Mathematical Problems of Computer Science 17, 1997, 57-62.

A New Fast Modular Exponentation Algorithm

G. H. Khachatrian , A. B. Andreasyan, K. P. Zelenko

Institute for Information and Automation Problems
of NAS of RA and YSU

The modular exponentation is the basis of many well known public key cryptosystems. In this paper a new fast exponentation algorithm is considered. A comparision of the algorithm with known ones is also presented.

1 Introduction

The fast exponentation problem is a key issue in the public key cryptography. Many well-known cryptographic systems are based on the computation of x^n where x is an element of GF(P) and n is an integer of 512 bit length and longer.

The exponentation is also used in many methods to find the prime numbers (for example in Rabin-Miller's method to find the 512 bit length prime number it is necessary to do about 500-600 exponentations) in digital signature systems, etc.

There are many known methods for computation of exponentation. All the methods are based on the multiplication in GF(P). Computational complexity of the exponentation can be estimated by the following formula

$$C(n) = M(n) + S(n),$$

where M(n) is the number of multiplications, S(n) shows how many times the intermediate result is squared. Note, that in all these methods for the fixed n, S(n) is constant. Therefore, for the fast exponentation it is necessary to reduce the number of multiplications. When precomputed results are stored, it is also important to reduce the number of squarings for precomputed values.

In this paper we consider a method which allows to reduce the memory size and the number of squarings for precomputed results by performing, in average, the same number of multiplications. In the Section 2 the comparison of the complexity of different methods of exponentation computing is discussed. The proposed algorithm for the fast exponentation is described and evaluated in Section 3.Its usage in modular expo- nentation and experemental results are considered in Section 4. The final conclusion is formulated in Section 5.

2 Comparison of different methods of exponentation

2.1

The well-known method of exponentation is the binary one [1]. For large pseudo- random numbers n this method takes (in average)

$$E(M(n)) = \frac{|n|}{2} \qquad (1)$$

multiplications, where $|n| = \lfloor \log_2^n \rfloor + 1$ is the bit length of n, and $S(n) = |n|$.
For the full estimation let's denote the necessary memory size by V(n). For the binary method $V(n) = 1$.

2.2
Straightforward generalization of the binary method is a h-ary method. In this algorithm 2^h precomputations must be done, where h is the bit length of the window. The average number of multiplications (including precomputations) is

$$E(M(n)) = \frac{|n|}{h} + 2^h - 2 \qquad (2)$$

and the number of squarings (including precomputations) is $S(n) = |n|$ and $V(n) = 2^h$.

2.3
There is a method based on the Lempel-Ziv(LZ) data compression algorithm[7]. This method parses the string from the least to the most significant bit by the LZ-algorithm[3], i.e. a binary "compression" tree is created, where the path n_i is a substring of the exponent from the root to node i, and node i contains the partial result x^{n_i}.

The exponent is calculated from the most significant bit on the basis of partial results obtained in the process of parsing. The estimation of complexity of this method is equal to

$$C(n) = L - h/2 + k + k/2 + h$$

where

$$M(n) = k + k/2 \quad S(n) = L - h/2 + h$$

(including precomputations), k is the number of nodes in the tree, h is the tree depth and L = log(n). As it is shown in [7] h = log(L) - loglog(L) and k = L/log(L).
Considering the last equations one can obtain

$$M(n) = 3L/2\log(L),\ S(n) = L + 1/2(\log(L) - \log\log(L)),\ V(n) = L/\log(L)$$

This method is effective, when the exponent is compressible.

2.4
There also exists a method based on the LZ-algorithm[2] , which uses the following representation of n

$$n = (d_1 0^{k_1} d_2 0^{k_2} d_3 \ldots d_t 0^t) \qquad (3)$$

where 0^{k_i} denotes the series of k_i zeros. The set of d_i has to satisfy the following conditions:

- the first symbol of every d_i is 1;
- d_i must have the tree code structure

X^n must be calculated as

$$x^n = (((x^{d_1})^{2^{k_1+|d_2|}} x^{d_2})^{2^{k_2+|d_3|}} \ldots)^{2^{k_t+|d_t|}} x^{d_t}$$

where $|d_i| = \lfloor \log d_i \rfloor + 1$.The set of d_i may be chosen by different ways. If

$D=\{1, 2,\dots,2^{h-1}-1, 2^{h-1}+1, 2^{h-1}+3,\dots, 2^{h}-1\}$
we have the case proposed in [2]. It is easy to see that the total number of symbols in the set D $3\cdot 2^{h-2}$.

In this case, including the precomputations, the average number of multiplications is equal to

$$E(TM(n)) = \frac{|n|}{h+1}+2^{h-1}-1 \quad , \tag{4}$$

and the total number of squarings is $S(n) = |n| + 3\cdot 2^{h-2} - 2$ and the memory size is

$$V(n) = 3\cdot 2^{h-2} - 1. \tag{5}$$

If n is the random sequence of independent binary variables and p is the probability of occurrence of the symbol 1, as it is determined in [4], then the average number of multiplications is equal to

$$E(M(n)) = \frac{|n|}{h-1+1/p}+2^{h-1}$$

3 The proposed method

As it was mentioned above the necessary memory size for the **2.4** algorithm is equal to (5). In order to decrease the memory size we propose to choose the set D by the following way

- each d_i must begin and end by 1, that is

$$D = (1, 3, 5, \dots, 2^{h}-1) \tag{6}$$

for example, for h=4 the following set is obtained (1, 11, 101, 111, 1001, 1011, 1101, 1111).

