

## 1 Introduction

Cryptography is becoming a fundamental building block to achieve security in data and telecommunications networks. It makes electronic commerce, payment systems, and transactions over open networks possible. It has become one of the main tools for privacy, trust, access control, corporate security, and countless other areas.

The most basic problem of cryptography is to secure communication over a public channel. The solution of this problem is a traditional form of cryptography called Secret Key Cryptography.

In Secret Key Cryptography, one secret key is shared between communicating parties, the same key is used for encryption and decryption. The cryptographic strength of such a system depends on the secrecy of the key since anyone knowing the key or with the ability to guess the key can decrypt the message. In most cases, the larger the key length, the harder it is to guess the key using exhaustive key search.

The biggest difficulty with this form of cryptography is the distribution of the key, without a secure key exchange mechanism the security of Secret Key Cryptography is compromised no matter how strong its algorithms and keys are.

In the mid-70s, a new form of cryptography was introduced called Public Key Cryptography as a practical solution for the key exchange and digital signature problems.

Public key cryptography is based on mathematical functions and relies on two mathematically related keys, encryption key made public and so called public key, and a

secret decryption key called private key. The security of the system relays on the difficulty of deriving the private key given knowledge of the public key and the cryptographic algorithm.

In order to assure the required level of security, mathematical functions used in Public Key Cryptography require operations on large integers of the size from 768 to 2048 bits, and elements of finite groups with element sizes in the range from 140 to 240 bits. Software implementations of such arithmetic operations are difficult since currently available processors have word size up to 64 bit.

Multiple algorithms have been developed to do multi-precision arithmetic operations efficiently, and several libraries implementing such algorithms exist both commercially and in the public domain, but so far no study has been done to compare and contest such libraries against each other.

This is the first study of such multi-precision arithmetic libraries using comprehensive criteria such as performance, support for public key primitive operations, ease of use and portability. The aim of this study is to evaluate the suitability of using the libraries for implementations of a wide range of Public Key Cryptosystems. In order to achieve such evaluation, a novel methodology for ranking the performance of the libraries is developed based on the performance of their primitive cryptographic operations.

I hope that this study will help developers of Public Key software implementations to make correct choices regarding the use of available libraries in their products based on knowledge of existing tradeoffs.

## 2 Background

### 2.1 Public Key Cryptosystems

The concept of Public Key cryptosystems was first introduced by Whitfield Diffie and Martin Hellman [6] in 1976. Their revolutionary idea proposed a solution to the key-exchange problem which led to the birth of Public Key cryptosystems.

Public Key cryptosystems rely on a pair of different keys, the Public Key and the Private Key. The Public Key is published, while the Private Key is kept secret. Any one with a Public Key can encrypt a message but only the person with the Private Key can decrypt.

In such a system the Private Key is always linked mathematically to the Public Key. Therefore, it is always possible to attack a Public Key system by attempting to derive the Private Key from the Public Key. Thus, it must be computationally infeasible to determine the Private Key given only the cryptographic algorithm and the Public Key.

Based on such characteristic, Public Key cryptosystems are categorized based on their security dependence on special mathematical problems called one-way functions which are easy to compute and hard to invert.

Table 1 lists three of the main mathematical problems (one-way functions) that are the basis of three categories of Public Key cryptosystems.

---

**Table 1:** Three Mathematical bases for Public Key Cryptosystems

	IFP <sup>(1)</sup>	DLP <sup>(2)</sup>		ECDLP <sup>(5)</sup>
		TDLP <sup>(3)</sup>	SDLP <sup>(4)</sup>	
Given	$N = p \cdot q$ $p$ and $q$ are prime	$y = g^x \text{ mod } p$ $p$ is a prime		$Q = kP$ $P$ is a point on $E^{(6)}$
Unknown	$p$ and $q$	$x$		$k$
Difficulty	computing $p$ and $q$ , with knowledge of $N$	computing $x$ , with knowledge of $y$ , $g$ and $p$		computing $x$ , with knowledge of $Q$ , $P$ and $E$

- (1) IFP: Integer Factorization Problem.  
(2) DLP: Discrete Logarithm Problem.  
(3) TDLP: Traditional Discrete Logarithm Problem.  
(4) SDLP: Special Discrete Logarithm Problem.  
(5) ECDLP: Elliptic Curve Discrete Logarithm Problem.  
(6)  $E$ : Elliptic curve.
- 

Integer Factorization Problem IFP: Computing  $N$  is easy since the multiplication of  $p$  and  $q$  can be done in polynomial time; computing the inverse i.e. factoring  $N$  in to its original primes, is extremely difficult and known as the Integer Factorization Problem.

Discrete Logarithm Problem: Computing  $y$  is a modular exponentiation that can be done in polynomial time; computing  $x$  from the knowledge of  $y$ ,  $g$  and  $p$  is extremely difficult and known as the Discrete Logarithm Problem. DLP can be further classified into two types, Traditional DLP where  $g$  is a generator of the prime field, and Subgroup DLP where  $g$  is a generator of a relatively small but sufficiently large subgroup of the prime field.

Elliptic Curve Discrete Logarithm Problem ECDLP: Computing  $Q$  is a scalar multiplication and can be done in polynomial time; computing  $k$  from the knowledge of  $Q, P$ , and  $E$  is extremely difficult and known as the Elliptic Curve Discrete Logarithm Problem.

Examples of cryptosystems from each category are listed in Table 2.

**Table 2:** Public Key Cryptosystems

	RSA	DSA/DH/ElGamal		ECDSA, ECDH, EC-ElGamal
Category	IFP	DLP		ECDLP
		TDLP	SDLP	
System Parameters	N/A	a prime modulus $p$ $g$ a generator of the prime field	a prime modulus $p^{(1)}$ , a prime divisor of $p-1$ , $q^{(2)}$ . $g$ a generator of a subgroup of order $q$	$GF(q^{(3)})$ , $E^{(4)}$ , a generator $G$ on $E$ and the order of $G$ ( $r$ )
Public Key	modulus $N$ and exponent $e$	$y = g^x \text{ mod } p$		point $P = kG$ on $E$
Private Key	exponent $d$	an integer $x$ where $0 < x < p-1$	an integer $x$ where $0 < x < q$	an integer $k$ where $0 < k < r$
Security Parameter	size of $N$	size of $p$	size of $p$ , size of $q$	size of $r$
Key Size	size of $N$	Size of $p$	size of $p$ , size of $q$	size of $q$
Usage	DS <sup>(5)</sup> , ENC/DEC <sup>(6)</sup> , Key exchange	ElGamal: ENC/DEC DH: key exchange	DSA: DS	ECDSA : DS ECDH: key exchange EC-ElGamal: ENC/DEC

(1)  $2^{L-1} < p < 2^L$  for  $512 < L < 1024$  and  $L$  a multiple of 64

(2)  $2^{159} < q < 2^{160}$

(3)  $q = \text{prime}$  or  $q = 2^m$

(4)  $E$  : Elliptic curve

(5) DS: Digital signature

(6) ENC/DEC: Encryption/Decryption

### 2.1.1 Security Parameters and Key Sizes

The best known attack on each cryptosystem requires an amount of computations proportional to the security parameter (Table 2) which is directly related to the underlying mathematical problem e.g. the larger the size of RSA [19] modulus the larger the amount of computation performed by GNFS to factorize  $N$ . Thus the security parameter (key size) must be sufficiently large as to make the best known attack infeasible.

Table 3 lists the best known attack against the underlying mathematical problems

---

**Table 3:** Best Known Attacks and Security Parameters

	RSA	DSA/DH/ElGamal	ECDSA, ECDH, EC-ElGamal
	IFP	DLP	ECDLP
Best know attack	GNFS <sup>(1)</sup>	a. GNFS <sup>(2)</sup> b. Parallel collision search <sup>(3)</sup>	Parallel collision search
Complexity	Sub Exponential	a. Sub Exponential b. Exponential	Exponential
Typical Security parameter size in bits	768 – 2048	a. 768 – 2048 b. 160 <sup>(4)</sup>	140 – 224

(1) GNFS: General Number Field Sieve

(2) against the prime field order

(3) against subgroup order

(4) DSA only

---

As a result, Public Key Cryptosystems has a variable security parameter (key size) that can be increased to achieve higher security as the best known attack improves.

### **2.1.2 Public Key Schemes**

Public Key Cryptosystems can be classified into three categories/schemes depending on the application:

- Encryption Schemes: for two parties to exchange messages, the sender encrypts the message using the recipient's Public Key. Examples of algorithms used for encryption are RSA, ElGamal and EC-ElGamal.
- Digital Signature Schemes: the sender signs a message with his Private Key. Signing is achieved by applying the Public Key algorithm to the message or to a block of data that is a function of the message. Examples of algorithms are RSA, DSA [16] and ECDSA [9].
- Key Exchange Schemes: two or more parties cooperate to share a secret key. Examples of algorithms used for key exchange are RSA, DH and ECDH

The following are examples of Digital Signature Schemes

#### 1. RSA Signature :

a. System Parameters: N/A

b. Key Generation:

- i. Generate two large random primes,  $p$  and  $q$  approximately the same size such that the modulus  $n = p \cdot q$  is of the desired size. Typical sizes of  $n$  are 768, 1024, 2048 bits.
- ii. Compute  $n = p \cdot q$  and  $\phi = (p-1)(q-1)$ .

- iii. Select the public exponent, a random integer  $e$ ,  $1 < e < \phi$ , such that  $\text{GCD}(e, \phi) = 1$ . In order to improve the efficiency of encryption, small public exponents are selected. Popular values are 3, 17 and  $65537 (2^{16} + 1)$ .
- iv. Compute the private exponent, the unique integer  $d$ ,  $1 < d < \phi$ , such that  $e \cdot d \equiv 1 \pmod{\phi}$ . Unlike the public exponent  $e$ , the private exponent should be roughly the same size of the modulus.

