A DISTRIBUTED MULTIPRECISION CRYPTOGRAPHIC LIBRARY DESIGN

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Software

by
Hüseyin HIŞIL

July 2005
İZMİR

We approve the thesis of **Hüseyin HIŞIL**

Date of Signature

. . . . . **22 July 2005**
Assoc. Prof. Dr. Ahmet KOLTUKSUZ
Supervisor
Department of Computer Engineering
İzmir Institute of Technology

. . . . . **22 July 2005**
Prof. Dr. Şaban EREN
Department of Computer Engineering
Ege University

. . . . . **22 July 2005**
Assist. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering
İzmir Institute of Technology

. . . . . **22 July 2005**
Prof. Dr. Muhsin Çiftçioğlu
Head of Department
İzmir Institute of Technology

. . . . . .
Assoc. Prof. Dr. Semahat ÖZDEMİR
Head of the Graduate School

ACKNOWLEDGEMENTS

Foremost, I would like to express my gratitude to my advisor, Assoc. Prof. Dr. Ahmet Koltuksuz, for his guidance, patience, and encouragement. He was the one who uplifted me when I was in trouble with critical decisions. His valuable support, and confidence have been the driving force of this thesis work.

Furthermore, I had the pleasure of working with Serap Atay who helped me in understanding the mathematical background of many algorithms. I should also thank to my room mates Selma Tekir and Şükran Asarcıklı for their patience and support.

I would also like to thank Asst. Prof. Murat Atmaca, Sultan Eylem Toksoy and Gökşen Bacak for their help in writing this thesis with LaTeX2e.

Finally, I should thank to my parents who always supported me throughout my education as well as in my graduate study.

ABSTRACT

Cryptographic schemes require specialized software libraries to work with large numbers on fixed-precision processors. The concept is known as multiple-precision computation. In this thesis, we aim to review the multiple-precision algorithms with the contemporary modifications. With this motivation, we develop a new multiprecision library named CRYMPIX and we carefully benchmark CRYMPIX in comparison with the fastest alternatives. We also develop a distributed wrapper for computationally expensive functions. Hence, we provide an abstraction method for the higher level cryptographic implementations by allowing them run in a distributed environment without containing any specialized code for distribution.

ÖZET

Kriptolojik uygulamaların sabit uzunluklu değişkenleri işleme yetisine sahip işlemciler üzerinde çalıştırılabilmesi için özelleşmiş yazılımlara ihtiyaç duymaktadır. Bu kavram çok-basamaklı (multiple-precision) sayı işlemleri olarak bilinmektedir. Bu tezde temel olarak çok-basamaklı sayılar için geliştirilen algoritmalar ve güncel modifikasyonları incelenmiştir. Bu motivasyonla, CRYMPIX olarak isimlendirilen yeni bir çok-basamaklı kütüphane tasarımna gidilmiştir. Çalışmada CRYMPIX'in alternatifleri ile karşılaştrmalı performans değerlendirmesi sunulmuştur. Ayrıca hesaplanması güç olan fonksiyonların dağıtık olarak çalışmasına imkan verecek bir katman geliştirilmiştir. Dolayısıyla, kriptolojik uygulamaların dağıtık mimariden soyutlanarak özelleşmiş koda ihtiyaç duyulmaksızın dağıtık hale getirilmesi sağlanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

This thesis is a combined study of building a multiple-precision number library and providing a distributed wrapper for its cryptographic functions. Designed either for cryptographic use or not, most of the multiple-precision number libraries implement arithmetic, logic and number theoretic routines. Our first aim in this thesis study, is to do research on the computational aspects of such primitives by implementing the most important functions within a new multiple-precision integer library named CRYMPIX. There are world-widely accepted code distribution solutions such as Message Passing Interface(MPI) Standard. Our second aim is to search for finding elegant ways of distributing the cryptographic computational mass over an MPI network of general use personal computers.

The efficiency of a cryptographic implementation mostly depends on the internally-used low-level cryptographic library. A cryptographic library is said to be competitive among its alternatives if it is engineered not only with the advanced level of coding but also with the careful selection of algorithms concerning their theoretical complexities and their inclination to the underlying hardware. However, finding the best tuning is always a tedious job because one has to switch between various algorithms with respect to their threshold values. On the other hand, once the library is developed, it is relatively easier to perform further scientific studies and go deeper inside the computational aspects of the cryptographic world. With this motivation, we strongly advise to code at least some functions if not all for every researcher who is in the field of cryptology.

It is known that the asymmetrical cryptosystems require multiple-precision arithmetic when they are run on fixed-precision processors. Let's consider the following example to understand what the term multiple-precision stands for; if an RSA implementation runs at 4096-bit key size, then at least 128 computer words are needed to store and process the key on a 32-bit architecture. Therefore, specialized algorithms are to be used for handling operations such as addition, multiplication and modular reduction. To address the necessity, many libraries are developed up to now of which the most popular ones are GNU GMP, Shamus Software MIRACL, LibTomMath, PARI/GP, LiDiA, BigNum,

Java BigInteger, Bouncy Castle, Magma, Maple, Mathematica, and MuPAD. All of these libraries are implemented for related but different purposes. Therefore, it is quite likely that one needs several of them to satisfy the one's specific scientific research needs. Surprisingly, our approach is to develop a newer one to learn more about them and distinguish between the elegant and the awful implementations as well as competing for a faster and a simpler code if possible.

Code distribution of a cryptosystem is a part of the folklore. However, the parallelization effort is repeated for each implementation where our struggle is on the development of a cryptographic distributed layer (CDL) that is transparent to the protocol level implementation and to the underlying low level cryptographic library. With this motivation, we exploit a property of key generation, encryption and decryption phases of most cryptosystems that is the independent nature of tasks. For instance, RSA encryption and decryption are done via several consequent and independent modular exponentiations. So that, our approach gains importance for a higher speed RSA implementation where the CDL could still be used for some other purpose even for cryptanalytic and/or non-cryptographic fields. We limit our discussion on the encryption phase of RSA cryptosystem of which we prepare a test case, implement it and provide the test results.

We begin our study in Chapter 2 by introducing the basic concepts that we shall encounter throughout this thesis and by making a preliminary introduction to multiple-precision computation. The subject is extended on contemporary algorithms throughout Chapter 3. We discuss the requirements of multiprecision library and our design and implementation in Chapter 4.

CHAPTER 2

COMPUTATIONAL ASPECTS OF CRYPTOGRAPHY

We provide some basic definitions in Section 2.1 that we will recall in the context of this thesis. The definitions are taken from (Menezes et al. 1996, pp.67-105.) and (Wagstaff 2002, pp.27-38.). Section 2.2 is a brief introduction to multiprecision arithmetic to give the reader a foresight for further discussions.

2.1 Basic Definitions

Definition 2.1.1. An algorithm is a well-defined computational procedure that takes a variable input and halts with an output.

Definition 2.1.2. If a and b are integers, then a is said to be congruent to b modulo n, written $a \equiv b\,(mod\ n)$, if n divides $(a.b)$. The integer n is called the modulus of the congruence.

Definition 2.1.3. The *integers modulo* n, denoted $\mathbb{Z}_n$, is the set of (equivalence classes of) integers $\{0, 1, 2, ..., n-1\}$. Addition, subtraction, and multiplication in $\mathbb{Z}_n$ are performed modulo n.

Definition 2.1.4. Let $a \in \mathbb{Z}_n$. The *multiplicative inverse* of $a\,(mod\ n)$ is an integer $x \in \mathbb{Z}_n$ such that $a \cdot x \equiv 1\,(mod\ n)$. If such an x exists, then it is unique, and a is said to be *invertible*, or a *unit*; the inverse of a is denoted by a^{-1}.

Definition 2.1.5. The greatest common divisor of two non-negative integers a and b is denoted as $gcd(a, b)$ and is equal to the largest integer that evenly divides both a and b.

Definition 2.1.6. For a, $b \in \mathbb{Z}^+$, if $gcd(a, b) = 1$ then a and b are relatively prime to each other.

2.2 Multiprecision

A numerical operation is said to be single-precision if the variables are presented by fixed-sized single computer words. For instance, on a 32-bit computer, numbers can only grow up to $2^{32} = 4294967296$. General use processors are designed to perform single-precision instructions. However, single-precision operations, by themselves, are not sufficient to carry out the basic arithmetic needs of cryptographic applications. These numbers grow up to thousands of bits and the underlying hardware variables cannot hold them. Therefore, each operation on large numbers must be carried out by a well defined strategy. A suitable approach is to split each large number into computer words and let them lay along some pieces of memory spaces and perform mathematical operations on such arrays by the special use of single-precision operations. This concept is known as multiple-precision and the large numbers are named multiple-precision numbers. Arbitrary-precision, multiprecision and bignum are synonyms. In the subsequent parts of this text, the term multiprecision is preferred to address multiple-precision.

2.2.1 Representation of Numbers and Notation

Representation of numbers in multiprecision arithmetic is just alike the common representation in base 10. The only difference is that the base is typically much larger in multiprecision systems. For instance, on a 32-bit processor, the base is selected equal or smaller than $2^{32} = 4294967296$. This type of representation is called radix representation or base representation and is given in Equation 2.1.

$$x = (x_{n-1}, x_{n-2}, x_{n-3}, ..., x_0)_\beta = \sum_{i=0}^{n-1} x_i \cdot \beta^i. \tag{2.1}$$

$\beta \geq 2$, and $0 \leq i < n$, with $0 \leq x_i < \beta$, and $x_{n-1} \neq 0$, and x is the multiprecision number. β is called the base and any positive integer x_i is called a digit of x in base β. Number of digits, n, is formulated as follows: $n = |x|_\beta = \lfloor \log_\beta x \rfloor + 1$.

2.2.2 Integer Arithmetic

In this section, we mainly focus on four basic operations of multiprecision integer arithmetic that are addition, subtraction, multiplication and division. The aim is to emphasize the importance of these primitives and clarify how they are used in cryptographic

implementations. The subject is going to be extended in mathematical and technical detail on Chapters 3 and 4.

2.2.3 Addition and Subtraction

Both addition and subtraction are carried out in $O(n)$ time. The addition or subtraction of two multiprecision numbers is needed in cryptographic algorithms and both are crucial in overall performance when used in the core loops of other operations. A perfect multiprecision library should utilize the underlying hardware for these operations.

2.2.4 Multiplication

Multiprecision multiplication is the heart of the prime field based cryptographic schemes such as RSA, ElGamal, and Diffie Hellman Key Exchange. The inner loop of the classic algorithm is a multiplication of a multiprecision vector by a constant single precision integer and an multiprecision addition. Therefore, the processes is done in $O(n^2)$ time with the classic algorithm. This technique is also known as School multiplication, Standard multiplication, Baseline multiplication and Basecase multiplication. The synonym Basecase multiplication is used within the context of this study.

The efficiency of most cryptographic libraries depend on the cost of multiprecision multiplication operation. Table 2.1 summarizes the popular multiplication methods, their complexities, and of their usage intervals.[1]

Basecase multiplication gets relatively slower as the input operand sizes grow. Fortunately, there are asymptotically faster multiplication schemes of which two of them are in cryptographic concern. These two techniques are known as Karatsuba Ofman multiplication and Toom-Cook 3-Way multiplication. Karatsuba Ofman multiplication is discovered in 1962 by a Russian mathematician. The idea is to get rid of the long multiplication by replacing it with 3 multiplications with half operand size and some shifting and addition operations. The recursive behavior of this technique decreases the theoretical complexity down to $O(n^{1,585})$ (Rosen 1998) which is far less than that of Basecase multiplication. The subject is going to be revisited in the following chapters. Toom-Cook multiplication is generalized version of Karatsuba multiplication. It is based

[1]GNU-GMP documentation at http://www.swox.com/gmp/.

Table 2.1. Algorithms in use for multiprecision multiplication.

Algorithm	**Complexity**	**Interval**
Basecase	$O(n^2)$	$0-1Kb$
Karatsuba	$O(n^{1.585})$	$1-6Kb$
Toom-Cook 3−Way	$O(n^{1.465})$	$6-24Kb$
FFT Based	$O(n^{\sim 1.4})$	$24Kb$−larger

on splitting the number to $n+1$ compartments. Toom-Cook 3-Way multiplication is the special form of Toom-Cook multiplication for cryptographic use. The complexity is $O(n^{1.465})$ (Rosen 1998) which is less than the complexity of Karatsuba multiplication. Fast Fourier Transform (FFT) based multiplication is even faster than these methods however it is out of the cryptographic concern since it is significant for very large numbers. For instance, GNU GMP library uses FFT based multiplication for numbers larger than $24K$ in size.

A typical question that arises quickly is which one of these techniques to use within the implementation. The preferred approach is to use all of them within special order bounded to some threshold values. Therefore, at some recursion depth, Toom-Cook multiplication is switched with Karatsuba multiplication and then the lower threshold value determines where Basecase multiplication starts.

2.2.5 Division

Cryptographic algorithms require modular reductions which can be done using multiprecision division. However, basecase multiprecision division is the most costly operation among basic arithmetic operations with $O(n^2)$ complexity. Therefore, most efforts in computational researches are on the elimination of multiprecision division operations. A recursive division algorithm is proposed in (Burnikel and Ziegler 1998). The method de-

creases the complexity to $2K(n) + O(n \log n)$ where $K(n)$ is the Karatsuba multiplication time.

2.2.6 Sign Management

A suitable approach to keep the sign of a multiprecision integer is using a distinct variable. This is called signed magnitude representation. Specialized hardware solutions uses the two's complement representation. Signed magnitude representation is more practical to be used in multiprecision arithmetic since additional effort for checking of the sign digit is negligible.

CHAPTER 3

ALGORITHMS IN USE

In this chapter, we provide the algorithms that are of importance in cryptographic implementations. In some cases, we have provided the ones that are left to the understanding of the researchers in related articles and books. We also provide graphical illustrations and numerical examples that will make it a lot easier to understand multiprecision operations.

3.1 Multiplication Algorithms

There are several algorithms developed for finding the product of two multiprecision operands. The multiplication algorithms are differed by their special usages and/or complexities and they represent the small pieces of the puzzle of high-performance. In other words, each of the algorithms is necessary for cryptographic computation.

