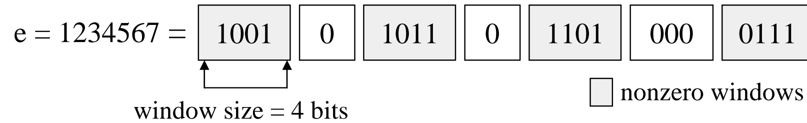
Output. $A = g^e \bmod m$

1. *Initialize* A Set $A \leftarrow g_{f_{s-1}} \bmod m$
2. *Loop function* For i from $(s - 2)$ down to 0 do the following:
   2.1 Set $A \leftarrow A^{2^{L(f_i)}} \bmod m$.
   2.2 If $f_i \neq 0$ then set $A \leftarrow A \cdot g_{f_i} \bmod m$.
3. *Final result*: Return A.

The above algorithm is adapted from [50]. The actual difference between the sliding window method and the K-ary method comes in the partitioning of the exponent e. Koç [22] provides two partitioning strategies to decompose the exponent. These strategies were initially proposed by Knuth [18] and Bos *et al.* [91]. The methods, though very similar in structure, differ in whether the length of a nonzero window must be constant($L(f_i) = k$), or can it be variable ($L(f_i) \leq k$), where r is the maximum length of the NZW, i.e. $r = max(L(f_i))$ for $i = 0, 1, 2, ...s - 1$ for all $f_i > 0$. In the following sections, algorithmic descriptions of these two partitioning strategies will be given.

University of Pretoria etd – Joseph, G (2005)

5.5 CONSTANT LENGTH NONZERO WINDOWS



The constant length nonzero window (CLNW) is a partitioning strategy that scans the bits of the exponent from the least significant to the most significant bit [50]. During scanning, it decomposes e into either a ZW or a NZW. The technique is described below:

- *Making ZW*. Check the incoming single bit: if it is a 0 then stay in ZW, else go to NW.
- *Making NZW*: Stay in NZW until all k bits are collected, where k is the maximum window size, then check the incoming single bit. If it is a 0 then go to ZW, else create a new NZW.

The CLNW technique produces zero windows of arbitrary length, and nonzero windows of length k. No adjacent ZW may occur, since adjacent zero windows are concatenated, while two NZW may be adjacent. An illustrative example ($e = 1234567$) of the formalization of the k-bit CLNW windows, where $k = 4$, is shown above. The CLNW state diagram is shown in Fig. 5.2.

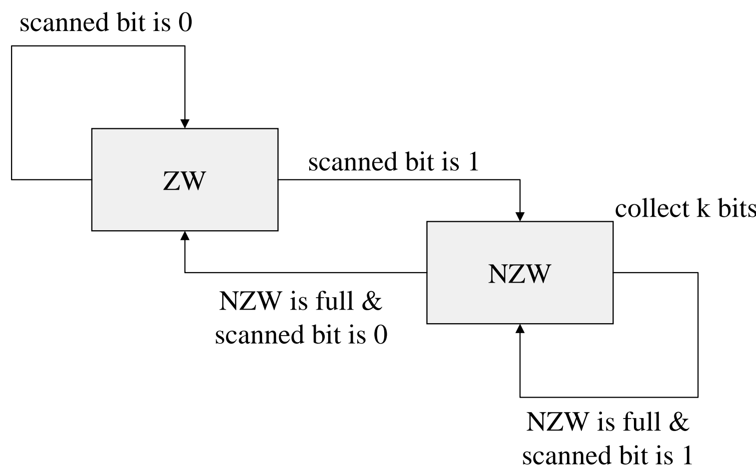


Figure 5.2: CLNW state diagram

5.5.1 The Algorithm

Given exponent e of n-bits, the window size k, an algorithm is established to create NZW and ZW f_i of length $L(f_i)$. The CLNW partitioning algorithm is shown as follows:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: CREATING CONSTANT LENGTH NONZERO WINDOWS

Input. Exponent e of n bits, and window size k

Output. Partition f_i of length $L(f_i)$ of exponent e for $i = 1, 2, ..., s-1$

Search the exponent For i from 0 to n do the following

1. *Create ZW* If $e_i = 0$
   1.1 $f_i \leftarrow 0$, $L(f_i) \leftarrow 1$ and set $i = i + 1$.
   1.2 While $e_i = 0$ set $L(f_i) \leftarrow L(f_i) + 1$ and $i = i + 1$.
2. *Create NZW* If $e_i = 1$
   2.1 $f_i \leftarrow e_i + e_{i+1} + ... + e_{i+k}$.
   2.2 Set $L(f_i) \leftarrow k$ and set $i = i + k$.
3. *Final result*: Return f_i and $L(f_i)$.

5.5.2 Computational Efficiency

In order to compute the minimum number of modular multiplications required by the CLNW partitioning strategy, a practical enumeration of the CLNW sliding window method ($g^e \bmod m$) must be performed Table 5.3 tabulates the simulation results in terms of the number of multiplications, squarings and precomputations required by the K-ary and CLNW sliding window methods.

Table 5.3: The computational efficiency of the K-ary and CLNW sliding window methods

	K-ary	CLNW sliding window				
n	T	k^*	P	S	M	T
128	167	4	8	125	25	157
256	325	4	8	253	50	311
512	635	5	16	509	84	609
1024	1246	6	32	1020	145	1197
2048	2439	7	64	2044	255	2363

The simulations for Table 5.3 were setup such that 1000 random exponent e samples were generated for the following n-bit sizes: $128, 256, 512, 1024, 2048$. The base g and modulus m were randomly generated 2048 bit integers utilizing the MIRACL pseudo random number generator. Counters were implemented in the precomputation, squaring and multiplication

University of Pretoria etd – Joseph, G (2005)

steps of the sliding window algorithm. In order to compute the optimal window size k^*, the simulations were configured to test each window size k in the range $2 \leq k < 32$ for the 1000 generated exponent values. The tests were repeated for each exponent bit-size.

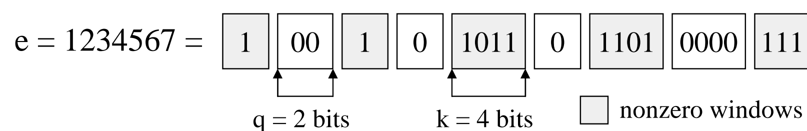
Koç [22] calculates the above results by modelling the CLNW partitioning strategy as a Markov chain. He states that for an arbitrary n-bit exponent e ($e = \sum_{i=0}^{s-1} f_i$ where s is the number of windows), the sliding window with CLNW technique requires:

- *Precomputations*: $P = (2^k - 2)/2 = 2^{k-1}$.
- *Squarings*: $S = k(s-1)$. This is simplified to $(\frac{n}{k} - 1)k = n - k$.
- *Multiplications*: M. A multiplication is performed if f_i is a NZW. Koç computes the number of nonzero windows statistically by a Markov chain process. The statistically computed multiplications are comparable to the multiplications shown in Table 5.3.

As shown in Table 5.3, the number of squarings are not equivalent to Koç's theoretical value of $n - k$. This is because the number of windows (s) is not necessarily equivalent to n/k. From the practical analysis performed, it was found that the number of squarings is dependent on the size of the most significant NZW in the exponent. Thus, the number of squarings $S = n - L(f_{s-1})$, which is often less than k.

The CLNW sliding window reduces the total number of multiplications required by the K-ary method by 3-7 % for $128 \leq k \leq 2048$. These improvements are due to the reduction of precomputations (odd NZWs) and multiplications (fewer NZWs) required by the CLNW sliding window algorithm.

5.6 VARIABLE LENGTH NONZERO WINDOWS



The CLNW technique starts with NZW when a one is encountered. Although the incoming $k-1$ bits may be zero, the algorithm appends them into the current NZW. The variable length nonzero window (VLNW) technique prevents such a NZW to exist [22]. In order to

University of Pretoria etd – Joseph, G (2005)

do this it requires two pivotal integer parameters: the maximum nonzero window length k and the minimum number of zeros q required to switch to the ZW.

The partitioning strategy is described as follows:

- *Create ZW*. Check the incoming single bit: if it is a 0 then stay in ZW, else go to NZW.
- *Create NZW*: Check the incoming q bits: if they are all zero then go to ZW; else stay in NZW. If not, add bits to nonzero window, and repeat process until either all k bits are collected or until q zeros are encountered. If k bits are collected then check the incoming single bit: if the bit is zero create a new ZW; else create a new NZW.

VLNW produces nonzero windows which start with a 1 and end with a 1. Two nonzero windows may be adjacent; however, the one in the least significant position will necessarily have k bits. Two zero windows will not be adjacent since they are concatenated. An illustrative example ($e = 1234567$) of the formalization of the k-bit VLNW windows, where $k = 4$ and $q = 2$, is shown above. The VLNW state diagram is shown in Fig. 5.3.

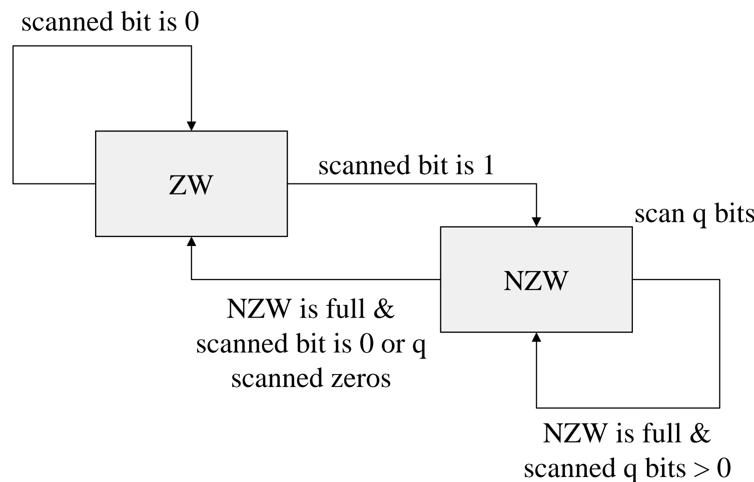


Figure 5.3: VLNW state diagram

5.6.1 The Algorithm

Given exponent e of n-bits, the window size k and the minimum number of zeros q required to switch to the ZW, an algorithm is established to create NZW and ZW f_i of length $L(f_i)$. The VLNW partitioning algorithm is shown as follows:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: CREATING VARIABLE LENGTH NONZERO WINDOWS

Input. Exponent e of n bits, and window size k and minimum number of zeros q

Output. Partition f_i of length $L(f_i)$ of exponent e for $i = 1, 2, ..., s-1$

Search the exponent For i from 0 to n do the following

1. *Create ZW* If $e_i = 0$
   - 1.1 $f_i \leftarrow 0$, $L(f_i) \leftarrow 1$ and set $i = i + 1$.
   - 1.2 While $e_i = 0$ set $L(f_i) \leftarrow L(f_i) + 1$ and $i = i + 1$.
2. *Create NZW* If $e_i = 1$
   - 2.1 Set $L(f_i) \leftarrow 1$ and $f_i \leftarrow 1$.
   - 2.2 Check incoming q bits:
     - 2.2.1 If all q bits zero then go to step 1
     - 2.2.2 Else set $L(f_i) = L(f_i) + q$ and add q bits to f_i
     - 2.2.3 Repeat 2.2 if $L(f_i) < k$
   - 2.3 Check if f_i has any leading zeros (l is the number of leading zeros).
     - 2.3.1 Eliminate leading zeros: $L(f_i) \leftarrow L(f_i) - l$.
   - 2.4 Set $i = i + L(f_i)$.
3. *Final result*: Return f_i and $L(f_i)$.

5.6.2 Computational Efficiency

For an arbitrary n-bit exponent e ($e = \sum_{i=0}^{s-1} f_i$ where s is the number of windows), the sliding window with VLNW technique requires:

- *Precomputations*: $P = (2^k - 2)/2 = 2^{k-1}$.
- *Squarings*: A squaring is performed for each bit of the exponent except for the most significant NZW.
- *Multiplications*: A multiplication is performed if f_i is a NZW.

The number of squarings and multiplications depend on the conditions set by Fig. 5.3. Koç [22] models these processes by a three state Markov chain. He provides a detailed analysis of this model and states that the optimal values of q are between 1 and 3 and the optimal window size k is between 4 and 6 for $128 \leq k \leq 2048$.

In order to verify statistical analysis stated in [22], a practical enumeration of the

University of Pretoria etd – Joseph, G (2005)

VLNW sliding window method ($g^e \bmod m$) was performed. The simulations were setup for the VLNW partitioning technique using the same configuration shown in Section 5.5.2.

To compute the optimal k^* and optimal q^*, the simulations were configured to test each window size k in the range $2 \leq k < 32$ and each q in the range $1 \leq q \leq k-1$. The tests were repeated for each exponent bit-size and the data collected was averaged over 1000 runs. The summarized results of the simulations are tabulated in Table 5.4.

Table 5.4: The computational efficiency of the CLNW and VLNW sliding window methods

	CLNW	VLNW sliding window					
n	T	k^*	q^*	P	S	M	T
128	157	4	3	8	125	25	157
256	311	4	3	8	253	50	311
512	609	5	4	16	509	84	609
1024	1197	6	5	32	1020	145	1197
2048	2363	7	6	64	2041	258	2363

Koç stated in [22] that VLNW technique would reduce the number of NZWs, thus being more computation efficient than the CLNW technique. From the practical enumeration performed, the VLNW technique decomposed the exponent into larger ZWs and shorter NZWs than the CLNW technique. However, the number of NZWs did not decrease. In fact for simulations where $q < k-1$, more NZWs were created.

If the VLNW technique is set to $q = k-1$, it decomposes the exponent identically to the CLNW technique. Thus, the number of precomputations, squarings and multiplications shown in Table 5.4 are identical to Table 5.3.

The summarized results in Table 5.4 show that when $q = k-1$, the total number of modular multiplications required by the modular exponentiation is at a minimum. Thus, the CLNW method is the better partitioning strategy to use in the sliding window method.

University of Pretoria etd – Joseph, G (2005)

5.7 ADDITION CHAINS

The optimal strategy to obtain the minimum number of modular multiplications in a modular exponentiation corresponds to a shortest path to the exponent value. This leads to the study of addition chains which is a research area for more than 100 years old [94].

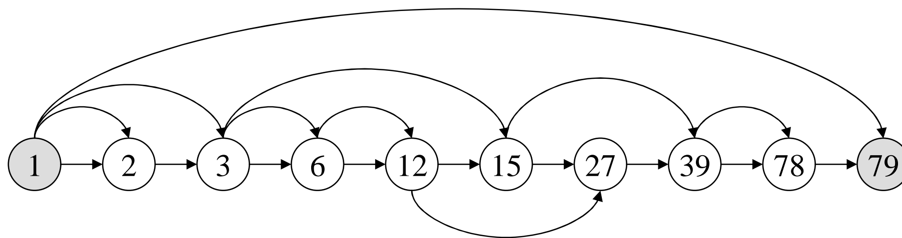
5.7.1 Description

Consider a sequence of integers $a_0, a_1, a_2, ..., a_{r-1}, a_r$ where $a_0 = 1$ and $a_r = e$. If the sequence is constructed in such a way that for all k there exist indices $i, j < k$ such that

$$a_k = a_i + a_j \tag{5.4}$$

then the sequence is an addition chain for e. The addition chain length is the number of elements r in the addition chain.

An addition chain can be represented by a directed graph, where the vertices are labelled a_k for $0 \leq k \leq r$. Arcs are drawn from a_j to a_k and from a_i to a_k as a representation of each step $a_k = a_i + a_j$ in Eq. 5.4. For example, the addition chain $1, 2, 3, 6, 12, 15, 27, 39, 78, 79$ corresponds to the following directed graph:



5.7.2 Addition Chain Heuristics

In terms of modular exponentiation, the addition chain method for an exponent e is computed as follows: Start with $g^1 \bmod m$, and proceed to compute $g^{a_k} \bmod m$ using the two previously computed values $g^{a_i} \bmod m$ and $g^{a_j} \bmod m$ as $g^{a_k} \bmod m = g^{a_i} \cdot g^{a_j} \bmod m$ [50]. The number of modular multiplications required is equal to the length of the addition chain. Thus, the task of minimizing the number of modular multiplications is equivalent to finding the minimal addition chain length.

