

# Implementation of Cryptographic Primitives

Shreya Rajkumar<sup>1</sup>, Dr. Chester Rebeiro<sup>2</sup>

Student ECE National Institute of Technology, Tiruchirappalli  
Tanjore Main Road, National Highway 67, India

Asst. Prof, Dept. of Computer Science and Engineering, Indian Institute of Technology, Madras

**Abstract** — The efficiency of cryptographic library depends on the implementation of multi-precision algorithms. In this paper, implementation of algorithms for modular addition, subtraction, comparison, Extended GCD Algorithm, Montgomery multiplication, Montgomery Exponentiation is discussed and the developed library is tested for correctness and analyzed on various platforms. The developed library works for very large numbers. It is also scalable from 8 bit to 64 bit for a wide range of platforms that includes embedded controllers and DSP processors.

**Keywords** — Chinese Remainder Theorem (CRT), Cryptography, Functionality testing, Montgomery Multiplication, Montgomery Exponentiation, Multi-precision Library, RSA Algorithm..

## I. INTRODUCTION

In today's computer-centric world, cryptography is most often associated with scrambling plaintext into cipher text by the process of encryption. Modern cryptography concerns itself with the following objectives:

- Confidentiality
- Integrity
- Non-repudiation
- Authentication.

In order to realize the above mentioned goals, implementation of cryptographic primitives requires integers of extremely large magnitude. This can be realized by using a multi precision library. There are already a large number of multi-precision libraries available. However, one of this proposed library's developed feature is that it is compatible with many systems. It does not use any predefined functions and there is no dynamic memory allocation. The advantages of these are the following :

- Memory is limited in embedded systems.
- Embedded systems can run for years which can cause wastage of memory due to fragmentation.
- Dynamic memory allocation is slow.
- Dynamic memory allocation makes it difficult to debug especially with limited debugging tools that are present on the embedded system.

## II. PROCEDURE FOR IMPLEMENTATION OF THE LIBRARY

The multi-precision number is represented as structure *bignum* which contain words of size specified by *word\_size*. The library is scalable to any magnitude by just giving the word size (*word\_size*) and input size (*No\_of\_bits*) as parameters.

The inputs and structure are defined in the following way.

The example here indicates input of 1024 bit and word size of 64 bits

- define *No\_of\_bits* 1024
- define *word\_size* 64
- define *max\_digits* (*No\_of\_bits*/*word\_size*)
- typedef unsigned long *word*;
- typedef struct  
  { *word* *digits*[*max\_digits*];  
    *int* *sign*;  
  } *bignum* *t*;

## III. ALGORITHMS

---

### Multi-Precision Addition

---

**procedure** ADD(*word* \**r*, *word* \**a*, *word* \**b*) ◀Inputs  
: array of words, performs  $c = a + b$   
2: *carry* ← 0  
3. for *i* from 0 to *n* do  
4: *word* *t* ← *a*[*i*]  
5: *t* ← *t* + *carry*  
6: *carry* ← (*t* < *carry*)  
7: *word* *l* ← *t* + *b*[*i*]  
8: *carry* ← *carry* + (*l* < *t*)  
9: *r*[*i*] ← *l*  
10: *i* ++

---

---

Multi-Precision Subtraction

---

1: **procedure** Sub(word \*r, word \*a, word \*b)  
 2: borrow ← 0  
 3: word temp1 ← a[i]  
 4: word temp2 ← b[i]  
 5: r[i] ← (temp1 – temp2 – borrow)  
 6: if (temp1 ≠ temp2) borrow = (temp1 < temp2)  
 7: i ++

---



---

Modular Addition

---

1: **procedure** ModularAdd(bignumber \*r, bignumber \* a, bignumber \*b, bignumber \*m)  
 2: ADD(r, a, b)  
 3: if (COMPARE(r,m) >= 0)  
 4: SUB(r, r,m)

---



---

Bignumber Comparison

---

1: **procedure** COMPARE(bignumber\*a, bignumber\*b)  
 2: word \*ap ← (a → digits) < digits is an array of words in the structure a  
 3: word \*bp ← (b → digits)  
 4: **for** i from n to 0 **do**  
 5: if (ap[i] ≠ bp[i])  
 6: **return** ((ap[i] > bp[i])?1 : -1)  
 7: **return** 0

---



---

Modular Subtraction

---

1: **procedure** ModularSUB(bignumber \* r, bignumber \* a, bignumber \* b, bignumber \* m)  
 2: if (COMPARE(a, b) < 0)  
 3: SUB(r, b, a)  
 4: SUB(r,m, r)  
 5: else  
 6: SUB(r, a, b)

---



---

Montgomery Multiplication:

---

1: **procedure** MontMul(bignumber \*m, bignumber \*a, bignumber \* b) < Performs (a\*b)R<sup>-1</sup> mod m  
 2: R ← 0  
 3: **for** i from 0 to (n – 1) **do**  
 4: ti ← (r0 + aib0)m1 mod b  
 5: R ← (R + aib + tim)/b  
 6: If R >= m then R <- R – m  
 7: **return** R

---



---

Extended GCD Algorithm

---

Given two positive integers x and y, the algorithm returns a, b and v such that ax + by = v, where v = gcd(x, y).

