Crympix: Cryptographic Multiprecision Library

Ahmet Koltuksuz and Hüseyin Hışıl

Izmir Institute of Technology, College of Engineering, Dept. of Computer Engineering, Gülbahçe, Urla, 35430 Izmir, Turkey
{ahmetkoltuksuz, huseyinhisil}@iyte.edu.tr

Abstract. This paper delineates the results gained throughout the development of a cryptographic multiprecision[1] integer library, CRYMPIX. To obtain the know-how for cryptographic computation and thus being able to create the high level cryptographic protocols in an in-house-fashion are the main reasons of this development. CRYMPIX is mainly designed to supply code readability and portability plus an increased performance over other similar libraries. The whole work is achieved by detailed investigation of current algorithms and multi-precision libraries. The selected algorithms are discussed by means of efficiency and various implementation techniques. The comparative performance measurements of CRYMPIX against other multiprecision libraries show that the overall performance of CRYMPIX is not behind its predecessors if not superior.

1 Introduction

The efficiency of a cryptographic implementation considerably depends on its low-level multiprecision library. A cryptographic library is said to be competitive among its alternatives if it is engineered with not only 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 some 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 of a cryptographic library for every researcher who is in the field of cryptology.

Either designed for cryptographic use or not, most of the current multiprecision libraries implement arithmetic, logic and number theoretic routines. CRYMPIX also offers those capabilities. What makes CRYMPIX different from its alternatives is its design criteria as well as its performance. Our measurements showed that the overall performance is not behind the other libraries. In

[1] Arbitrary-precision, multiprecision and bignum are synonyms. In the subsequent parts of this text, the term multiprecision is preferred to address the multiple-precision.

P. Yolum et al.(Eds.): ISCIS 2005, LNCS 3733, pp. 884–893, 2005.

this paper, we explain the principles of which CRYMPIX is developed by plus will compare its performance with the others.

It is known that the asymmetrical cryptosystems require multiprecision arithmetic when they are run on fixed precision processors. For instance, if an RSA implementation uses 4096-bit key size then at least 128 computer words is needed to store and process this key on 32 bit architecture. To address this necessity, many libraries are developed up to now. The most popular ones among these libraries are GNU GMP, Shamus Software MIRACL, LibTomMath, PARI/GP, BigNum, Java BigInteger, Bouncy Castle, Magma, Maple, Mathematica, and MuPAD. All of these 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.

Excluding the scientific interpreters, the efficiency of a cryptographic library is directly proportional to the overall performance of some well known number theoretical routines such as modular powering, greatest common divisor and multiplication. Therefore, almost all of these libraries contain specialized parts for several different architectures. So, it is clear that the implementation has a tendency of multiplying very rapidly in terms of coding efforts which in turn requires handling of multiple libraries in one project thus the growing pains of code management.

In this study, we discuss how to minimize the development effort without causing any performance degradation. Finally, we compare the outcome of our design decisions with that of some other libraries.

2 Basic Design Criteria

Common design criteria of most multiprecision libraries are representation of numbers, programming language selection, memory management, portability, and functionality [1]. 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.

2.1 Representation of Numbers

Almost all multiprecision libraries use positive integer vectors that are analogous to the radix representation that is given in below equation 1.

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

CRYMPIX also uses this representation. 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. Radix representation is further explained in [7].

2.2 Programming Language

The preferred languages 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 paper since CRYMPIX aims to be an educational library in which the most suitable algorithms are being implemented for cryptographic use. Nevertheless, we have included a performance table that may give the reader an idea of how Assembly affects the performance in Table 1.

On Table 1, MIRACL 4.8, GMP 4.1.4, Java BigInteger and CRYMPIX are benchmarked via their integer multiplication function. We prepared test suits of

Table 1. Integer Multiplication benchmark results. (microseconds).

Size	**CRYMPIX**		**MIRACL**		**GMP**		**Java BigInteger**
	C, v1	C, v2	C	C+Asm	C	C+Asm	
1K	21	11	17	6	23	4	32
2K	69	41	68	26	74	15	132
4K	219	133	277	104	235	47	512
8K	673	410	1097	411	731	154	2630