University of Pretoria etd – Joseph, G (2005)

The methods introduced so far, namely, the Binary method, the K-ary method and the sliding window methods are in fact methods of generating addition chains for the given exponent [50]. Consider for example $e = 55$, the addition chains generated by the discussed methods are shown below:

Table 5.5: Addition chain of exponentiation methods for $e = 55$

Method	Window										
Binary method	$k = 1$	1	2	3	6	12	13	26	27	54	55
K-ary method	$k = 2$	1	2	3	6	12	13	26	52	55	
CLNW sliding window method	$k = 2$	1	2	3	6	12	13	26	52	55	
VLNW sliding window method	$k = 3$	1	2	3	5	6	7	12	24	48	55

Heuristics are practical methods that create addition chains for a particular integer. However, they do not guarantee the shortest addition chain length for that particular integer. The methods shown in Table 5.5 are all heuristics for generating short addition chains. However, creating minimum length addition chains for very large integers (512-bits and above) is extremely difficult [91, 95]. Through heuristics, methods to compute an addition chain for very large integers is feasible.

The heuristics, thus far, implemented a basic exponentiation step (square-and-multiply) and a window decomposition step to reduce the computation of the addition chain for a large exponent e to the computation of an addition sequence by choosing an appropriate set of integers which are much smaller than e. The computation of short addition chain lengths can be further improved by producing a short addition chain sequence for those integers. A proposed heuristic applying this additional steps is described in the following section.

5.7.3 The Algorithm

The addition chain algorithm consists of three steps:

- *Window decomposition*: This step partitions the exponent e into smaller windows w_i.
- *Make sequence*: This step creates an addition sequence for a set of integers utilizing a specialized algorithm proposed by Bos and Coster [91].
- *Addition chain method*: A modified binary exponentiation step for addition chains.

University of Pretoria etd – Joseph, G (2005)

5.7.3.1 Window Decomposition

There are three different techniques to decompose the exponent e into smaller windows. These windowing techniques are the fixed window decomposition, constant length nonzero window decomposition and variable length nonzero window decomposition.

100	110	100	100	000	010	001	101	000	111

Fixed window decomposition. The exponent is decomposed into windows of fixed k-bit length, as shown above. For further details, refer to Section 5.3.

1	0011	0	1001	000	0001	000	1101	00	0111

Constant length nonzero window decomposition. The CLNW decomposition produces zero windows of arbitrary length, and nonzero windows of length k. For further details, refer to Section 5.5.

1	00	1101	00	1	000000	1	000	1101	000	111

Variable length nonzero window decomposition. The VLNW decomposition produces zero windows of arbitrary length, and nonzero windows of a maximum length k. This creates larger ZWs and smaller NZWs. For further details, refer to Section 5.6.

5.7.3.2 Make Sequence

The initial addition chain sequence, the proto-sequence, consists of 1 and 2 and the decomposed window values in ascending order. The sequence is then increased with insertions of intermediate values required to obtain the required window values.

Bos *et al.* [91] describe four algorithms to create the addition sequence: Approximation, Division, Halving and Lucas algorithms. Good sequences can be found utilizing the Division and Lucas algorithms. However, implementing these algorithms is much more time consuming and complex, sometimes infeasible to implement [91].

University of Pretoria etd – Joseph, G (2005)

A combination of the Halving and Approximation algorithms provides a faster, less complex solution to creating the addition sequence. Thus, the *Make sequence* algorithm is as follows [57]:

ALGORITHM: CREATE ADDITION CHAIN SEQUENCE

Input. Decomposed exponent windows w_i

Output. Addition chain (AC) a_i with vertices v_{i_1} and v_{i_2}

1. *Create Initial set*. a_i consists of 1,2, and the ascending window values w_i
2. Set Element E to largest value in initial set.
3. While $E > 2$ do the following:
   - 3.1 Set S to the next smaller element after E.
   - 3.2 *Halving*. If $E - S > S$ then do the following:
     - 3.2.1 If E odd then increase the set with $(E - 1)$.
       Add vertices $v_{i_1} = 1$ and $v_{i_2} = E - 1$ to AC
     - 3.2.2 If E even then increase set with $(E/2)$. Add vertices $v_{i_1} = v_{i_2} = E/2$ to AC
   - 3.3 *Approximation*. Else if $E - S \leq S$ then do the following:
     - 3.3.1 Let $\exists(x, y) \in$ Set. If $x + y = E$, add vertices $v_{i_1} = x$ and $v_{i_2} = y$ to AC.
     - 3.3.2 Else increase set with $(E - S)$. Add vertices $v_{i_1} = S$ $v_{i_2} = E - S$ to AC.
   - 3.4 *Set Element to next smaller element in set* $E \leftarrow S$
4. Return addition chain a_i with vertices v_i and v_j

Vertices v_{i_1} and v_{i_2} are computed in correlation with addition chain element a_i, such that during the precomputation phase of the exponentiation: $g^{a_i} \bmod m$ can be computed as follows:

$$g^{a_i} \bmod m = g^{v_{i_1}} \cdot g^{v_{i_2}} \bmod m \tag{5.5}$$

The vertices v_{i_1} and v_{i_2} for their respective a_i may contain values less than a_i, so that Eq. 5.5 holds true. The following figure depicts the use of the above algorithm for the uncompleted addition chain $a = 1, 2, 6, 42$.

University of Pretoria etd – Joseph, G (2005)

STEP	ADDITION CHAIN (AC)	OPERATION	MODIFIED AC
1	$1, 2, 6, \mathbf{42}$	Halving: Insert 21	$1, 2, 6, 21, 42$
2	$1, 2, 6, \mathbf{21}, 42$	Halving: Insert 20	$1, 2, 6, 20, 21, 42$
3	$1, 2, 6, \mathbf{20}, 21, 42$	Halving: Insert 10	$1, 2, 6, 10, 20, 21, 42$
4	$1, 2, 6, \mathbf{10}, 20, 21, 42$	Approx: $6 + 4 = 10$, insert 4	$1, 2, 4, 6, 10, 20, 21, 42$
5	$1, 2, 4, \mathbf{6}, 10, 20, 21, 42$	Approx: $2 + 4 = 6$, no insert	$1, 2, 4, 6, 10, 20, 21, 42$
6	$1, 2, \mathbf{4}, 6, 10, 20, 21, 42$	Approx: $2 + 2 = 4$, no insert	$1, 2, 4, 6, 10, 20, 21, 42$
7	$1, \mathbf{2}, 4, 6, 10, 20, 21, 42$	Terminate algorithm	

Figure 5.4: Approximation and Halving for proto-sequence $\{1, 2, 6, 42\}$

5.7.3.3 Addition Chain Exponentiation

After the exponent has been decomposed into smaller windows and the make sequence algorithm creates the required addition chain sequence, the addition elements can be utilized in a modular exponentiation. The algorithm for the modular exponentiation is as follows:

ALGORITHM: ADDITION CHAIN EXPONENTIATION

Window decomposition. Break e into w_i words of $L(w_i)$-bit length for $i = 0, 1, 2, ...s - 1$

Make sequence. Make addition chain sequence a_i with vertices v_{i_1} and v_{i_2} from w_i.

Precomputation. Compute and store $g_{a_i} \bmod m = g_{v_{i_1}} \cdot g_{v_{i_2}} \bmod m$ for $a_i = 1, 2, ..., a_r$

Input. Base g, modulus m and partitioned exponent e

Output. $A = g^e \bmod m$

1. *Initialize* A Set $A \leftarrow g_{w_{s-1}} \bmod m$
2. *Loop function* For i from $(s - 2)$ down to 0 do the following:
   2.1 Set $A \leftarrow A^{2^{L(w_i)}} \bmod m$.
   2.2 If $w_i \neq 0$ then set $A \leftarrow A \cdot g_{w_i} \bmod m$.
3. *Final result*: Return A.

To analyze the addition chain method, an exponentiation graph of the exponent $e = 192000470$ is shown in Fig. 5.5.

University of Pretoria etd – Joseph, G (2005)

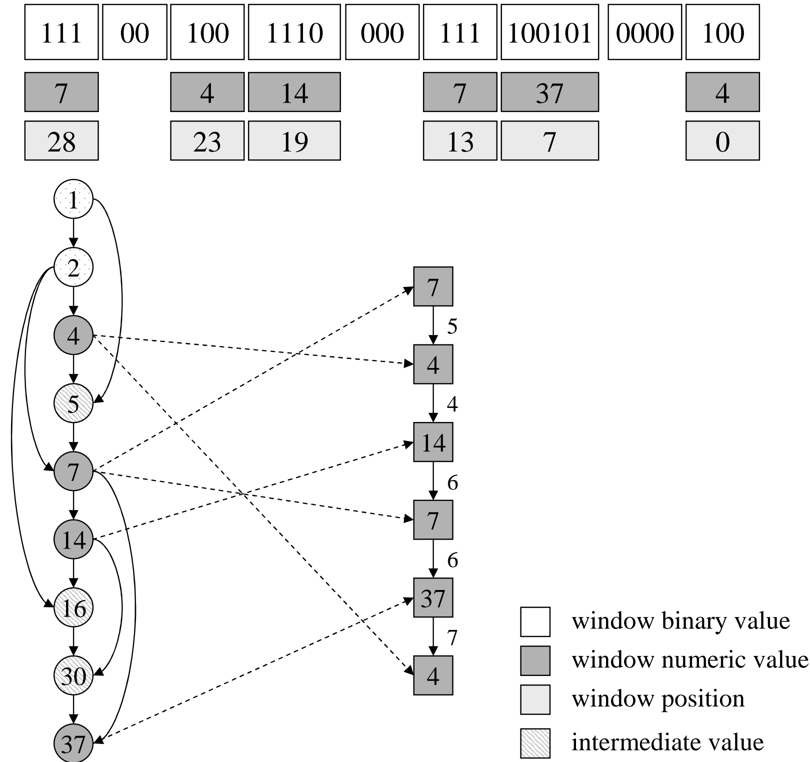


Figure 5.5: Exponentiation graph for $e = 192000470$ [57]

There are two chains in Fig. 5.5. The left chain is the addition sequence created to compute the required window values tabulated at the top of the figure. The right chain is the actual exponentiation flow that occurs. The numbers between the boxes on the right chain are the amount of squarings that occur between each window, which is determined by the position of the window. The number of modular multiplications required to compute exponent $e = 192000470$ are 28 squarings, 5 multiplications and 8 precomputations. The total is 41 operations, whilst the Binary method would require 45 operations to compute the same number.

In general, it is the case that using addition sequences instead of precomputed tables allows one to use bigger window sizes, giving shorter addition chains for the original exponent. The larger the window, the greater the amount of precomputations and thus fewer main-loop multiplications. The following section will provide practical enumeration of the variations of the addition chain heuristic namely the fixed window, CLNW and VLNW decompositions.

University of Pretoria etd – Joseph, G (2005)

5.7.4 Practical Enumeration

In order to compute minimum number of modular multiplications required by the addition chain heuristic, a practical enumeration must be performed. The enumerations were performed under the following conditions:

Algorithms tested:

- The modular exponentiation algorithm, $g^e \bmod m$, implemented was the addition chain exponentiation method shown in Section 5.7.3.3.
- Each enumeration utilized a different window decomposition technique: fixed window decomposition, CLNW decomposition and VLNW decomposition.
- The multiplication method utilized in the enumeration was the Karatsuba-Ofman with Comba method
- The reduction method utilized in the enumeration was the Montgomery method.

Programming conditions:

- Each algorithm was implemented using standard ANSI C coding. The base B was chosen as 2^{32}, hence used basic operations on integers of **unsigned int** type.
- The base g and m were randomly generated 2048 bit integers utilizing the MIRACL pseudo random number generator.
- The enumerations were conducted for 1000 randomly generated samples of the exponent e.
- Counters were implemented in the precomputation, squaring and multiplication steps of the exponentiation algorithm.
- In order to compute the optimal window size k^*, the enumerations were configured to test each window size k in the range $2 \leq k < 32$ for the 1000 generated exponent values.
- The tests were repeated for each exponent bit-size: $128, 256, 512, 1024$ and 2048.

University of Pretoria etd – Joseph, G (2005)

The results collected was averaged over 1000 iterations and the respective enumeration graphs and summary tables are presented for each window decomposition technique.

Fixed window decomposition. The enumeration graph, shown in Fig. 5.6, depicts the average number of modular multiplications against the various window sizes. The shaded bar is the optimal window size that provides the minimum number of modular multiplications.

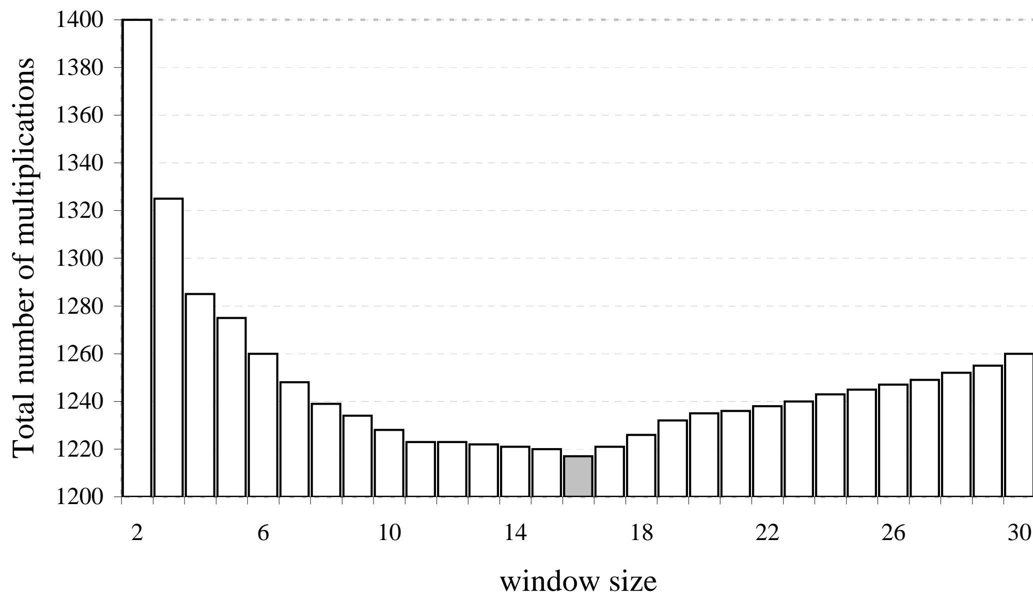


Figure 5.6: Enumeration graph for the fixed window addition chain method for 1024-bit exponent

Table 5.6 tabulates the results of the above enumeration in terms of the number of multiplications, squarings and precomputations required for n-bit exponent.

Table 5.6: Computational efficiency of the fixed window addition chain method

n	k^*	P	S	M	T
128	16	41	112	7	160
256	16	59	240	15	314
512	16	92	496	31	619
1024	16	147	1008	62	1217
2048	16	230	2032	127	2389

University of Pretoria etd – Joseph, G (2005)

Constant length nonzero window decomposition. The enumeration graph, shown in Fig. 5.7, depicts the average number of modular multiplications against the various window sizes. The shaded bar is the optimal window size that provides the minimum number of modular multiplications.

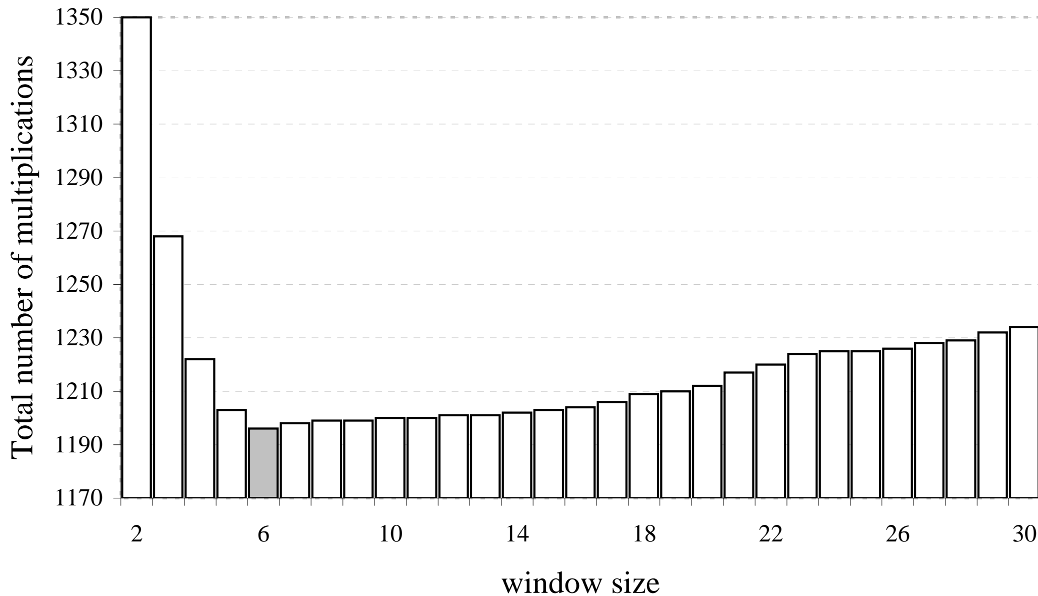


Figure 5.7: Enumeration graph for the constant length nonzero window addition chain method for 1024-bit exponent

Table 5.7 tabulates the results of the above enumeration in terms of the number of multiplications, squarings and precomputations required for n-bit exponent.