The Public Key is  $(n, e)$ , the Private Key is  $(n, d)$ .

c. Signature Generation

- i. Compute  $m' = R(m)$ ,  $0 \leq m' \leq n-1$ , where  $R$  is a redundancy function
- ii. Compute  $s = m'^d \pmod{n}$
- iii. Send  $s$  as the digital signature of  $m$ .

d. Signature Verification

- i. Compute  $m' = s^e \pmod{n}$
- ii. Verify that  $m' \in$  message space of  $R$ , if not reject the signature.
- iii. Recover the message  $m = R^{-1}(m')$

2. DSA DLP/SDLP:

a. System Parameters

- i. Select a prime number  $q$  such that  $2^{159} < q < 2^{160}$
- ii. Select a prime number  $p$  where  $2^{L-1} < p < 2^L$  for  $512 < L < 1024$  and  $L$  a multiple of 64, with the property that  $q$  divides  $(p - 1)$ .

- iii. Select a generator  $g$  of the prime field  $GF(p)$  with order  $q$ .

DSA System Parameters are  $(p, q, g)$

b. Key Generation

- i. Select a random integer  $x$  such that  $1 \leq x \leq q - 1$
- ii. Compute  $y = g^x \text{ mod } p$

The Public Key is  $(p, q, g, \text{ and } y)$ , the Private Key is  $(x)$ .

c. Signature Generation

- i. Select a random secret integer  $k$ ,  $1 \leq k \leq q - 1$ .
- ii. Compute  $r = (g^k \text{ mod } p) \text{ mod } q$ .
- iii. Compute  $s = k^{-1} \cdot (\text{SHA-1}(m) + x \cdot r) \text{ mod } q$ .
- iv. Send  $(r, s)$  as the digital signature of  $m$ .

d. Signature Verification

- i. Compute  $w = s^{-1} \text{ mod } q$ .
- ii. Compute  $u1 = w \cdot \text{SHA-1}(m) \text{ mod } q$  and  $u2 = r \cdot w \text{ mod } q$ .
- iii. Compute  $v = (g^{u1} \cdot y^{u2} \text{ mod } p) \text{ mod } q$ .
- iv. If  $v = r$ , accept signature.

### 3. ECDSA ECDLP

a. System Parameters

- i. Elliptic Curve  $E$  over  $GF(q)$  where  $q = p$  (prime) or  $q = 2^m$  with order  $\#E$ . Typical sizes of  $q$  range from 140 to 224.
- ii.  $r$ , a prime divisor of  $\#E$ .

iii.  $\mathbf{G}$ , a point on  $E$  generating a subgroup of order  $r$ .

b. Key Generation

i. Select a random integer  $s$  where  $1 \leq s \leq r - 1$ .

ii. Compute point  $\mathbf{W} = s \cdot \mathbf{G}$ .

The Public Key is  $\mathbf{W}$ , the Private Key is  $s$ .

c. Signature Generation

i. Select a one-time random message Private Key  $k$ ,  $1 \leq k \leq r - 1$ .

ii. Compute message Public Key,  $\mathbf{R} (x_R, y_R) = k \cdot \mathbf{G}$ .

iii. Compute  $c = x_R \bmod r$ , if  $c = 0$  goto i.

iv. Compute  $d = k^{-1} \cdot (\text{SHA-1}(m) + x \cdot c) \bmod r$ , if  $d = 0$  goto i.

v. Send  $(c, d)$  as the digital signature of  $m$ .

d. Signature verification

i. Compute  $w = d^{-1} \bmod r$ .

ii. Compute  $u1 = \text{SHA-1}(m) \cdot w \bmod r$  and  $u2 = c \cdot w \bmod r$ .

iii. Compute EC point  $\mathbf{V} = u1 \cdot \mathbf{G} + u2 \cdot \mathbf{W}$ , if  $\mathbf{V} = \mathbf{O}$  reject signature otherwise  $\mathbf{V} = (x_V, y_V)$ .

iv. Compute  $c' = x_V \bmod r$ .

v. If  $c' = c$  accept signature.

## **2.2 Libraries:**

Public Key Cryptosystems based on previously described mathematical problems require operations with large integers ranging from 768 to 2048 bit and elements of large finite fields ranging from 140 to 240 bits. These operations need to be optimized for high-speed implementations. Arithmetic and number theoretical operations with such large operand sizes are hard to implement, thus the majority of developers implementing Public Key Schemes use existing multi-precision and number theoretical libraries.

In this study, eight libraries are compared against each other, based on a set of evaluation criteria, in order to help developers decide which library to choose.

The libraries used in the comparison are listed in Table 4. All the libraries are general purpose multi-precision arithmetic, and number theoretical libraries with the exception of CryptoPP, MIRACL and OpenSSL which are specifically targeting cryptographic schemes.

**Table 4:** Libraries

	Language	Support	Written/Maintained by	License
CLN[2]	C++	integer, rational, float, complex, mod integer, polynomials	Written by Bruno Haible Maintained by Richy Kreckel	GNU General Public License
CryptoPP[5]	C++	cryptographic primitives and schemes	Wei Dia	Copyrighted as a compilation
GMP[7]	C	integer, rational, float	Maintained by Torbjörn Granlund	GNU General Public License
LiDIA[13]	C++	integer, rational, float, complex, mod integer, polynomials, elliptic curves, factorization, number fields,	LiDIA group University of Darmstadt	LiDIA group
MIRACL[20]	C	cryptographic primitives and schemes	Michael Scott	Shamus Software Ltd.
NTL[17]	C++	integer, float, polynomials	Victor Shoup	GNU General Public License
OpenSSL[18]	C	cryptographic primitives and schemes	OpenSSL team <a href="http://www.openssl.org">www.openssl.org</a>	Apache-style licence
PIOLOGIE[8]	C++	integer, rational	Written by Sebastian Wedeniwski	<a href="http://www.hipilib.de">www.hipilib.de</a>

CLN, LiDIA and NTL where compiled using GMP as the underlying multi-precision library as recommended by the library developers to achieve maximum speed. The structure of GMP has six function categories; two of them are of interest:

- **mpz:** high-level functions for signed/unsigned integer arithmetic
- **mpn:** low-level functions that operate on natural numbers. Most mpn functions contain machine-dependent code, written in assembly or C.

The mpn functions are used by other function categories including mpz functions.

CLN and NTL use GMP mpn functions and build a different user interface, while LiDIA uses mpz functions.

The libraries versions used are listed in Table 5.

---

**Table 5: Libraries Versions**

Library	Version used	Current Version
CLN	1.1.5	1.1.6
CryptoPP	5.1	5.1
GMP	4.1.2	4.1.2
LiDIA	2.1pre7	2.1pre7
MIRACL	4.82	4.82
NTL	5.3.1	5.3.1
OpenSSL	0.9.7c [12/23/1998]	0.9.7d [03/17/2004]
PIOLOGIE	scientific edition V1.3.2	-

---

There are two different editions of PIOLOGIE; normal editions, dependent on special processors, compilers and operating system and special editions, independent of these surroundings.

A special scientific edition V1.3.2 was used in this thesis, this edition is distributed under the terms and conditions of the GNU General Public License.

### **2.2.2 Operations**

In this section we will categorize operations into four categories with respect to the previous Public Key categories in Table 2 and Digital Signature Schemes in section 2.1.2:

- System Parameters Operations: Operations involved in the generation of system parameters.
- Key Generation Operations: Operations involved in the generation of Public and Private Keys.
- Public Key Operations: Operations performed using the Public Key.
- Private Key Operations: Operations performed using the Private Key.

Table 6 shows the primitive operations involved in the previous categories.

**Table 6: Primitive Operations**

	RSA	DSA/DH/ElGamal	ECDSA, ECDH, EC-ElGamal
	IFP	DLP	ECDLP
System parameters Operations	N/A	Primality Testing Finding a generator $a$ of the prime field	generate/choose $E$ Find a base point on $E$ with a large prime order
Key generation Operations	Multiplication GCD xGCD <sup>(1)</sup>	Modular Exponentiation	scalar multiplication
Public Key Operations	Modular Exponentiation	Modular Exponentiation Multiplication <sup>(*)</sup> xGCD <sup>(*)</sup>	scalar multiplication addition
Private Key Operations	Modular Exponentiation	Modular Exponentiation	scalar multiplication addition

(\*) DSA and ElGamal only

(1) xGCD: Extended GCD to compute multiplicative inverse

Typical sizes of operands involved are shown in Table 3. Based on sizes and types of operands we can further organize these operations into two main sets; operations on large integers which involve integers of sizes ranging from 768 to 2048 bits and operations on elliptic curve points defined over finite fields of sizes ranging from 140 to 224 bits.

### **2.3 Evaluation Criteria:**

The libraries are compared to each other using the following criteria:

#### **Licenses**

Do you have to pay for it?

#### **Documentation and Ease of use**

The documentation of a library is particularly important. It must be able to show in a clear and efficient way the basics that are needed to use its functions in the proper way. Examples in the documentation are important since they provide the user with a quick, effective and practical idea of the way the library functions/classes have to be used.

Ease of use of a library includes easiness of install and the simplicity and clarity of the interface.

#### **Supported compilers**

How many compilers are supported? Evaluation of this criterion is based on the information provided by the library suppliers.

### **Support for Public Key Cryptosystems**

All previous criteria are one way or another context-independent, in the sense that they are general criteria against which a library can be evaluated and found to be more or less attractive.

Support for Public Key Cryptosystems is based on the support of primitive arithmetic and number theoretical operations (Table 2) needed by the three main categories of Public Key cryptosystems discussed in section 2.2.2, and on complete implementation of Public Key Schemes in the library.