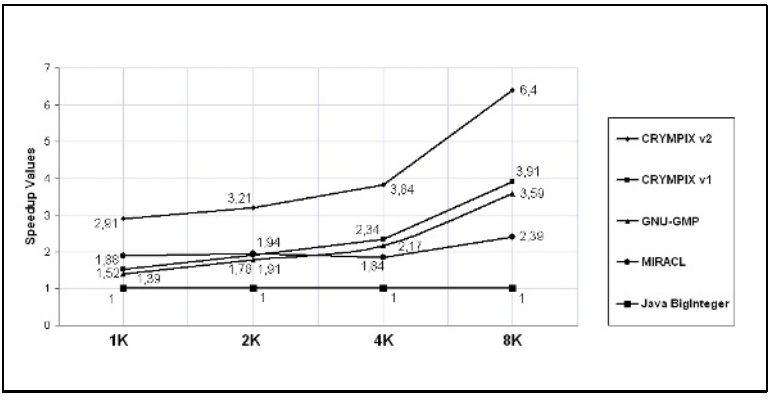

Fig. 1. Speedup values obtained by the results in Table 1

1K, 2K, 4K, and 8K each having 1000 randomly selected inputs. We decoupled the I/O time to get more accurate results. Excluding Java BigInteger, 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.

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. Fig. 1 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 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 Fig. 1 we have provided the performance comparison of libraries for C only built at optimization level 2 (excluding Java BigInteger).

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 `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 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 is slower; on the other hand, code development is far easier.

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 code below introduces CRYMPIX with an integer addition example.

```
CRYMPIX Code Example for Integer Addition.

CZ a, b, c;
crympix_init(100, 20);   // Max words, max instances.
...
a = cz_init();
b = cz_init();
c = cz_init();
...
cz_add(c, a, b);   // c = a + b.
...
cz_kill(a);
cz_kill(b);
cz_kill(c);
...
crympix_finalize();
```

2.4 Code Readability and Portability

Code readability has been one of the major concerns in CRYMPIX library right from the start. Therefore, function bodies are written as plain as possible and the code organization, a standardized naming and indentation are applied throughout the development. 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. At the lowest level, we handle single-precision arithmetic operations with C macros. A vector layer; which is on top of that, manipulates the operations between a positive integer array and a single-precision operand. The below code provides an idea about the vector layer.

```
Vector Layer Code example.

#define ccm_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++){ \
        cvm_mul_2_add_2(_t, _an[_i], _b, _zn[_i], _t.spu[HIGH]); \
        _zn[_i] = _t.spu[LOW]; \
    } \
    _carry = _t.spu[HIGH]; \
}
```

At the low-level function layer which comes after vector layer, the arithmetic functions are implemented and the relevant code example is given below.

```
Low-level Function Layer Code example.

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

    ccm_mul_1(z[bl], z, b, bl, a[0], 0);
    for(i = 1; i < al; i++){
        ccm_inc_n_mul_1(z[i + bl], (z + i), b, bl, a[i], 0);
    }
}
```

The layered approach simplifies the function bodies, prevents code repetitions; hence less tedious development phase.

2.5 Selection of Algorithms

Almost all libraries use the similar algorithms in high speed multiprecision arithmetic. Therefore, we limit our decisions with algorithm selection criteria.

Table 2. Algorithms in use for multiprecision multiplication

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

Addition, Subtraction and Shift. Addition and subtraction are done as they are explained by Knuth in [5] and Menezes in [7]. The operation starts from the least significant word and carry/borrow bits are transferred to the following steps of the algorithm. For shifting multiprecision numbers the basic bitwise operators of C language are convenient to use. Generally, operations such as addition, subtraction, clone, shift, and compare are relatively cheaper therefore all of the cryptographic libraries employ the similar suites.

Multiplication. The efficiency of most cryptographic libraries depend on the cost of multiprecision multiplication operation. Table 2 summarizes the popular multiplication methods, their complexities, and of their usage intervals.

In the cryptographic applications Basecase [5,7] and Karatsuba [5,6] multiplication algorithms are frequently used. Although the above seen FFT algorithm is asymptotically faster, it is more costly as far as the cryptographic applications concerned.

Division. CRYMPIX uses basecase division algorithm explained in Knuth [5]. If numbers get slightly larger than 1500 bits, then Divide-and-conquer algorithm [2] which is a recursive variant of the basecase division, gets to be utilized often and GMP includes it too.

Greatest Common Divisor (GCD), Extended 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 [4,8] 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 by Jebelean [3] in Table 3. We have used approximative condition of GCD and double-precision techniques. The speedup values are given in Fig. 2.

We also provided the performance comparison of CRYMPIX Lehmer GCD and GMP Generalized GCD in Table 4. The expected value is a constant speedup around $0,75$ which is actually a slow down factor. This is depicted in Fig. 3. The lower performance of CRYMPIX below the expected value in smaller operands is due to the absence of binary GCD implementation.