3.1.1 Basecase Multiplication

Basecase multiplication algorithm is discussed in (Knuth 1997, p.268.) and (Menezes et al. 1996, p.595.) and is formulated in Equation 3.1. Let x and y be two not necessarily distinct multiprecision operands with $|x|_\beta = m$ and $|y|_\beta = n$. The product is stored in the number z having $|z|_\beta = m + n$ memory places that are allocated and cleared initially. Generally, operands are equal in size in cryptographic implementations.

$$z = xy = \sum_{i=0}^{m-1} x_i \beta^i \cdot \sum_{j=0}^{n-1} y_j \beta^j = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_i y_j \beta^{i+j} \tag{3.1}$$

The inner loop in Equation 3.1 is constructed to calculate inner-products and to add them to the current value of z. The implementation tricks are covered in Chapter 4. The Basecase multiplication algorithm is given in Algorithm 3.1. The term $t_{H,L}$ represents a double precision variable where t_H is the the upper and t_L is the lower half of the variable.

The naive illustration in Figure 3.1 may give the reader how the computer words are used throughout the operation. Suppose that we want to multiply two

input : x, y with $|x|_\beta = m$ and $|y|_\beta = n$.

output: Returns the product $z = x \cdot y$ with $|z|_\beta = m + n$.

for $i = 0$ **to** $m - 1$ **do**
  $t_H = 0$.
  for $j = 0$ **to** $n - 1$ **do**
    $t_{H,L} = z_{i+j} + x_i \cdot y_j + t_H$.
    $z_{i+j} = t_L$.
  end
  $z_{i+j} = t_H$.
end

return z.

Algorithm 3.1. Basecase Multiplication Algorithm.

integers 3981788410 and 16318719 in base 10. Then, $3981788410 \times 16318719 = 64977686180246790$. To perform the operation at word level, radix conversion is applied to $(3981788410)_{10} = (11101101010101010100010011111010)_2$ and $(16318719)_{10} = (111110010000000011111111)_2$. Efficient radix conversion algorithms are explained by (Knuth 1997, pp.319-327). The remaining operations are included in Figure 3.2.

No overflow occurs in any of the carry words since $(\beta-1)+(\beta-1).(\beta-1)+(\beta-1) = (\beta^2 - 1)$ where the summed terms represents; previously calculated sum, inner product and carry words respectively. The largest possible outcome, $(\beta^2 - 1)$, does not exceed the length of two digits in base β hence Basecase multiplication algorithm does not have any upper bound on the length of the operands. However, Basecase multiplication requires n.m single precision multiplications that makes it less attractive for numbers having 32 or more digits in base β. Note that 32 digit is an empirical value due to our implementation measurements. This value tends to change on various hardware.

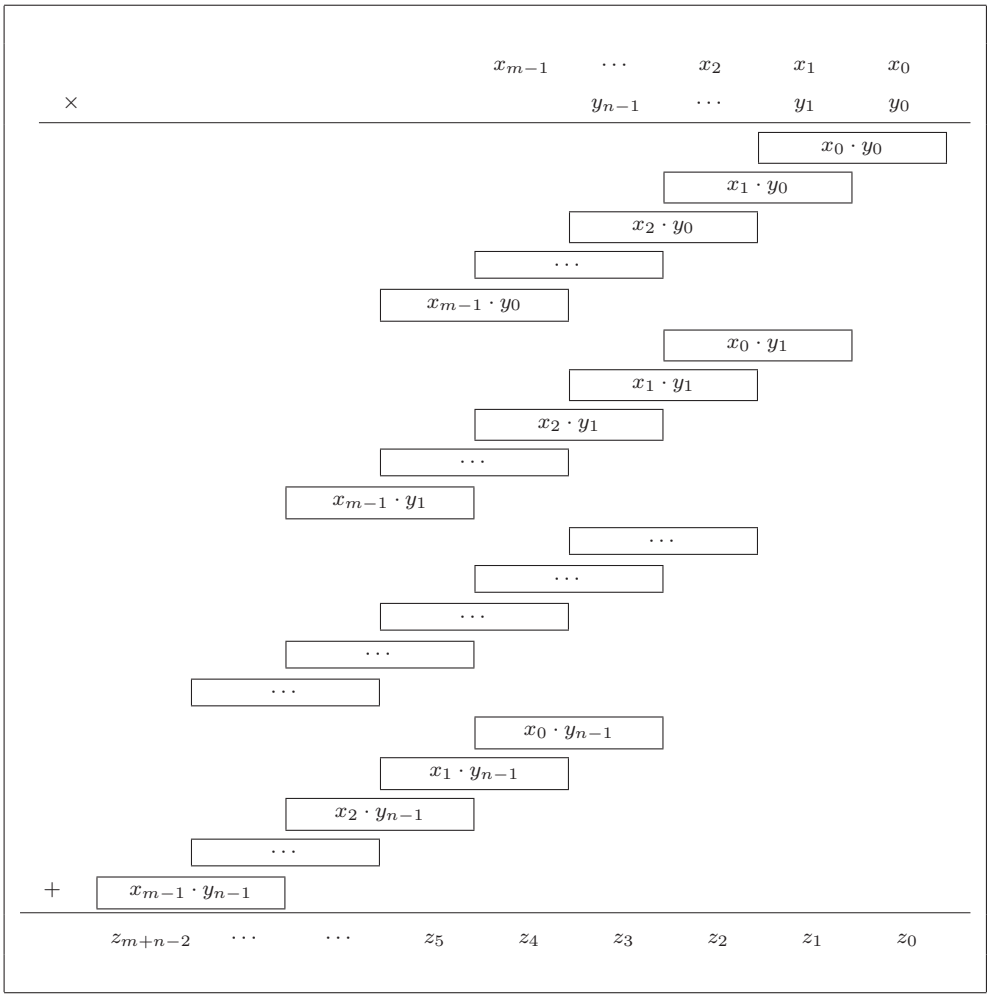

Figure 3.1. Basecase Multiplication Scheme

				11101101	01010101	01000100	11111010
$\times$					11111001	00000000	11111111
			11101100	01100111	11101111	10110101	00000110
		00000000	00000000	00000000	00000000	00000000	
$+$	11100110	11010111	11110000	00010111	00101010		
	11100110	11011000	11011100	01111111	00011001	10110101	00000110

Figure 3.2. Basecase Multiplication Example

3.1.2 Comba Multiplication

Another basecase multiplication is described by (Comba 1990). The method requires $O(n^2)$ time and enables efficient parallelization by eliminating upward carry propagation of inner-product operation in Basecase multiplication. The algorithm produces the product from least significant word to most significant word, one at each outer iteration by summing the equal order single-precision product and fixing the carry bits externally. Comba multiplication is not used in most of the libraries because it is much more suitable for hardware implementations. Nevertheless, Comba multiplication finds its use in Half multiplication scheme explained in Section 3.1.3. A simple numerical example to perform $3981788410 \times 16318719 = 64977686180246790$ is given in Figure 3.3. The algorithm is visualized in Figure 3.4.

					11101101	01010101	01000100	11111010
×						11111001	00000000	11111111
							11111001	00000110
						00000000	00000000	
						01000011	10111100	
					11110011	00101010		
					00000000	00000000		
					01010100	10101011		
				01000010	00100100			
				00000000	00000000			
				11101100	00010011			
			01010010	10101101				
			00000000	00000000				
+		11100110	10000101					
	11100110	11011000	11011100	01111111	00011001	10110101	00000110	

Figure 3.3. Comba Multiplication Example

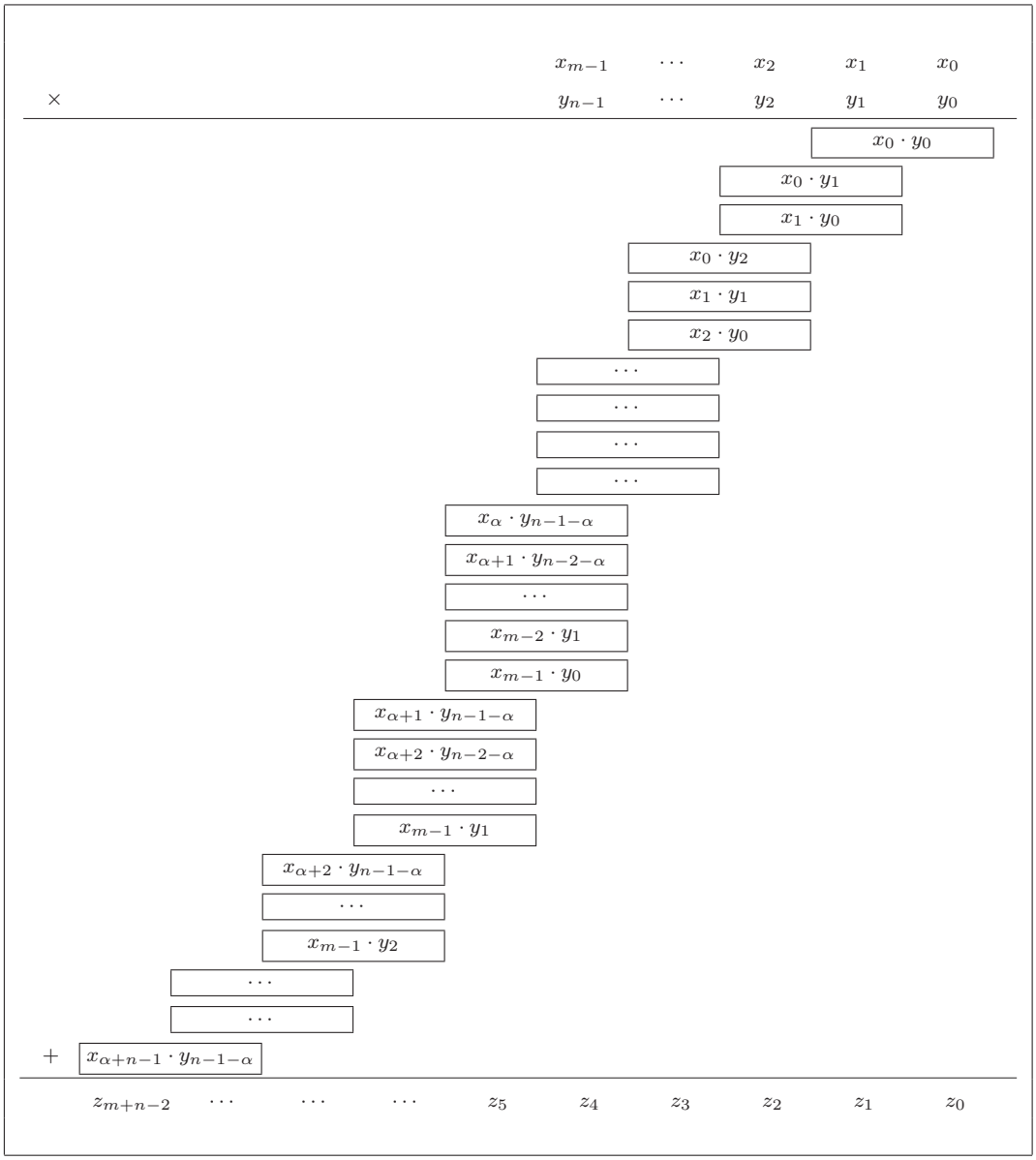

Figure 3.4. Comba Multiplication Scheme

3.1.3 Lower Half Product

The lower half of the product is needed in some cases. For instance, fully-recursive Montgomery Reduction (see Section 3.3.3), uses half multiplication to operate at a lower complexity. Half multiplication can be achieved by ignoring the upper half of the full product. However, this approach leads to inefficient implementation since we loose time for the extra computation of the upper half. This is the part where Comba multiplication is taken into consideration because the algorithm is developed to give one least-significant-word of the final product at each outer iteration. Thus, Half multiplication is achieved by terminating the operation when sufficiently many words are computed. In addition, the lower half sized partitions can take the advantage of the fastest multiplication method available by leaving the upper half to Comba type multiplication. Furthermore, the scheme can be applied recursively. Pseudocode is provided in Algorithm 3.2 and the approach is illustrated in Figure 3.5. Note that this is a naive approach for half product operation. More efficient partitioning techniques are discussed in (Mulders 2000).

input : x, y with $|x|_\beta = |y|_\beta = n$, n is even.
output: Returns the half product $z = x \cdot y \ (mod\ \beta^n)$ with $|z|_\beta = n$.

if $n <$ *CombaThreshold* **then**

$z = x \cdot y \, (mod\ \beta^{\frac{n}{2}})$. `//Use Comba method for lower half product.`

else

$z = x \, (mod\ \beta^{\frac{n}{2}}) \cdot y \, (mod\ \beta^{\frac{n}{2}})$. `//Use the fastest method.`

$z = z + \left(x \, (mod\ \beta^{\frac{n}{2}}) \cdot \frac{y}{\beta^{\frac{n}{2}}} \right) \cdot \beta^{\frac{n}{2}}$. `//Use Algorithm 3.2.`

$z = z + \left(\frac{x}{\beta^{\frac{n}{2}}} \cdot y \, (mod\ \beta^{\frac{n}{2}}) \right) \cdot \beta^{\frac{n}{2}}$. `//Use Algorithm 3.2.`

end

return z.

Algorithm 3.2. Lower Half Product Algorithm.

Division and modular reduction operations of Algorithm 3.2 has no computational

load since they represent the lower half of the number when modular reduction is used and the upper half of the number when division is used.

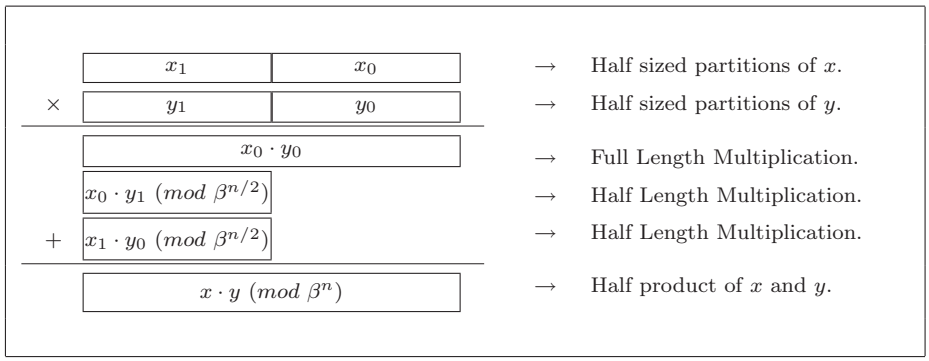

Figure 3.5. Lower Half Product Scheme.

3.1.4 Squaring

Squaring can be performed faster than long hand multiplications that are discussed in Section 3.1.1 and Section 3.1.2. Fast squaring is explained in detail by (Menezes et al. 1996, p.597.) and (Koc 1994, p.41.). Therefore, we will skip the section in short. Nevertheless, the idea is to reduce total number of operations by eliminating the double execution of the same single-precision products. This approach is summarized in Equation 3.2.

$$z = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i x_j \beta^{i+j} = 2 \cdot \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} x_i x_j \beta^{i+j} + \sum_{j=0}^{n-1} x_i^2 \cdot \beta^{2i}. \tag{3.2}$$

It is irrelevant to give the algorithm here since the corresponding pseudocode can be derived from Algorithm 3.1 easily.