### **Performance of Primitive Cryptographic Operations**

The primitive cryptographic operations tested are chosen from Table 2 and organized into two main sets according to the operand sizes and types:

Large integer operation: multiplication, modular exponentiation, GCD, xGCD (multiplicative inverse) and Primality testing with operands sizes 768, 1024 and 2048 bits.

Operations on EC points: point addition and scalar multiplication with base point order lengths 163, 233 and 409 bits for EC2 and 162, 226 and 386 bits for ECP.

All Libraries were compiled using GNU C/C++ compiler on three Operating Systems Windows XP, P4-RedHat 9.0 and Solaris 5.8, using instructions provided by the libraries writers. Measurements are analyzed to obtain a general overall ranking of the libraries with respect to one another on each platform based on an overall rank of each operation.

### 3 Algorithms

The following are the algorithms used by the libraries to implement the operations on large integers and EC points:

#### **3.1 Large Integers**

##### **3.1.1 Multiplication**

Multiplication Algorithms implemented in the libraries are summarized in Table 7. Three of these algorithms are actually used for the input sizes typically used in Public Key cryptography (Table 3).

1. Classical Multiplication: long multiplication described in [11], carried out using two loops thus when multiplying two  $n$ -digits numbers a total of  $(n + 1)^2$  single precision multiplications are computed. Squaring requires  $(n^2 + n)/2$  single precision multiplications.
2. Comba Multiplication [4]: unraveling the loops in classical multiplication and reducing the memory access. Operands must have a fixed length.
3. Karatsuba Multiplication [11]: multiplying  $2n$ -digit numbers by performing three multiplications of  $n$ -digit numbers along with two additions and two subtractions as opposed to four multiplications of  $n$ -digit numbers. The key issue in the algorithm is to determine the break-point (threshold) when the cost of addition and subtraction are insignificant relative to the cost of multiplication. The Karatsuba algorithm can be used recursively where either classical or Comba multiplication algorithms are used below the threshold.

**Table 7: Multiplication Algorithms**

Multiplication Algorithm	Complexity	Description
Classical	$O(n^2)$	double loop
Comba	$O(n^2)$	Unraveling double loop and reducing memory access
Karatsuba , Karatsuba-Comba	$O(n^{\lg 3})$	Divide-and-conquer method, Can be combined with Classical (Karatsuba) or Comba (Karatsuba-Comba) algorithms
Toom-Cook T-C [4]	$O(n^{2.146} \log n)$	
Fast Fourier Transformation FFT [4]	$O(n \log n \log \log n)$	

Table 8 summarizes the libraries implementations of multiplication algorithms and their thresholds.

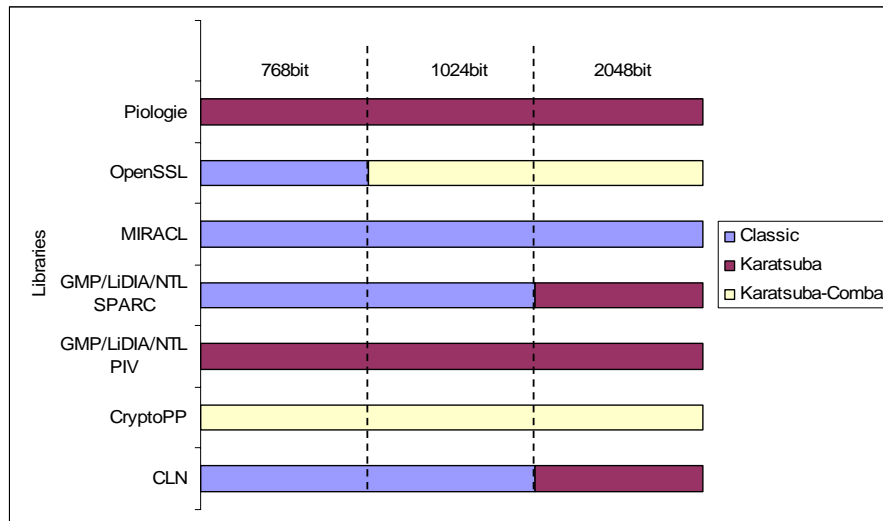
**Table 8: Multiplication Algorithm Ranges**

	CLN	CryptoPP	GMP/LiDIA/NTL PIV	GMP/LiDIA/NTL UltraSPARC- Solaris	OpenSSL	PIOLOGIE
Classical	[0,1120)	N/A	[0,576)	[0,1280)	[0,512)	[0,256]
Comba	N/A	[0,256]	N/A	N/A	N/A	N/A
Karatsuba	[1120,80000)	(256,) <sup>(*)</sup>	[576,4448)	[1280,7104)	[512,) <sup>(*)</sup>	(256,160000)
T-C	N/A	N/A	[4448,188416)	[7104,122800)	N/A	N/A
FFT	[80000,)	N/A	[188416,)	[122800,)	N/A	[160000,)

(\*) Karatsuba-Comba

Figure 1 shows the different operand sizes used and the algorithms used for multiplication. GMP is the only library that adjusts the threshold not only depending on

operand sizes, but also on the underlying architecture. This directly affects LiDIA and NTL which uses GMP implementations of Multiplication algorithms.



**Figure 1: Multiplication Algorithms for Different Key Sizes**

CryptoPP implementation of the Karatsuba-Comba algorithms requires the inputs to be powers of 2. In case they are not, the input is rounded to the next power of 2 before applying algorithm e.g. an input of size 768 bit is rounded up to 1024 bits, which explains the ratios of 768 to 1024 bit inputs.

OpenSSL implementation of Karatsuba-Comba algorithm requires the inputs to be of powers of 2, otherwise Classical multiplication is used.

MIRACL implements Classical multiplication for all sizes.

### **3.1.2 Modular exponentiation**

Modular Exponentiation Algorithms implemented in the libraries are summarized in Table 9.

---

**Table 9:** Modular Exponentiation Algorithms

Algorithm	Complexity	Per-computations
Left-to-right	$L(e)-1$ squarings, $W(e)-1$ multiplications	No
Right-to-left	$L(e)-1$ squarings, $W(e)-1$ multiplications	No
Left-to-right k-ary	$L(e)-1$ squarings, $[L(e)-1/k]-1$ multiplications	Yes
Simultaneous multiple exponentiation with sliding window		Yes Number of exponents = 1
k-ary sliding window		Yes

---

Table 10 summarizes library implementations of the previous algorithms and their thresholds.

---

**Table 10:** Modular Exponentiation Algorithm Ranges

	CLN	CryptoPP	GMP	LiDIA	MIRACL	NTL	OpenSSL	PIOLOGIE
LR	[2,8]	N/A	[2,32]	N/A	N/A	[2,512)	N/A	N/A
RL	N/A	N/A	N/A	[2,)	N/A	N/A	N/A	[2,)
LR k-ary	(8,)	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SME	N/A	[2,)	N/A	N/A	N/A	N/A	N/A	N/A
k-ary SW	N/A	N/A	(32,)	N/A	[2,)	[512,)	[2,)	N/A

LR: Left-to-Right

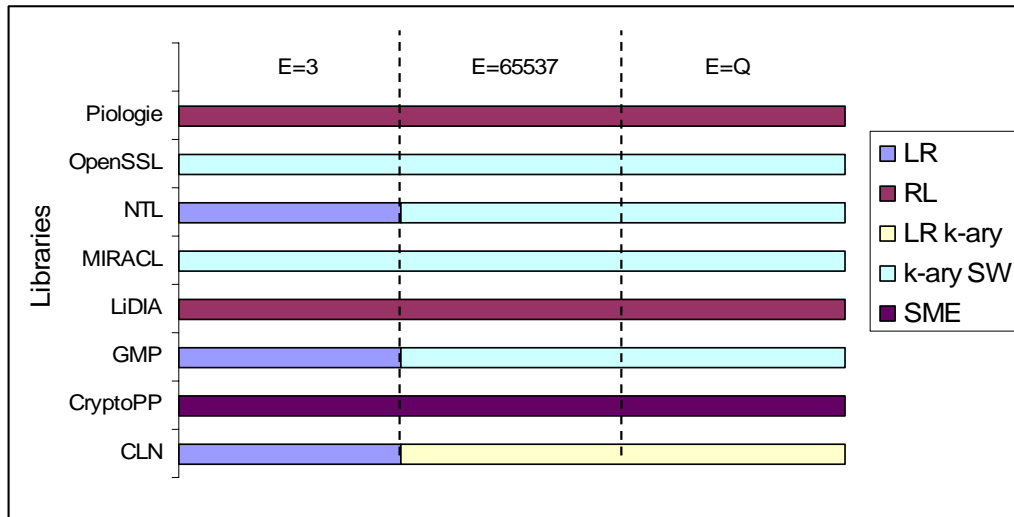
RL: Right-to-Left

SME: Simultaneous Multiple Exponentiation

SW: Sliding Window

---

Figure 2 shows the algorithms used by the libraries for 3 exponents, 3, 65537 and a random exponent the same size as the modulus.



LR: Left-to-Right  
 RL: Right-to-Left  
 SW: Sliding Window  
 SME: Simultaneous Multiple Exponentiation

**Figure 2:** Modular Exponentiation Algorithms for Different Exponent Sizes

### 3.1.3 GCD and xGCD

Table 11 summarizes the libraries implementations for the GCD and xGCD algorithms.

**Table 11:** GCD and xGCD Algorithms

	GCD	xGCD
CLN	Lehmer	Lehmer
CryptoPP	Euclid	Binary
GMP/LiDIA/NTL	Generalized Binary	Lehmer
MIRACL	Lehmer	Lehmer
OpenSSL	Binary	Binary
PIOLOGIE	Generalized binary	Euclid

### **3.2 EC points**

Operations on elliptic curves is limited to four libraries; CryptoPP, LiDIA, MIRACL and OpenSSL (ECP only).