Table 3. Standard Lehmer GCD vs. Modified Lehmer GCD (microseconds)

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

Table 4. 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

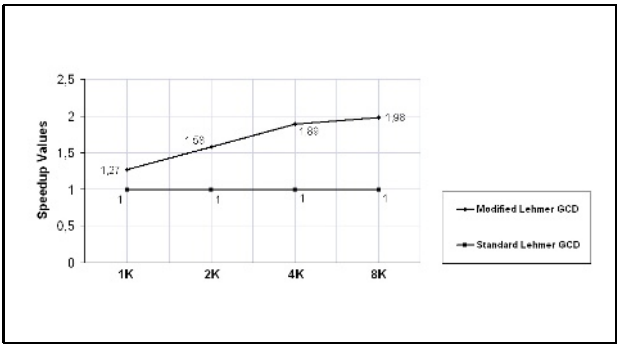

Fig. 2. Speedup values for Modified Lehmer GCD over Standard Lehmer GCD, derived from Table 3

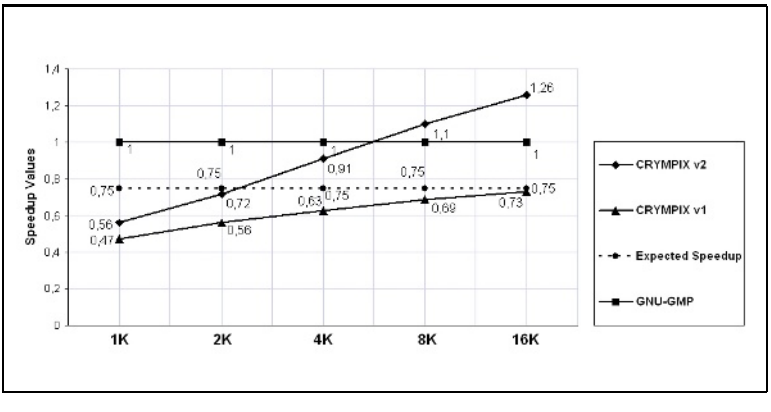

Fig. 3. Speedup values for CRYMPIX Lehmer GCD over GMP Generalized GCD, derived from Table 4

Table 5. 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

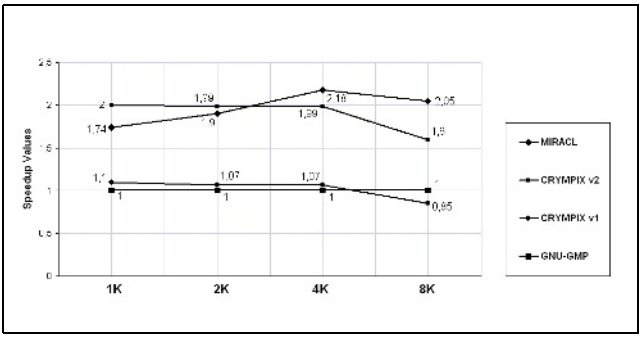

Fig. 4. Speedup values for CRYMPIX and MIRACL over GMP in modular exponentiation

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 ToomCook-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 5 with time measurements of modular powering for 1K, 2K, 4K and 8K numbers with GMP, CRYMPIX, and MIRACL. Fig. 4 provides corresponding speedup values.

3 Results and Contribution

In this study, 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. The first release 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. Our next study will be on the layered adaptation of this library to distributed environments.

References

1. Bosselaers A., Govaerts R., Vandewalle J.: A Fast and Flexible Software Library for Large Integer Arithmetic. Proceedings 15th Symposium on Information Theory in the Benelux, Louvain-la-Neuve (B). **82-89** (1994)
2. Burnikel C.: Fast Recursive Division. Max-Planck-Institut fuer Informatik Research Report. **MPI-I-98-1-022** (1998)
3. Jebelean T.: Improving the Multiprecision Euclidean Algorithm. Proceedings of DISCO'93, Springer-Verlag LNCS 772. **45-58** (1993)
4. Jebelean T.: A Generalization of the Binary GCD Algorithm. ISSAC 93. **111-116** (1993)
5. Knuth D.E.: The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 3rd edition, Addison-Wesley. (1998)
6. Koc C.K.: High Speed RSA Implementation. RSA Laboratories. **TR201** (1994)
7. Menezes A.: Handbook of Applied Cryptography. CRC Press, 608. (1993)
8. Weber K.: The Accelerated Integer GCD Algorithm. ACM Transactions on Mathematical Software, v.2. **111-122** (1995)