3.1.5 Karatsuba Multiplication

An asymptotically faster method with $O(n^{\log_2 3})$ running time is proposed in (Karatsuba and Ofman 1962). To multiply x and y where $x \geq y$, let $n = |x|_\beta$ be the digit

count of x and $L = \lceil \frac{n}{2} \rceil$. We partition both operands using L such that $x = x_1 \cdot \beta^L + x_0$ and $y = y_1 \cdot \beta^L + y_0$. Then,

$$\begin{aligned} xy &= (x_1\beta^L + x_0)(y_1\beta^L + y_0) \\ &= x_1y_1\beta^{2L} + x_1y_0\beta^L + x_0y_1\beta^L + x_0y_0 \qquad (3.3) \\ &= x_1y_1\beta^{2L} + [(x_1 + x_0)(y_1 + y_0) - (x_1y_1 + x_0y_0)]\beta^L + x_0y_0. \qquad (3.4) \end{aligned}$$

Equation 3.3 has a recursive nature however it is not superior to Basecase multiplication in terms of time. On the other hand, Equation 3.4 that is illustrated as in Figure 3.6, eliminates one of the $L \cdot L$ multiplication. The method replaces a $2L \cdot 2L$ multiplication with 3 $L \cdot L$ computationally easier multiplication. Furthermore, the recursive nature of the method finds its best use when it is applied to a specific recursion depth. As a consequence of addition and subtractions, the developer has to cope with the carry bits. The implementation details are discussed in Chapter 4. An elegant implementation of Karatsuba multiplication is mandatory for every cryptographic libraries.

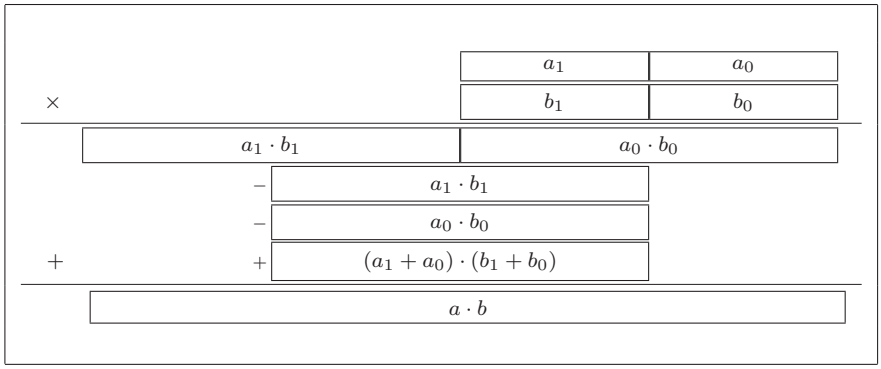

Figure 3.6. Karatsuba Multiplication Scheme

For instance, let's apply one level Karatsuba multiplication for the numbers $(11101101010101010100010011111010)_2$ and $(11111001101001011111111)_2$. The evaluation of the method is given in Figure 3.7.

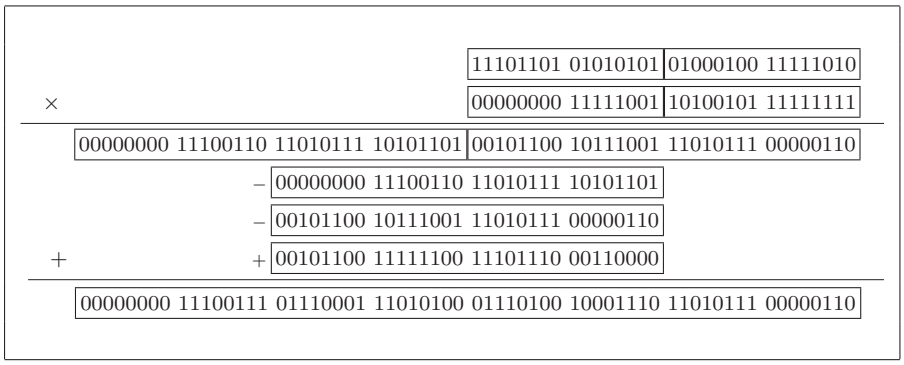

Figure 3.7. Karatsuba Multiplication Example

3.1.6 Toom-Cook 3-Way Multiplication

Toom-Cook multiplication algorithm is based on polynomial multiplication. Actually, it is a method of adapting polynomial multiplication for multiplying integers. From a different point of view, it is the generalization of Karatsuba multiplication where the numbers are divided into $n+1$ partitions instead of 2. The mathematical background of Toom-Cook multiplication and polynomial multiplication goes deeper in (Knuth 1997, pp.295-313). In this thesis, we merely explain the special case of Toom-Cook algorithm that is significant in cryptographic implementations. The method is called Toom-Cook 3-way multiplication since the operands are split into 3 partitions.

Let $u = (u_2, u_1, u_0)_k$ and $v = (v_2, v_1, v_0)_k$ be the operands to be multiplied where $n = |x|_\beta = |y|_\beta$ and $n = 3k$ for some $k \in \mathbb{Z}^+$, note that we took equal-sized operands for simplicity of operations. To explain the way of multiplying two operands we start with defining polynomials $U(x)$ and $V(x)$. The multiplication of the polynomials $U(x) = U_2x^2 + U_1x^1 + U_0x^0$ and $V(x) = V_2x^2 + V_1x^1 + V_0x^0$ yields the coefficients of polynomial $W(x)$ such that

$$W(x) = U(x) \cdot V(x) = W_4x^4 + W_3x^3 + W_2x^2 + W_1x^1 + W_0x^0. \tag{3.5}$$

It is straight forward to show $W(k) = U(k) \cdot V(k) = u \cdot v$. If we can find a fast way of evaluating coefficients W_i for $i \in \mathbb{Z}^+$ and $0 \leq i \leq 4$ of $W(x)$ then we are done with the

desired product $u \cdot v$. To find the five unknown coefficients, we obtain 5 linear equations at arbitrary points of the polynomial $W(x)$. Three of these points are very common to be selected:

$$W(0) = U(0) \cdot V(0) = W_0 = U_0 \cdot V_0. \tag{3.6}$$

$$W(1) = U(1) \cdot V(1) = W_4 + W_3 + W_2 + W_1 + W_0 = (U_2 + U_1 + U_0) \cdot (V_2 + V_1 + V_0). \tag{3.7}$$

$$W(\infty) = U(\infty) \cdot V(\infty) = W_4 = U_2 \cdot V_2. \tag{3.8}$$

The remaining two points are selected at $W(2)$ and $W(3)$ in Knuth. GNU-GMP uses $W(2)$ and $W(\frac{1}{2})$ which seems to be more advantageous to in implementation.

3.1.7 Other Multiplication Algorithms

There are other multiplication algorithms which are explained in computational algebra resources. We will limit our discussion with currently mentioned algorithms since the most suitable methods to be used in cryptographic applications. For very large numbers reader should follow the publications of Schönhage A., Strasse V., and Zuras D..

3.2 Greatest Common Divisor Algorithms

Many situations in cryptography require the computation of the greatest common divisor (gcd) of two positive integers (Menezes et al. 1996). Greatest Common Divisor from Definition 2.1.6 computation is crucial in most cryptographic implementations. For instance, Greatest Common Divisor is used in RSA key generation, in n-Residue systems and even in Modular Exponentiation. In the rest of this document Greatest Common Divisor is abbreviated as GCD. Summarized by the recursive formula $gcd(x,\ y) = gcd(x\ (mod\ y),\ x)$ for $x \geq y$, the first GCD algorithm, also the first nontrivial one, is defined by Euclid in 300 B.C. (Knuth 1997, pp.333-336.). In 1938, the algorithm is modified by Lehmer for efficient multiprecision computation. Lehmer's GCD algorithm is explained in Section 3.2.2. A completely different approach, Binary GCD, is given in Section 3.2.3 and its multiprecision variant is discussed in Section 3.2.4. Both type of the

algorithms has the extended versions. Extended GCD is used for finding multiplicative inverse.

3.2.1 Euclid GCD Algorithm

The correctness of this algorithm is proved in number theoretic books. However, Euclid GCD algorithm suffers from computational efficiency because of the multiprecision division at each step. Therefore, the algorithm works at bit level and is not implemented in most libraries. It is a symbolic example to give the computational basis of GCD. The extended version of GCD algorithm is given in Algorithm 3.3 which based on the algorithm in (Menezes et al. 1996, p.67.). The Extended GCD algorithm computes $d = gcd(x, y)$ as well as xd and yd satisfying $x \cdot xd + y \cdot yd = d$.

input : x, y where $x \geq 0$ and $y \geq 0$ with $x \geq y$.

output: Returns $d = gcd(x,\ y)$ with $x \cdot xd + y \cdot yd = gcd(x,\ y)$.

$xd = 1,\ yd = 0.$

if $y \neq 0$ **then**

$xt = 0,\ yt = 1.$

while $y \neq 0$ **do**

$q = \left\lfloor \frac{x}{y} \right\rfloor,\ r = x - q \cdot y,\ x = xd - q \cdot xt,\ y = yd - q \cdot yt.$

$x = y,\ y = r,\ xd = xt,\ xt = x,\ yd = yt,\ yt = y.$

end

end

$d = x.$

return $\{d, xd, yd\}$.

Algorithm 3.3. Extended Euclid GCD Algorithm.

3.2.2 Lehmer GCD Algorithm

Lehmer's modification to Classic GCD benefits an exact quotient guess to eliminate many expensive divisons. Lehmer GCD algorithm is improved with three strategies by (Jebelean 1993a). The extended version with Jebelean's double digit and approximative approaches is summarized in Algorithm 3.4. The algorithm is based on (Menezes et al. 1996, p.607.) and assumes that the input x and y values are equal in size in terms of computer word count. If the number sizes differ, then one has to use a preliminary modular reduction.

3.2.3 Binary GCD Algorithm

Euclidean algorithms deals with the most significant bits and/or words of the operands. In 1967, Josef Stein published an algorithm in which the operands are processed from the least significant bits and/or words. The newer approach uses the following well known properties of $gcd()$ function:

a. If x and y are both even, then $gcd(x,\ y) = 2 \cdot gcd(x/2,\ y/2)$.

b. If x is even and y is odd, then $gcd(x,\ y) = gcd(x/2,\ y)$.

c. $gcd(x,\ y) = gcd(x - y,\ y)$.

d. If x and y are both odd, then $x - y$ is even, and $|x - y| < max(x, y)$.

The algorithm is suitable for numbers that are few word long. The extended version of the algorithm is given in (Menezes et al. 1996, p.608.).

3.2.4 Generalized GCD Algorithm

A k-ary version of Binary GCD algorithm by the independent studies of (Jebelean 1993b) and (Weber 1995) enables a practically faster method when compared to Lehmer GCD and Sorenson's k-ary GCD algorithms. The method eliminates one word of both operands by a modular conjugation step and an exact divison step respectively.

input : $x,\ y$ integers, $x > 0$ and $y > 0$ and $|x|_\beta = |y|_\beta$ with $x \geq y$.

output: Returns $d = gcd(x,\ y)$ with $x \cdot xd + y \cdot yd = d$.

$xd = 1,\ yd = 0.$

while $b \neq 0$ **do**

$m = |x|_\beta,\ n = |y|_\beta.$

$\overline{x} = \dfrac{(x_{m-1},\ x_{m-2},\ x_{m-3})}{x_{m-1} + 1}.$ `// `$\overline{x}$` is double-precision.`

$\overline{y} = \dfrac{(y_{n-1},\ y_{n-2},\ y_{n-3})}{x_{m-1} + 1}.$ `// `$\overline{y}$` is double-precision.`

$A = 1,\ B = 0,\ C = 0,\ D = 1,\ q = 0,\ q' = 0.$

while $(\overline{y} + C \neq 0)$ *and* $(\overline{y} + D \neq 0)$ *and* $(q = q')$ **do**

$q = \left\lfloor \dfrac{\overline{x} + A}{\overline{y} + C} \right\rfloor,\ \ q' = \left\lfloor \dfrac{\overline{x} + B}{\overline{y} + D} \right\rfloor.$

if $q = q'$ **then**

$t = A - q \cdot C,\ \ A = C,\ \ C = t.$

$t = B - q \cdot D,\ \ B = D,\ \ D = t.$

$t = \overline{x} - q \cdot \overline{y},\ \ \overline{x} = \overline{y},\ \ \overline{y} = t.$

end

end

if $B = 0$ **then**

$T = x\ (mod\ y),\ \ x = y,\ \ y = T.$

$T = xd + \left\lfloor \dfrac{x}{y} \right\rfloor \cdot yd,\ \ yd = xd,\ \ xd = T.$

else

$T = (x \cdot A + y \cdot B),\ \ U = (x \cdot C + y \cdot D),\ \ x = T,\ \ y = U.$

$T = (xd \cdot |A| + yd \cdot |B|),\ \ U = (xd \cdot |C| + yd \cdot |D|),\ \ xd = T,\ \ yd = U.$

end

Correct the sign of xd and yd.

end

return $\{d, xd, yd\}$.

Algorithm 3.4. Extended Lehmer GCD Algorithm.

input : x, y where $x \geq 0$ and $y \geq 0$ with $x \geq y$.

output: $gcd(x, y)$.

while $x \neq 0$ *and* $y \neq 0$ **do**

if $x < y$ **then**

$swap(x,\ y)$.

end

if $|x|_\beta = |y|_\beta$ **then**

Find c with modular conjugation for $x_0 \cdot c + y_0 \equiv 0\ (mod\ \beta^2)$.

Derive single words (u, v) from c such that $u \cdot c - v \equiv 0\ (mod\ \beta^2)$.

Compute $x = |x \cdot u + y \cdot v|$.

else

$x = x\ (mod\ y)$. `//bmod operation.`

end

end

return x.

Algorithm 3.5. Generalized GCD Algorithm.

3.2.5 Other GCD Algorithms

There are other Euclidean or Binary GCD based algorithms defined by various researchers. For further investigation, reader should refer to the related studies of Damien Stehlé, Tudor Jebelean, Arnold Schönhage, Sidi Mohammed Sedjelmaci, John Sorenson, Kenneth Weber, and Paul Zimmermann.