#### **2.2.1 Addition**

Addition of EC points is done using the following algorithms depending on the underlying field:

1.  $GF(p)$

$$\mathbf{T}(x_t, y_t) + \mathbf{S}(x_s, y_s) = \mathbf{R}(x_r, y_r)$$

$$x_r = \lambda^2 - x_t - x_s$$

$$y_r = \lambda(x_t - x_s) - y_t$$

Where  $\lambda = (y_s - y_t) / (x_s - x_t)$  if  $\mathbf{T} \neq \mathbf{S}$  and  $\lambda = (3x_t^2 + a) / (2y_t)$  if  $\mathbf{T} = \mathbf{S}$  (doubling).

2.  $GF(2^m)$

$$\mathbf{T}(x_t, y_t) + \mathbf{S}(x_s, y_s) = \mathbf{R}(x_r, y_r)$$

If  $\mathbf{T} \neq \mathbf{S}$

$$\lambda = (y_t - y_s) / (x_t - x_s)$$

$$x_r = \lambda^2 + \lambda + x_t + x_s + a$$

$$y_r = \lambda(x_t + x_s) + x_r + y_t$$

If  $\mathbf{T} = \mathbf{S}$  (doubling)

$$\lambda = (x_t + y_t) / x_t$$

$$x_r = \lambda^2 + \lambda + a$$

$$y_r = x_t + x_r (\lambda + 1)$$

### **2.2.2 Scalar Multiplication**

Table 12 summarizes the libraries implementations of EC point scalar multiplication.

---

**Table 12:** Elliptic Curve Scalar Multiplication Algorithms

	Scalar Multiplication
CryptoPP	Simultaneous Sliding Window
LiDIA	Left-to-Right
MIRACL	wNAF-based interleaving [15]
OpenSSL	wNAF-based interleaving

---

## 4 Performance of Primitive Cryptographic Operations

### 4.1 Methodology

Measurements were conducted in two different ways depending on the platform. The first method of testing determines the number of clock cycles required to perform the operation referred to as RDTSC method. The second method determines the amount of time in milliseconds required to perform the operation referred to as Timing method.

---

**Table 13:** Platforms and Compilers Used

Processor/hardware	OS/Compiler	Measurement method
2.00 GHz Pentium IV Processor, 512 MB ram <sup>(*)</sup>	Windows XP Cygwin GNU C/C++ 3.3.1	RDTSC
2.00 GHz Pentium IV Processor, 512 MB ram <sup>(*)</sup>	P4-RedHat Linux 9.0 GNU C/C++	RDTSC
Sun: 2x ??? MHz UltraSPARC-Solaris-II, 4-MB E-cache, ??? MB RAM	Solaris 5.8	Timing

(\*) Same Machine

---

#### **4.1.1 RDTSC method:**

Uses the RDTSC [10] (read time-stamp counter) instruction to access the time-stamp counter.

The time-stamp counter present on Intel processors beginning with the Pentium processor is a 64 bit model specific register that is incremented every clock cycle.

The CUID instruction is used as a serializing instruction to prevent out-of-order execution.

---

```

#define CUID    asm volatile(".byte 0x0f, 0xa2" : : : "eax", "ebx", "ecx", "edx")

#define _RDTSC(ts) asm volatile(".byte 0x0f, 0x31;movl %%edx,%0;    movl %%eax,%1" :
"=r" (ts.HighPart) , "=r" (ts.LowPart): : "%edx", "%eax")

LARGE_INTEGER getRDTSC()
{
    LARGE_INTEGER ts;
    CUID;
    _RDTSC(ts);
    return ts;
}

```

---

**Figure 3** Calling RDTSC Instruction

---

The overhead associated with the instructions call is also calculated and subtracted from the final result.

Both instructions were called several times before testing the operation to flush the instruction cache.

---

```

for( i = 0; i < 100 ; i++ )
{
    start = getRDTSC() ;
    for( j = iterations; j > 0; j-- )
        operation();
    finish = getRDTSC() ;
    ClockCycles[i] = Subtract(finish, start)/iterations - overhead;
}

```

---

---

### Figure 4 RDTSC Method

---

#### **4.1.2 Timing method**

Due to the lack of CPU cycle counter in the UltraSPARC-Solaris II the function `gettimeofday()` [1] was used to calculate the time consumed in the execution of the operation. The function gives resolutions up to microseconds

```
for( i = 0; i < 100 ; i++ )
{
    gettimeofday(&start, NULL);
    for( j = iterations; j > 0; j--)
        operation();
    gettimeofday(&finish, NULL);
    time[i] = Subtract(finish, start)/iterations;
}
```

---

### Figure 5 Timing Method

---

#### **4.1.3 Operations and Input Sets**

1. Large Integer Operations: two groups of inputs where used.
  - a. Group A: a group of randomly generated integers containing three sets of sizes 768, 1024 and 2048 bits. Each set contains three large integers denoted as  $I_i$ ,  $J_i$  and  $K_i$ , where  $i$  is the size of the integer in bits. The values of  $I_i$ ,  $J_i$  and  $K_i$  are listed in Appendix A.
  - b. Group B: a group of randomly generated large prime numbers containing three sets of sizes 768, 1024 and 2048 bits. Each set contains 10 large

prime integers denoted as  $P_{ij}$ , where  $i$  is the size of the integer in bits and  $j$  is the number of the prime in the set.

Table 14 summarizes the operations tested and the associated group of inputs

**Table 14:** Large Integer Operations tested

Operation	Input Group	Comments
Multiplication	A	result = $I_i * J_i$
Mod Exp E=3	A	result = $(I_i)^3 \bmod K_i$
Mod Exp E=65537	A	result = $(I_i)^{65537} \bmod K_i$
Mod Exp Large E <sup>(*)</sup>	A	result = $(I_i)^J \bmod K_i$
Primality	B	result = IsPrime( $P_{ij}$ )
GCD	B, A	result = GCD( $P_{ij}, K_i$ )
xGCD	B, A	result = xGCD( $P_{ij}, K_i$ )

(\*) Exponent has the same size of the modulus

With respect to a particular library under a particular OS:

Each operation using Group A is tested using either RDTSC method or Timing method as shown in Figures 4 and 5.

The operation is tested using three input sizes which produces three sets of 100 values (one value for one iteration of the experiment) one set for each input size.

Each set of 100 values is sorted and the minimum value is recorded and denoted as  $AMIN_i$  ( $i = 768, 1024$  and  $2048$  bits).

The final set of raw results for each operation tested on a particular library under a particular OS is denoted by  $(LIB-OP)_{OS} = \{ AMIN_{768}, AMIN_{1024}, AMIN_{2048} \}$ .

The same approach is used with operations using Group B, except that each operation is tested using 10 different inputs ( $P_{ij}$ ) of the same size e.g. for 768 bits we have 10 inputs  $P_{768j}$   $0 \leq j \leq 9$  and for each  $j$  we will have 100 values.

For each  $P_{ij}$  the 100 values are sorted and the minimum recorded and denoted as  $BMIN_{ij}$ . The 10 minimum values for each input size ( $BMIN_{768j}$ ,  $BMIN_{1024j}$ ,  $BMIN_{2048j}$ ) are then averaged,

$$BAVG_i = \frac{1}{10} \sum_{j=0}^9 BMIN_{ij}$$

The final set of raw results for each operation on a particular library under a particular OS is denoted by  $(LIB-OP)_{OS} = \{BAVG_{768}, BAVG_{1024}, BAVG_{2048}\}$ .

## 2. EC Operations

EC point operations tested are point Addition and Scalar Multiplication with input sizes for EC2 163, 233 and 409 bits and ECP 162, 226 and 386 bits.

For each EC curve two randomly generated points  $T_i$  and  $S_i$  ( $i = 163, 233, 409$  for EC2,  $i = 162, 226, 386$  for ECP) were used as inputs.

Addition: result =  $T_i + S_i$ .

Scalar Multiplication: result =  $(r - 2) T_i$  where  $r$  is the order of the base point.

The results are collected as described for operations on large integers using Group A of inputs.

The EC curves and points are listed in Appendix A.

## **4.2 Ranking**

### **4.2.1 Operation ranking**

After complete values of execution times for all operations under eight libraries were available, the operations rankings were calculated as follows:

(With respect to an operation OP tested on eight libraries under a particular operating system OS)

We rearrange the execution times of operation OP into three sets of eight values (one value for each execution time under a particular library). The minimum value in each set is determined and all values in a given set are divided by that value. The resulting values, denoted by  $(LIB-r_i)_{OS}$  represent operation OP rank on library LIB for a given input size  $i$ . For an operation OP,  $(LIB-r_i)_{OS} = 1.00$  corresponds to the fastest library, a rank equal to  $r$  means that a given library is  $r$  times slower for OP for a given size  $i$ , compared to the fastest library.

Operation OP overall rank on a library LIB denoted by  $(LIB-R)_{OS}$  is simply the Geometric Mean of its ranks for three different input sizes:

$$(LIB-R)_{OS} = \sqrt[3]{\prod_i (LIB - r_i)_{OS}}$$

### **4.2.2 Library Ranking**

As a result we will have a set of operation rankings for each library on each OS; the overall rank of the library denoted by  $(LR)_{OS}$  on a particular OS is determined by calculating the geometric mean of its individual operation ranks.

$$(\text{LR})_{\text{OS}} = \sqrt[N]{\prod_{k=1}^N (\text{LIB} - R)^k}_{\text{OS}}$$

Where N is the number of operations considered. N = 6 for large integer operations, N=2 for EC point operations. The two sets of operations are considered separately because EC point operations are not supported by all libraries.

## 5 Results

### 5.1 Documentation and ease of use

Figure 6 summarizes the results for documentation and ease of use.