Table 5.7: Computational efficiency of the constant length nonzero window addition chain method

n	k^*	P	S	M	T
128	4	8	125	25	157
256	5	15	253	42	310
512	5	16	509	84	609
1024	6	31	1020	145	1196
2048	7	63	2044	255	2362

Variable length nonzero window decomposition. The enumeration graph, shown in Fig. 5.8, depicts the average number of modular multiplications against the various window sizes.

University of Pretoria etd – Joseph, G (2005)

The shaded bar is the optimal window size that provides the minimum number of modular multiplications.

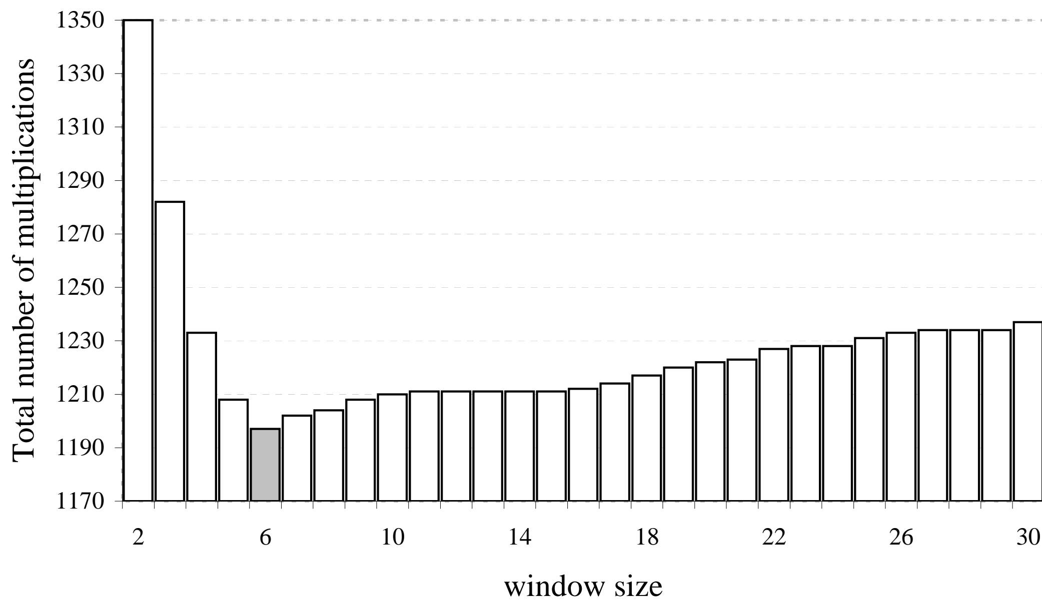


Figure 5.8: Enumeration graph for the variable length nonzero window addition chain method for 1024-bit exponent

Table 5.8 tabulates the results of the above enumeration in terms of the number of multiplications, squarings and precomputations required for n-bit exponent.

Table 5.8: Computational efficiency of the variable length nonzero window addition chain method

n	k^*	q^*	P	S	M	T
128	4	3	8	124	24	156
256	5	4	15	253	41	309
512	5	4	16	509	84	609
1024	6	5	31	1020	146	1197
2048	7	6	61	2044	256	2361

5.7.5 Discussion

The addition chain methods discussed in this section determine the total number of precomputations. In comparison to the sliding window methods for larger window sizes, the addition chain method requires significantly fewer precomputations. However, the addition

University of Pretoria etd – Joseph, G (2005)

chain methods do not improve on the sliding window methods in terms of the total number of modular multiplications, i.e. at its optimum window size it requires the same number of modular multiplications as the sliding window methods.

Fig. 5.6 to Fig. 5.8 shows a distinct minimum for T. This is due to the fact that with increasing window sizes the number of windows decrease while the size of the most significant window increases. This results in fewer multiplications and squarings [92]. However, when increasing the window sizes further, the addition chain for creating the window elements becomes longer, hence increasing the number of precomputations. The addition chains created, for these larger window sizes, are not the optimum because the make sequence algorithm employed is inefficient to identify short sequences out of larger integers.

The sequence algorithm has the following effects:

- It depends on the structure of the exponent, more specifically on the decomposed window values (w_i), as it creates the addition chain with respect to these values. The fewer number of different window values, the smaller the addition chain created.
- The values in the addition chain can only be twice their previous number (Halving operation) or the sum of two previous numbers (Approximation operation) in the sequence.
- Though the algorithm does not provide the shortest addition chain length, it provides a practical solution to find a shortish addition chain for the exponent.

From Fig. 5.6, it is noticed that though the optimal window size $(k = 16)$ produces a distinct minimum of 1217 total multiplications, it requires 147 precomputations. Since the relationship between performance and memory usage is important to the efficient implementation of the modular exponentiation algorithm, especially in resource-constrained environment, this high number of precomputations would require a large amount of stored memory compared to the sliding window methods. Hence, the fixed window addition chain method is inefficient in this regard.

Kunihiro and Yamamoto [96, 97, 98] proposed further systematic algorithms which

University of Pretoria etd – Joseph, G (2005)

create short addition chains for large Hamming weight exponents using smaller window sizes.

Understanding the relationship between performance and memory usage is vital in implementing the modular exponentiation in a resource constrained environment. The addition chain methods can require a large number of precomputations, hence memory. Certain of the precomputations are temporarily required and may be removed once it is utilized, and some are required for the majority of the chain. Sauerbrey [92] provides methods to reutilize memory space for storage of many precomputations.

5.8 THEORETICAL LIMITS

The computation of the shortest addition chain for a positive integer e is known to be an NP-complete problem [95]. This implies that all possible chains leading to e must be computed in order to obtain the shortest one. Since addition chains were introduced by Scholz [18] in 1937, its bounding properties have been established:

The upper bound on the length of the shortest addition chain for e is equal to:

$$L(e) \leq \log_2 e + H(e) - 1 \tag{5.6}$$

where $H(e)$ is the Hamming weight and $L(e)$ is the length of the addition chain. This upper bound corresponds to the number of operations required by the Binary method as long as the $H(e)$ is not small. Though the Classical exponentiation method has a much larger addition chain than the Binary method, anything worse than the Binary method is simply ignored by cryptographers.

The lower bound was established by Schönhage [99]:

$$L(e) \geq \log_2 e + log_2 H(e) - 2.13 \tag{5.7}$$

Theoretical analysis and the asymptotic bounds of addition chains are defined in [94, 100, 101, 102, 103, 95].

University of Pretoria etd – Joseph, G (2005)

5.9 EXPERIMENTAL RESULTS

To obtain a practical comparison for the discussed exponentiation methods to be used in a public-key environment, two tests must be performed:

- Hamming weight analysis
- Timing analysis

The simulations were done on a Pentium III processor running at 550 MHz with 256 Mbyte main memory under Windows XP Home Edition platform using a Borland C Builder 6.0 compiler. The simulations were performed under the following conditions:

Algorithms tested:

- The exponentiation algorithms that were tested were the Binary method, the K-ary method, the Sliding window method (utilizing CLNW decomposition) and the Addition-chain method (utilizing CLNW decomposition).
- Each method is configured to utilize its optimal window-size.
- The multiplication method utilized in the simulations was the Karatsuba-Ofman with Comba method
- The reduction method utilized in the simulations was the Montgomery method.

Programming conditions:

- Each algorithm was implemented using standard ANSI C coding. The base B was chosen as 2^{32}, hence used basic operations on integers of **unsigned int** type.
- The base g and m were randomly generated 1024 bit integers utilizing the MIRACL pseudo random number generator.

Hamming weight analysis parameters:

- 1000 samples of the exponent was generated for specific Hamming weight probabilities: $0.05, 0.5$ and 0.95.
- The tests were repeated for each exponent bit-size: $128, 256, 512, 1024$ and 2048.

University of Pretoria etd – Joseph, G (2005)

- A counter (C), representing the addition chain length of the algorithm, was incremented each time a precomputation, squaring or multiplication step occurred in the exponentiation algorithm.
- The respective normalized addition chain lengths of each of the exponentiation methods were calculated using the averaged value of C over 1000 iterations. A normalized addition chain length is the ratio of the addition length of e to the bit size of e, i.e. $\acute{L}(e) = \frac{C}{n}$.

Timing analysis parameters:

- The simulations were conducted for 1000 random exponent e samples for the specified bit size. $128, 256, 512, 1024$ and 2048.
- One iteration consisted of a single run of the exponentiation algorithm for each exponentiation algorithm. The total time period of each test was 20 seconds.
- Each simulation was run until the total time period had elapsed and the number of iterations exceeded 20. The average time was calculated as a function of the total time elapsed divided by the total number of iterations.
- The timing of the once-off precalculations were not taken into account (i.e. window decomposition, addition-chain sequence creation, calculation of $\acute{m}$). However, argument transformations, precomputations and postcomputations were taken in the run-time, as they were computed within the modular exponentiation operation.

Fig. 5.9 to Fig. 5.11 plots a normalized addition chain length of each exponentiation method against bit-length of the exponent for different Hamming weight probabilities. Fig. 5.12 gives the average run-time of the various exponentiation methods utilizing the optimal window-size over an even distribution of ones in the exponent. The graphs below have the following parameters:

- Size of the exponent: the length of the exponent in bits.
- Normalized addition-chain length: The averaged normalized addition chain length over 1000 iterations for the specific exponent bit-size.
- Schnönhage's limit: the lower bound of the shortest possible addition chain length.
- Time: The averaged run-time for one iteration of the exponentiation method.

University of Pretoria etd – Joseph, G (2005)

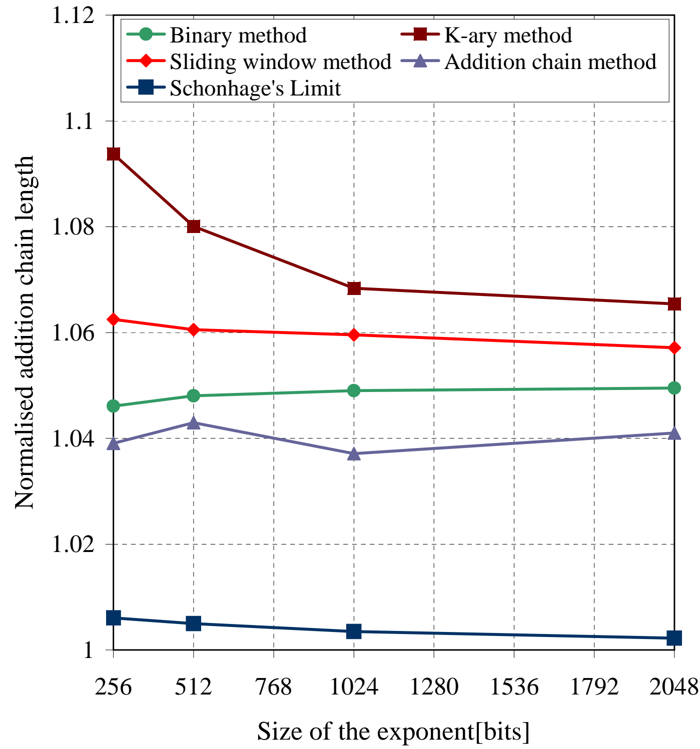


Figure 5.9: Normalized addition chain lengths for 5% $H(e)$ exponent

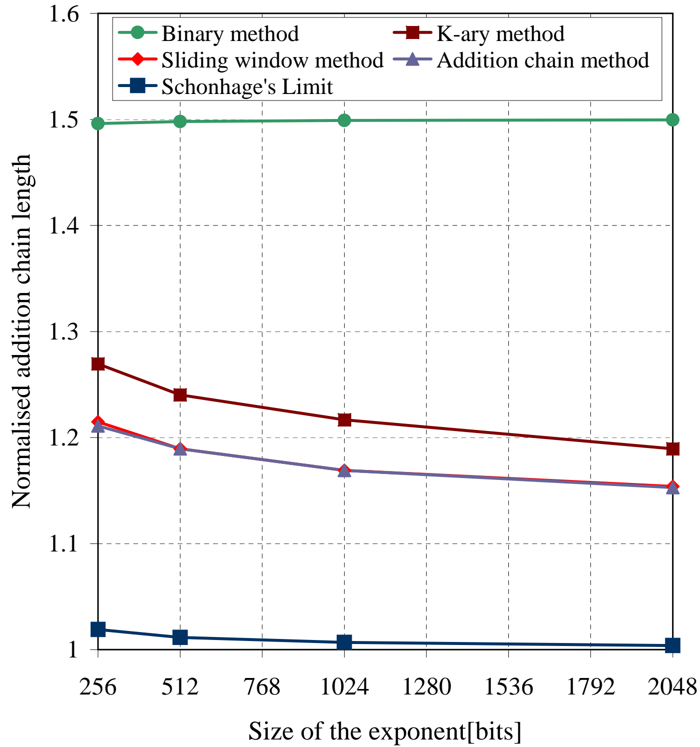


Figure 5.10: Normalized addition chain lengths for 50% $H(e)$ exponent

University of Pretoria etd – Joseph, G (2005)

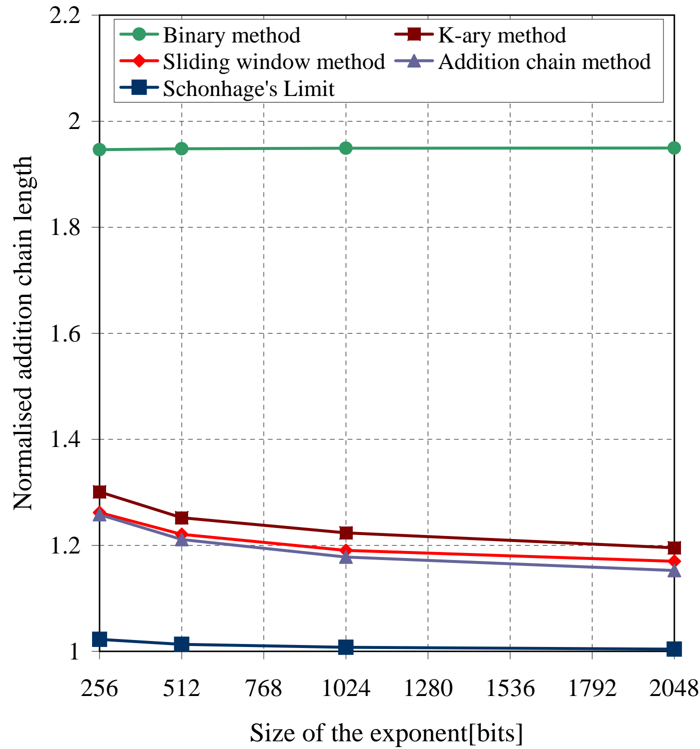


Figure 5.11: Normalized addition chain lengths for 95% $H(e)$ exponent

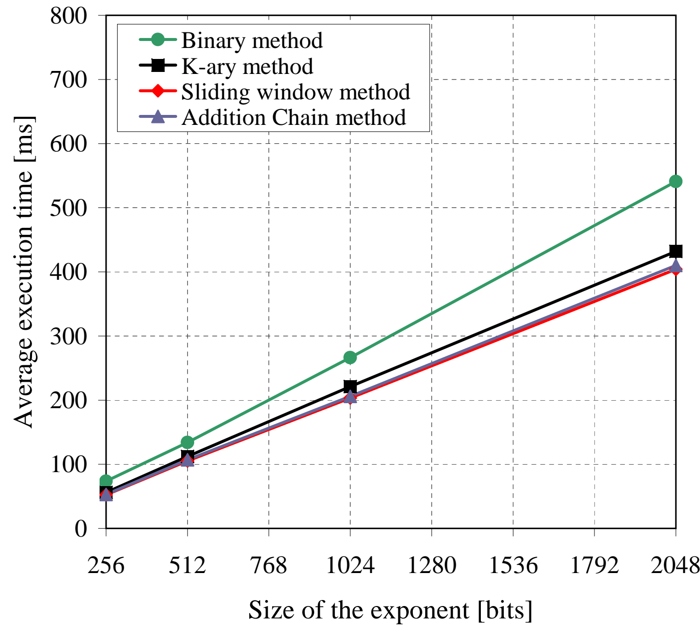


Figure 5.12: Time analysis of the exponentiation methods

University of Pretoria etd – Joseph, G (2005)

5.10 DISCUSSION

It is shown in Fig. 5.9 that the Addition-chain and the Binary methods give favorable results when the exponent has low Hamming weight. In Fig. 5.10 the Addition-chain method and Sliding window provides the best results for an even distribution of ones in the exponent. Fig. 5.11 shows that when the Hamming weight of the exponent is large the Addition chain method gives the shortest chain, whilst the Binary method gives the longest.