3.3 Exponentiation Techniques

Modular exponentiation is the most time consuming operation among the others which are covered within the previous sections. Nevertheless, the operation is to be processed quickly to enable efficient use of cryptographic schemes such as RSA and ElGamal encryption and decryption and Diffie-Helmann key exchange. Therefore, any single information about the characteristic of the operation is important to lower the number of multiplications and save valuable time. There are plenty of techniques for doing faster exponentiation and modular reduction. The best strategy is to combine the suitable techniques together. In this study, we include the most suitable techniques for general use personal computers. The modular exponentiation is achieved by a modular reduction step after each multiplication. Thus, any exponentiation technique such as Successive Squaring in Section 3.3.1 and Variable Length Windows Sliding in Section 3.3.2 in $\mathbb{Z}^+$ can be modified to work for $\mathbb{Z}_m, m \in \mathbb{Z}^+$. Unfortunately, modular exponentiation is a bit level operation that depends on the exponent length and that makes it relatively slower even with the best methods known.

3.3.1 Successive Squaring

In cryptographic bounds, it would be computationally infeasible to do y group multiplications to evaluate x^y that would require $O(n)$ operations. Successive Squaring is the main approach that lowers the time to $O(\log\ n)$. Although, the complexity function still depends on the size of exponent, it is now in computational margins for the numbers up to few thousand bits long. The technique is also called Binary Method or Square and Multiply Method. Successive Squaring method scans the exponent bits from left-to-right or right-to-left to evaluate the answer quickly.

3.3.2 Variable Length Windows Sliding

Exponent scanning can be utilized by Window Sliding technique. In the execution of the method, a precomputation phase determines some small powers and then they are used in the multiplication step of Successive Squaring by eliminating many of the multi-precision multiplications in Chapter 3.3.1. The method becomes faster when the database size for precomputed values is selected to minimize the total number of multiplications

including the precomputation phase and when the sliding windows are formed in variable length to maximize the number of digits set to $(1)_2$. The subject is further examined in (Koc 1994).

3.3.3 Montgomery Modular Multiplication

A different way of computing modular multiplication is introduced in (Montgomery 1985). The method shows how to eliminate the expensive reduction step with long hand division by replacing it with preferably two fast multiplications or a full length basecase multiplication and some addition and shift operations. Montgomery's method is based on the shifting of calculations modulo n into a complete residue system defined by the Equation 3.9.

$$R(r,n) = \{i \cdot r \,(mod\ n) \mid 0 \leq i < n\}. \tag{3.9}$$

Montgomery Reduction computes the value of Equation 3.11. However, a straight forward implementation will require a multiprecision multiplication and division. It is far cheaper to compute $t \cdot r^{-1}\,(mod\ n)$ via Equation 3.10 which gives the same result with Equation 3.11. The idea behind this scene is to work with modulo r without violating residue n.

$$\begin{aligned}
\frac{t + m \cdot n}{r}\,(mod\ n) &\equiv \frac{t + [t \cdot n^{'}\,(mod\ r)] \cdot n}{r}\,(mod\ n) \qquad (3.10)\\
&\equiv \frac{t + \left[t \cdot n^{'} - \left\lfloor \frac{t \cdot n^{'}}{r} \right\rfloor \cdot r\right] \cdot n}{r}\,(mod\ n)\\
&\equiv \frac{t + t \cdot n \cdot n^{'} - n \cdot r \cdot \left\lfloor \frac{t \cdot n^{'}}{r} \right\rfloor}{r}\,(mod\ n)\\
&\equiv \frac{t + t \cdot (r \cdot r^{-1} - 1)}{r} - n \cdot \left\lfloor \frac{t \cdot n^{'}}{r} \right\rfloor (mod\ n)\\
&\equiv \frac{t \cdot r \cdot r^{-1}}{r}\,(mod\ n)\\
&\equiv t \cdot r^{-1}\,(mod\ n). \qquad (3.11)
\end{aligned}$$

The algorithm holds for arbitrary value of r where $r > n$ and $gcd(r,n) = 1$ and if r is selected $r = \beta^{|n|_\beta}$, then the reduction step is carried out easily with β-sized-word hardware. The methods needed for Montgomery's Modular Multiplication are shown in Algorithms 3.6 and 3.7.

input : $x,\ y,\ n$.

output: Returns $x \cdot y \,(mod\ n)$.

$\bar{x} = x \cdot r \,(mod\ n),\ \ \bar{y} = y \cdot r \,(mod\ n)$. `//Precomputation.`

$t = \bar{x} \cdot \bar{y}$. `//Use fastest multiplication available.`

$\bar{z} = t \,(mod\ n)$. `//Use Algorithm 3.7.`

$z = \bar{z} \cdot r^{-1} \,(mod\ n)$. `//Postcomputation.`

return z.

Algorithm 3.6. Montgomery Multiplication.

Algorithm 3.6 is rather a transformation phase for the operands. Firstly, they are converted to n-residue forms. After the Montgomery Multiplication step, a back conversion is applied to the n-residue product.

input : t, n.

output: Returns $u = t \cdot r^{-1} \,(mod\ n)$.

$m = t \cdot n' \,(mod\ r)$.

$u = (t + m \cdot n) \,/\, r$.

if $u \geq n$ **then**
    $u = u - n$.

end

return u.

Algorithm 3.7. Montgomery Reduction (REDC).

At the first glance, Algorithm 3.7 seems to be much slower than classic reduction. However, the cost of division and modulus operations via r is negligible since r is of the form $\beta^{|n|_\beta}$. The remaining expensive operations $(t \cdot n')$ and $(m \cdot r)$ are also carried out easily by a basecase approach that is introduced in (Dussé and Kaliski Jr. 1991). On the other hand, one can still need to work asymptotically faster by computing $(t \cdot n')$

and $(m \cdot r)$ with non-basecase multiplication schemes. We included a time comparison on MIRACL's two different *modexp*() functions that shows the performance of either of the approaches. Table 3.1 and Figure 3.8 together provides the possible speedup. The experiment is done with DEBIAN/LINUX with GNU/GCC Compiler at optimization level $-O2$ with ANSI-C. Target hardware is Intel Centrino 1.4Ghz processor and 768MB of main memory. The margins of the trade off may still vary in other libraries and in Assembly enabled builds.

Table 3.1. Modular exponentiation times for two different compilations of MIRACL library. (milliseconds).

Length	**1K**	**2K**	**4K**	**8K**
MIRACL ModExp with Basecase REDC	28	208	1611	12613
MIRACL-KCM ModExp with Recursive REDC	31	204	1298	8132

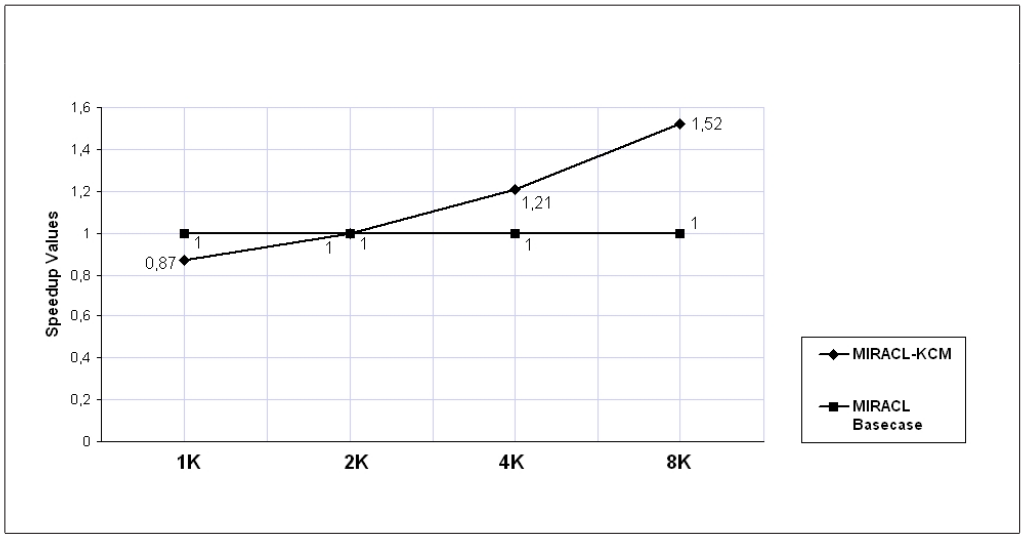

Figure 3.8. Speedup values of MIRACL-KCM modexp() with recursive REDC function over MIRACL modexp() with basecase REDC function.

Although Montgomery's method is a different way of doing modular multiplication, it is relatively slower because of its precomputation and postcomputation phases. Yet, it is highly preferable when used in modular multiplication. Algorithm 3.8 shows how to exploit Montgomery's approach in the computation of modular exponentiation.

input : $x,\ y$ where $y \geq 1$ and $y = (y_{i-1},\ ...,\ y_2,\ y_1,\ y_0)_2$.

output: Returns $x \cdot y\,(mod\ n)$.

Set k to optimal window size.

$x_0 = x$.

for $i = 0$ **to** $k - 1$ **do**
$dbase[i+1] = dbase[i] \cdot x^2$.

end

$r = \beta^{|n|_\beta}$.

$\bar{x} = x \cdot r\,(mod\ n)$.

$\bar{z} = 1 \cdot r\,(mod\ n)$.

while $i > 0$ **do**

if $y_{i-1} = 0$ **then**
$\bar{x} = \bar{x}^2 \cdot r^{-1}\,(mod\ n)$.

$i = i - 1$.

else
Determine next window with a value of W with bit length L.

$\bar{x} = \bar{x}^{2^L} \cdot dbase[W] \cdot r^{-1}\,(mod\ n)$.

end

end

$z = 1 \cdot \bar{z} \cdot r^{-1}\,(mod\ n)$.

return z.

Algorithm 3.8. ModExp, Window-Sliding, Montgomery Multiplication.

3.4 Modular Reduction

Modular reduction is nothing but a shuffling mechanism of numbers within a group that must be carried out as quick as possible. The reduction of a single integer a in modulus n is carried out by the multiprecision division operation. A multiprecision division with remainder suffices the modular reduction operation. If several reductions are to be made for a single modulus, then Barrett reductions is used. Previously mentioned Montgomery REDC function is used for reducing odd moduli numbers in modular exponentiation. All three approached are discussed in (Bosselaers et al. 1994b) in detail. Thus, we will skip any further discussion of the part.

CHAPTER 4

CRYMPIX: ANALYSIS, DESIGN AND IMPLEMENTATION

The project CRYMPIX is an educational, open-ended, and multi-layered cryptographic library. CRYMPIX aims to provide any service that is related with cryptology as an open-ended library. Thus, it is highly experimental and code refactoring never ends in time. Major and minor design changes are considerable and encouraged. CRYMPIX is a multi-layered library. The library is developed with layers each having a different level of abstraction that means each of them provides distinct a solution to the cryptographic needs, starting from low-level operations such as hardware abstraction to high-level operations such as specialized number theoretic techniques for faster implementations of cryptosystems. The subject is extended in Section 4.3.4. As an educational library, CRYMPIX aims to provide the known solutions for various problems by giving researcher the chance of trying new ideas and compare them with the classic and contemporary approaches.

In sections from 4.1 to 4.5, we discuss the current development issues of CRYMPIX in comparison with the others. Besides, implementation notes are covered in detail to reveal the differences between the theory and the practice.

4.1 Requirements of a Cryptographic Library

The overall performance is the major requirement of multiprecision libraries. Thus, all decisions should be made to provide the maximum efficiency by utilizing the underlying hardware at its peak. In addition, the faster algorithms should be triggered when operands are too large to be processed by the basecase approaches. Therefore, the best performance is a result of the union of theoretic knowledge-base with good programming skills.

Another requirement is the code portability which is not a big issue when a portable programming language such as ANSI C is used. However, the portable code may behave relatively slower on various hardware. For instance, if the target hardware doesn't have

any built-in division instruction, then the design parameters may vary to preserve computation speed although even when a machine independent language and compiler are referred. Thus, developer should first determine the target architecture that the library is expected to run on. As a consequence, machine dependent coding is also required for the inner-most loops/layers of the library. For instance, assembly coding of some primitive operations may enhance the overall performance for some specific hardware. The speed-up is empirically around 5 when processor support is supplied. Most of the competing multiprecision libraries provide special processor support to gain the maximum efficiency.

Each multiprecision number occupies a considerable space on the memory. This value varies between 512 bytes and 8K in cryptographic applications. Hence, custom memory management may become crucial when many numbers are created and killed rapidly within an application. We do not supply any measurements here since it is behind our scope.

Error handling is a general subject in software engineering. Likewise, a multiprecision library should include necessary error controls without causing serious degradation in overall performance. Thus, most libraries enables error handling as a compile time option and it is used during the development phase. Note that the term error is used both for exceptional cases such a out of memory exception and mathematical errors such as divide-by-zero error.

Code readability and simplicity are the minor requirements. A well designed multiprecision library is expected to have a simple abstract programming interface (API) for the integration of higher level implementations. For a better code readability, the context of the code should reflect the construction of the algorithm at some level that makes the new developers understand the previous work done.

4.2 Design Criteria of CRYMPIX

Common design criteria of most multiprecision libraries are representation of numbers, programming language selection, memory management, portability, error handling and functionality (Bosselaers et al. 1994a). We will skip error handling and functionality since there are less important in our case. A well designed library is expected to satisfy optimum decisions and utilize the underlying hardware at its peak. In the following sections, we describe the design parameters of CRYMPIX and compare and contrast it with that of the corresponding parameters of other libraries.

4.2.1 Programming Language and Portability

The languages that are preferred in multiprecision library development are Assembly, C, C++, FORTRAN, and Java. Excluding Assembly, the performance of any given cryptographic library depends on the coding talents of developer as well as the chosen design criteria. It is clear that performance of Assembly will always be one step ahead hence the exclusion.

ANSI C is selected as the development language of CRYMPIX. Pointer arithmetic and structural features and portability of ANSI C code play the most important role in our decision. Easy integration with Message Passing Interface (MPI) is also a distinguishing factor. In most of the other cryptographic libraries some inner-most loops are delivered to user with Assembly on the compile time as an answer to the demand of high speed computation. We are going to limit our discussion only with C and the C based versions of other libraries in this thesis since CRYMPIX aims to be an educational library in which the most suitable algorithms are being implemented for cryptographic use.

4.2.2 Representation of Numbers

To represent the numbers, almost all multiprecision libraries use positive integer vectors that are analogous to the radix representation (Equation 2.1). CRYMPIX also uses this representation. We provide a simple diagram that shows how a number is placed into the memory of a 8-bit architecture in Figure 4.1.