---

<b>Documentation</b>	<b>Sufficient</b>		GMP LiDIA MIRACL NTL	
			CLN	PIOLOGIE
	<b>Insufficient</b>	Crypto++	OpenSSL	
		<b>Hard</b>	<b>Ease of use</b>	<b>Easy</b>

---

**Figure 6** Documentation and Ease of Use

---

PIOLOGIE's simple structure makes it the easiest of all, on the other hand CryptoPP is the hardest due to its complex structure and insufficient documentation.

GMP, LiDIA, MIRACL and NTL have a complex structure but their documentation, documentation examples and test suites decrease their difficulty especially for a beginner.

### 5.2 License

All libraries are free for non-commercial use.

For commercial use, all libraries are free except for LiDIA, MIRACL and PIOLOGIE.

The license fee for LiDIA depends on the packages of interest and under which conditions the end product is sold.

MIRACL Single user license costs 1,000 Euros for first user/machine and 500 Euros per additional, Server license costs 5000 Euros and updates are free.

PIOLOGIE Professional Edition, a static library dependent on processor, compiler, and operating system costs 200 Euros per machine, Server Edition same as Professional Edition costs 800 Euros per server machine, up to 10 clients, Enterprise Edition, complete collection of static libraries and source code costs 2000 Euros.

### 5.3 Supported Compilers

CLN, GMP and LiDIA can only be compiled under GNU C/C++ which actually was the main reason the compiler was used for the performance tests.

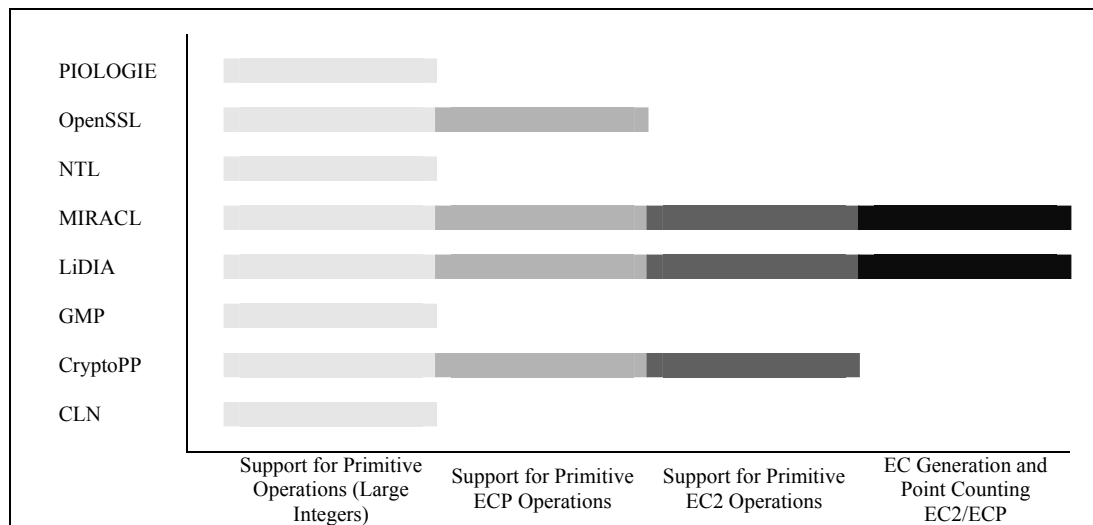
SUN WorkShop C++							
MIPSpro C++							
MSVC							
KAI C++							
VisualAge C++							
IBM CSet++							
IBM C++							
HP C++							
HP aC++							
GNU C/C++	MSVC	MSVC					
Digital C++	Intel C/C++	Microsoft eMbedded Visual C++	MSVC				
front end	GNU C/C++	GNU C/C++	Intel C/C++				
Borland C/C++	Sun WorkShop, Forte C++	DEC C	GNU C/C++				
Apogee C++	CodeWarrior Pro	CodeWarrior Pro	Borland C/C++	GNU C/C++			
Watcom C++	Borland C++ Builder	Borland C/C++	ARM C	MSVC	GNU C/C++	GNU C/C++	GNU C/C++
PIOLOGIE	CryptoPP	OpenSSL	MIRACL	NTL	CLN	GMP	LiDIA

**Figure 7** Supported Compilers

Figure 7 summarizes the compilers supported by each library.

## 5.4 Support For Public Key Cryptosystems

### 5.4.1 Support for Primitive Operations



(\*) EC generation and point counting

**Figure 8** Support for Primitive Operations

Figure 8 summarizes the libraries support for primitive operations on large integers, ECP and EC2. Support for ECP operations is limited to CryptoPP, LiDIA, MIRACL and OpenSSL. Support for EC2 is limited to CryptoPP, LiDIA and MIRACL.

EC generation and point counting is supported by LiDIA and MIRACL only.



MIRACL implements cryptographic primitives in IEEE P1363. It also has implementations for AES and SHA (1, 256, 384, and 512).

OpenSSL implements DH, DSA and RSA, and a collection of Secret Key ciphers, hash functions and MAC functions

## **5.5 Performance Results**

### **5.5.1 Operations on Large Integers**

As described in section 4.1, the following figures show the overall ranking of the libraries on the three platforms used for performance testing.

Each figure contains a table and a bar chart. The table lists the individual operation ranking of each library and the overall ranking of the library (Geometric Mean of individual operation rankings). The bar chart shows the overall ranking of the libraries sorted in ascending order.

#### **Over all Ranking for Operations on Large Integers**

The tables presented in the next figures show the overall ranking of the libraries for Operations on Large Integers based on the following operation rankings:

MUL: Multiplication ranking,

E=3: Modular Exponentiation Ranking with exponent = 3.

E=65537: Modular Exponentiation Ranking with exponent = 65537.

Large E: Modular Exponentiation Ranking with exponent of the same size as the modulus.

GCD, xGCD: GCD and xGCD Rankings.

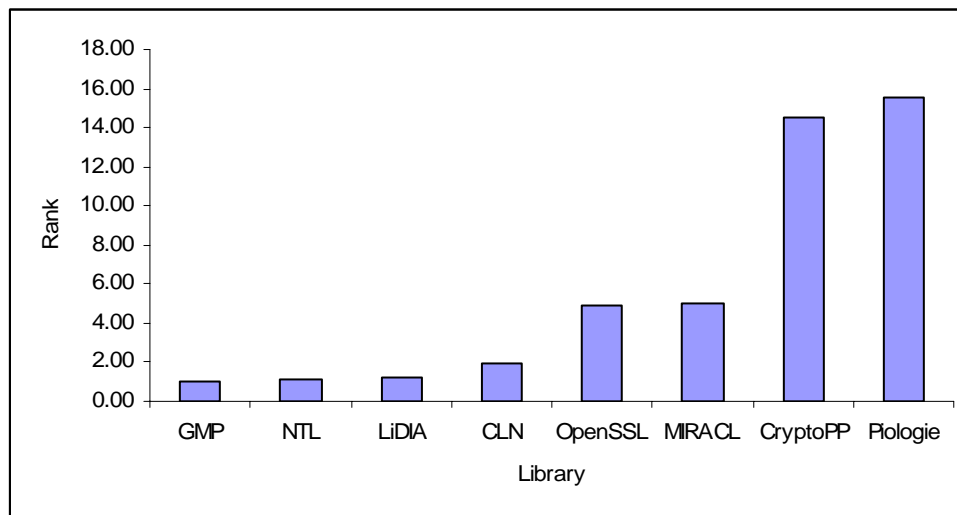
For all OS, LiDIA and NTL use GMP functions for Multiplication, GCD and xGCD; which makes their rankings in these operations very much close to GMP. For Modular Exponentiation, NTL and LiDIA have their own implementations, with NTL choice of algorithms similar to GMP.

CLN has its own implementations for all operations.

GMP has the best Ranking on three platforms followed by NTL, LiDIA and CLN.

### **P4-WinXP**

Library	MUL	E = 3	E = 65537	Large E	GCD	xGCD	P4-WinXP Rank
CLN	2.12	2.23	2.25	2.79	1.34	1.37	1.95
CryptoPP	7.11	15.17	4.71	4.04	464.90	9.99	14.56
GMP	1.00	1.00	1.00	1.00	1.01	1.08	1.01
LiDIA	1.08	1.45	1.08	1.65	1.03	1.10	1.21
MIRACL	3.58	22.40	4.56	2.62	5.15	3.15	5.00
NTL	1.01	1.42	1.17	1.18	1.00	1.00	1.12
OpenSSL	2.75	8.07	2.65	2.33	8.31	12.17	4.90
PIOLOGIE	7.60	7.40	6.63	10.65	16.41	213.30	15.51



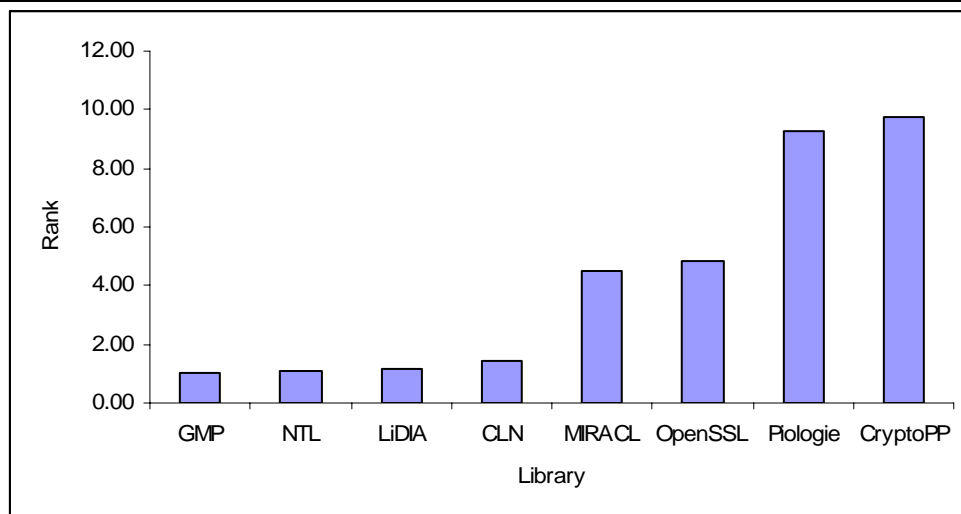
**Figure 10** P4-WinXP Rankings

Under P4-WinXP, MIRACL and OpenSSL are very close, OpenSSL Multiplication and Modular Exponentiation Rankings are better than MIRACL; while MIRACL GCD and xGCD Ranks are better than OpenSSL.