The Hamming weight of the exponent generally affects the number of multiplications (M) and squarings (S) that occur in a modular exponentiation. The larger the Hamming weight of the exponent the greater the number of multiplications. This is due to a smaller probability of zero windows in large Hamming weight exponents. However, the number of precomputations play an important role in the total number of modular multiplications (T). The K-ary and Sliding window methods have a fixed amount of precomputations which is independent to the Hamming weight of the exponent. The Binary method has no precomputations and depends entirely on the Hamming weight of the exponent.

The Addition-chain method utilizes its Make-sequence algorithm to adapt itself to the Hamming weight of the exponent to create a low number of precomputations. In terms of larger Hamming-weight exponents, the Make-sequence algorithm creates an addition-chain sequence of very large window values and very small window values. This is due to the large amount of large valued windows, that are numerically close to each other, created by the window decomposition employed. In low Hamming weight exponents, the Make-sequence algorithm needs only to create an addition chain sequence to a smaller value.

The methods analyzed all fall, on average, 10-20% short of Schnöhage's lower bound. The normalized addition chain length indicates that as the exponent size increases, the Schnöhage limit tends to one. This is expected as the $\log_2 e$ term of Eq. 5.7 becomes dominant. This is directly related to the number of squarings that occur in the modular exponentiation. The 10-20% difference of the exponentiation methods is attributed to the number of multiplications and precomputations. For low Hamming weight exponents, the required number of precomputations and multiplications is reduced due to greater amount of zeros in the exponent.

University of Pretoria etd – Joseph, G (2005)

The effect of each of the exponentiation methods on the number of modular multiplications can be attributed to the following:

- The length of the exponent approximately determines the number of squarings that occur.
- The length of the most significant window is the only factor that influences the number of squarings.
- The number of nonzero windows determined the number of loop multiplications that occur.
- The precomputations are determined by the window size and its respective value, i.e. the length of the addition sequence required to compute the different window values.

Whilst the Binary method is generally the slowest method, in special cases this method can provide shorter or at least comparable addition chains, especially for low Hamming weight exponents. The two additional advantages the Binary method has over its exponentiation counterparts is that it is most simple algorithm to implement and secondly it makes use of the least memory resources as it requires no precomputations.

It is also interesting to note that for a chosen exponent $e = 65537$, the Binary method would provide the shortest addition chain. This choice for the exponent is popular for the public exponent of the RSA encryption scheme, since the computation is relatively efficient and large enough to make trivial attacks infeasible.

The window methods introduce an additional step whereby the exponent is decomposed into windows of a certain bit length. It may be thought of taking k-bit windows in the binary representation of the exponent, calculating the powers in the windows one by one, squaring them k times to shift them over, and then multiplying by the power in the next window. The window decomposition, however, requires a precomputation of all the possible window values that may occur. The techniques for defining windows have a great effect on the modular multiplication count.

The basic window method, i.e. the K-ary method, decomposes the exponent into fixed k-bit

University of Pretoria etd – Joseph, G (2005)

windows. This decomposition creates window values in the range of $0 \leq w_i \leq 2^k - 1$, which would require $2^k - 1$ precomputations.

To further reduce the number of precomputations, the window decomposition can be performed such that only odd window values can exist, i.e. creating windows where the LSB is one. Also by decomposing the exponent in such a way to decrease the number of nonzero windows, the number of multiplications are also decreased. The CLNW and VLNW window decomposition techniques decompose the exponent into zero windows of any length and nonzero windows of a fixed of maximum bit-length. It was shown in Section 5.6.2 that the CLNW decomposition was a more efficient window decomposition technique. The K-ary method is 8% better the Binary method, whilst the Sliding window methods are 5-7% better than the K-ary method. The sliding window algorithms are easy to program, introducing negligible overhead.

The Addition-chain method introduces an additional step which tries to reduce the number of precomputations that occur by creating an addition chain sequence that from the decomposed window values. This step, due to Bos and Coster [91], creates a sequence starting from the largest window value, then finding the next smallest value by either halving its value or finding whether two previous window values' sum adds to the desired value. The process is repeated until the sequence contains all the required values. However, this sequence-building step becomes inefficient to find short sequences at larger integers, hence for a random distribution of ones in the exponent it does not improve on the sliding-window methods.

Moreover, since the sliding window method is effectively an ”on-the-fly” window method, the use of the Addition chain method may not be a necessity for evenly distributed exponents. The addition chain method provides the shortest addition chain lengths for exponents with very low or very large Hamming weights, as the sequence-building algorithm provides ”shortish” addition sequences for these window values.

The sequence building step is complex and time-consuming to compute. However, this step would not effect the performance of a fixed-exponent exponentiation algorithms where the precalculations are done before the actual modular exponentiation. Though it

University of Pretoria etd – Joseph, G (2005)

was shown that the Addition chain method was the better method to utilize for exponents of low and large Hamming weight exponents, considering that Addition chain method are on average 1-10% better than other methods, it seems perfectly reasonable to use the other methods to its simplicity.

The chapter results show that the addition chain length is dependent on the method chosen and the method is which the exponent is decomposed. Certain methods are optimal for certain exponents, hence it is unlikely that will be single method to generate the shortest addition chain.

5.11 CHAPTER SUMMARY

Different methods for modular exponentiation were examined, implemented and evaluated in this chapter. These methods are compared with respect to the average number of modular multiplications needed to accomplish exponentiation for various Hamming weight exponents. The main factors influencing the modular multiplication count have been stated. The Addition-chain method discussed provides the the best or comparable short addition chain lengths for different Hamming weight exponents. However, its complex sequence-building step is inefficient in identifying short sequences for evenly distributed exponents. The CLNW Sliding window method provides similar performances to the Addition chain method and should be used in these cases. Fig. 5.12 confirms the research obtained in this chapter.

University of Pretoria etd – Joseph, G (2005)

CHAPTER SIX
FAST EXPONENT TECHNIQUES

一寸光陰一寸金,寸金難買寸光陰.

A CHINESE PROVERB[1]

The techniques described thus far are applicable to all the public-key cryptosystems discussed in Chapter 2. Further speed enhancements can be made by modifying or manipulating the exponent of the modular exponentiation. This, however, only works in cryptosystems where the exponent is fixed, i.e. the RSA cryptosystem.

The RSA encryption is a very fast operation, as the encryption exponent (e) is often chosen to be a small prime with a low Hamming weight (typically $e = 65537$). However, the decryption procedure is very slow, due to the fact the decryption exponent is generally a very large integer. This fact remains a problem in many applications of the RSA algorithm.

Constructive work in this area of cryptography has provided some significant speed enhancements to the decryption process, most notably the use of the Chinese Remainder Theorem (CRT) in the decryption process by Quisquater *et al.* [23]. However, the applicability of choosing a suitable decryption exponent will further enhance the speed of the RSA decryption. This chapter will look at three novel ways of choosing the decryption exponent (with specific prime generation techniques) that will lead to a substantial reduction in the RSA decryption time.

[1] Translated in English: "An inch of time is worth an inch of gold, but it's hard to buy one inch of time with one inch of gold"

University of Pretoria etd – Joseph, G (2005)

The chapter will implement and evaluate the various improvements performed on the exponent. It will also provide a security analysis of selecting a certain decryption exponent and conclude with a summary of the work done.

6.1 RSA DECRYPTION

The RSA algorithm generates two distinct large primes p and q to create the modulus m i.e. $m = pq$. Utilizing Euler's totient function of m, $\phi(m) = (p-1)(q-1)$, the encryption exponent e is then chosen such that

$$\gcd(e, \phi(m)) = 1 \tag{6.1}$$

The decryption exponent d is computed using the extended Euclidean algorithm [18]:

$$d = e^{-1} \bmod (\phi(m)) \tag{6.2}$$

where d and m are relatively prime [25]. The decryption of the message M is computed as follows:

$$M = C^d \bmod m \tag{6.3}$$

where C is the ciphertext generated from $C = M^e \bmod m$. The correctness of Eq. 6.3 is shown in [50]. In order to compute the number of operations required by Eq. 6.3, let the size of p and q be $k/2$-bits. Since $m = pq$, then m and d are k-bit integers. Thus, the required number of operations required by Eq. 6.3 is calculated as

$$\frac{3k}{2}(k)^2 = \frac{3k^3}{2}$$

6.2 FAST DECRYPTION USING CRT

When the modulus m is the product of two primes p and q, a significant performance improvement can be achieved through the using the Chinese Remainder Theorem (CRT). This method, proposed by Quisquater and Convreur [23], only works when p and q are factors of the modulus m. The CRT enables the computation of the modular exponentiation

University of Pretoria etd – Joseph, G (2005)

modulo m to be performed using two modular exponentiations modulo p and q, which is half the size of m:

$$\begin{aligned} M_p &= C^d \bmod p \\ M_q &= C^d \bmod q \end{aligned} \tag{6.4}$$

After Eq. 6.4 is computed, the message M is computed by the application of the Chinese Remainder Theorem. There are two algorithms that computes the CRT: Gauss' CRT [18] and Garner's CRT (GCRT) [104]. The following subsection discusses the Chinese remainder theorem and its properties.

6.2.1 The Chinese Remainder Theorem (CRT)

If the integers $p_1, p_2, ..., p_k$ are pairwise relatively prime (that is $\gcd(p_i, p_j) = 1$), then the system of simultaneous congruences

$$\begin{aligned} x &\equiv u_1 \bmod p_1 \\ x &\equiv u_2 \bmod p_2 \\ &\vdots \\ x &\equiv u_k \bmod p_k \end{aligned} \tag{6.5}$$

has a unique solution modulo $p = p_1 p_2 ... p_k$ [25]. Using Gauss's algorithm, the solution u to the simultaneous congruences in Eq. 6.5 may be computed as

$$\sum_{i=1}^{k} u_i c_i P_i (\bmod p) \tag{6.6}$$

where $P_i = p_1 p_2 ... p_{i-1} p_{i+1} ... p_k = \frac{p}{p_i}$ and c_i is the multiplicative inverse of P_i modulo p_i, i.e. $c_i = P_i^{-1} \bmod p_i$. Applying Fermat's theorem [105] to RSA decryption, the computation of Eq. 6.4 becomes

$$\begin{aligned} M_p &= C^{d_p} \bmod p \\ M_q &= C^{d_q} \bmod q \end{aligned} \tag{6.7}$$

where

$$\begin{aligned} d_p &= d \bmod (p-1) \\ d_q &= d \bmod (q-1) \end{aligned} \tag{6.8}$$

d_p and d_q are half the size of d, which reduces the time required by the decryption process. In order to obtain M, utilizing Eq. 6.6 and Eq. 6.7 the following is obtained

$$M = M_p c_p \frac{pq}{p} + M_q c_q \frac{pq}{q} \bmod m = M_p c_p q + M_q c_q p \bmod m \tag{6.9}$$

where $c_p = q^{-1} \bmod p$ and $c_q = p^{-1} \bmod q$. This simplified to

$$M = M_p \cdot (q^{-1} \bmod p) \cdot q + M_q \cdot (p^{-1} \bmod q) \cdot p \pmod{m} \tag{6.10}$$

Garner's CRT algorithm (GCRT) [104], on the other hand, computes the final integer u by first computing a triangular table of values:

$$\begin{array}{lllll} u_{11} & & & & \\ u_{21} & u_{22} & & & \\ u_{31} & u_{32} & u_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ u_{k1} & u_{k2} & u_{k3} & \cdots & u_{kk} \end{array}$$

where the first column of the values u_{i1} are the given values of u_i. The values in the remaining columns are computed sequentially using the values from the previous columns using recursion

$$u_{i,j+1} = (u_{ij} - u_{jj})c_{ji} \bmod p \tag{6.11}$$

where c_{ij} is the multiplicative inverse of p_j modulo p_i, i.e.

$$c_{ji}p_j = 1 \bmod p_i \tag{6.12}$$

For example u_{32} is computed as $u_{32} = (u_{31} - u_{11})c_{13} \bmod p_3$ where $c_{13}p_1 = 1 \bmod p_3$. The final value of u is computed as

$$u = u_{11} + u_{22}p_1 + u_{33}p_1p_2 + ... + u_{kk}p_1p_2...p_k$$

which does not require a final modulo p reduction [50].

University of Pretoria etd – Joseph, G (2005)

Applying the GCRT algorithm to the RSA decryption, first compute Eq. 6.7 and Eq. 6.8. The triangular table becomes

$$\begin{array}{ll} M_{11} & \\ M_{21} & M_{22} \end{array}$$

where

$$\begin{aligned} M_{11} &= M_1 \\ M_{21} &= M_2 \\ M_{22} &= (M_{21} - M_{11})(p^{-1} \bmod q) \bmod q \end{aligned} \tag{6.13}$$

Therefore the RSA decryption using GCRT is computed as

$$M = M_1 + \Big[(M_2 - M_1) \cdot (p^{-1} \bmod q) \bmod q\Big] \cdot p \tag{6.14}$$

The GCRT method is more advantageous than the standard CRT computation for RSA decryption. This is due to two reasons:

- The GCRT method requires a single inverse computation $p^{-1} \bmod q$, which can be precomputed and saved.
- The GCRT method does not require a final reduction by p.

6.2.2 Computational Efficiency

In order to compute the total number of operations required by the RSA decryption $(M = C^d \bmod m)$ using the CRT, the following assumptions are made:

- The size of the primes p and q is $k/2$-bits respectively, where $m = pq$.
- The modulus m and the decryption exponent d are k-bits in length.
- d_p, d_q and $p^{-1} \bmod q$ are precomputed.

To construct $C^d \bmod (pq)$ by GCRT, initially requires computation of Eq. 6.7 and Eq. 6.8. Hence, each exponentiation (i.e. M_1 and M_2) requires $\frac{3}{2}k \cdot frack2)$ operations.

University of Pretoria etd – Joseph, G (2005)

The total computation of M (including the combination of M_1 and M_2 by GCRT) will consist of one $k/2$-bit subtraction, two $k/2$-bit multiplications and one k-bit addition. Hence, the total number of operations is

$$2\frac{3k}{2}\left(\frac{k}{2}\right)^2 + 2\left(\frac{k}{2}\right)^2 + \frac{k}{2} + k = \frac{3k^3}{8} + \frac{k^2+3k}{2}$$

The RSA decryption without CRT requires $\frac{3k^3}{2}$ operations. Thus, just considering the higher-order terms, the decryption using CRT will be approximately four times faster.

A comparison of the relative performance in Fig. 6.1 shows that the respective times of performing an RSA decryption with CRT will improve the decryption speed as theoretically expected. Fig. 6.1 was implemented utilizing the algorithmic and programming conditions shown in Section 6.7.