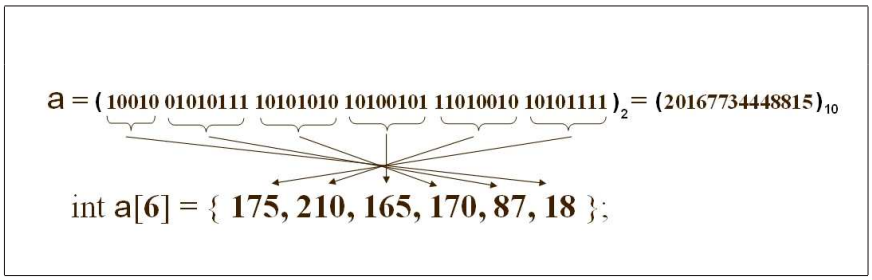

Figure 4.1. Representation of multiprecision numbers within the memory.

The number is partitioned into compartments and is laid along a memory space with the first variable being set to the least significant digit of the number. Note that, Figure 4.1 is a symbolic illustration that does not point out all implementation details.

4.2.3 Memory Management

Since all asymmetrical cryptosystems uses modular arithmetic, we are able to know how much the numbers grow. In this case, it is possible to prevent memory fragmentation if we fix the size of each number. Furthermore, memory allocation cost can be further decreased if a specialized kernel layer is utilized for the implementation. The kernel is responsible for fast memory allocation and subsequent release service. The whole memory needed by the application is allocated when the system initialized. This type of approach is crucial in embedded and/or real-time systems. To prevent the system run out of memory, exceeding allocations can be made by `malloc()` function. In other words, system starts dynamic memory allocations if and when necessary.

MIRACL's design is partially similar to above discussion. The space need for each number is fixed and is declared to the system as a runtime parameter. The memory allocation is done via `malloc()` function. Each number that is passed to a function is assumed to be initialized. To overcome the slowness of memory allocation from heap by `malloc()` function, MIRACL uses an inner workspace. This approach prevents exhaustive memory allocation and release problem.

Memory allocation in GMP is done with `malloc()` function. The system automatically increase memory space for each number when needed. This approach is open to

memory fragmentation which slows down GMP. However, GMP remedies this omission by using the stack memory. If the overall performance does not satisfy the requirements, the user is allowed to do custom memory allocation.

Java BigInteger API is designed to meet object oriented programming criteria. There is no limitation or space preallocation for the numbers. JVM and its garbage collector determine the overall performance. When compared to C libraries, BigInteger API is slower; on the other hand, code development is far easier.

4.2.4 Code Readability

We have observed that there are three major code portability styles in the libraries mentioned above. In the first style; which is a naive approach, the architecture-depended code is blended together with the original one. They are separated with compile time pragmas. This approach is open to *spaghetti-like* coding. The second approach is to place architecture-depended code in separate files. This approach is used in GMP library. Since GMP is developed by collection of volunteer people, no code support problem arises. A third approach is to decouple architecture-depended codes via C macros. This approach is used partially in GMP. CRYMPIX's design is solely based on this above mentioned third approach.

4.3 Implementation of CRYMPIX

We have included some implementation notes in this section to clarify the scene that CRYMPIX is developed on. The code in Figure 4.2 introduces CRYMPIX with an integer addition example.

```
MI a, b, c;
crympix_init(100, 20);   // Max words, max instances.
...
a = mai_init();
b = mai_init();
c = mai_init();
...
mai_add(c, a, b);   // c = a + b.
...
mai_kill(a);
mai_kill(b);
mai_kill(c);
...
crympix_finalize();
```

Figure 4.2. CRYMPIX Code Example for Integer Addition.

4.3.1 Programming Language and Portability

We have already mentioned that ANSI C is selected as the programming language and explained the reasoning. Nevertheless, we have included a performance table that may give the reader an idea of how non-portable coding affects the performance. On Table 4.1, portable and non-portable versions of MIRACL 4.8 and GNU-GMP 4.1.4 are benchmarked via their integer multiplication function. We prepared test suits of 1K, 2K, 4K, and 8K each having 1000 pseudo-randomly selected inputs. We decoupled the I/O time to get more accurate results. All tests are done with GNU GCC compiler at optimization levels $O2$. The test is performed on a Intel Pentium IV 1400 Mhz processor Redhat 9.0 box.

We merely start the section with an specialized-code-support discussion to empha-

size that specialized processor support supplies only a constant speedup over portable versions that is the reasoning we build our library on the portable basis. In addition, one still has the chance of implementing the inner-most operations with machine dependent code to gain the necessary speedup. Figure 4.3 shows the performance ratio between the assembled version and the portable version for GNU-GMP and MIRACL libraries on Pentium like processors. Note that, this figure does not indicates any performance comparison between the libraries.

Table 4.1. Integer Multiplication benchmark results in microseconds.

Size	**MIRACL**		**GMP**	
	C	C+Asm	C	C+Asm
1Kb	17	6	23	4
2Kb	68	26	74	15
4Kb	277	104	235	47
8Kb	1097	411	731	154

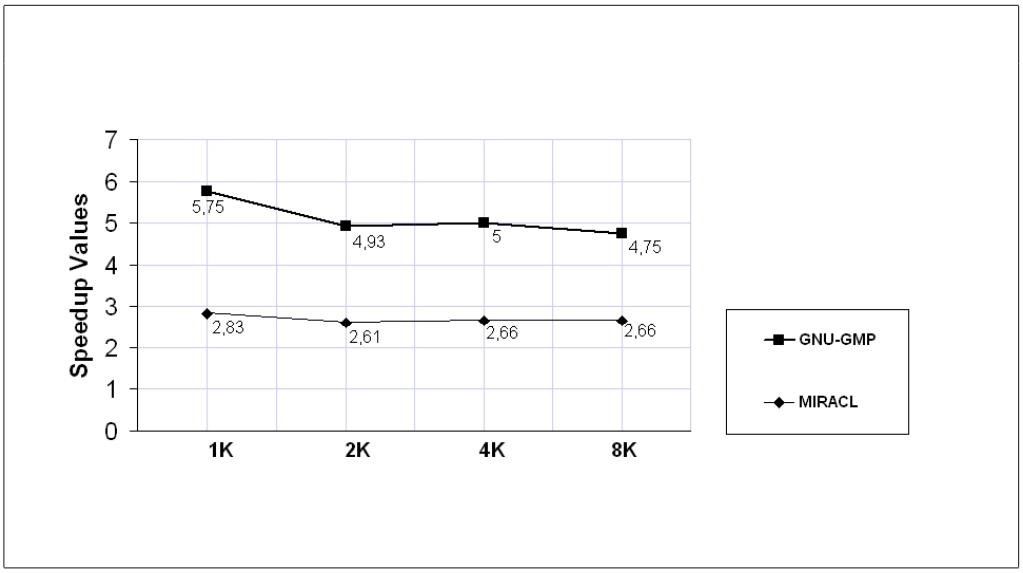

Figure 4.3. Speedup values obtained by the results in Table 4.1.

4.3.2 Representation of Numbers

Since a multiprecision number is composed of many computer words, the start and the end addresses should be known. Also the sign of the integer should also be stored. The integers are handled by the following structure shown in Figure 4.4.

```
typedef struct {
    POS l;  /* Number of digits */
    POS *n; /* Starting address of digits */
} MV_T, *MV;

typedef struct {
    POS t;  /* Type of the object */
    SIGN s; /* Sign of the integer */
    MV v;   /* Vector part of the integer */
} MI_T, *MI;
```

Figure 4.4. Structure for CRYMPIX integer.

The data type `POS` is actually the underlying hardware word that corresponds to `unsigned int` in GNU GCC compiler. `POS *num` is the starting address of the number and `POS len` is used to store the number of used words to store the whole number. `SIGN sign` represents the sign of the integer. The numbers is positive when `sign` is set to macro `POSITIVE` and negative for `NEGATIVE`. Variable `memid` tells the specialized kernel where the logical location of the number is and `type` is an unused future variable to indicate that type of the number. In our case, it is set to the paragma `INTEGER`. Note that a multiprecision CRYMPIX integer has the structural type `CZ` with `C` symbolizing CRYMPIX and `Z` indicating the numbers is in $\mathbb{Z}$. Please see Figure 4.5 in Section 4.3.3 for a better illustration of number representation.

4.3.3 Memory Management

CRYMPIX is designed to manage its own memory. Stack memory is not used for manipulating multiprecision numbers. The whole memory needed by the application is reserved by an initialization function. A tiny kernel supplies a fast memory allocation and release service on the preallocated space. The kernel uses a circular array data structure to speed up the allocation and release operations. Size of each number is fixed to prevent memory fragmentation. There is no built-in garbage collector mechanism in C so that programmer is responsible for the life cycle of each number. The scheme is illustrated in Figure 4.5.

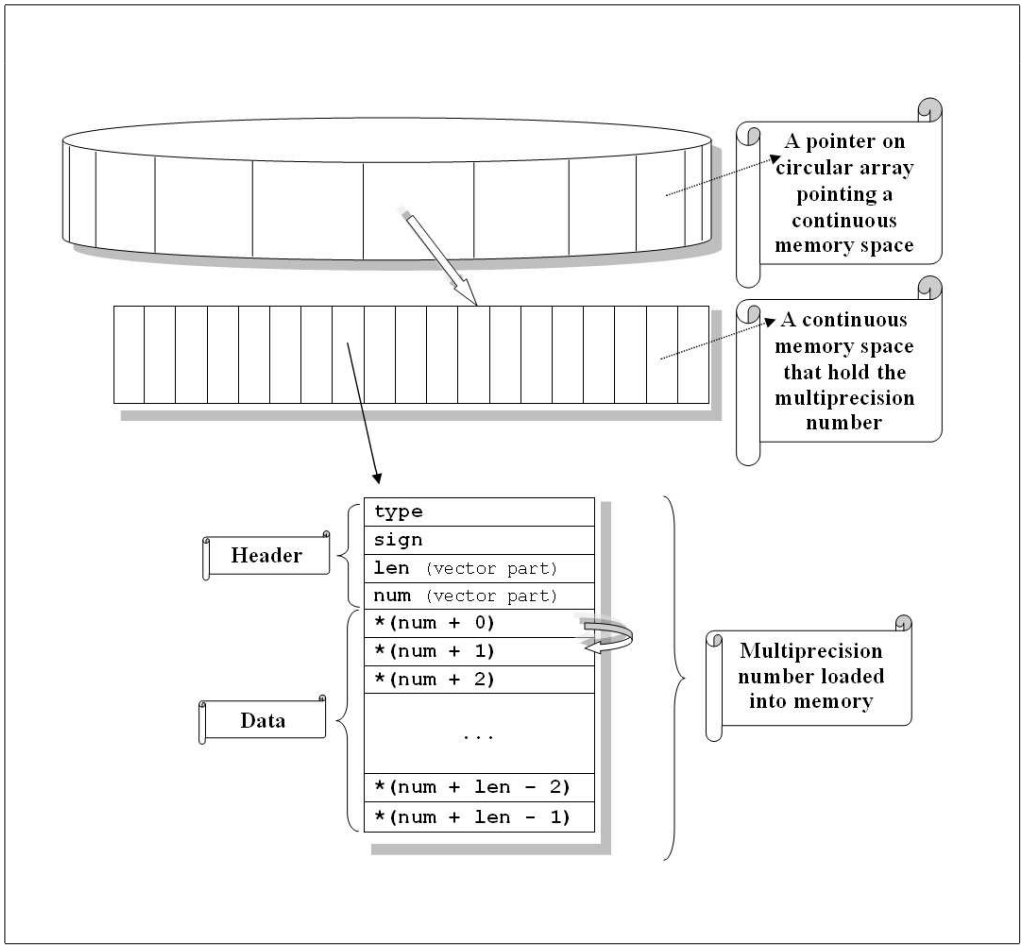

Figure 4.5. Manipulation of numbers in CRYMPIX.

Kernel controls a circular array of pointers that are linked to a predefined amount of continuous memory space. This space is used to store the multiprecision numbers. A multiprecision number is composed of two parts: header and data. Header holds necessary information about the structure of the multiprecision integer in terms of magnitude, and memory location. Note that header and data are located within the same continuous memory slot to fit the design to easy integration with Message Passing Interface (MPI) for Distributed API Layer. This approach is implemented as given in Figure 4.6.

```
MI mai_init(){
    MI new;
    new = (MI)k_init();
    new->t = TYPE_MI;
    new->s = POSITIVE;
    new->v = (MV)((CHAR *)new + sizeof(MI_T));
    new->v->l = 0;
    new->v->n = (POS *)((CHAR *)new->v + sizeof(MV_T));
    return new;
}
```

Figure 4.6. Initialization of multiprecision integer.

Here, `cz_init()` is a kernel level function that allocates the continuous memory slot. Header and data link is constructed with `new->num = (POS *)((CHAR *)new + sizeof(CZ_T))`.

4.3.4 Code Readability and Layered Approach

Code readability has been one of the major concerns in CRYMPIX library right from the start. Logical layers are decoupled by C macros at lower levels and by functions at the higher. Function bodies are written as plain as possible and the code organization,

standardized [1] naming, inline comments, and indentation are applied throughout the development. We will skip the details because we hope the code examples will clarify the relevant standards. Currently, CRYMPIX has 48 API functions and this number is expected to reach about 120 with its first release. CRYMPIX has 16.7 lines of code on average per function.

CRYMPIX is built on a layered approach with each layer having a distinct usage. The layers evolved after many design modifications and refactoring phases. The model is visualized in Figure 4.7. The layered approach simplifies the function bodies, prevents code repetitions; hence less tedious development phase.

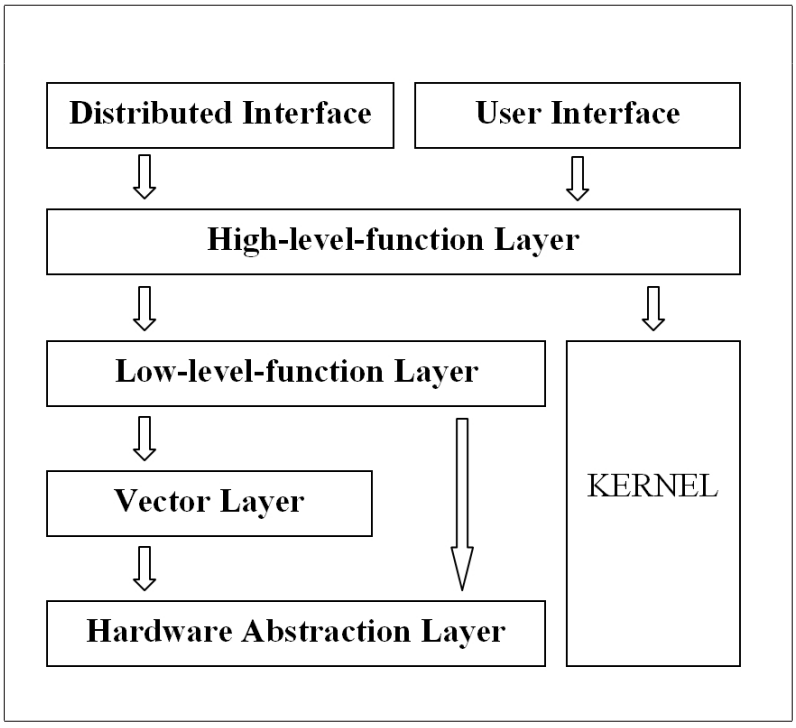

Figure 4.7. Representation of multiprecision numbers within the memory.