CryptoPP GCD Rank is larger than xGCD Rank unlike all other libraries; PIOLOGIE has the opposite situation; in both cases this is due to the choice of algorithm (Table 11).

### **P4-RedHat**

Library	Mul	E = 3	E = 65537	Large E	GCD	xGCD	P4-RedHat Rank
CLN	1.50	1.27	1.39	1.87	1.37	1.27	1.43
CryptoPP	4.49	9.19	3.79	5.04	65.96	16.82	9.78
GMP	1.00	1.00	1.01	1.00	1.00	1.08	1.01
LiDIA	1.00	1.10	1.06	1.84	1.00	1.09	1.15
MIRACL	3.60	21.06	4.30	2.77	3.99	2.36	4.52
NTL	1.01	1.20	1.10	1.29	1.01	1.00	1.10
OpenSSL	2.80	7.12	2.43	2.43	8.93	12.49	4.86
PIOLOGIE	5.35	5.01	5.07	8.79	21.95	24.22	9.27



**Figure 11** P4-RedHat Rankings

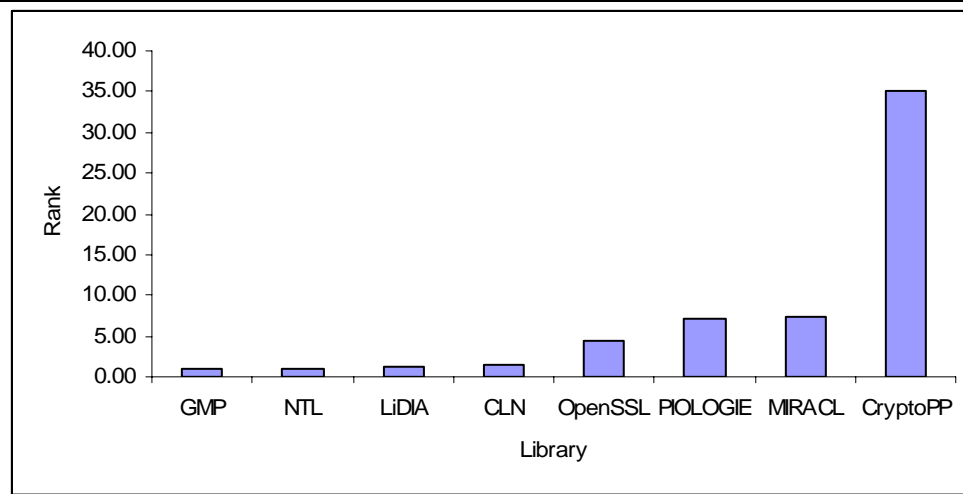
Under P4-RedHat, CryptoPP and PIOLOGIE have better Rankings than P4-WinXP; the order of GMP, NTL, LiDIA and CLN is the same.

MIRACL Rank is slightly better than OpenSSL due to GCD and xGCD Rankings.

PIOLOGIE Rank is slightly better than CryptoPP due to Modular Exponentiation Rank with E=3 and GCD Rank.

### UltraSPARC-Solaris

Library	Mul	E = 3	E = 65537	Large E	GCD	xGCD	UltraSPARC-Solaris Rank
CLN	1.21	1.60	1.70	1.98	1.67	1.40	1.58
CryptoPP	16.43	38.52	18.10	17.68	184.68	49.08	34.99
GMP	1.00	1.00	1.00	1.00	1.00	1.12	1.02
LiDIA	1.00	1.20	1.10	1.55	1.02	1.14	1.16
MIRACL	9.08	23.85	2.98	7.41	8.77	3.71	7.33
NTL	1.00	1.25	1.15	1.13	1.05	1.00	1.09
OpenSSL	2.16	7.80	2.81	2.45	7.67	7.74	4.36
PIOLOGIE	3.51	4.13	4.06	5.95	9.13	37.22	7.01



**Figure 12** UltraSPARC-Solaris Rankings

Under UltraSPARC-Solaris, GMP, NTL, LiDIA and CLN have the same order.

PIOLOGIE has a better Ranking, OpenSSL Rank remains the same.

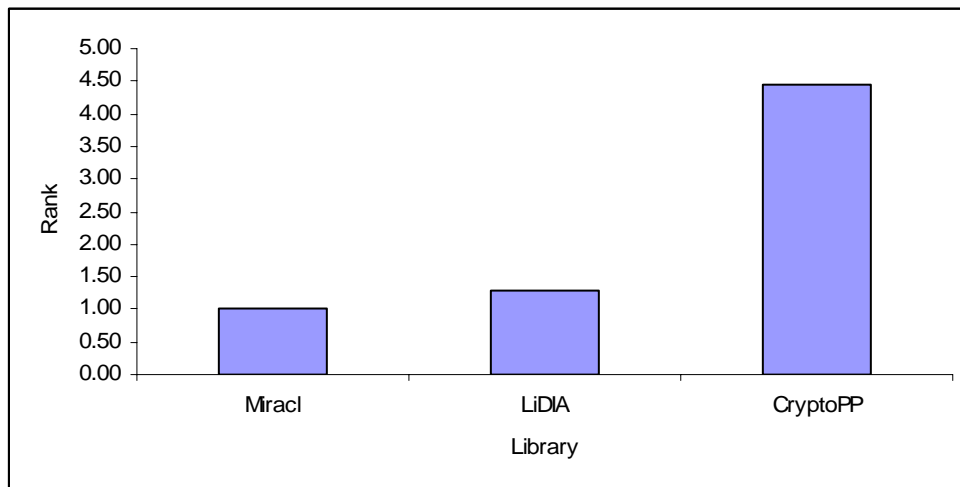
### 5.5.2 Operations on EC Points

#### Overall Ranking of Operations on EC2 Points

#### P4-WinXP

---

Library	ADD	Scalar MUL	P4-WinXP Rank
CryptoPP	4.56	4.35	4.46
LiDIA	1.33	1.22	1.28
MIRACL	1.00	1.00	1.00



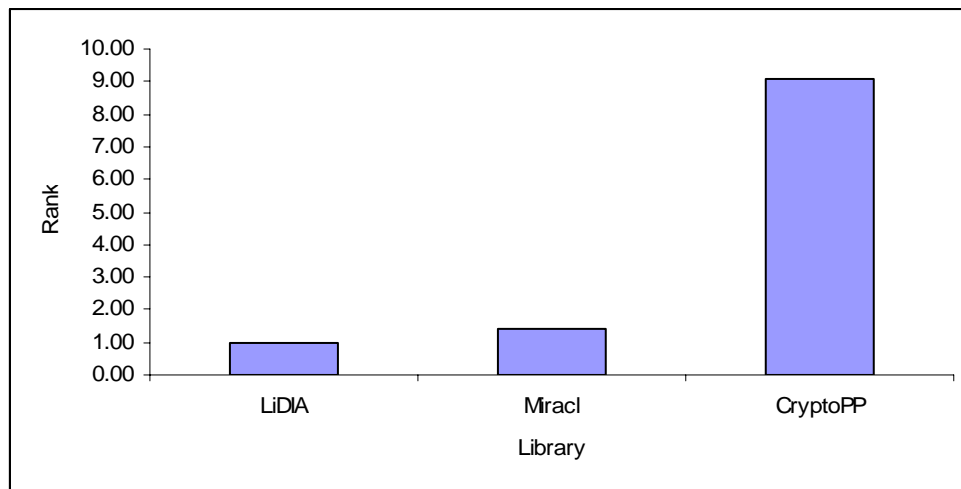
**Figure 13 P4-WinXP EC2 Rankings**

---

## P4-RedHat

---

Library	ADD	Scalar MUL	P4- RedHat Rank
CryptoPP	9.55	8.67	9.10
LiDIA	1.00	1.00	1.00
MIRACL	1.48	1.32	1.40



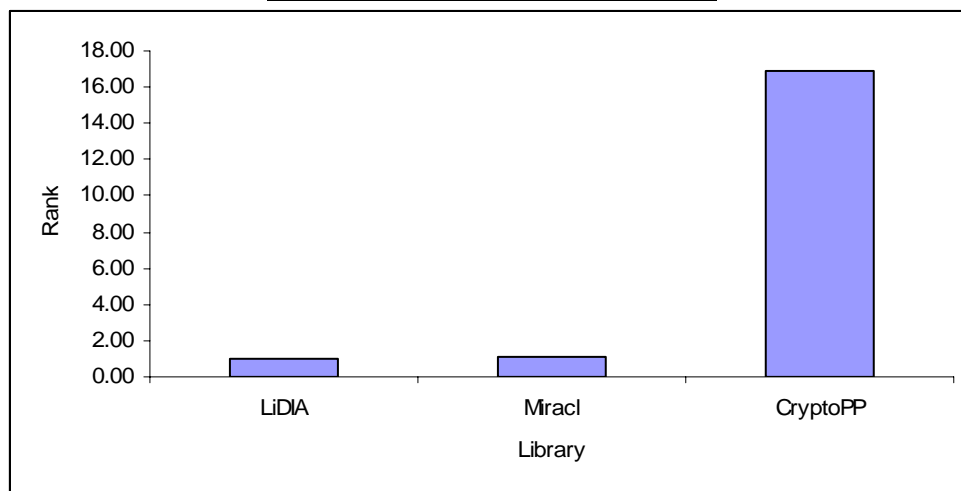
**Figure 14** P4-RedHat EC2 Rankings

---

## UltraSPARC-Solaris

---

Library	ADD	Scalar MUL	UltraSPARC- Solaris Rank
CryptoPP	17.10	16.65	16.87
LiDIA	1.00	1.00	1.00
MIRACL	1.17	1.15	1.16