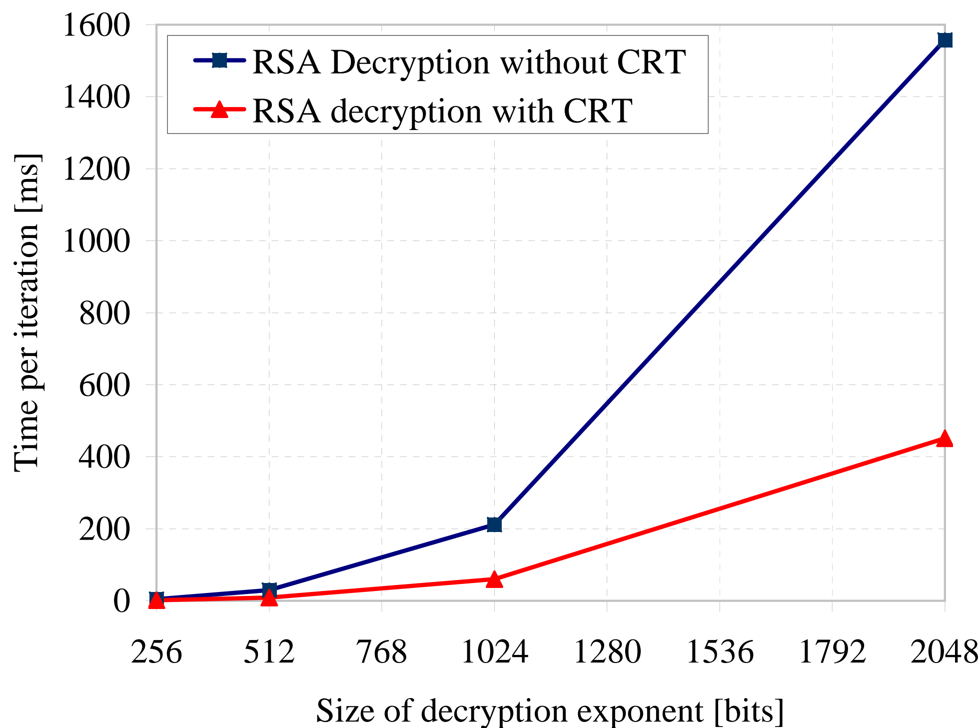


Figure 6.1: Comparison of RSA decryption with and without CRT

University of Pretoria etd – Joseph, G (2005)

6.3 FAST DECRYPTION BY CHOOSING THE DECRYPTION EXPONENT (METHOD I)

Conventional RSA prime generation chooses the encryption exponent and then utilizes the extended Euclidean algorithm to compute the decryption exponent. In general, the encryption operation is very fast, since the publicly-known encryption exponent can be chosen to be a small integer with low Hamming weight (a popular choice is $2^{16} + 1$). However, the decryption is very slow, even with the application of the Chinese Remainder Theorem, due to the fact that the decryption exponent is very large.

In this chapter three novel techniques are introduced where instead of choosing the encryption exponent, the decryption exponent is chosen. In this section, the first of the new methods for choosing the decryption exponent is presented:

Let p and q be two distinct primes where $p < q$. The difference r is represented as

$$r = q - p \tag{6.15}$$

The decryption exponent is chosen as follows

$$d \equiv (p^2 \cdot q) \bmod \phi(m) \tag{6.16}$$

where $\phi(m) = (p-1)(q-1)$. The encryption exponent e is then computed as the inverse of the decryption exponent (using the extended Euclidean algorithm [18]):

$$e \equiv d^{-1} \bmod \phi(m) \tag{6.17}$$

Note that, in general, the size of e will be approximately the same as the size of the modulus m. Reformulating Eq. 6.8, the decryption exponents for the half exponentiations are expressed as:

University of Pretoria etd – Joseph, G (2005)

$$\begin{aligned} d_p &= d \bmod (p-1) \\ &= p^2 \cdot q \bmod (p-1)(q-1) \bmod (p-1) \\ &= q \bmod (p-1) \\ &= (r+p) \bmod (p-1) \\ &= (r+p+1-1) \bmod (p-1) \\ &= (r+1) \bmod (p-1) \end{aligned} \tag{6.18}$$

and similarly

$$\begin{aligned} d_q &= d \bmod (q-1) \\ &= p^2 \cdot q \bmod (p-1)(q-1) \bmod (q-1) \\ &= p^2 \bmod (q-1) \\ &= (q-r)^2 \bmod (q-1) \\ &= (q+1-1-r)^2 \bmod (q-1) \\ &= (1-r)^2 \bmod (q-1) \\ &= (r-1)^2 \bmod (q-1) \end{aligned} \tag{6.19}$$

With the above formulations, one can compute

$$\begin{aligned} C^d \bmod p &= C^{d_p} \bmod p \\ &= C^{(r+1) \bmod (p-1)} \bmod p \\ &= C^{(r+1)} \bmod p \end{aligned} \tag{6.20}$$

and similarly

$$\begin{aligned} C^d \bmod q &= C^{d_q} \bmod q \\ &= C^{(r-1)^2 \bmod (q-1)} \bmod q \\ &= C^{(r-1)^2} \bmod q \end{aligned} \tag{6.21}$$

The decrypted message is finally obtained by applying the Chinese Remainder Theorem to Eq. 6.20 and Eq. 6.21. Note that since $r+1$ is smaller than p, Eq. 6.18 can be expressed as $d_p \equiv r+1$.

University of Pretoria etd – Joseph, G (2005)

Analyzing the Eq. 6.20 and Eq. 6.21, the speed of the decryption lies in the size of decryption exponent, i.e. $r-1$ and $r+1$. By choosing the difference r between the two primes p and q to be small, it is possible to obtain a very short exponent, which would lead to a substantial reduction in decryption time.

Note that if the Hamming weight of $r+1$ is chosen to be low, assuming r is approximately the same size as p and q, the modular exponentiations in Eq. 6.20 and Eq. 6.21 can be improved. Section 6.6 will describe these methods whereby the difference of the primes is chosen to either have a low Hamming weight or a small numerical difference.

6.4 FAST DECRYPTION BY CHOOSING THE DECRYPTION EXPONENT (METHOD II)

The second method proposed is similar to the method proposed Section 6.3. However, in this case the sum r is defined as:

$$r = p + q \tag{6.22}$$

The decryption exponent is chosen to be

$$d \equiv (p \cdot q) \bmod \phi(m) \equiv m \bmod \phi(m) \tag{6.23}$$

where $\phi(m) = (p-1)(q-1)$. The encryption exponent e is then computed using the extended Euclidean algorithm $e \equiv d^{-1} \bmod \phi(m)$. Utilizing Eq. 6.8 for this case, the decryption exponents for the half exponentiations is then calculated as:

$$\begin{aligned} d_p &= d \bmod (p-1) \\ &= p \cdot q \bmod (p-1)\,(q-1) \bmod (p-1) \\ &= q \bmod (p-1) \\ &= (r-p) \bmod (p-1) \\ &= (r-p-1+1) \bmod (p-1) \\ &= (r-1) \bmod (p-1) \\ &\equiv r-1 \end{aligned} \tag{6.24}$$

University of Pretoria etd – Joseph, G (2005)

and similarly

$$\begin{aligned} d_q &= d \bmod (q-1) \\ &= p \cdot q \bmod (p-1)(q-1) \bmod (q-1) \\ &= p \bmod (q-1) \\ &= (r-q) \bmod (q-1) \\ &= (r-q-1+1) \bmod (q-1) \\ &= (r-1) \bmod (q-1) \\ &\equiv r-1 \end{aligned} \tag{6.25}$$

With the above formulations, one can compute

$$\begin{aligned} C^d \bmod p &= C^{d_p} \bmod p \\ &= C^{(r-1)} \bmod p \end{aligned} \tag{6.26}$$

and similarly

$$\begin{aligned} C^d \bmod q &= C^{d_q} \bmod q \\ &= C^{(r-1)} \bmod q \end{aligned} \tag{6.27}$$

The Chinese remainder theorem is applied to Eq. 6.26 and Eq. 6.27 to obtain the message M. Since r is the sum of the two primes, it would be favorable to choose $r-1$ with a low Hamming weight exponent in order to reduce the decryption time. A method to create r as a low Hamming weight integer is described in Section 6.6.

6.5 FAST DECRYPTION BY CHOOSING THE DECRYPTION EXPONENT (METHOD III)

The third method of choosing the decryption exponent is to compute d as the sum of p and an integer multiple of q. In this section, the third of the new methods for choosing the decryption exponent is presented.

Let p and q be two distinct primes where $p < q$. The difference in primes r is represented as

$$r = q - p \tag{6.28}$$

University of Pretoria etd – Joseph, G (2005)

Now choose the decryption exponent as follows

$$d \equiv (nq + r) \bmod \phi(m) \tag{6.29}$$

where $\phi(m) = (p-1)(q-1)$, where n is the odd multiple of q. Hence, the half exponentiations are expressed as:

$$\begin{aligned} d_p &= d \bmod (p-1) \\ &= (nq + r) \bmod (p-1)(q-1) \bmod (p-1) \\ &= n(p+r) + r \bmod (p-1) \\ &= n(p+1-1+r) + r \bmod (p-1) \\ &= r(n+1) + n \bmod (p-1) \end{aligned} \tag{6.30}$$

and similarly

$$\begin{aligned} d_q &= d \bmod (q-1) \\ &= (nq + r) \bmod (p-1)(q-1) \bmod (q-1) \\ &= n(q+1-1) + r \bmod (q-1) \\ &= (r+n) \bmod (q-1) \end{aligned} \tag{6.31}$$

With the above formulations, one can compute

$$\begin{aligned} C^d \bmod p &= C^{d_p} \bmod p \\ &= C^{(r(n+1)+n) \bmod (p-1)} \bmod p \\ &= C^{r(n+1)+n} \bmod p \end{aligned} \tag{6.32}$$

and similarly

$$\begin{aligned} C^d \bmod q &= C^{d_q} \bmod q \\ &= C^{r+n \bmod (q-1)} \bmod q \\ &= C^{r+n} \bmod q \end{aligned} \tag{6.33}$$

The message M is obtained applying the Chinese remainder theorem to Eq. 6.32 and Eq. 6.33. In order to satisfy d, where $d \equiv nq + r$, n has to be odd. This is due to the fact that r, the difference of primes p and q, will always be even. Hence in order for the condition $\gcd(e, \phi(m))$ to be satisfied, d and n has to be odd. The generation of the difference r is shown in Section 6.6.

University of Pretoria etd – Joseph, G (2005)

6.6 PRIME GENERATION

The prime generation techniques focuses on creation of r from the generated primes p and q. Three methods, derived for the proposed methods shown in Section 6.3 to Section 6.5, are described in the following subsections.

6.6.1 Low Hamming Weight Prime Difference

In order to create r to have a low Hamming weight, where r is the difference between primes p and q, the following steps are taken:

- Create r such that the MSB is always one and the LSB is zero with a bit-length approximately the same size as p. Note that r will always be even since p and q are primes.
- Randomly disperse $H(r)$ ones through the length of r, where $H(r)$ is the desired Hamming weight of r.
- Generate prime p and add it to r to obtain q.
- If q is not a prime, the process has to repeated until a prime q is found.

The algorithm for the above method is given as follows:

ALGORITHM: PRIME GENERATION - LOW HAMMING WEIGHT PRIME DIFFERENCE

Input. Random integer $p = (p_k, p_{k-1}, ..., p_1, p_0)_2$ of k-bits length

Output. Prime p and q with difference r

1. *Create prime* p.
   1.1 If p is even: $p \leftarrow p + 1$.
   1.2 While p is not prime: $p \leftarrow p + 2$.
2. *Create prime* q. While q is not prime do the following:
   2.1 Initialize $r \leftarrow 0$
   2.2 *Create difference* r. For i from 1 until $i \leq (k-1)$ do the following:
      2.2.1 $j = R(l)$.
      2.2.2 $r_{j+i} \leftarrow 1$ where r_{j+i} is the $(i+j)$th binary position in r
      2.2.3 $i \leftarrow i + j$.
   2.3 Compute $q \leftarrow p + r$.
3. Return p, q and r.

University of Pretoria etd – Joseph, G (2005)

The algorithm employs $R(l)$ which is a random number generator to insert approximately k/l binary ones into r, hence determine the Hamming weight of r. A low Hamming weight r is favorable since it would imply that p and q may be Hamming weight close but not necessarily numerically close.

The visualization of the algorithm is shown in Fig. 6.2.

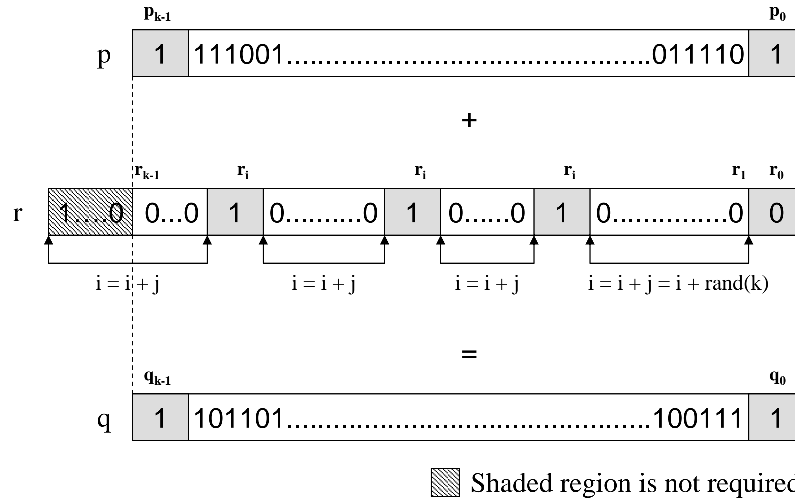


Figure 6.2: Creation of a low Hamming weight prime difference r

6.6.2 Small Prime Difference

To create the small difference between p and q: they need to numerically close. Their Hamming weight will be identical except that for their least significant l bits, in which they can differ significantly.

The method first creates a prime p and then utilizing a random generator $R(r)$ creates r that is l bits long. Prime q is then computed by the sum of p and r. If q is not prime, then the process is repeated until a prime q is found.

The algorithm for the method is given as follows:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: PRIME GENERATION - SMALL PRIME DIFFERENCE

Input. Random integer $p = (p_k, p_{k-1}, ..., p_1, p_0)_2$ of k-bits length

Output. Prime p and q with difference r

1. *Create prime* p.
   1.1 If p is even: $p \leftarrow p + 1$.
   1.2 While p is not prime: $p \leftarrow p + 2$.
2. *Create prime* q. While q is not prime do the following:
   2.1 *Create* r. $r \leftarrow R(r)$ of l-bits length.
   2.2 *Ensure* r *is even*. If $r_0 = 1$ then set $r_0 \leftarrow 0$.
   2.3 *Compute* q. $q = p + r$.
3. Return p, q and r.

The visualization of the algorithm is shown in Fig. 6.3.

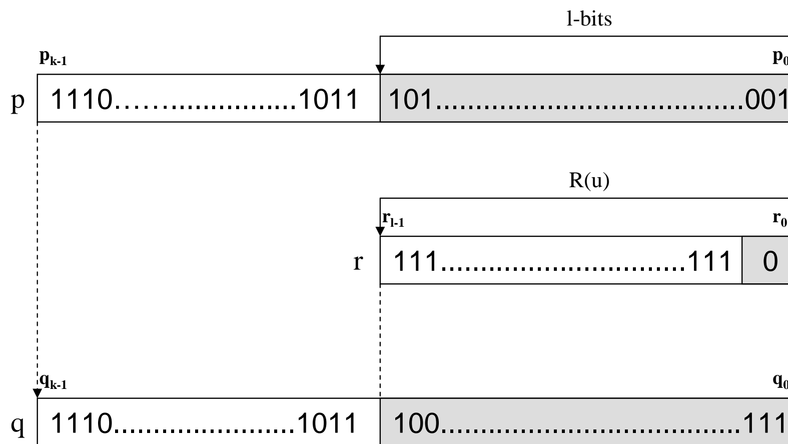


Figure 6.3: Creation of a small difference r

6.6.3 Low Hamming Weight Prime Sum

Given that $p + q = r$, it can be noticed that the sum r is larger than p and q. However, since Eq. 6.26 and Eq. 6.24 requires $(r - 1)$ to have a low Hamming weight, the method creates $(r - 1)$ such that the MSB is always one and the LSB is zero. The bit-length of r is always one bit larger than the size of p and is randomly populated with $H(r)$ ones, where $H(r)$ is the desired Hamming weight of r. The generated prime p is then subtracted from r to obtain q. If q is not a prime, the process has to repeated until prime q is found. Hence, the following algorithm can be formulated:

University of Pretoria etd – Joseph, G (2005)

ALGORITHM: PRIME GENERATION - LOW HAMMING WEIGHT PRIME SUM

Input. Random integer $p = (p_k, p_{k-1}, ..., p_1, p_0)_2$ of k-bits length

Output. Prime p and q with sum r

1. *Create prime* p.
   1.1 If p is even: $p \leftarrow p + 1$.
   1.2 While p is not prime: $p \leftarrow p + 2$.
2. *Create prime* q. While q is not prime do the following:
   2.1 Initialize $r \leftarrow 0$ with $r_k \leftarrow 1$
   2.2 *Create* $(r - 1)$. For i from $(k - 1)$ down-to 0 do the following:
      2.2.1 $j = R(l)$.
      2.2.2 $r_{i-j} \leftarrow 1$ where r_{i-j} is the $(i - j)$th binary position in r
      2.2.3 $i \leftarrow i - j$.
   2.3 *Set* $(r - 1)$ *to be low Hamming weight*. $r_0 \leftarrow 1$.
   2.4 *Compute* q. Compute $q \leftarrow r + 1 - p$.
3. Return p, q and $r \leftarrow r + 1$.