At the hardware abstraction layer (HAL), we handle single-precision arithmetic

[1]NASA C Style Guide, Software Engineering Laboratory Series, SEL-94-003, Goddard Space Flight Center, Greenbelt, Maryland 20771

operations with C macros on an imaginary double-precision processor which is capable of processing full-length single-precision multiplications. A possibly better approach is to supply this layer for a n-precision imaginary processor for a small value of n with the help of Comba-type multiplication. (See Section 3.1.2). We are currently adapting the mentioned technique to CRYMPIX. The below code in Figure 4.8 is an example macro which performs an inner product operation, $z = a \cdot b + z + carry$ that was explained in Section 3.1.1. The below code in Figure 4.8 is specialized for compilers that supports the double-digit data type namely `long long`. We skip the single-precision version because it occupies a large space for an example.

```
#define km_mul_2_add_2(_zz, _a, _b, _d_, _carry){ \
    _zz.dpu = ((DPU)_a * _b + _d_ + _carry); \
}
```

Figure 4.8. Hardware Abstraction Layer Code example.

A vector layer; which is on top of HAL, manipulates the operations between a positive integer array and a single-precision operand. The below code in Figure 4.9 provides an idea about the vector layer. This time the operation is to compute $zn = an \cdot b$ where zn and an are multiprecision numbers and b is a single-precision multiplier. Note that the code uses previously shown inner-product macro. Note that, the `DPUP` data type is supplied by the previously discussed imaginary processor.

```
#define maim_inc_n_mul_1(_carry, _zn, _an, _al, _b, _pad)if(1){ \
    DPUP _t; \
    POS _i; \
    \
    _t.spu[HIGH] = _pad; \
    for(_i = 0; _i < _al; _i++){ \
        km_mul_2_add_2(_t, _an[_i], _b, _zn[_i], _t.spu[HIGH]); \
        _zn[_i] = _t.spu[LOW]; \
    } \
    _carry = _t.spu[HIGH]; \
}
```

Figure 4.9. Vector Layer Code example.

At the low-level function layer which accesses both to the hardware abstraction layer and to the vector layer, implements the basic arithmetic, logic and bitwise functions. The term *low-level* is used to emphasize that the functions of this layer are provided with the simplified pointer access to multiprecision operands and each input is assumed to be correct by means of starting address and length. Thus, error handling is omitted. In the relevant code example in Figure 4.10, processes the low-level basecase multiplication operation, $z = a \cdot b$, for the multiprecision positive integers z, a, and b.

```
void main_mul_basecase(POS *z, POS *a, POS al, POS *b, POS bl){
    POS i;

    maim_mul_1(z[bl], z, b, bl, a[0], 0);
    for(i = 1; i < al; i++){
        maim_inc_n_mul_1(z[i + bl], (z + i), b, bl, a[i], 0);
    }
}
```

Figure 4.10. Low-level Function Layer Code example.

High-level function layer is responsible of preparing the operands to the low-level function layer. Error handling, sign and length management and proper selection of algorithms is done within this layer. The functions of this layer has slightly more lines of code. So that we provide a trimmed code sample of multiprecision multiplication in Figure 4.11. Interface Layer is a symbolic layer which separates the inner functions and the user level functions.

```
...
if(a == b){
    if(a->len < THRESHOLD_KARATSUBA_SQR){
        if(a->len < THRESHOLD_SQR){
            main_mul_basecase(r->num, a->num, al, a->num, al);
        }
        else{
            main_sqr_basecase(r->num, a->num, al);
        }
    }
    else if(a->len < THRESHOLD_TOOMCOOK3_SQR){
        while((al % 2) != 0){
            a->num[al] = 0;
            al++;
        }
        t = cz_init();
        main_sqr_karatsuba(r->num, a->num, al, t->num);
        main_kill(t);
    }
    else{
        while((al % 3) != 0){
            a->num[al] = 0;
            al++;
        }
        t = main_init();
        main_sqr_toomcook3(r->num, a->num, al, t->num);
        main_kill(t);
    }
}
...
```

Figure 4.11. High-level Function Layer Code example.

4.3.5 Implementation Details

This section provides implementation details of some primitives that are implemented in CRYMPIX. Instead of supplying a full documentation we merely give the significant part of the implementation which are mostly focused on multiplication and GCD computation. Thus, we simply skip the discussion of basic operations such as cloning, shifting, comparing, sign inversion, hamming weight, bit count and word count.

4.3.5.1 Addition and Subtraction

When adding two operands a and b, we can save some time if the result is stored in any of the input operands such as $a = a + b$. In such cases, CRYMPIX performs an accumulation operation given in Figure 4.12. The macro computes $zn = zn + an$ where zn is composed of zl words and an is represented by al computer words.

```
#define maim_inc_n(_carry, _zn, _zl, _an, _al, _pad)if(1){ \
    POS _i; \
    _carry = _pad; \
    for(_i = 0; _i < _al; _i++){ \
        km_inc_1(_carry, _zn[_i], _an[_i], _carry); \
    } \
    if(_zl > _al){ \
        maim_inc_1(_carry, (_zn + _al), (_zl - _al), _carry); \
    } \
}
```

Figure 4.12. Vector Layer Accumulation operation in CRYMPIX.

Addition is performed whenever the result is stored on a distinct instance such as $c = a + b$. The code sample is provided in Figure 4.13. Here, `chm_add_2(...)` is a HAL layer macro and `ccm_add_1(...)` is vector layer macro. Note that all operands are threatened as positive although they may not be. The sign management is controlled by the high-level function layer as provided in below Figure 4.14.

```
...
#define maim_add_1(_carry, _zn, _an, _al, _b)if(1){ \
    POS _i; \
    _zn[0] = _an[0] + _b; \
    _carry = (_zn[0] < _an[0]); \
    for(_i = 1; ((_i < _al) && (_carry != 0)); _i++){ \
        _carry = (_an[_i] == (0-1)); \
        _zn[_i] = 0; \
    } \
    mavm_clo((_zn + _i), (_an + _i), (_al - _i)); \
}

#define maim_add_n(_carry, _zn, _an, _al, _bn, _bl, _pad)if(1){ \
    POS _i; \
    _carry = _pad; \
    for(_i = 0; _i < _bl; _i++){ \
        km_add_1(_carry, _zn[_i], _an[_i], _bn[_i], _carry); \
    } \
    if(_al > _bl){ \
        maim_add_1(_carry, (_zn + _bl), (_an + _bl), (_al - _bl), _carry); \
    } \
}
...
```

Figure 4.13. Vector Layer Addition operation in CRYMPIX.

```
void mai_add(MI z, MI a, MI b){
    VALIDATE(z != NULL);
    VALIDATE(a != NULL);
    VALIDATE(b != NULL);
    if(a == b){
        if(z == a){
            maim_inc_n(z->v->n[a->v->l],a->v->n,a->v->l,a->v->n,a->v->l,0);
        }
        else{
            maim_add_n(z->v->n[a->v->l],z->v->n,a->v->n,a->v->l,a->v->n,a->v->l,0);
        }
    }
    else{
        if(mai_compare_abs(a, b) == LESS){
            MI_SWAP(a, b);
        }
        if(z == a){
            if(a->s == b->s){
                maim_inc_n(z->v->n[a->v->l],a->v->n,a->v->l,b->v->n,b->v->l,0);
            }
            else{
                maim_dec_n(z->v->n[a->v->l],a->v->n,a->v->l,b->v->n,b->v->l,0);
            }
        }
        else{
            if(a->s == b->s){
                maim_add_n(z->v->n[a->v->l],z->v->n,a->v->n,a->v->l,b->v->n,b->v->l,0);
            }
            else{
                maim_sub_n(z->v->n[a->v->l],z->v->n,a->v->n,a->v->l,b->v->n,b->v->l,0);
            }
        }
    }
    if(z->v->n[a->v->l] == 0){
        z->v->l = a->v->l;
    }
    else{
        z->v->l = a->v->l + 1;
    }
    z->s = a->s;
}
```

Figure 4.14. High-level Function Layer Addition operation in CRYMPIX.

4.3.5.2 Multiplication

In the cryptographic applications Basecase and Karatsuba multiplication algorithms are frequently used. CRYMPIX implements both algorithms. We have already gave basecase multiplication in Figure 4.10. Note that the implementation does not start with an initially cleared product space. Instead, the product z is assigned to the result of first vector multiplication by saving one iteration. This approach is not important for larger numbers but can be useful when small operands are in use. Figure 4.15 provides the performance comparison with the naive approach. The speedup values are obtained from Table 4.2.

Table 4.2. Comparison of CRYMPIX's implementation of Basecase multiplication with naive approach in small length operands. (microseconds).

Length (bits)	**64**	**128**	**256**	**512**
Basecase Mul.	181	392	1153	4175
Improved Basecase Mul.	213	447	1277	4425

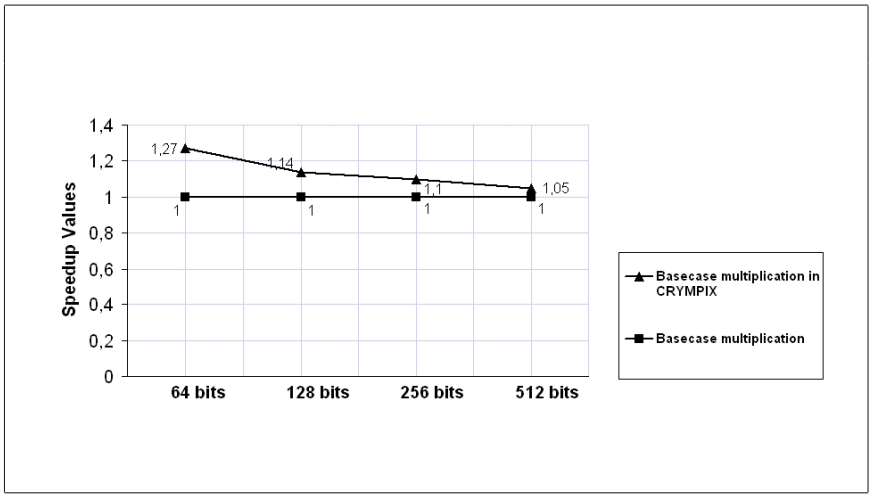

Figure 4.15. Speedup values of CRYMPIX's basecase multiplication over the naive approach.

We provide the full source of Karatsuba multiplication in Figure 4.16. The most important implementation trick is the use of variables z and t. These memory spaces are used in partitioned form as a consequence of the recursion. Note that the algorithm switches to basecase multiplication at a predefined compile time threshold value. This is the tuning point where basecase multiplication is carried out faster than Karatsuba multiplication. The implementation of ToomCook multiplication is left as a future work in CRYMPIX since it is only significant for 8K operands.

```
void main_mul_karatsuba(POS *z, POS *a, POS al, POS *b, POS bl, POS *t){
    POS ca, cb, ct, c, abl;

    if((al < THRESHOLD_KARATSUBA_MUL) || (bl < THRESHOLD_KARATSUBA_MUL)){
        main_mul_basecase(z, a, al, b, bl);
    }
    else{
        al >>= 1;
        bl >>= 1;
        abl = (al + bl);
        maim_add_n(ca, z, a, al, (a + al), al, 0); /* (ca,zL) = aL + aH */
        maim_add_n(cb, (z + al), b, bl, (b + bl), bl, 0); /* (cb,zH) = bL + bH */
        main_mul_karatsuba(t, z, al, (z + al), bl, (t + abl)); /* t = zL * zH */
        c = (ca & cb);
        if(ca == 1){
            maim_inc_n(ct, (t + bl), al, (z + al), bl, 0);
            c += ct;
        }
        if(cb == 1){
            maim_inc_n(ct, (t + al), bl, z, al, 0);
            c += ct;
        }
        main_mul_karatsuba(z, a, al, b, bl, (t + abl)); /* zL = aL * bL */
        main_mul_karatsuba((z + abl), (a + al), al, (b + bl), bl, (t + abl));
        maim_dec_n(ct, t, abl, z, abl, 0); /* t = t - zL */
        c -= ct;
        maim_dec_n(ct, t, abl, (z + abl), abl, 0); /* t = t - zH */
        c -= ct;
        maim_inc_n(ct, (z + (abl >> 1)), (abl + (abl >> 1)), t, abl, 0);
        c += ct;
        maim_inc_n(ct, (z + abl + (abl >> 1)), (abl >> 1), (&c), 1, 0);
    }
}
```

Figure 4.16. Karatsuba multiplication in CRYMPIX.

4.3.5.3 Greatest Common Divisor

The basic algorithm for GCD computation is Euclid's algorithm with $O(n^2)$ complexity. The algorithm is modified by Lehmer to fit the fixed-precision processors. Another method of GCD computation is the Binary GCD algorithm. This algorithm is faster when the numbers are few words long. For larger numbers Binary GCD algorithm is modified by many researchers. Jebelean and Weber proposed Accelerated/Generalized GCD algorithm which is faster than Lehmer GCD algorithm (Jebelean 1993b, Weber 1995) by a factor of 1,45. CRYMPIX includes a slightly modified version of Lehmer GCD algorithm. It is used both for GCD and Extended GCD computations. We have provided a comparison between Lehmer GCD algorithm and its modified variant proposed in (Jebelean 1993a). We have used approximative condition of GCD and double-precision techniques. The speedup values of Table 4.3 are given in Figure 4.17.

Table 4.3. The time needed to compute GCD of two operands with Standard Lehmer GCD algorithm and the with the modified version. (microseconds).

