Multiprecision ANSI C Library for Implementation of Cryptographic Algorithms on Microcontrollers

Jan Říha
Department of Digital Design
Faculty of information technology
Czech Technical University in Prague
Email: rihaja11@fit.cvut.cz

Jakub Klemsa
Department of Telecommunication Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Email: jakub.klemsa@fel.cvut.cz

Martin Novotný
Department of Digital Design
Faculty of information technology
Czech Technical University in Prague
Email: martin.novotny@fit.cvut.cz

***Abstract*—Current cryptographic algorithms work with operands that are several times wider than the machine word, e.g., the still popular RSA algorithm shall use at least 2 048-bit keys. Such algorithms therefore require libraries that implement multiprecision arithmetic. Existing libraries are either not tailored for microcontrollers, or they implement an incomplete set of multiprecision operations, which limits the implementation of some unusual cryptographic algorithms on microcontrollers.**

In this work, we present a novel ANSI C library that implements also some less common operations like, e.g., multiprecision integer division. The library was designed with respect to the use on microcontrollers and has been tested on ARM M4-based microcontroller Microchip CEC1302.

I. Introduction

Most of the common cryptographic algorithms used today can be easily implemented using one of the existing cryptographic libraries. However, if one aims to implement a little more exotic algorithm, none of the existing libraries is suitable. For this reason, we decided to implement a new multiprecision arithmetic library called *bigi*, which is intended as a building block of cryptographic algorithms, however, to some extent, it can be used in a general way as well.

This paper is structured as follows: First, we discuss the state-of-the-art in Section II and give essential preliminaries for integer arithmetic's in Section III. In Section IV, we describe our implementation in detail and provide a brief evaluation in Section V.

II. State of the Art

Implementations of multiprecision (aka. arbitrary precision) arithmetic's can be divided into two groups according to the operations implemented, i.e., concerning their usage.

The first group consists of multiprecision libraries for general mathematical computations, such as GNU Multiple Precision Arithmetic Library (GMP) [1] for C/C++, gmpy2 [2] for Python, Java BigInteger [3], and many more. These libraries are usually used in computing systems such as Wolfram Mathematica and Maple, or for scientific purposes. The GMP library is also used in the source code of the GCC compiler collection. Libraries for general computations are not optimized for the use with microcontrollers because they use dynamic memory allocation and are designed to handle large numbers that exceed the memory size of most microcontrollers.

The second group consists of multiprecision libraries that are employed as a building block of cryptographic operations, which are often handling with numbers bigger than the machine word. The most popular libraries are OpenSSL [4] and WolfSSL [5]. There are also libraries dedicated and optimized for microcontrollers, such as ARM-Crypto-Lib [6], AVR-Crypto-Lib [7], and mbedtls [8]. None of these cryptographic libraries implements integer division, which might be necessary for some specific cryptographic schemes (e.g., Paillier cryptosystem [9]). Moreover, ARM-Crypto-Lib and AVR-Crypto-Lib have not been maintained for a long time.

The bigdigits [10] library fits into both groups. This library implements all basic arithmetic operations and offers optimized versions of operations that are used in cryptographic algorithms (such as modular exponentiation). However, it is primarily dedicated for the use on 64-bit processors. Moreover, modular multiplication is implemented by generic multiplication and the reduction step is performed by division. This approach is naïve, hence, if using this library, we would need to implement modular multiplication and modular exponentiation operations. Dynamic allocation can be disabled during compilation, but this requires considerable interference with the library source code.

Another reason for the decision to implement own multiprecision integer library was given by the limitations of the development tools for the target platform. The MikroC for ARM compiler works somewhat differently than the commonly used compilers (GNU GCC toolchain).

To conclude, to the best of our knowledge, there is no existing multiprecision integer library suitable for implementation of standard as well as less common cryptographic algorithms on microcontrollers. Also, the transfer of existing libraries would entail considerable effort comparable to the creation of a new library. Further, by creating a custom library, the homogeneity of the code is ensured, and the algorithms used are efficient and optimized for the use on the target platform.

III. Preliminaries

In order to cover the needs of the broad family of cryptographic algorithms, the library shall implement basic arithmetic operations, namely addition, subtraction, bit shifts, multiplication, and integer division, as well as their modular counterparts. Besides that, modular inversion and greatest common divisor (GCD) are necessary. In the following text, we summarize efficient implementations of the operations mentioned above.