**Figure 15** UltraSPARC-Solaris EC2 Rankings

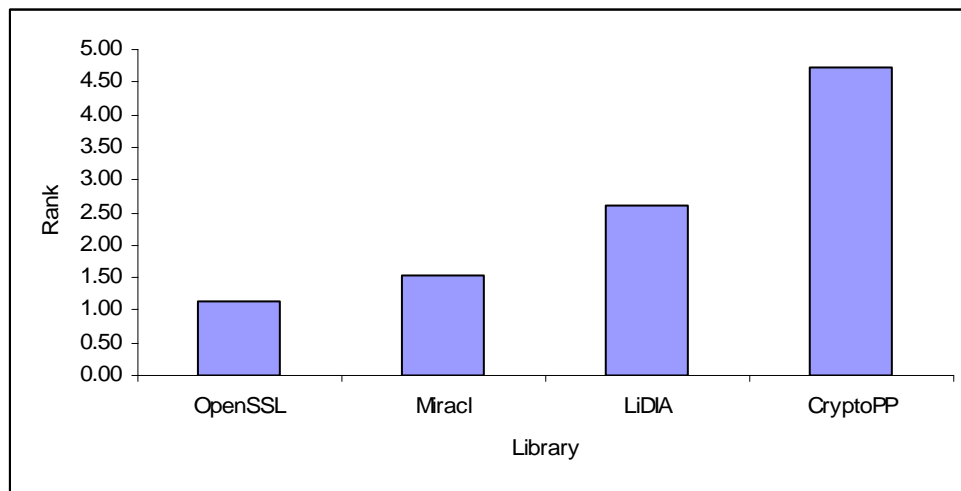
---

## Overall Ranking of Operations on ECP Points

### P4-WinXP

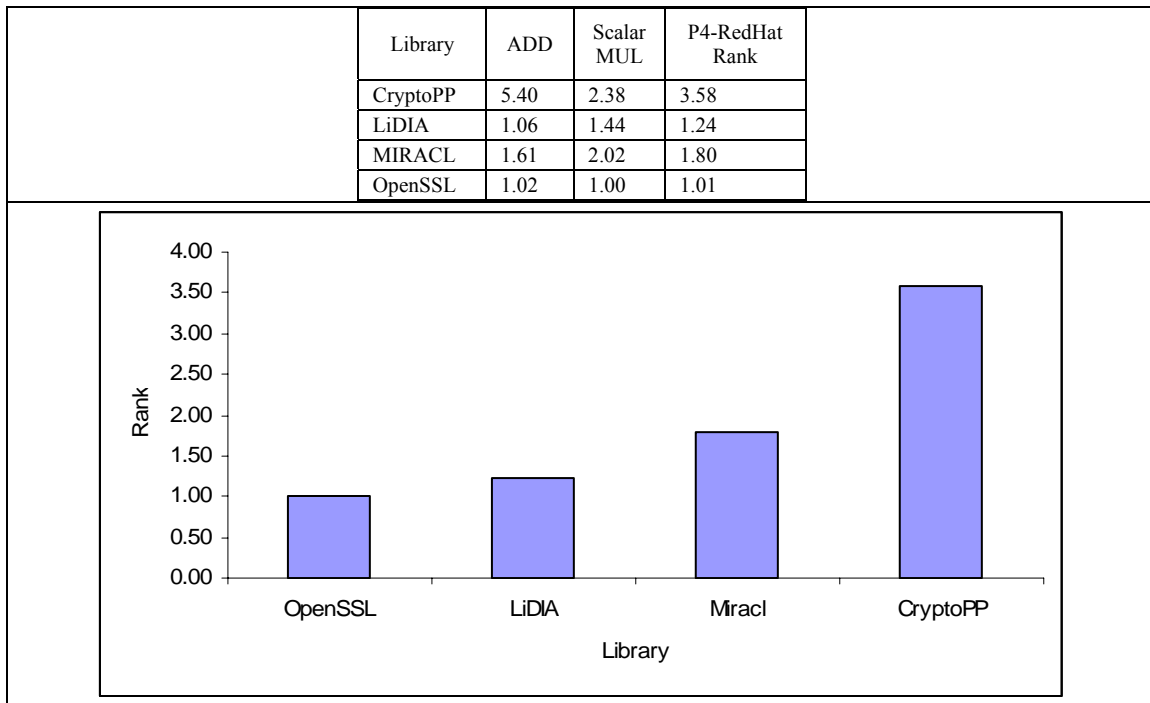
---

Library	ADD	Scalar MUL	P4-WinXP Rank
CryptoPP	6.02	3.72	4.73
LiDIA	1.97	3.45	2.61
MIRACL	1.05	2.24	1.53
OpenSSL	1.30	1.00	1.14



**Figure 16** P4-WinXP ECP Rankings

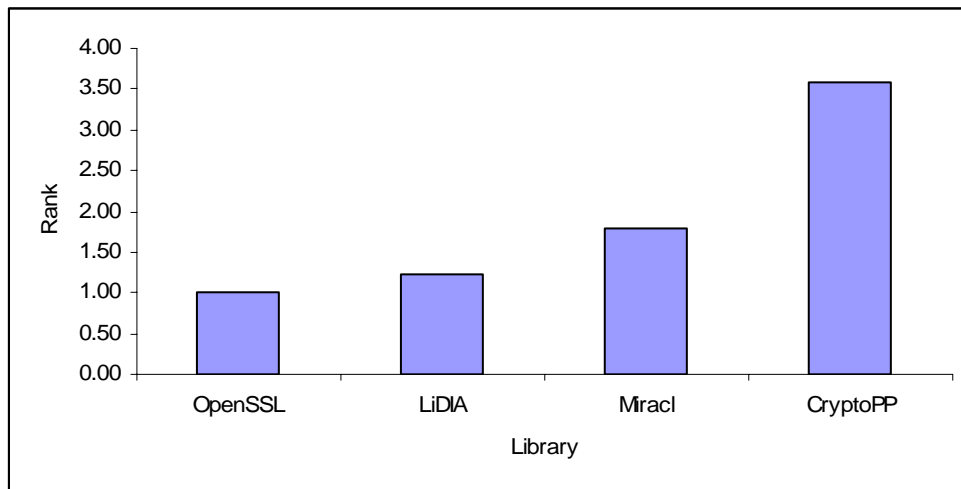
---

**P4-RedHat****Figure 17 P4-RedHat ECP Rankings**

## UltraSPARC-Solaris

---

Library	ADD	Scalar MUL	UltraSPARC- Solaris Rank
CryptoPP	9.46	7.15	8.23
LiDIA	1.05	1.73	1.35
MIRACL	1.49	2.60	1.97
OpenSSL	1.00	1.00	1.00



**Figure 18** UltraSPARC-Solaris ECP Rankings

---

## 6 Observations and Comments

### 6.1 Primality Testing

Generating prime numbers is a prerequisite in Public Key cryptosystems (Table 2).

All the libraries provide primality testing functions, mainly using trial division by small primes and Miller-Rabin test. Due to the variations of the implementations, the operation was removed from the ranking of the libraries.

Table 11 shows the number of trial divisions and Miller-Rabin iterations used in each library using 768, 1024 and 2048 bit inputs. The functions tested are listed in Appendix C.

---

**Table 15:** Primality Testing Parameters

	Trial Divisions	M-R Iterations	M-R base
CLN	2 to 67	50	Random
CryptoPP	2 to 32719	1	3
GMP	-	variable	Random
LiDIA	-	variable	Prime 3, 5...
MIRACL	2 to 997	2	Prime 3, 5
NTL	768 bit: 2 to 5381 1024 bit: 2 to 9221 2048 bit: 2 to 22627	variable	Random
OpenSSL	-	variable	Random
PIOLOGIE	-	-	-

---

All libraries using variable Miller-Rabin iterations have been tested using 5 iterations.

In GMP, instead of trial division the function multiplies small primes until overflowing a single 32 bit word, then divides the input by the small primes product (before overflow), and searches for factors in the remainder.