Length (bits)	**1K**	**2K**	**4K**	**8K**
Standard Lehmer	201	557	1746	6228
Modified Lehmer	158	351	921	3131

The algorithm has been already discussed in Algorithm 3.4. Therefore, we will only provide the approximation phase in Figure 4.18 that is replaced with bit nailing. Here, the most significant 3 words of the operands are divided by the most significant word of the operand x which gives a better chance to form larger coefficient before each multiplication step. Note that the approximative condition is done with a double-precision approach which further increases the coefficient sizes. Table 4.3 and Figure 4.17 supplies necessary information that delineates the performance improvement of double-digit approximative condition over the naive approach.

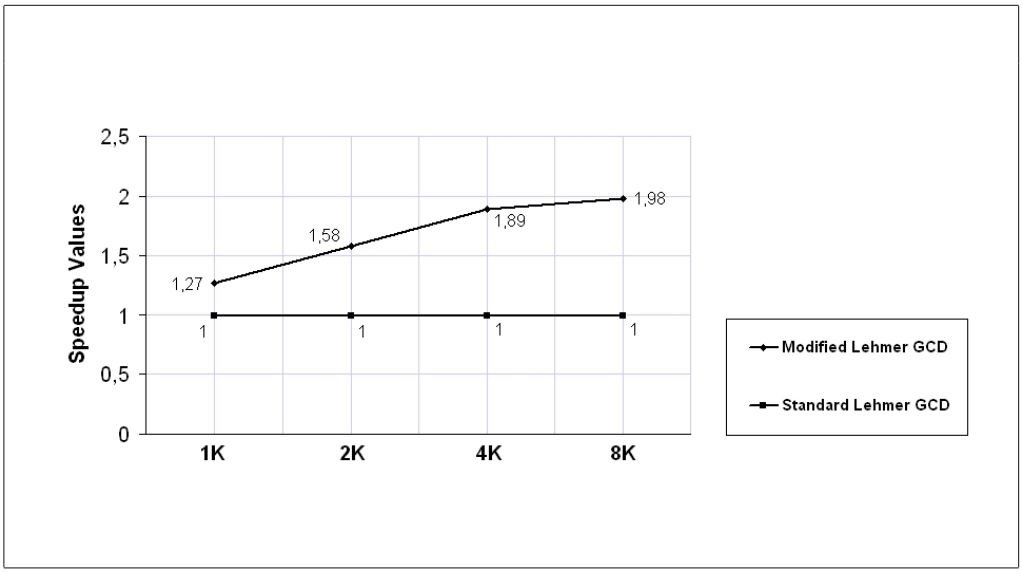

Figure 4.17. Speedup values of Modified Lehmer GCD algorithm over Standard version.

```
...
if(xl > THRESHOLD_GCD_LEHMER_LONG_USW){
    dp = xn[xl - 1] + 1;
    dp += (dp == 0); /* dp is set to 1 if there is no need to approximation. */
    /* u = x[3,2,1] / x[3] */
    km_q_and_r(ua.spu[HIGH], s.spu[HIGH], xn[xl - 1], xn[xl - 2], dp);
    s.spu[LOW] = xn[xl - 3];
    km_div(ua.spu[LOW], s.dpu, dp);
    /* v = y[3,2,1] / x[3] */
    km_q_and_r(va.spu[HIGH], s.spu[HIGH], yn[xl - 1], yn[xl - 2], dp);
    s.spu[LOW] = yn[xl - 3];
    km_div(va.spu[LOW], s.dpu, dp);
}
...
```

Figure 4.18. Quotient approximation in Lehmer GCD implementation.

4.3.5.4 Modular Exponentiation

Several different techniques for modular exponentiation has been discussed in Section 3.3. CRYMPIX uses successive squaring algorithm with left-to-right exponent scanning and variable-length-window-sliding technique with variable window size and Montgomery's multiplication via Basecase REDC function with Karatsuba multiplication. Recursive REDC implementation is left as a future study. Figure 4.19 provides the CRYMPIX's implementation of window sliding technique.

```
...
while(i > 0){
    mavm_ith_bit(ei, e->v->n, (i - 1));
    if(ei == 0){
        mfp_nres_mul(xd, xd, xd, nd, n);
        i--;
    }
    else{
        li = mfpn_find_window(e->v->n, (i - 1), ws);
        k = 0;
        lj = li;
        while(lj != 0){
            mfp_nres_mul(xd, xd, xd, nd, n);
            lj >>= 1;
            k++;
        }
        t = (MI)cds_array_get(lookup, li >> 1);
        mfp_nres_mul(xd, t, xd, nd, n);
        i -= k;
    }
}
...
```

Figure 4.19. Implementation of sliding windows technique.

4.4 Distributed Architecture

A distinctive feature of CRYMPIX is its inclination to distributed computing. In most distributed cryptographic implementations, developers blend the cryptographic codes with the ones needed for distribution. As a consequence, the application becomes hard to handle in terms of code readability and support. In addition, the development effort is repeated for every single implementation. CRYMPIX is designed from the scratch to solve this problem by providing an easy scalable distribution mechanism and decouple cryptographic functions from distribution functions.

The parallelization of a task can be achieved in several ways. For instance, special hardware for systolic parallelization is used in Binary GCD variants to speedup the operation which is directly proportional the number of supplied arithmetic units. Hence, a single GCD computation halts n times faster for a given n unit parallel environment. This concept is also known as perfect parallelism [2]. In distributed environments cryptographic functions benefit perfect parallelism.

All distribution functions are included in distributed layer of CRYMPIX. This layer directly/only accesses to the high level layer functions and distributed the computational mass over an MPI network. In this study we merely work on the several modular exponentiations that is used in many asymmetric cryptosystems. Any further implementations are left as future work. We observe that network overhead is negligible at all common key sizes. The experiment environment is constructed by 8 identical PCs each having a Pentium IV 2.4 Ghz processor and 512 MB of RAM. All computers are connected via a 100 Mbps Ethernet LAN using a switch. Operating system is Debian Linux with 2.4.02 kernel. Distributed platform is constructed with MPI (Message Passing Interface) v.1.2.6. The test bed consists of 64 * 1K, 32 * 2K, 16 * 4K and 8 * 8K numbers. I/O time for reading those numbers from the disk is completely discounted (not added to the results). Scattering, gathering and execution time is measured distinctly where the execution time stands for simultaneous modular exponentiations. The model is summarized in Figure 4.20.

[2]Such tasks are also known as embarrassingly parallel that means actually there is no special effort needed to partition the problem into smaller tasks.

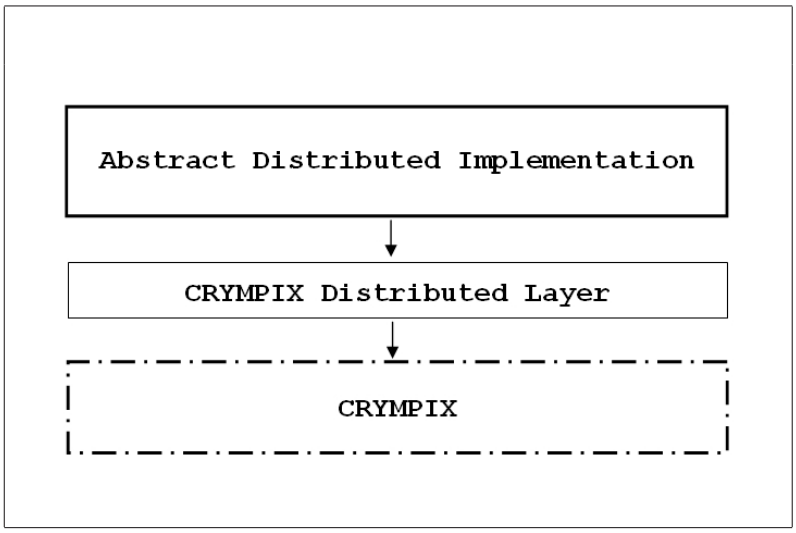

Figure 4.20. Distributed Wrapper for CRYMPIX.

We have provided a distributed formation of modular exponentiation within the distributed layer. Note that, CRYMPIX uses successive squaring algorithm with left-to-right exponent scanning and variable-length-window-sliding technique with variable window size and Montgomerys multiplication with Karatsuba algorithm to compute modular exponentiations. This is a naive case study just to give the basic idea. We warn the reader that the real life implementations will be composed of more complex scenarios. For instance, matrix solving is used in Discrete Logarithm Attacks and a gauss elimination of a extremely sparse matrix of integers in a modular base will contain many group communications within the distributed layer. Therefore there perfect parallelism may not possible for all scenarios. The pseudocode of several ordinary modular exponentiations is given in Figure 4.21.

The software model in Figure 4.20 can be extended for other multiprecision libraries because the inner representation is nearly the same and functions of distributed layer calls other libraries' functions via some adapter sublayers. The idea is visualized in Figure 4.22.

```
procedure mod_exp_n(z[], M[], e[], n[], len){
    Scatter each array depending on len.
    Convert z[i], m[i], and n[i] to nresidue form.
    Call nresidue_mod_exp_n(z[i], M[i], e[i], n[i]).
    Convert z[i], m[i], and n[i] to normal form.
    Gather results.
    return.
}
```

Figure 4.21. A case study in CRYMPIX distributed layer.

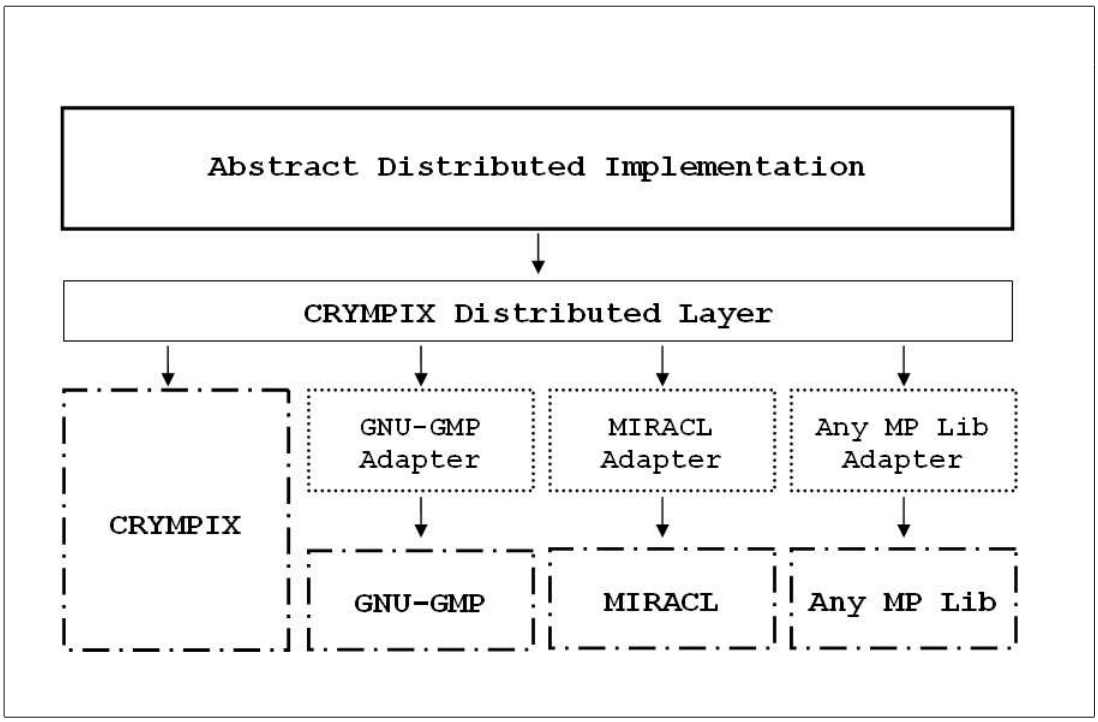

Figure 4.22. Distributed wrappers for several libraries.

We repeat the experiment for 1, 2, 4, and 8 computers with the same setup. The outcome of the experiments approves the perfect parallelism. The speedup is obvious hence we only give the performance results that are provided in Table 4.4. Note that scatter and gather times are nearly zero that the values only represent the execution times.

Table 4.4. Distributed layer test case results. (milliseconds).

# of PCs	64 * 1K	32 * 4K	16 * 2K	8 * 8K
1 PCs	1788	6407	24931	92020
2 PCs	912	3284	12661	45965
4 PCs	472	1654	6233	22820
8 PCs	223	793	3118	11645

4.5 Benchmark of CRYMPIX

We have already discussed that the main requirement of multiprecision libraries is the performance and stated that all decisions are made to maintain the maximum performance. Note that CRYMPIX is a C only library so that it does not really utilize the underlying hardware. However, we have also explained the reasoning behind this decision. In this chapter, we provide the performance benchmark of CRYMPIX that makes everything meaningful. We also include fair comparisons of the measurements with C-builds of GNU-GMP and MIRACL. The section is composed of benchmark of multiplication, GCD, and modular exponentiation functions. We skip all other functions since they are already used within given functions in question and most of them such as addition, runs in $O(n)$ time. Therefore, there is no need to do any performance test within our scope.

We should also state that a completely fair comparison between libraries is not possible all the time because libraries implements different algorithms even with different modifications. Thus, we only include the functions of the latest versions.

In the following experiments, MIRACL 4.8, GMP 4.1.4, and CRYMPIX are benchmarked via their most important functions. For multiplication function we also included Java BigInteger library. We decoupled the I/O time to get more accurate results. Excluding Java BigInteger API, all tests are done with GNU GCC compiler at optimization levels O0, O1, and O2. The whole test is repeated on Intel Centrino M 1400 Mhz, Intel P4 1700 Mhz, and IBM RISC RS/6000 133 Mhz processors with no options on memory. As an operating system we used YellowDog 2.3 Linux on IBM RISC RS/6000 machine and Redhat Linux 9.0 and Microsoft Windows XP/SP2 on Intel machines. To port GNU GCC compiler to Windows we used CYGWIN platform. Java BigInteger benchmark is done on Java Virtual Machine (JVM) of Sun Microsystems, Inc., Java2 Standard Development Kit (J2SDK) v1.4.2 and applied on Intel boxes and on both Redhat Linux and Microsoft Windows XP. The whole measurements have provided us with so much data and since the speedup values are nearly constant we give results of only Intel Centrino M 1400 MHz processor with Redhat Linux operating system. The above defined test environment is used throughout this study.

4.5.1 Multiplication

We prepared test beds of various sized operands, 1K, 2K, 4K, and 8K, each having 1000 randomly selected inputs. ISO C'99 standard has introduced a new data type, namely *long long*, which enabled full length single-precision multiplication with C language. CRYMPIX v2 and MIRACL takes the advantage of the new double-precision data type. CRYMPIX v1 and GMP don't use this facility. What separates CRYMPIX v1 and CRYMPIX v2 is a simple compile time macro. We merely include this feature to do fair comparisons with the other libraries. The benchmark results of multiplication functions are provided in Figure 4.5 and the speedup diagram is given in Figure 4.23.