A. *Integer Multiplication*

The basic algorithm for multiplication of large numbers is described by Knuth [11]; its implementation in C is given by Warren [12]. Besides Knuth's algorithm, several more integer multiplication algorithms exist. Karatsuba [13] published an algorithm with $\mathcal{O}(n^{\log_2 3})$ complexity in 1962, however, the fastest known algorithm is by Schöngage-Strassen [14] with $\mathcal{O}(n \cdot \log n \cdot \log \log n)$ complexity. It is used for very large numbers, e.g., the GMP library employs it for numbers longer than 1 700 machine words.

B. *Integer Division*

Integer division is the most complicated basic arithmetic operation. The basic algorithm for multiprecision integer division is described in [11] as Algorithm D. According to Knuth, this algorithm is an adaptation of schoolbook division. A full overview of division algorithms for large integers can be found in [15].

C. *Modular Reduction*

Modular arithmetic is based on the modular reduction operation which, given a number x and modulus n, returns $b = x \bmod n$. The naïve way of implementing modular reduction is through integer division, which returns both the quotient and the remainder. However, this is inefficient due to the computational complexity of integer division.

Barrett reduction [16] is less computationally demanding compared to integer division—it employs only multiplication, addition, subtraction and bit shift operations. Detailed description and a C implementation are given in [17].

D. *Modular Multiplication*

There are several ways of implementing modular multiplication. The naïve approach is to follow classical multiplication by a modular reduction, i.e., in the first step, standard LSB multiplication is performed, and in the second step, the intermediate value is reduced by the modulus.

Another way of implementing modular multiplication is MSB multiplication using the double-and-add algorithm, which only employs left shift and addition. After each shift or addition, a trial division is performed. This approach is less memory demanding than the previous one.

In 1985, Peter Montgomery published a method for fast modular multiplication of large integers [18]. The main idea behind Montgomery's approach is to convert numbers into a different representation (aka. Montgomery domain), which eliminates the need for trial division by the modulus n and also decreases memory requirements of modular multiplication. Due to pre-calculation of parameters, Montgomery multiplication is only useful when multiple multiplications with the same modulus are performed, which poses a typical use in modular exponentiation.

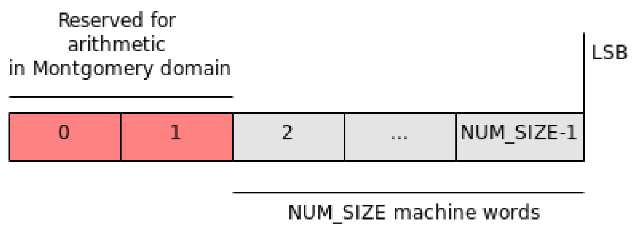

Fig. 1. Number representation in *bigi* library

E. *Exponentiation*

(Modular) exponentiation is the core operation of many asymmetric cryptosystems, such as RSA [19], Diffie-Hellman [20], Paillier cryptosystem [9], Goldwasser-Micali [21], etc.

Left-to-right binary exponentiation (aka. square-and-multiply algorithm) is the basic algorithm for exponentiation [11]. Each iteration of the algorithm consists of one squaring which is followed by multiplication whenever a corresponding bit of an exponent is equal to one. However, this approach is vulnerable to side channel attacks, namely timing attacks [22] and simple power analysis (SPA) [23].

Montgomery ladder [18] is a variant of left-to-right binary exponentiation with the difference that each iteration consists of exactly one squaring and one multiplication, thus making the power consumption and execution time independent of the Hamming weight of the exponent. This approach makes implementation less vulnerable to timing attack and significantly less vulnerable to SPA attack.

Within modular exponentiation, modular multiplication is used. If Montgomery domain is used for modular multiplication, a slight modification of the original algorithm is required and the base must be given in or converted to the Montgomery domain.

IV. Implementation

Our library *bigi* works over a static array of unsigned integer type elements. The size of the array is determined during compilation according to predefined constants, namely size of the machine word on the target platform, and required number of bits in the number representation. For Montgomery multiplication, two elements are added in order to prevent overflow [24], as depicted in Figure 1, which shows the structure of an array representing a number in our library. Endianity of the number was chosen for easier debugging.