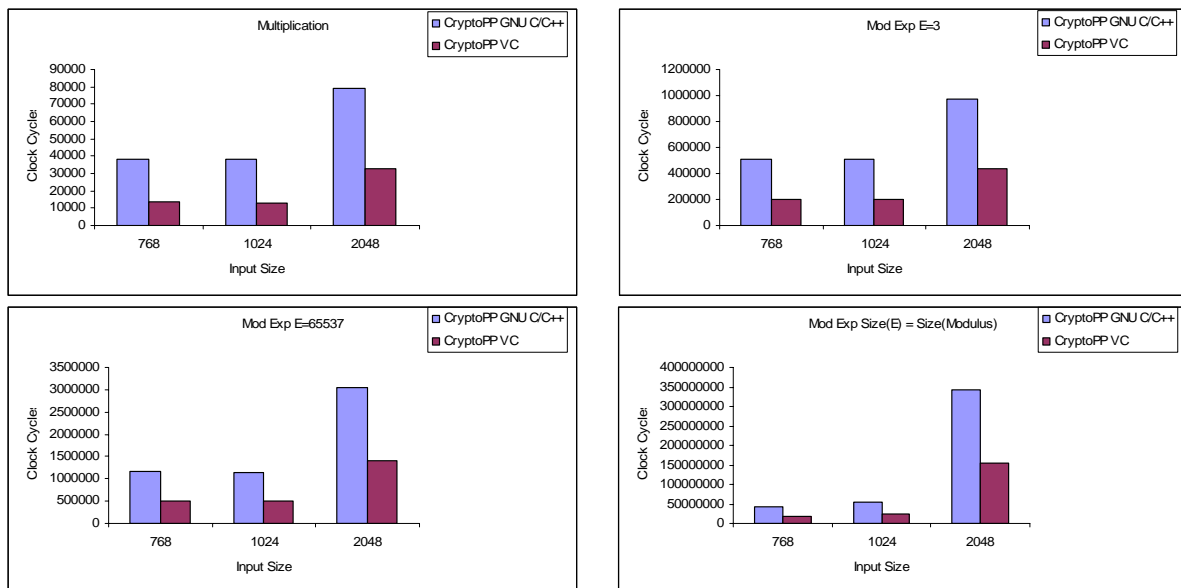
In PIOLOGIE, the function factorizes the input -1, which takes a considerable amount of time thus not included in testing.

In CryptoPP, the tested function performs an additional Lucas tests with base 3.

## **6.2 CryptoPP GNU C/C++ vs MS VC++ 6.0**

The library's performance was tested under MS VC++ 6.0 and results compared to results obtained under GNU C/C++.

Figure 19 shows the ratios of execution times for GNU C/C++ vs MS VC++ 6.0 for multiplication and modular exponentiation for three input sizes.



**Figure 19** CryptoPP GNU C/C++ vs VC++ 6.0

The figure above clearly shows that the library compiled under MS VC++ 6.0 is more than twice as fast as GNU C/C++. This is due to the library's optimization for Pentium IV (SSE2 instructions) under MS VC++ 6.0 versus its generic Pentium optimization under GNU C/C++.

MIRACL and PIOLOGIE were also tested on MS VC++ 6.0, with no significant change in their performance as compared to GNU C/C++.

### **6.3 Performance**

From the results obtained, two factors mainly affect the library's performance:

- Targeting the underlying architecture: this can be clearly observed through ranks of libraries using GMP as the underlying low-level arithmetic library and CryptoPP performance in section 6.2. GMP implements optimized assembly code targeting both platforms (PIV and UltraSPARC II) used in the experiments.
- Algorithms: choice of algorithm, algorithm thresholds and the implementation of the algorithm affects the performance of the library. The choice of algorithm effect on performance can be clearly observed through the performance of CryptoPP GCD operation over all platforms. Thresholds and different implementations of the same algorithm can be observed through the difference in performance of Multiplication and Modular Exponentiation between GMP and CLN

## 7 Conclusion

### 7.1 Final Evaluation

The final evaluation is based on categorizing the libraries in groups based on their support for Public Key Cryptosystems and performance, then differentiating between libraries in each group using other criteria.

Figure 20 shows the different groups based on Support for PKC and Performance:

<b>Support</b>	<b>PKS<sup>(1)</sup></b>		CryptoPP	MIRACL	OpenSSL	
	<b>EC2</b>	<b>PG<sup>(2)</sup> PC<sup>(3)</sup></b>	CryptoPP	MIRACL	LiDIA	
	<b>ECP</b>	<b>PG PC</b>	CryptoPP	MIRACL	LiDIA OpenSSL	
	<b>LINT</b>	<b>High</b>	CryptoPP	MIRACL	OpenSSL	CLN LiDIA NTL GMP
<b>Low</b>		PIOLOGIE				
		low		Performance		high

(1)PKS: Public Key Schemes.

(2)PG: Point Generation.

(3)PC: Point Counting.

**Figure 20** Support vs Performance

### **Support for Large Integers**

- GMP, NTL, LiDIA, and CLN: best performance under all platforms tested, with GMP the fastest and CLN the slowest in the group. LiDIA is the only library in the group that needs a license for commercial use.

For a developer targeting operations on large integers, GMP would be the best choice in terms of performance, the tradeoff if the amount of time and effort needed for implementation and portability.

- OpenSSL, MIRACL: trail libraries from the first group in terms of overall performance. OpenSSL is faster than MIRACL for all operations except GCD and xGCD which affects OpenSSL ranking on P4-WinXP. MIRACL also needs a license for commercial use.

Best choice for fast development of Public Key Cryptosystem based on operations on large integers while having an acceptable performance as compared to other libraries. Both libraries have implementations of cryptographic scheme primitives.

- CryptoPP, PIOLOGIE: PIOLOGIE performance under UltraSPARC-Solaris is better than CryptoPP for all operations, under P4-WinXP and RedHat, CryptoPP is faster except for two operations, GCD and Modular Exponentiation with exponent = 3.

CryptoPP is the best choice for fast development, complete implementations of a wide range of Cryptographic schemes involving large integers; the drawback is its performance as compared to other libraries

### **Support for EC2**

LiDIA has the best performance under P4-RedHat and UltraSPARC-Solaris. Under P4-WinXP, MIRACL has the best performance. CryptoPP is the slowest under all platforms.

LiDIA is the best choice for performance, with less portability.

MIRACL is the best choice for portability and acceptable performance.

CryptoPP is best choice for fast development, portability and coverage of schemes.

### **Support for ECP**

OpenSSL has the best performance under all platforms, LiDIA performance is better than MIRACL on P4-RedHat and UltraSPARC-Solaris, while under P4-WinXP, MIRACL is better than LiDIA. CryptoPP has the lowest performance.

For a developer targeting ECP cryptosystems, OpenSSL is the best choice, it has the best performance, portable and free.

### **Support for Public Key Schemes**

Although Public Key Schemes implemented in CryptoPP, MIRACL and OpenSSL where not compared for performance, one can estimate their performance based on primitive operations' performance. Accordingly, OpenSSL is expected to have better performance followed by MIRACL and CryptoPP respectively. In terms of support, CryptoPP has more support for Public Key Schemes and cryptography in general, followed by MIRACL and OpenSSL.

OpenSSL has the best performance compared to MIRACL and CryptoPP, but less coverage of cryptographic schemes, best choice for performance.

MIRACL has acceptable performance, but is not free for commercial use.

CryptoPP has the best coverage of cryptographic schemes in general, but low performance compared to OpenSSL, best choice for fast development.

## **7.2 Summary**

If performance is critical, and the developer has limited resources and a lot of time devoted for implementing Public Key Schemes, the best choice is GMP, the developer will have to develop the cryptographic schemes from scratch.

If performance is critical and the developer has medium amount of time and resources for implementation, the best choice is OpenSSL.

If performance is critical and the developer has limited time and a lot of resources for implementation, the best choice is MIRACL.

If performance is not critical, and the developer has limited resources and time the best choice is CryptoPP.

## **7.2 Improvements/Future Work**

Improvements include adding more libraries to the comparison, testing performance under more platforms. Future work includes evaluating the libraries support for new Public Key Cryptosystems such as NTRU.

## References

- [1] R. E. Bryant and D. O'Hallaron. *Computer Systems, A Programmer's Perspective*. Prentice-Hall, 2003.
- [2] CLN: Class Library for Numbers  
<http://www.ginac.de/CLN/>
- [3] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993
- [4] P. G. Comba. *Exponentiation cryptosystems on the IBM PC*. *IBM Systems Journal*, vol. 29, n. 4, pp. 526–538, 1990.
- [5] Crypto++ Library 5.1: a Free C++ Class Library of Cryptographic schemes  
<http://www.eskimo.com/~weidai/cryptlib.html>
- [6] W. Diffie and M. Hellman. *New Directions in Cryptography*. *IEEE Transactions on Information Theory*, IT-22, no. 6, pp. 644-654, 1976.
- [7] The GNU MP Library  
<http://www.swox.com/gmp/>
- [8] HiPiLib – Piologie  
<http://www.hipilib.de/piologie.htm>
- [9] IEEE P1363: Standard Specifications for Public-Key Cryptography.  
<http://grouper.ieee.org/groups/1363/>
- [10] Intel Corporation. *IA-32 Intel® Architecture, Software Developer's Manual*, vol 2B: Instruction Set Reference, N-Z  
<http://developer.intel.com/design/pentium4/manuals/25366713.pdf>
- [11] D.E. Knuth. *The Art in Computer Programming. Vol2 : Seminumerical Algorithms*. Addison-Wesley, 2nd.Ed. 1981.
- [12] C.K. Koc. *Analysis of sliding window techniques for exponentiation*. *Computers and Mathematics with Applications*, vol.30, n.10, pp.17–24,195.
- [13] LiDIA: A C++ Library For Computational Number Theory  
<http://www.informatik.tu-darmstadt.de/TI/LiDIA/>
- [14] A. Meneses, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
- [15] B. Möller. *Algorithms for multi-exponentiation*. *Selected Areas in Cryptography – SAC 2001 (2001)*, S. Vaudenay and A.M. Youssef (Eds.), LNCS 2259, pp. 165–180.
- [16] NIST, U.S. Department of Commerce. *Digital Signature Standard*. FIPS PUB 186, 1994.
- [17] NTL: A Library for doing Number Theory  
<http://www.shoup.net/ntl/>
- [18] OpenSSL: The Open Source toolkit for SSL/TLS  
<http://www.openssl.org/>

- [19] R. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*. Communications of the ACM, vol. 21, no.2, pp. 158-164, 1978.
- [20] Shamus Software Ltd – MIRACL  
<http://indigo.ie/~mscott/>
- [21] Standards for Efficient Cryptography Group. *SEC 2: Recommended Elliptic Curve Domain Parameters*. Version 0.6, 1999.
- [22] S. Y. Yan. *Number Theory for Computing*. Springer-Verlag 2000