Table 4.5. Integer Multiplication benchmark results in microseconds.

Size	**CRYMPIX**		**MIRACL**	**GMP**	**Java BigInteger**
	C, v1	C, v2	C	C	
1Kb	21	11	17	23	32
2Kb	69	41	68	74	132
4Kb	219	133	277	235	512
8Kb	673	410	1097	731	2630

Figure 4.23 indicates that CRYMPIX is competitive on all test beds. MIRACL has an embedded Karatsuba/Comb routine but it is used for more costly operations such as modular exponentiation, thus it is relatively slower in this experiment. The overall performance of Java BigInteger API varies with respect to the JVM but this library is slower in all circumstances and it is developed with the basecase algorithms in most cases. On the other hand, it is far easier to develop applications on such an object oriented environment. We used this library only to generate the test beds data. In Figure 4.23 we

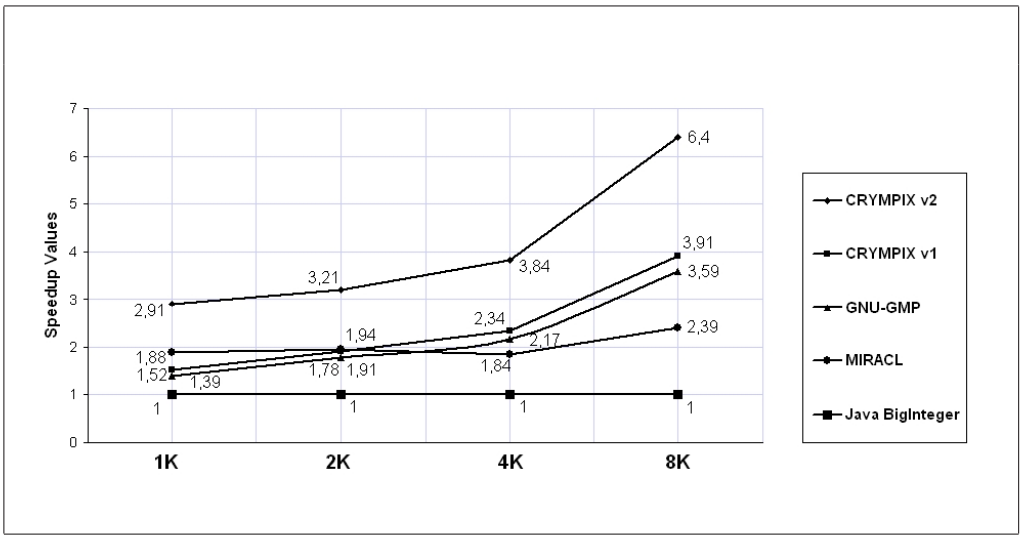

Figure 4.23. Speedup values when assembly support is used.

have provided the performance comparison of libraries for C only built at optimization level 2 (excluding Java BigInteger).

4.5.2 Greatest Common Divisor (GCD)

We provide the performance comparison of CRYMPIX Lehmer GCD and GMP Generalized GCD in Table 4.6. The expected value is a constant speedup around $0,75$ which is actually a slow down factor for CRYMPIX. This is because GMP performs 3 one-digit multiprecision multiplication at each step as a consequence of modular conjugation that is coupled with a *bmod* operation. CRYMPIX's Lehmer GCD function has to do 4 such multiplications to lower the operands by one computer word. This is depicted in Figure 4.24. CRYMPIX is also slower on smaller operands because it hasn't contained bit level Binary GCD yet. CRYMPIX v2 takes the advantage of full length single-precision multiplication. MIRACL's GCD function is a single-digit approximative Lehmer variant. It is relatively slower on all test beds that we do not include values for MIRACL here. Generalized GCD implementation and its distributed version is left as a future task in CRYMPIX.

Table 4.6. CRYMPIX Lehmer GCD vs. GMP Generalized GCD. (microseconds).

Length	1K	2K	4K	8K	16K
CRYMPIX v1 GCD	186	474	1372	4449	15767
CRYMPIX v2 GCD	157	368	957	2802	9161
GNU-GMP GCD	88	266	874	3101	11592

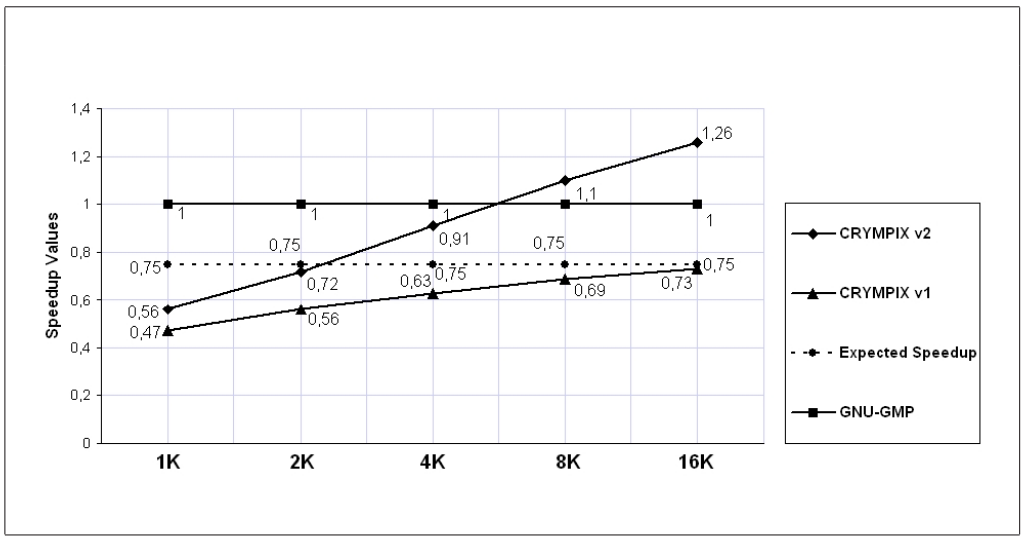

Figure 4.24. Speedup values for CRYMPIX Lehmer GCD over GMP Generalized GCD, derived from Table 4.6.

4.5.3 Modular Exponentiation

Modular exponentiation is the most expensive operation among the other multi-precision operations. A competitive implementation takes the advantage of almost all techniques to speedup the operation. CRYMPIX uses successive squaring algorithm with left-to-right exponent scanning and variable-length-window-sliding technique with variable window size and Montgomery's multiplication with Karatsuba algorithm. MIRACL-KCM is the generated code for embedded systems. The speed underlying MIRACL-KCM references from the recursive implementation of Montgomery REDC function with half

multiplication technique. In the 8K test bed, GMP triggers Toom-Cook-3-way multiplication hence all speedup values tend to decrease in 8K test bed. CRYMPIX will be updated to benefit such techniques in the future. We constructed Table 6 with time measurements of modular powering for 1K, 2K, 4K and 8K numbers with GMP, CRYMPIX, and MIRACL. Figure 4.25 provides corresponding speedup values.

Table 4.7. Modular exponentiation for GMP, CRYMPIX, and MIRACL. (milliseconds).

Length	**1K**	**2K**	**4K**	**8K**
GMP Mod. Exp.	54	389	2841	16734
MIRACL-KCM Mod. Exp.	31	204	1298	8132
CRYMPIX v1 Mod. Exp.	49	363	2650	19526
CRYMPIX v2 Mod. Exp.	27	195	1423	10411

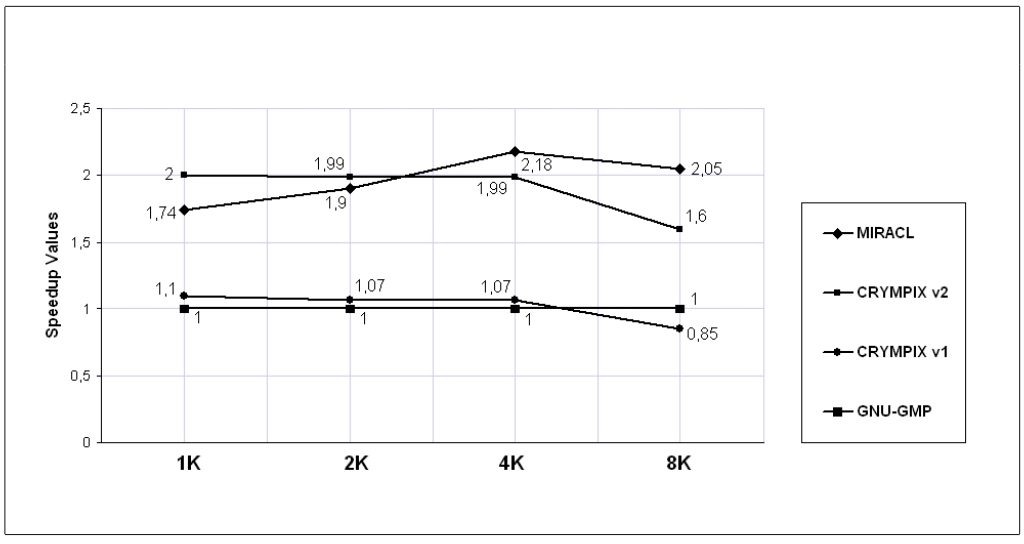

Figure 4.25. Speedup values for CRYMPIX and MIRACL over GMP in modular exponentiation, derived from Table 4.7.

CHAPTER 5

CONCLUSION

In this study, we aim to review multiprecision concept in detail. To gain the know-how, we have introduced a new cryptographic multiprecision library, CRYMPIX. We also provided a fair performance comparison between some libraries by providing technical comments. CRYMPIX which is developed in ANSI C, is able to take the advantage of `long long` data type of ISO C'99 whenever possible. CRYMPIX includes low level routines for multiprecision arithmetic in prime fields. The overall performance of CRYMPIX is equal to its predecessors and in some instances even superior.

A typical challenge is that struggles a researcher is what library to use. All competing multiprecision libraries have distinct powerful sides which makes it hard to select and even tedious. For instance, our benchmark results showed that GNU-GMP is very fast on GCD computation, on the other, MIRACL is the fastest in modular powering. Furthermore, each of these libraries are being developed that this scheme may vary in time. Therefore, a researcher should at least be aware of the algorithms behind the scene of multiprecision libraries to be able to decide which library best fits to the specific research topic. We still face another problem that is whether a selected fastest function can be further utilized. Therefore, one should know low level operations such as core inline assembly tricks and/or compiler supported double-precision operations.

The mathematical background that is supplied in this thesis is far ahead the current implementation of CRYMPIX. Thus, we leave some functions such some jacobian symbol, square root, and radix conversion functions as future study. For instance, CRYMPIX is supposed to provide Binary GCD, Binary Extended GCD, Generalized GCD, and fully-recursive REDC functions. Furthermore, the real life implementations of cryptosystems benefits some other optimizations such as fixed based and fixed exponent exponentiation techniques. CRYMPIX is going to include all such methods within its first release.

CRYMPIX is built on an imaginary processor. We should supply a optimal n-precision imaginary processor taking the advantage of Comba multipliers to further utilize the underlying hardware. As a consequence, no ANSI-C arithmetic operators such be used

in multiprecision operations on low-level and high-level function layers. This approach may even lead to a specialized crypto-compiler based on ANSI-C standard.

The model of the distributed layer of CRYMPIX is made clear. However, we only provide a test scenario of several ordinary exponentiations to show its applicability. The distributed layer should contain all key functions related to cryptography even to cryptanalysis.

In conclusion, the first release of CRYMPIX is expected to include all functions significant for cryptography. Support for specific processors is not in the short term schedule. After the first stable release, the project is going to be extended over binary field arithmetic.

REFERENCES

Bosselaers A., Govaerts R., and Vandewalle J. 1994a. "A fast and flexible software library for large integer arithmetic", In proceedings 15th Symposium on Information Theory in the Benelux, Louvain-la-Neuve (B), London, UK, pp. 82–89.

Bosselaers A., Govaerts R., and Vandewalle J. 1994b. "Comparison of three modular reduction functions", In CRYPTO '93: Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology, London, UK, Springer-Verlag, ISBN 3-540-57766-1, pp. 175–186.

Burnikel C. and Ziegler J. 1998. "Fast recursive division", Technical Report, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1998, MPI-I-98-1-022.

Comba P.G. 1990. "Exponentiation cryptosystems on the IBM PC", *IBM Systems Journal*, Vol. 29, No. 4, pp. 526–538.

Dussé S.R. and Kaliski Jr.B.S. 1991. "A Cryptographic Library for the Motorola DSP560001", In I. Damgård, editor, Proc. Advances in Cryptology - EUROCRYPT '90, New York, Springer Verlag, Vol. 473 of *Lecture Notes in Computer Sciences*, pp. 230–244.

Jebelean T. 1993a. "Improving the multiprecision euclidian algorithm", In DISCO '93: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, , London, UK, Springer-Verlag, ISBN 3-540-57235-X, pp. 45–58.

Jebelean T. 1993. "A generalization of the binary gcd algorithm", In ISSAC '93: Proceedings of the International Symposium on Symbolic and Algebraic Computation,

New York, NY, USA, ACM Press, ISBN 0-89791-604-2, pp. 111–116.

Karatsuba A. and Ofman Y. 1962. "Multiplication of many-digital numbers by automatic computers", *Doklady Akad*, Vol.145, No. 14-15, pp. 293–294.

Knuth D.E. 1997. "The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201896842.

Koc C. 1994. "High-Speed RSA Implementation", 100 Marine Parkway, Suite 500 Redwood City, CA 94065-1031, Technical Report TR 201.

Menezes A.J., Vanstone S.A., Oorschot P.C. 1996. "Handbook of Applied Cryptography", CRC Press, Inc., Boca Raton, FL, USA, ISBN 0849385237.

Montgomery P. 1985. "Modular Multiplication without Trial Division", *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521.

Mulders T. 2000. "On Short Multiplications and Divisions". *AAECC*, Vol. 11 No. 1 pp. 69–88.

Rosen K.H. 1998. "Discrete Mathematics And Its Applications", McGraw-Hill Science/Engineering/Math, Boston, MA, USA, 4th edition edition, ISBN 0072899050.

Wagstaff S. 2002. "Cryptanalysis of Number Theoretic Ciphers", CRC Press, Inc., Boca Raton, FL, USA, ISBN 1584881534.

Weber K. 1995. “The Accelerated Integer GCD Algorithm”. *ACM Trans. Math. Softw.*, ISSN 0098-3500, Vol. 21 No. 1 pp. 111–122.