The *bigi* library consists of four files: `bigi.h`, `bigi.c`, `bigi_io.h` and `bigi_io.c`. The header file `bigi.h` includes definitions of function prototypes and the following data types:

- `bigint`: an array of machine-word-sized unsigned integers representing a number,
- `bigint_type`: a machine-word-sized unsigned integer,

- `bigint_type_big`: a double-machine-word-sized unsigned integer.

The header file `bigi_io.h` provides prototypes of functions for a conversion of a `bigint` from/to a string, and for a dump to a terminal or serial link. I/O functions are separated so that they can be easily replaced according to the platform or user requirements.

A. Basic Arithmetic Operations

Basic arithmetic operations include addition, subtraction, bit shifts, division and multiplication. Addition, subtraction and bit shifts are straightforward to implement, hence we will only discuss multiplication and division in this section.

1) Multiplication: The current implementation follows [11], however, for the future versions of our library, we plan to implement Karatsuba's algorithm.

2) Division: The division algorithm we employ was proposed in [11] and its C implementation can be found in [12]. On the one hand, other algorithms mentioned in Section III-B have better asymptotic time complexity, on the other hand, their implementation is much more complicated and their speedup only applies to numbers which are significantly larger than those expected to be processed by our library.

B. Modular Arithmetic Operations

As stated before, modular arithmetic is the core of most asymmetric cryptosystems. For this reason, state-of-the-art algorithms were chosen for its implementation.

1) Modular Multiplication: Since the target platform is a microcontroller, the choice of modular multiplication algorithm was affected by the memory demand. For this reason, Montgomery's approach was chosen for modular multiplication. It only requires $n+2$ machine words to store intermediate results [24], where n is the number of machine words in the representation of the number.

Multiplication in the Montgomery domain requires a precomputation of a parameter derived from the modulus, and converting both operands to the Montgomery domain. In order to multiply two operands, a total of three calls of Montgomery multiplication are required.

The transformation of operands and the precomputation represent an overhead that pays off whenever several modular multiplications are performed consecutively—typically with modular exponentiation or when all computations following the modular multiplication can be done in the same Montgomery domain.

Because of the overhead, modular multiplication is implemented also in a second way—a standard MSB multiplication using the double-and-add algorithm. This algorithm needs the same number of machine words to store the intermediate result as the number representation itself.

2) Modular Exponentiation: Modular exponentiation is implemented by the Montgomery ladder [25], which provides a protection against timing attacks—simply because squaring and multiplication operations take place in each iteration. The main disadvantage of this algorithm is the longer calculation time compared to the square-and-multiply approach.

Multiplication and squaring in the Montgomery ladder are implemented in two ways. The first, naïve, uses standard MSB multiplication. The second way employs multiplication in the Montgomery domain [24].

3) Modular Reduction: Modular reduction is implemented according to Barrett [16] which requires a precomputation of a parameter μ. This parameter can be used repeatedly until the modulus changes. It can also be used to implement modular multiplication, however this approach requires more memory because the whole product must be stored as an intermediate value.

4) Modular Inversion: Inverse element of $a \in \mathbf{Z}_n^*$ is a number denoted by a^{-1}, for which it holds that $a \cdot a^{-1} \equiv 1 \pmod{n}$. One way to compute a modular inverse is to employ the Extended Euclidean Algorithm, another possibility is to use Euler's theorem:

$$a^{\phi(n)} \equiv 1 \pmod{n}, \tag{1}$$

where $\phi(\cdot)$ is Euler's totient function. If we multiply the equation by a^{-1} (assuming the inverse exists), we get

$$a^{\phi(n)-1} \equiv a^{-1} \pmod{n} \tag{2}$$

A disadvantage of this procedure is the implicit assumption of the existence of the inverse and also the fact that it is necessary to know $\phi(n)$. Note that calculation of $\phi(n)$ is as computationally demanding as factorization of n.

Therefore, for a library to be generally usable, modular inversion is implemented using Extended Euclidean Algorithm, which can detect possible absence of the inverse.

C. Advanced Functions

At this moment, the only advanced function implemented in our library is the greatest common divisor (GCD). GCD of two integers a, b is the greatest positive integer d, such that it holds $d \mid a$ and $d \mid b$.

In the algorithm 14.54 in [17], a binary GCD algorithm is used. The search method only employs subtraction, division by two, and multiplication by two. Note that division and multiplication by two are simple operations that can be implemented as right and left bit-shifts, respectively.

V. Evaluation

The *bigi* library was tested on Microchip's CEC1302 microcontroller, which is based on an ARM M4 core.