1: g = 1  
 2: While x and y are both even : x = x/2, y = y/2, g = 2g  
 3: u = x, v = y, A = 1, B = 0, C = 0, D = 1  
 4: **for** u is even do the following **do**  
 5: u = u/2  
 6: If A and B are even, then A = A/2, B = B/2; otherwise, A = (A + y)/2, B = (B – x)/2.  
 7: **for** v is even do the following **do**  
 8: v = v/2  
 9: If C and D are even, then C = C/2, D = D/2; otherwise, C = (C + y)/2, D = (D – x)/2.  
 10: If u >= v, then u = u – v, A = A – C, B = B – D  
 11: otherwise, v = v – u, C = C – A, D = D – B.  
 12: If u = 0, then a = C, b = D, and return(a, b, g.v); otherwise go to Step4

---



---

Montgomery Exponentiation:

---

1: **procedure** MontExp(bignumber \*m, bignumber \*a, bignumber \*e) < Performs ae mod m  
 2: x1 ← MontMul(a, R2 mod m), A ← R mod m  
 3: **for** i from n to 0 **do**  
 4: A ← MontMul(A, A)  
 5: If ei = 1 then A ← MontMul(A, x1)  
 6: A ← MontMul(A, 1)  
 7: **return** A

---

RSA was implemented along with CRT in order to speed up the calculations during decryption.

**IV. IMPLEMENTATION AND RESULTS**

**Functionality testing :**

Python scripts were written in order to test the functionality of the library developed. Python’s inbuilt bignum library was used to check the correctness of the developed C library.

```

First Number is :
12432848055194002703081943631899563108714179450438354999750685331976673395864557
07332274035417903438311241636261874613510395681689963781567561313950691622894596
20764659132269800992075640138530551757000908779654815829106316474330589833456814
456645668556626982214658953971805427335081631318600414005103177582083

Second number is :
15847000253206254881303061265876227741665192055109720085614979737167414915283621
99957226735098937002399597563855714647040793882325215901811087621678420800952922
62159254763566922181518570726370913009492762653981461721607998877925597545339290
108187975800806781722369843449992450158303865486389859399266161165376

The addition result is :
2827984830840025758438500489777579085037937150554807508536566506914408831148179
07289500770516840440710839200117589260551189564015179683378648935629112423847518
82923913895836723173594210864901464766493671433636277550714315352256187378796104
564833644357433763937028797421797877493385496804990273404369338747459

Addition Success
    
```

Fig. 1. Testing the developed addition algorithm with Python’s in-built Multiprecision library.

```

The first number is :
1476668844968131965015231686829398574487167318855269352386469012910391462359872
34355839413066841510682438108060305009655010995861072959167319357730901568040387
8942724086644147975265209260787246863347543814236780832276965025730954755539511
267181597404394797746801971603609969270550578737293059288250135010

The Second number is :
15174044980849893173591085040701594769893937167402342190611087283019027541552487
60210972590063517204476917210805105553953021728385866022778065379996696294942588
20260177653980140940072939815915520864592168119792268485907742454228827306355441
67156923464503927316878117515939867549726281795770694380040663288060

The Subtracted Result is :
-136973761358817612085758533587219619540676984854707283827244038172798839531650
0367753886487028330534086733999990750529875206287975872686133344422300613813854
94131740524531572614254641888983679617825741373836859040268004595165573183080149
054485107490459979339131315544336257580455731217033401320752413153050

Subtraction Success
shreya@shreya-Inspiron-5547:~$
    
```

Fig. 2. Testing the developed subtraction algorithm with Python’s in built Multiprecision library.

The RSA algorithm implemented using the developed multi-precision library is evaluated on different platforms and the total time taken by the RSA algorithm which includes encryption and decryption is measured in seconds.

Evaluation of RSA on Intel 64-bit System

Input Size	Without CRT	With CRT
128	0.000081	0.000019
256	0.000502	0.000304
512	0.001680	0.010923
1024	0.091220	0.077648

Evaluation of RSA on LPC Xpresso 1347

Input Size	Without CRT	With CRT
128	2.58000	1.224000
256	19.674000	12.958000
512	169.754000	118.123000

We can see that in the above cases, RSA with CRT outperforms RSA without CRT.

**V. CONCLUSION**

This paper explains implementation details of multi-precision arithmetic library which is key for performing cryptographic operations. The RSA algorithm is implemented and CRT technique is used for faster implementation. The library of multi-precision arithmetic operations can be used in implementing not only RSA but also various cryptographic primitives like ECC, Elgamal etc. The main advantage of the developed library when compared to other available libraries is that it is supported by almost all the systems as neither dynamic memory allocation nor predefined functions were used.

**ACKNOWLEDGMENT**

I would like to thank IIT Madras for the facilities provided. I would also like to thank my parents for their support and encouragement.

**References**

- [1] P. C. v. O. Alfred J. Menezes. Handbook of Applied Cryptography. CRC Press, August
- [2] Koltuksuz A., Hışıl H. (2005) Crympix: Cryptographic Multiprecision Library. In: Yolum ., Güngör T., Gürgen F., Özturan C. (eds) Computer and Information Sciences - ISCIS 2005. ISCIS 2005. Lecture Notes in Computer Science, vol 3733. Springer, Berlin, Heidelberg.
- [3] Koc, C.K.: High Speed RSA Implementation. RSA Laboratories. TR201 (1994)
- [4] Bosselaers, A., Govaerts, R., Vandewalle, J.: A Fast and Flexible Software Library for Large Integer Arithmetic. In: Proceedings 15th Symposium on Information Theory in the Benelux, Louvain-la-Neuve (B), pp. 82–89 (1994)