The algorithm employs $R(l)$ which is a random number generator to insert k/l binary ones into r, hence determines the Hamming weight of r. The algorithm can be visualized in Fig. 6.4.

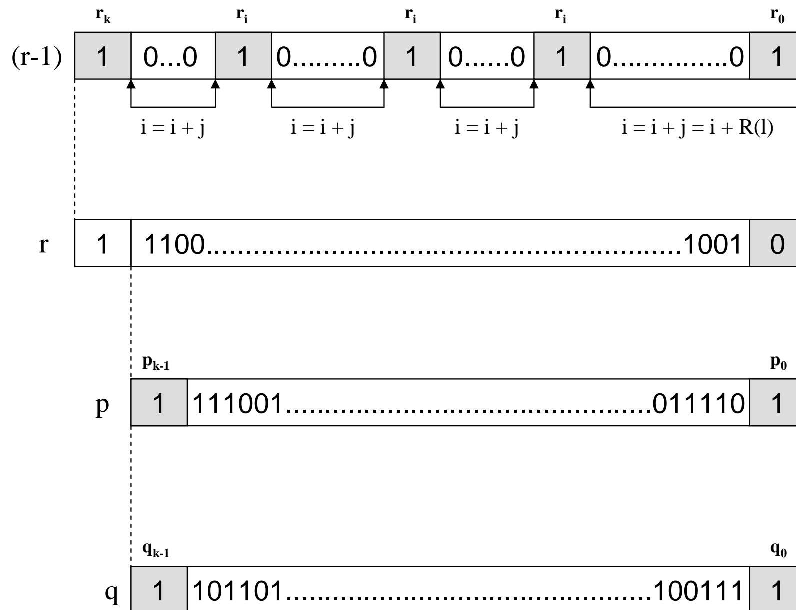


Figure 6.4: Creation of a low Hamming weight prime sum r

University of Pretoria etd – Joseph, G (2005)

6.7 EXPERIMENTAL RESULTS

In order to obtain practical times for the discussed decryption methods, specific simulations must be performed. Simulations were performed on a Pentium III processor running at 550 MHz with 256 Mbyte main memory under Windows XP Home Edition platform using a Borland C Builder 6.0 compiler. The simulations were performed under the following conditions:

Decryption algorithms tested:

- The RSA decryption algorithm utilizing the Chinese remainder theorem, proposed by Quisquater and Convreur [23].
- The proposed RSA decryption method choosing $d = p^2q \bmod \phi(m)$ utilizing the low Hamming weight prime difference shown in Section 6.6.1.
- The proposed RSA decryption method choosing $d = p^2q \bmod \phi(m)$ utilizing the small prime difference shown in Section 6.6.2.
- The proposed RSA decryption method choosing $d = m \bmod \phi(m)$ utilizing the low Hamming weight prime sum shown in Section 6.6.3.
- The proposed RSA decryption method choosing $d = 3q + r \bmod \phi(m)$ utilizing the small prime difference shown in Section 6.6.2.

Algorithm basis:

- The exponentiation algorithms utilized the Sliding window method(utilizing VLNW decomposition).
- Each exponentiation algorithm was configured as to utilize its optimal window-size for the respective exponent bit-length.
- The multiplication method utilized in the simulations was the Karatsuba-Ofman with Comba method
- The reduction method utilized in the simulations was the Montgomery method.

University of Pretoria etd – Joseph, G (2005)

Programming conditions:

- Each algorithm was implemented using standard ANSI C coding. The base B was chosen as 2^{32}, hence used basic operations on integers of **unsigned int** type.
- The message M was a randomly generated 2048 bit integer which was encrypted by the respective public exponent utilized.
- In addition, the exponent was created by the prime generation method implemented for the test case.
- For verification that decryption process was successful, the decrypted message was compared to the original message.
- Though a lot of effort has been done to remove the overhead generated by the compiler, the test is still subjected to a little overhead generated by the platform and compiler.

Timing analysis parameters:

- One iteration consisted of a single run of the two decryption algorithms. The total time period of each test was 20 seconds.
- Each simulation was run until the total time period had elapsed and the number of iterations exceeded 20. The average time was calculated as a function of the total time elapsed divided by the total number of iterations.
- The timing of the precomputations were not taken into account, however argument transformations and postcomputations were taken, as they were computed within the modular exponentiation.

Fig. 6.5 to Fig. 6.8 plots the average time against either the Hamming weight or bit-length of r depending on the prime generation technique utilized for the simulation. The dotted line shown in each of the figures is the average time required by a standard RSA decryption using the CRT (with the conventional selection for the public exponent e and the private exponent d).

The RSA encryption time for the new methods had an average timing of $1512.23ms$.

University of Pretoria etd – Joseph, G (2005)

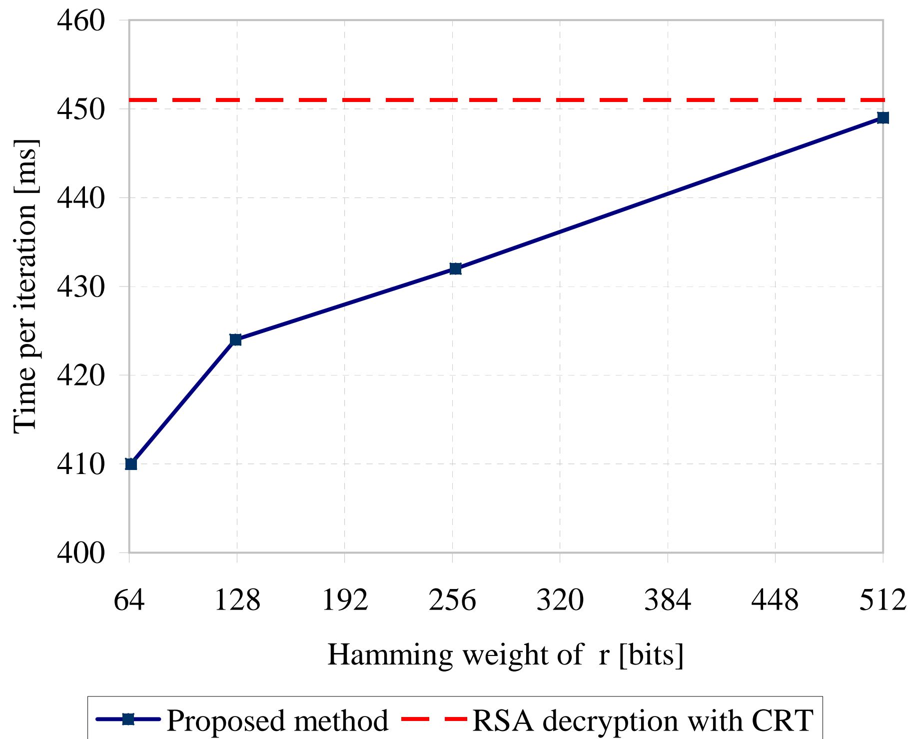


Figure 6.5: Time analysis of Method I shown in Section 6.3 applying prime generation shown in Fig. 6.2 (average RSA encryption time $1512.23ms$)

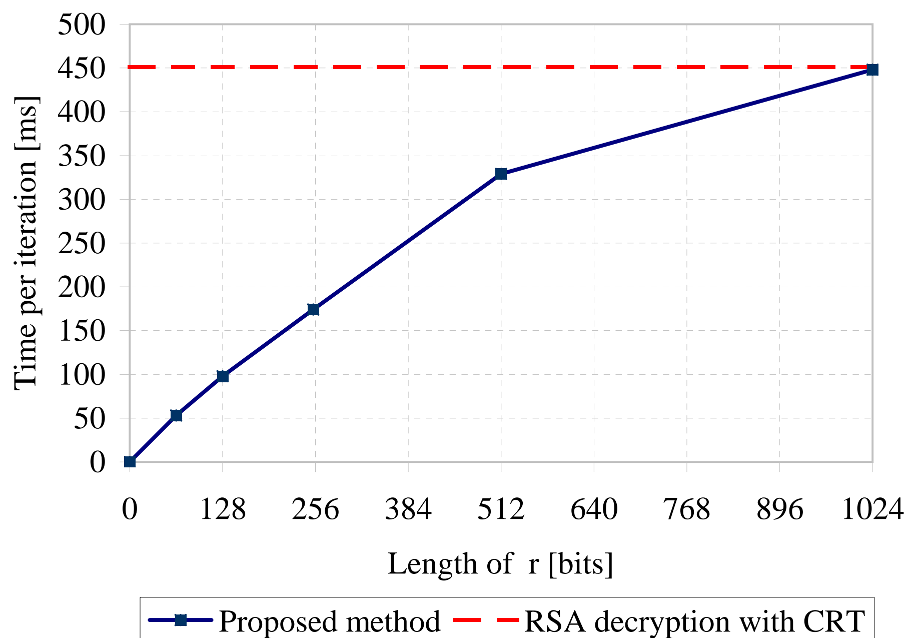


Figure 6.6: Time analysis of Method I shown in Section 6.3 applying prime generation shown in Fig. 6.3 (average RSA encryption time $1512.23ms$)

University of Pretoria etd – Joseph, G (2005)

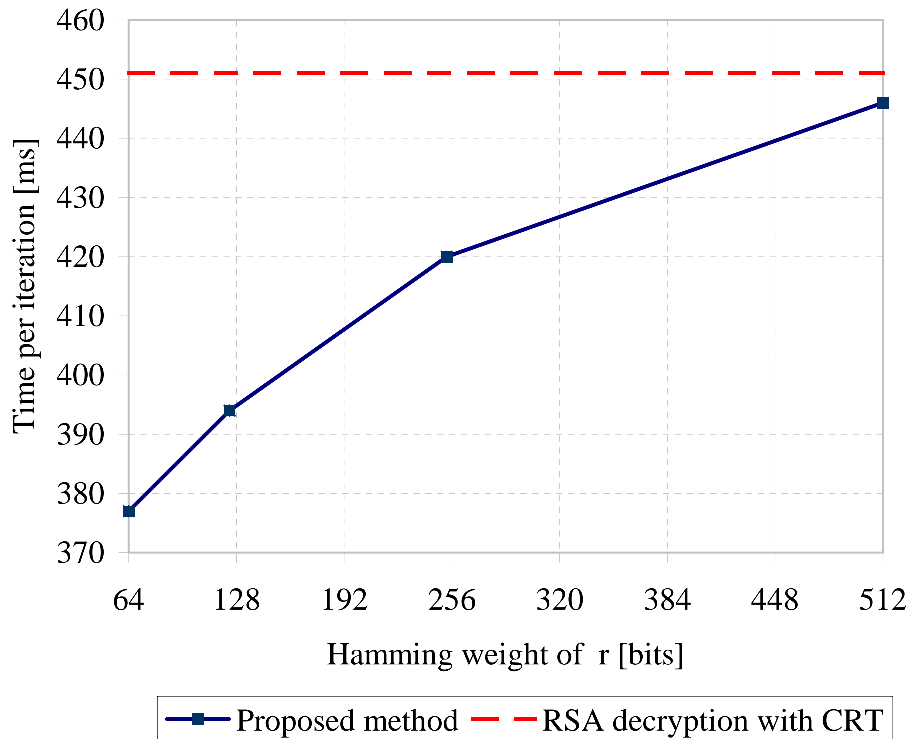


Figure 6.7: Time analysis of Method II shown in Section 6.4 applying prime generation shown in Fig. 6.4 (average RSA encryption time $1512.23ms$)

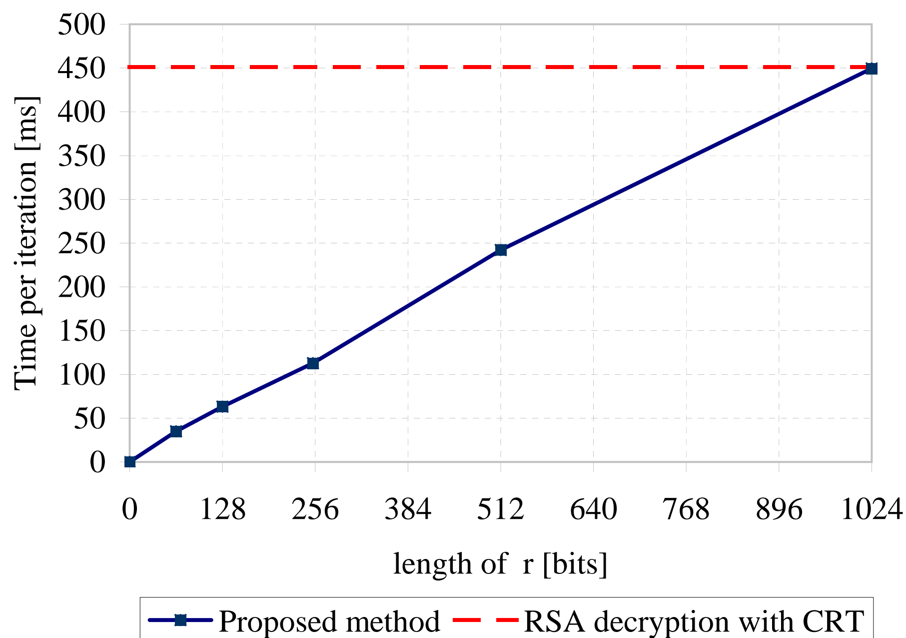


Figure 6.8: Time analysis of Method III shown in Section 6.5 applying prime generation shown in Fig. 6.3 (average RSA encryption time $1512.23ms$)

University of Pretoria etd – Joseph, G (2005)

6.8 DISCUSSION

6.8.1 Performance Analysis

Fig. 6.5 shows the decryption times for the RSA method described in Section 6.3 by applying the low Hamming weight prime generation shown in Section 6.6.1. Method I, when utilizing a low Hamming weight r, provides a maximum improvement of 10% against the RSA decryption (as the Hamming weight of r increases, the improvement of the method decreases).

The low Hamming weight $(r-1)$ is only advantageous for one of the half exponentiations, i.e. $M_p = C^{r-1} \bmod p$. The exponent in the second half exponentiation, i.e. $(r-1)^2 \bmod (q-1)$, will not necessarily produce a low Hamming weight exponent. Thus, the speed improvement of Method I relies on the computation of M_p.

Fig. 6.6 shows the timing results for Method I if the prime generation method created primes with a small difference between them (shown in Section 6.6.2. The times shown in Fig. 6.6 show a greater improvement than the times shown in Fig. 6.5. By creating a small difference between the primes, the decryption exponent utilized in the half exponentiations in Fig. 6.6 varied from 64-bits to 1024-bits whilst the exponent in Fig. 6.5 exponent was of a constant 1024-bit length.

The timing results utilizing Method II are shown in Fig. 6.7. Comparing Fig. 6.5 and Fig. 6.7, it can be seen Method II is faster than Method I when a low Hamming weight exponent is generated. Utilizing Method I, the derivation of $(r-1)^2 \bmod (q-1)$ did not always achieve a low Hamming weight. However utilizing Method II, the derived exponent for the half exponentiations was $r-1$, which is chosen to be a low Hamming weight integer. Thus, a speed improvement of Method II was obtained on both half exponentiations.