Including the library into an existing project is simple and straightforward—it requires just adding the header and C files into corresponding compilation path or an IDE project.

The representation of a number in our library only consists of an array of machine-word-sized unsigned integers, which makes understanding of the code easier. It also allows the user to implement custom functions to extend the library.

The whole library was compiled by *MikroC Pro for ARM* compiler by Mikroelektronika and it occupies only $184\,\text{kB}$ of program memory.

Since the library is still under active development, we have not made any speed comparisons with existing libraries such as ARM-Crypto-Lib. We only compared the speed of modular exponentiation in our library to the speed of the hardware RSA accelerator on the CEC1302 microcontroller: our (software) implementation of 2 048-bit modular exponentiation is only three times slower.

VI. Conclusions

We presented a novel ANSI C library which also implements less common operations like, e.g., multiprecision division. This enables an implementation of some specific cryptographic algorithms. The library is dedicated for an easy use on microcontrollers; it was tested on ARM microprocessor CEC1302. After compilation, the library occupies only 184 kB of program memory, which makes it available for a wide range of microcontrollers. The library is available under MIT license from [26].

VII. Future Work

We aim to change the multiplication algorithm for Karatsuba's multiplication algorithm and we also plan to add certain assembly level optimizations. Our further goal is to implement and evaluate some types of side channel attack countermeasures.

Acknowledgment

This work was supported by the Grant Agency of the Czech Technical University in Prague, grants No. SGS17/213/OHK3/3T/18 and SGS19/109/OHK3/2T/13

References

[1] Free Software Foundation, "The gnu multiple precision arithmetic library [online]," 2016, [cit. 2019-03-23]. [Online]. Available: https://gmplib.org/

[2] Python Software Foundation, "gmpy2 2.0.8 [online]," 2018, [cit. 2019-03-23]. [Online]. Available: https://pypi.org/project/gmpy2/

[3] ORACLE, "Class biginteger [online]," 2014, [cit. 2019-03-23]. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

[4] OpenSSL Management Committee, "Openssl," 2019, [cit. 2019-04-03]. [Online]. Available: https://www.openssl.org/

[5] wolfSSL Inc., "Embedded tls library," 2019, [cit. 2019-04-03]. [Online]. Available: https://www.wolfssl.com/

[6] das labor, "Arm-crypto-lib [online]," 2014, [cit. 2019-04-03]. [Online]. Available: https://wiki.das-labor.org/w/ARM-Crypto-Lib/en

[7] das labor, "Avr-crypto-lib [online]," 2015, [cit. 2019-04-03]. [Online]. Available: https://wiki.das-labor.org/w/AVR-Crypto-Lib/en

[8] ARM Holdings, "mbed tls," 2019, [cit. 2019-04-03]. [Online]. Available: https://tls.mbed.org/

[9] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology — EUROCRYPT '99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

[10] DI Management, "Bigdigits multiple-precision arithmetic source code [online]," 2016, [cit. 2019-03-23]. [Online]. Available: https://www.di-mgt.com.au/bigdigits.html

[11] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997.

[12] H. S. Warren, *Hacker's Delight*, 2nd ed. Addison-Wesley Professional, 2012.

[13] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Dokl. Akad. Nauk SSSR*, vol. 145, pp. 293–294, 1962.

[14] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, Sep 1971.

[15] K. Hasselström, *Fast Division of Large Integers – A Comparison of Algorithms*. Stockholm, Royal Institute of Technology, Department of Numerical Analysis and Computer Science, 2003, Master's Thesis.

[16] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[17] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation - Math. Comput.*, vol. 44, pp. 519–519, 04 1985.

[19] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[20] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[21] S. Goldwasser and S. Micali, "Probabilistic encryption &amp; how to play mental poker keeping secret all partial information," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '82. New York, USA: ACM, 1982, pp. 365–377.

[22] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[23] P. Kocher, J. Jaffe, and J. Jun, "Introduction to differential power analysis," in *Advances in Cryptology - CRYPTO '99*. Berlin, Springer-Verlag, 1999, pp. 388–397.

[24] J. W. Bos and P. L. Montgomery, "Montgomery arithmetic from a software perspective," *IACR Cryptology ePrint Archive*, vol. 2017, p. 1057, 2017.

[25] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[26] J. Říha, "bigi library," 2019, [cit. 2019-04-12]. [Online]. Available: https://github.com/takyrajdr/bigi