Though Method II is faster than Method I when utilizing low Hamming weight exponents, the security of this system can be easily compromised. This security risk is shown in the Section 6.8.5.

Method III chose the exponent such that $d \equiv nq + r$ whereby n had to odd. For the

University of Pretoria etd – Joseph, G (2005)

most efficient generation of r, n was chosen as 3. Thus, Fig. 6.8 shows the timing results for Method III whereby $d \equiv 3q + r$. The Method III's derived exponents, i.e. $(r + 3)$ and $4r + 3$, are approximately half the size of exponent $(r - 1)^2$ and similar in bit-length to $r + 1$ that are created by Method I. Thus, the times shown in Fig. 6.8 are better than those obtained in Fig. 6.6 by 25%.

One drawback of the proposed methods is that the encryption speed has decreased due to a much larger public exponent e. Although e is much larger than what is conventionally used (i.e. $e = 65537$), it is assumed the encryption is done by faster processors, whereas the decryption is done on applications with limited resources i.e. smart cards.

Choosing the decryption exponent poses certain security risks. These risks are analyzed in the following subsections.

6.8.2 Simple Factoring Attack on the Modulus

There is a simple algorithm for factoring the modulus when the difference between the two prime factors is small. Consider the following identity:

$$\left(\frac{q+p}{2}\right)^2 - pq = \left(\frac{q-p}{2}\right)^2 \tag{6.34}$$

The term $((q + p)/2)^2$ can be found in a linear search through all the perfect squares, starting from the modulus $n = pq$. The correct square is found when the difference between the square and the modulus is itself a perfect square. Once the two terms $(q + p)/2$ and $(q - p)/2$ have been found, it is easy to factor the modulus.

Suppose the difference r between the two prime factors, i.e. $p + r = q$ is chosen to be a 64-bit number. Then the exhaustive linear search to find the required perfect squares has the computational complexity of $O(2^{64})$.

6.8.3 Wiener's Attack on Short Decryption Exponents

Wiener has previously considered the situation when the decryption exponent for RSA is chosen too "small" [60]. We give a brief review of his attack. One tries to find d, knowing

University of Pretoria etd – Joseph, G (2005)

that

$$de = 1 \bmod (lcm(p-1)(q-1)) \tag{6.35}$$

when e and $n = pq$ are known. Of course, neither p or q are known. Expressing

$$de = 1 + \frac{k}{h}(p-1)(q-1) = 1 + \frac{k}{h}(n-p-q-1) \tag{6.36}$$

where k and h are integers with $\gcd(k, h) = 1$. Now divide dy dn:

$$\frac{k}{hd} - \frac{e}{n} = \frac{k}{hd}\left(\frac{1}{p} + \frac{1}{q} - \frac{1}{n}\right) - \frac{1}{dn} \tag{6.37}$$

If r is small, then p and q are not far from $\sqrt{n}$. If d is too small (of the order of $n^{\frac{1}{4}}$, k, h and d can be recovered from $\frac{k}{hd}$, which is continued fraction approximation of the known number $\frac{e}{n}$. However, in our case, the decryption exponent will not be small, and so Wiener's attack will not be possible.

6.8.4 Fermat and Lehman Attacks

The small difference between p and q is often attacked by factoring algorithms. Namely, the Fermat factoring technique and Lehman attacks may be applied. However, the question remains how small may the difference between the primes be in order to guarantee security against these attacks.

The ANSI X9.31 standard [69] defines a method for digital signature and verification of message using reversible public key cryptosystems with message recovery, i.e. RSA. The standard provides criteria for the generation of public and private keys required for secure use of the algorithm.

The standard states that in order to prevent Fermat factoring and Lehman attacks, the difference between p and q must be larger than 2^{412}. The mathematical details of this can be read in [106] and its applications in [107]. Hence r should have a length of at least 412-bits.

6.8.5 Security Risk of Method II

At first glance Method II seems to provide secure decryption, however closer examination of the decryption exponent d suggests otherwise. From Eq. 6.22 the decryption exponent d

University of Pretoria etd – Joseph, G (2005)

can be expressed in terms of r as follows:

$$\begin{aligned} d &\equiv m \bmod \phi(m) \\ &\equiv p \cdot q \bmod (p-1)(q-1) \\ &\equiv p \cdot (q-1+1) \bmod (p-1)(q-1) \\ &\equiv p \bmod (q-1) \\ &\equiv (r-q) \bmod (q-1) \\ &\equiv r-1 \end{aligned} \tag{6.38}$$

In order to compute $M = C^d \bmod m$ where C, m and $d \equiv r-1$ is known, it becomes very easy for an intruder to compute d. Since $(r-1)$ is chosen to have a low Hamming weight (l), the total number of possibilities that d may be for a k-bit number is

$$\binom{k}{l} = \frac{k!}{l!(k-l)!} \tag{6.39}$$

From Eq. 6.39 it is shown the number of possibilities that d has significantly decreased compared to the possibilities required by a conventional RSA implementation (2^k). Since d is a low Hamming weight integer, it is prone to Hamming weight attacks i.e. Baby-step Giant-step attacks. Hence this method, though faster, is insufficient for security applications.

6.9 CHAPTER SUMMARY

The chapter gives a brief description of the conventional RSA decryption. Utilizing the Chinese Remainder Theorem, the RSA decryption performance is dramatically increased. This chapter describes and implements three novel ways to select the decryption exponent to further improve the decryption speed.

Method I, implemented for low Hamming weight or small exponents, provided an improvement on the conventional RSA decryption utilizing the CRT. Method II, although an improvement on the speed from methods discussed in Section 6.1 and Section 6.2, was insecure to be used in security applications. Method III, however, described in Section 6.5 employing prime generation that created a small r provided exceptional timing results. The

University of Pretoria etd – Joseph, G (2005)

security of the proposed methods were tested against various attacks and can implemented in various security applications.

By carefully choosing the difference between the two primes that form the modulus, it is possible to obtain a decryption exponent d whose size can be substantially reduced. When applying the Chinese Remainder Theorem for the decryption algorithm, this results in a dramatic reduction of the decryption time.

University of Pretoria etd – Joseph, G (2005)

CHAPTER SEVEN

CONCLUSION

"Everything that has a beginning has an end."

THE ORACLE, *Matrix Revolutions*

The main objective of this dissertation is to improve the implementation efficiency of widely used modular exponentiation-based public-key cryptosystems. The execution time of the these cryptosystems are based on specific mathematical algorithms. These algorithms have to be optimized for platforms ranging from super-computers to smart cards. It is especially the lower-end platforms that require algorithms to be efficient and consume as little resources as possible. Throughout this dissertation specific attention was paid to the properties of these mathematical algorithms, specifically their performance characteristics.

7.1 ASSESSMENT OF STUDY

In order to demonstrate the improvements recommended by the dissertation on the implementation efficiency of the public-key cryptosystems, a case study must be undertaken whereby these improvements can be compared against an industry-standard multi-precision integer library. MIRACL[1] was chosen as the case study of this dissertation, though other libraries such as GMP, LIP and OpenSSL have multiple precision integer arithmetic routines, due to that fact that all its routines have been thoroughly optimized for speed and efficiency in terms of standard portable C.

[1] The MIRACL library (Multi-precision Integer and Rational Arithmetic C Library) was created by Michael Scott and consists of well over 100 routines that cover all aspects of multi-precision arithmetic required for public-key cryptosystems.

University of Pretoria etd – Joseph, G (2005)

Table 7.1 shows a summary of the list of algorithms utilized by an ”out-of-the-box” built MIRACL to perform a modular exponentiation. This table also illustrates the recommendations of the dissertation for each step introduced in Fig. 1.1.

Table 7.1: Summary of algorithms used by Miracl and Proposed methods

	MIRACL	Proposed
Multiplication	Classical [18]	3 level Karatsuba-Ofman with Comba [19]
Reduction	Montgomery [21, 52]	Montgomery [21, 52]
Exponentiation	5-bit VLNW sliding window $(q^* = 2)$ [22]	5,6,7-bit CLNW sliding window [22]

Table 7.2 shows a summary of the results obtained by MIRACL as well as the recommended optimal results for $512, 1024, 2048$-bit modular exponentiations. The last row of Table 7.2 shows the time required by the respective methods to perform a 2048-bit modular multiplication. The last column of Table 7.2 shows the percentage improvement of the timing results obtained by the proposed method compared to MIRACL.

The results of Table 7.2 were obtained on a Pentium III processor running at 550 MHz with 256 Mbyte main memory under Windows XP Home Edition platform using a Borland C Builder 6.0 compiler. Fig. 7.1 shows the speed improvement of the proposed method compared to MIRACL.

Table 7.2: Summary of results by Miracl and Proposed methods

Exponent (e)	MIRACL		Proposed		Improvement
Bit-size	Time	#mult	Time	#mult	%
512	$444.13ms$	622	$387.58ms$	609	12.73
1024	$827.20ms$	1229	$759.10ms$	1200	8.23
2048	$1729.00ms$	2444	$1496.85ms$	2365	13.43
Mod. mult.	$941.36\mu s$	-	$758.93\mu s$	-	19.37

University of Pretoria etd – Joseph, G (2005)

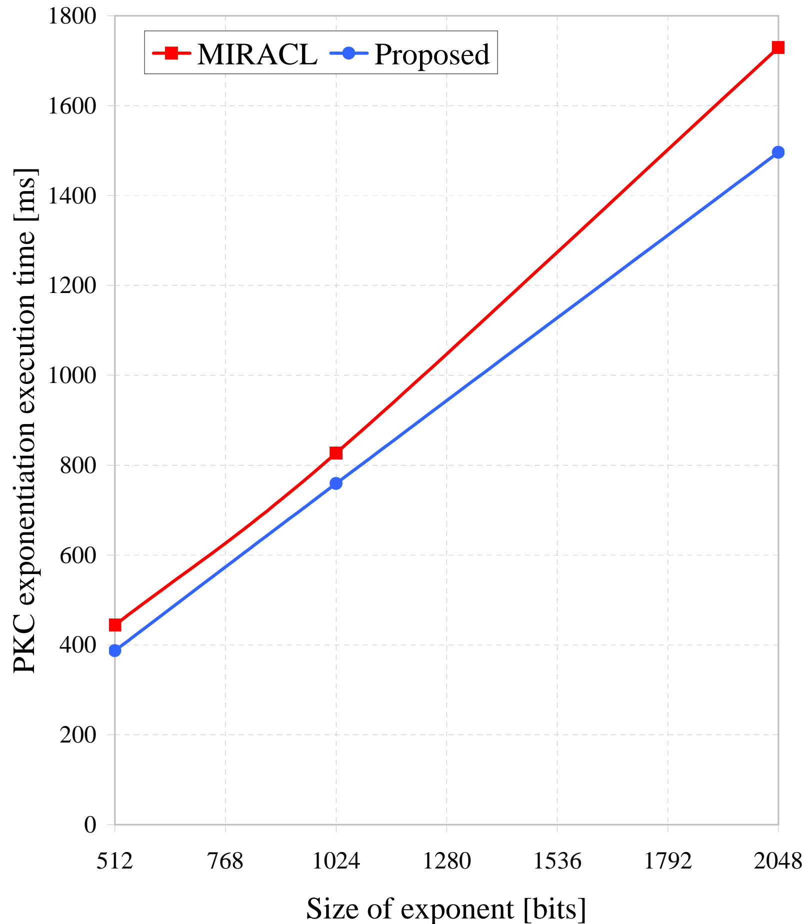


Figure 7.1: MIRACL and Proposed methods results

In general, the execution time of public-key cryptosystems used in industry are based on the MIRACL algorithms shown in Table 7.1. Thus, depending on the processor and memory requirements, the optimized method can provide improvements up to 13%.

From the results shown in Table 7.2 it is evident that multiplication is one of the most important factors influencing performance. Simulation results in Chapter 3 showed that the Classical multiplication algorithm was slow compared to the Karatsuba-Ofman algorithm. The dissertation shows that by applying the Karatsuba-Ofman algorithm recursively to an optimal recursion depth, significant performance gains up to 20% can be achieved. From the simulation results shown in Chapter 4, it has been have verified that the Montgomery reduction algorithm provides superior performance over the Classical and Barrett reduction algorithms.

University of Pretoria etd – Joseph, G (2005)

The speed improvements of the proposed method can be attributed to the fewer number of modular multiplications required by the chosen exponentiation method. An invaluable contribution of our dissertation, shown in chaprefFastExponentiationTechniques, describes the windowing algorithms that utilize different techniques of breaking the exponent structure into unique windows to compute shorter addition chain sequences. These improvements are gained by utilizing two sliding-window algorithms, the CLNW and VLNW techniques, which depends greatly on the exponent, and in particular its weight. The research of these two sliding-window methods, backed up with mathematical and simulation results, showed that the optimal choice of the windowing-strategy is the CLNW method.

An alternative approach was investigated to directly find the shortest addition chain leading to a particular exponent by utilizing a heuristic by Bos *et al.* [91] that approximates the shortest chain to the exponent. This heuristic, when used in conjunction with a windowing-strategy, did not produce much better approximations to the shortest addition chain than the sliding-window exponentiation algorithms. Thus the heuristic, though it promises significant improvements in theory, finds similar addition chains to the sliding-window algorithms. In future significant improvements might still be made here.

The main goal of the dissertation was extended even further in Chapter 6 when three novel approaches were implemented for improving the decryption efficiency of the RSA algorithm. The three novel methods, by carefully choosing the difference between the two primes that form the modulus, obtains a decryption exponent whose size can be substantially reduced. When applying the Chinese Remainder Theorem to the RSA decryption algorithm, a dramatic reduction of the decryption time is obtained. However these improvements are done at the expense of some increase in the encryption complexity.

Security tests have shown that Method II is not secure, and should not be used in practice. The other two methods were tested against various known attacks, and resisted these well. Methods I and III represent a noteworthy contribution to the field of cryptography and provides up to a 45% improvement on the decryption speed for 1024-bit modular exponentiation when ANSI X9-31 security considerations are taken into account.

University of Pretoria etd – Joseph, G (2005)

7.2 SUMMARY AND FURTHER RESEARCH

This research has uncovered some areas where further research is required. It would result in a large improvement in the performance of any exponentiation-based algorithm if the processes of multiplication and reduction can be combined into a single step. An efficient algorithm for calculating the shortest addition chain for a given exponent still has to be found. More work on the security of methods I and III can be carried out. These novel methods represent a whole new family of potential improvements to RSA decryption. More work can be done on exploring these new opportunities, and finding the optimal combination of encryption and decryption efficiency. Design and development of suitable hardware solutions for these mathematical algorithms evaluated in this dissertation is left as a possible research project.

It can be said that the optimal algorithm performance for modular exponentiation-based public-key cryptosystems has not been found yet, but our work should be in itself a contribution to the field of cryptography and serve as a valuable platform for further research.

University of Pretoria etd – Joseph, G (2005)

REFERENCES

[1] P.F. Syverson, "Limitations on design principles for public-key protocols," *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 62–73, 1996.

[2] P.G. Comba, "Exponentiation on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[3] C. Kaufman, R.Perlman and M. Speciner, *Network Security: Private communication in a public world*. Prentice Hall, 1995.

[4] ITU-T Recommendation X.800, "Security Architecture For Open Systems Interconnection For CCITT Applications," *3/91, Geneva*, 1991.

[5] W. Diffie, M.E. Hellman, "New directions in cryptography," *IEEE Transactions on Computers*, vol. IT-22, pp. 644–654, June 1976.

[6] R.L. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *CACM*, vol. 21, pp. 120–126, 1978.

[7] T. ElGamal, "A public-key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, 1985.

[8] FIBS 186 Federal Information Processing Standards Publication 186, "Digital Signature Standard," *U.S. Department of Commerce/N.I.S.T., National Technical Information Service*, 1994.

[9] P. Rogaway, D. Coppersmith, "A software-optimised encryption algorithm," *Proceedings of Fast Software Encryption*, pp. 56–61, December 1993.

[10] T. Azar, *High-Speed Algorithms & Architectures For Number-Theoretic Cryptosystems*. PhD thesis, Oregon State University, 1997.

[11] G.Joseph, W.T. Penzhorn, "Implementation and design of fast multiplication techniques for public-key cryptosystems in smart cards," *SATNAC 2003 Conference at George, South Africa, 7-10 September 2003*, pp. 172–175, 2003.

[12] G. Joseph, W.T. Penzhorn, "High-speed algorithms for public-key cryptosystems," *Proceedings to AFRICON 2004 Conference at Gabarone, Botswana, 16-19 September 2004*, vol. 1, pp. 945–952, 2004.

University of Pretoria etd – Joseph, G (2005)

[13] G. Joseph, W.T. Penzhorn, "High-speed algorithms for public-key cryptosystems in an e-commerce environment," *SATNAC 2004 Conference at Stellenbosch, South Africa, 8-11 September 2004*, pp. 121–126, 2004.

[14] G. Joseph, W.T. Penzhorn, "High-speed Algorithms for the RSA Cryptosystem," *submitted to IEEE Transactions on Computers*, 2005.

[15] G. Joseph, W.T. Penzhorn, "Fast RSA decryption method," *submitted to South African Computer Journal (SACJ)*, 2005.

[16] N. Joubert, "The Design, Implementation and Testing of a Fast Exponentiation Algorithm for RSA," Internal Report, Department of Electrical, Electronic and Computer Engineering, University of Pretoria, November 2003.

[17] W.T. Penzhorn, "Fast decryption Algorithms for the RSA cryptosystem," *Proceedings to AFRICON 2004 Conference at Gabarone, Botswana, 16-19 September 2004*, vol. 1, pp. 361–364, 2004.

[18] D.E. Knuth, *The Art of Computer Programming - Seminumerical Algorithms*, vol. 2. Massachusetts: Addision-Wesley, 2 ed., 1981.

[19] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics - Doklady*, vol. 7, pp. 595–596, 1963.

[20] P.D. Barrett, "Implementing the Rivest Shamir Adleman public-key encryption algorithm on a standard digital signal processor," *Advances in Cryptology - CRYTO '86 (LNCS 263)*, pp. 311–323, 1987.

[21] P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[22] C.K. Koc, "Analysis of Sliding Window Techniques for Exponentiation," *Computers and Mathematics with Applications*, vol. 30, no. 10, pp. 17–24, 1995.

[23] J.J. Quisquater, "A digital signature scheme with extended recovery." preprint, 1995.

[24] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, Second ed., 1996.

[25] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, First ed., 1997.

[26] S.M. Bellovin, M.Merritt, "Encrypted key exchange: Password-based protocols secure against Dictionary attacks," *Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy*, pp. 72–84, 1992.

[27] K.C. Goss, "Cryptographic method and apparatus for public key exchange with authentication," *U.S. Patent* $4,956,863$, September 1990.

[28] P. van Oorschot and M. Wiener, "On Diffie-Hellman key agreement with short exponents," *Advances in Cryptology - EUROCRYPT '96 (LNCS 1070)*, pp. 332–343.

University of Pretoria etd – Joseph, G (2005)

[29] S.C. Pohlig, M.E. Hellman, "An Improved Algorithm for Computing Logarithms in GF(p) and Its Cryptographic Significance," *IEEE Transactions on Information Theory*, vol. 24, pp. 106–111, January 1978.

[30] Z. Shmuley, "Composite Diffie-Hellman public-key generating systems are hard to break," Technical Report 356, Computer Science Department, Technicon, Haifa, Israel, February 1985.

[31] K.S. McCurley, "A key distribution system equivalent to factoring," *Journal of Cryptography*, vol. 1, no. 2, pp. 95–106, 1988.

[32] N. Koblitz, *A course in Number Theory and Cryptography*. Graduate Texts in Mathematics, Springer-Verlag, 1987.

[33] E. Hughes, "An encrypted key transmission protocol," *Advances in Cryptology - CRYPTO '94 (LNCS 456)*, August 1994.

[34] W. Diffie, P.C. van Oorschot, M.J. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes and Cryptography*, vol. 2, pp. 107–125, 1992.

[35] R.A Rueppel, "Key agreements based on function composition," *Advances in Cryptology - CRYPTO '88 (LNCS 330)*, pp. 3–10, 1988.

[36] C.P. Waldvogel, J.L. Massey, "The probability distribution of the Diffie-Hellman key," *Advances in Cryptology AUSCRYPT '92 (LNCS 718)*, pp. 492–504, 1993.

[37] Y.Yacobi, "A key distribution paradox," *Advances in Cryptology - CRYPTO '90 (LNCS 537)*, pp. 268–273, 1991.

[38] M.J. Wiener, "Performance comparison of public-key cryptosystems," *RSA Laboratories Cryptobytes*, vol. 4(1), pp. 1–5, 1998.

[39] S.Saryazdi, "An extension to ElGamal public-key cryptosystem with a new signature scheme," *Proceedings of the 1990 Bilkent International Conference on New Trends in Communication Control and Signal Processing*, pp. 195–198, 1990.

[40] T. Beth, "Efficient Zero-Knowledge scheme for smart cards," *Advances in Cryptology - EUROCRYPT '86 (LNCS 341)*, pp. 77–84, 1988.

[41] C.C. Chang, S.J. Hwang, "Cryptographic authentication of passwords," *Proceedings of the 25th Annual 1991 IEEE International Carnahan Conference on Security Technology*, pp. 126–130, October 1991.

[42] W.J. Jaburek, "A generalisation of ElGamal's public-key cryptosystem," *Advances in Cryptology - EUROCRYPTO '89*, pp. 23–28, 1990.

[43] D.W. Kravitz, "Digital Signature Algorithm," *U.S. Patent no. 5, 231, 668*, July 1993.

[44] FIPS 180, "Secure hash function," *Federal Information Processing Standards Publication 180, U.S. Department of Commerce/N.I.S.T National Technical Information Service, Springfield, Virginia*, May 1993.

University of Pretoria etd – Joseph, G (2005)

[45] C.P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[46] D. Naccache, D. M'Raihi, S.Vaudenay, D. Raphaeli, "Can DSA be improved? Complexity trade-offs with the digital signature standard," *Advances in Cryptology EUROCRYPT '94 (LNCS 950)*, pp. 77–85, 1995.

[47] S.M. Yen, *Design and computation of public-key cryptosystems*. PhD thesis, National Cheng Hung University, April 1994.

[48] K.S. McCurley, "Open letter from the Sandia National Laboratorieson the DSA," *N.I.S.T*.

[49] C.H. Lim, P.J. Lee, "Security of interactive DSA batch verification," *Electronics Letters*, vol. 30, pp. 1592–1593, September 1994.

[50] C.K. Koc, "High-speed RSA implementation," Version 2.0, RSA Laboratories, November 1994.

[51] J.J. Quisquater, C.Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics Letters*, pp. 905–907, October 1982.

[52] S.R. Dusse, B.S. Kaliski , "A cryptographic library for the Motorola DSP 56000," *Advances in Cryptology - EUROCRYPT '90 (LNCS 473)*, pp. 230–244, 1991.

[53] E.F. Brickell, "Survey of hardware implementations of RSA," *Advances in Cryptology CRYPTO '89 (LNCS 435)*, pp. 368–370, 1989.

[54] C. Koc, "RSA Hardware Implementation," TR-801, RSA Laboratories, 1996.

[55] R.L. Rivest, "RSA chips (Past/Present/Future)," *Advances in Cryptology EUROCRYPT 84 LNCS(209), Springer-Verlag*, pp. 159–165, 1984.

[56] J.F. Dhem, *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Université Catholique de Louvain, Faculté des Sciences Appliquées, Laboratoire de Mircoélectronique, Louvain-la-Neuve, Beligique, May 1998.

[57] J. Sauerbrey, *Langzahl-Modulo-Arithmetik für kryptographische Verfahren*. Wiesbahen: DUV Informatik, Dt. Univ. -Verl., 1993.

[58] G.L. Miller, "Riemann's hypothesis and tests for primality," *Journal of Computer and System Sciences*, vol. 13, pp. 300–317, 1976.

[59] J. Håstad, "Solving simultaneous modular exponentiations of low degree," *SIAM Journal of Computing*, vol. 17, pp. 336–341, 1988.

[60] M.J. Wiener, "Cryptanalysis of short RSA secret exponents," *IEEE Trans. Information Theory*, vol. IT-36, no. 3, pp. 553–558, 1990.

[61] B.S. Kaliski, M. Robshaw, "The secure use of RSA," *CryptoBytes*, pp. 7–13, 1995.

University of Pretoria etd – Joseph, G (2005)

[62] D. Coppersmith, “Finding a small root of a univariate modular exponentiation,” *Advances in Cryptology EUROCRYPT ’96 (LNCS 1070)*, pp. 155–165, 1996.

[63] G.I. Davida, “Chosen signature cryptanalysis of the RSA (MIT) public-key-cryptosystem,” Technical Report TR-CS-82-2, Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, WI, 1982.

[64] J.M. DeLaurentis, “A further weakness in the common modulus protocol for the RSA cryptoalgorithm,” *Cryptologia*, vol. 8, pp. 253–259, 1984.

[65] G.J. Simmons, M.J. Norris, “Preliminary comments on the M.I.T. public-key cryptosystem,” *Cryptologia*, vol. 1, pp. 406–414, 1977.

[66] R.L. Rivest, R.D. Silverman, “Are ’strong’ primes needed for RSA?.” November 1999.

[67] A. Shamir, “RSA for paranoids,” *Cryptobytes*, vol. 1, pp. 1–4, 1995.

[68] ISO/IEC 9796, ed., *Information Technology security techniques - Digital signature scheme giving message recovery*, (Geneva, Switzerland), International Organization for Standardization, First ed., 1991.

[69] ANSI X9.31-1998, *Digital signatures using reversible public key cryptography for the finicial service industry (rDSA)*. American National Standards Institute, 1998.

[70] P.C. van Oorschot, “A comparison of practical public key cryptosystems based on integer factorization and discrete logarithms,” *Contempary Cryptology: The Science of Information Integrity*, pp. 289–322, 1992.

[71] B.Tuckerman, “The 24th Mersenne Prime,” *Proceedings to National Academy of Science*, vol. 68, pp. 2319–2330, 1970.

[72] M. Scott, “Comparison of methods for modular exponentiation on 32-bit Intel 80x86 processors.” Informal draft, School of Computer Applications, Dublin City University, June 1996.

[73] K. Geddes, S. Czapor and G. Labahn, *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, 1992.

[74] M. Welschenbach, *Cryptography in C and C++*. Apress Publications, Springer-Verlag, 2001.

[75] A. Weimerskirch and C.Paar, “Generalizations of the Karatsuba algorithm for efficient implementations,”

[76] C.K. Koc, C.Hung, “Fast algorithm for modular reduction,” *IEEE Proc.: Computers and Digital Techniques*, vol. 145(4), 1998.

[77] A. Bosselaers, R. Govaerts, J. Vandewalle, “Comparison of three modular reductions,” *Advances in Cryptology - Crypto ’93 (LNCS 773), Springer-Verlag*, pp. 175–186, 1994.

University of Pretoria etd – Joseph, G (2005)

[78] H. Morita, C.H. Yang, “A modular multiplication algorithm using lookahead determination,” *IEICE Trans. Fundamentals*, vol. E76-A, no. 1, 1993.

[79] N. Takagi, “A modular multiplication algorithm with triangle additions,” *11th Symp. on Computer Arithmetic, IEEE Computer Society Press*, pp. 272–276, 1993.

[80] C.H. Lim, H.S. Hwang, P.J. Lee, “Fast modular reduction with precomputation,” *Proc. of Korea-Japan Joint Workshop on Information Security and Cryptology*, October 1998.

[81] C.D. Walter, “Faster modular multiplication by operand scaling,” *Advances in Cryptology - CRYPTO ’91 (LNCS 576)*, pp. 313–323, 1992.

[82] D. Naccache, D. M’Stilti, “A new modulo computation algorithm,” *Recherche Operationelle - Operations Research (RAIRO-OR)*, no. 24, pp. 307–313, 1990.

[83] K. Hensel, “Theorie der algebraischen Zahlen,” *Leipzig*, 1908.

[84] M. Shand, J. Vuillemin, “Fast implementation of RSA cryptography,” *Proceedings of the 11th IEEE Symposium on Computer Arthimetic*, pp. 252–259, 1993.

[85] J.C. Bajard, L.S. Didier, P. Komerup, “An RNS Montgomery modular multiplication algorithm,” *IEEE Transaction on Computers*, vol. 47, pp. 766–776, July 1998.

[86] T. Blum, C.Paar, “Montgomery modular exponentiation on reconfigurable hardware,” *14th IEEE Symposium on Computer Arithmetic*, pp. 70–77, 1999.

[87] P. Behrooz, *Computer arithmetic algorithms and hardware designs*. Oxford University Press Inc., 2000.

[88] C.D. Walter, S.E. Elridge, “Hardware implementation of Montgomery modular multiplication algorithm,” *IEEE Trans. Comp.*, vol. 42, pp. 693–699, 1993.

[89] D. Naccache, D. M’Raihi, “Montgomery-suitable cryptosystems,” *Algebraic Coding Lecture Notes in Computer Science*, vol. 781, 1994.

[90] C. Koc, T. Acar, B.S. Kaliski, “Analyzing and comparing Montgomery multiplication algorithms,” *IEEE Micro*, no. 16, pp. 26–33, 1996.

[91] J. Bos, M. Coster, “Addition-chain heuristics,” *Advances in Cryptology - EUROCRYPT ’89 (LNCS 435), Springer-Verlag*, pp. 400–407, 1990.

[92] J. Sauerbrey, A. Dietel, “Resource requirements for the application of addition chains in modulo exponentiation,” *Advances in Cryptology - EUROCRYPT ’94 (LNCS 513)*, pp. 174–182, 1994.

[93] H. Cohen, *A Course in Computational Algebraic Number Theory*. Springer Verlag, Berlin, 1993.

[94] A.G. Thurbur, “Efficient generation of minimal length addition chains,” *SIAM Journal of Computing*, vol. 28, no. 4, pp. 1247–1263, 1999.

University of Pretoria etd – Joseph, G (2005)

[95] P.Downey, B. Leony and R.Sethi, "Computing sequences with addtion chains," *SIAM Journal of Computing*, vol. 3, pp. 638–696, 1981.

[96] N. Kinohiro, H. Yamamoto, "Window and extended window methods for addition chain and addition-subtraction chain," *IEICE Trans. Fundamentals*, vol. E81-A, pp. 72–81, January 1998.

[97] N. Kinohiro, H. Yamamoto, "New methods for generating short addition chains," *IEICE Trans. Fundamentals*, vol. E83-A, pp. 60–67, January 2000.

[98] N. Kinohiro, H. Yamamoto, "Optimal addition chains classified by Hamming weight," *IEICE Technical Report ISEC96-74*, 1997.

[99] A. Schönhage, "The lower bound on the length of addition chains," *Theoretical Computer Science*, vol. 1, pp. 1–12, 1975.

[100] E.G. Thurbur, "On addtion chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$," *Duke Mathematics Journal*, vol. 40, pp. 907–913, 1973.

[101] E.G. Thurbur, "The Scholz-Braurer problem on addition chains," *Pacific Journal Mathematics*, vol. 49, pp. 229–242, 1973.

[102] E.G. Thurbur, "Addition chains and solutions of $l(2n) = l(n)$ and $l(2^n - 1) = n + l(n) - 1$," *Discrete Mathematics*, vol. 16, pp. 279–289, 1976.

[103] A. Brauer, "On addition chains," *Bull. Mathematics Society*, vol. 45, pp. 736–739, 1939.

[104] H.L. Garner, "The residue number systems," *IRE Transactions on Electronic Computers*, vol. 8, pp. 140–147, June 1959.

[105] W. Stallings, *Cryptography and Network Security*. Prentice Hall, New Jersey, Second ed., 1999.

[106] R.D. Silverman, *Fast generation of random strong RSA primes*. The 1998 RSA Data Security Conference Proceedings, Cryptographers Track, 1998.

[107] B. de Weger, "Cryptanalysis of RSA with small prime difference," *Applicable Algebra in Engineering Communication and Computing, Springer Verlag*, vol. 13, pp. 17–28, 2002.