This algorithm uses h -bit length window, sliding on the exponent bits. The representation of n in the formula (3) is considered , where D is chosen as described above.
In general, for the h- bit length window we will have 2^{h-1} elements and, accordingly, $2^{h-1}-1$ precomputations must be done and stored. So, in each step of computation the window of the length h is considered and precomputed result of the content of window is used. The next window which should be considered must be the one starting with "1" . If n is a random sequence of binary independent distributed variables with $Pr(n_i=1) = p$ and $Pr(n_i=0)=q$,where $q=1-p$, then the average distance between starting points of different windows will be equal to the mathematical expectation.

$$L(n)= p\cdot h+ p\left[q(h+1)+q^{2}(h+2)+\dots+q^{r}(h+r)+\dots\right] \tag{7}$$

By simplifying (7) we get

$$L(n) = p\cdot h + p\left[\frac{q\cdot h}{1-q}+q(1+2\cdot q+3\cdot q^{2}+\dots+r\cdot q^{r-1}+\dots)\right]$$

$$L(n) = h + \frac{q}{p} = h+\frac{1}{p}-1 \tag{8}$$

The average number of multiplications will be equal to

$$E(M(n)) = \frac{|n|}{L(n)} + 2^{h-1}-1$$

and

$$E(M(n)) = \frac{|n|}{h+1/p-1} + 2^{h-1}-1 \qquad (9)$$

When $p = q = 1/2$ the average distance will be equal to $L(n)= h+1$ and substituting p and q in (9)

we obtain $$E(M(n)) = \frac{|n|}{h+1} + 2^{h-1}-1 \qquad (10)$$

If $q>p$, then the average distance will be increased by q/p and the average number of multiplications will be decreased by one.

As we see, the average number of multiplications is the same as in **2.4,** but the total number of squarings is equal to $S(n) = |n|+1$. Since we keep only odd degrees of x, then memory size is $V(n) = 2^{h-1}$. They can be easily computed if we increase the memory size by one and store x^2. This will speed up the precomputations, that is each of the i-th exponent will be obtained by multiplying x^2.

$$X^i = X^{i-1} \cdot X^2 \quad i=1,2^{h-1} \qquad (11)$$

In order to obtain the optimal average number of multiplications for the corresponding length of window, the function (10) should be minimized over h. Solving (10) for the n=256, 512, 1024, 2048 we get , that h=4, 5, 6, 7.The number of bits in the window can be taken 4 for 256, 5 for 512, 6 for 1024, 7 for 2048.

DESCRIPTION OF THE ALGORITHM:

This algorithm uses h -bit length window, sliding on the exponent bits (started from the most significant bit). The value of h is determined by the bit length of the exponent . Then x^2 and x^{d_i}, where $d_i=\{1,3,5,...,2^h-1\}$, are precomputed by (11). For accumulation of the intermediate results, the additional memory initialized with 1 is used. The first bit of the exponent must be 1.

```
1. Check the next bit of n.
If it is equal to 0 then square the intermediate result and go to step 1.
2. If remaining bits of n are less than h, then h=h-1 and go to step 4.
3. Read the h bits of n and find corresponding binary representation of $d_i$    $i=1,2^{h-1}$.
Then perform $\lfloor \log d_i \rfloor$ squarings, multiply the intermediate result by the $x^{d_i}$ and perform ($h - \lfloor \log d_i \rfloor$) squarings.
    Return to step 1.
4. If h > 0 go to step 3.
5. End.
```

4 Modular exponentation and experimental results.

For computing X^n mod P the above described algorithm is used.The number of squarings is equal to the number of bits in the exponent plus the number of precomputed values. Since, in squaring operation multipliers are the same, then it can be done nearly twice faster tha multiplying with different multipliers.

Let suppose that the number of bits in argument, exponent and modulo are equal. Th number of bits in intermediate result after each operation exceeds twicely that of the modulo P and, so it must be partially reduced by modulo P. The classical and Montgomery's partia reduction [5] are used in modular exponentation. Execution times are shown in Table 1. The implementation is written in ANSI C [6] to be portable to any computer system. The arithmeti parts are obtained with optimized assembly code for Intel 486 processor.

Length of the argument in bits	Binary method classi-cal	h-ary method classi-cal	Proposed method classi-cal	Proposed method with Mongo-mery's
256	0.131	0.113	0.104	0.097
512	0.725	0.616	0.565	0.440
1024	4.994	4.174	3.824	3.051
2048	39.000	31.552	28.576	22.768

Table 1. Execution times (in seconds) for modular exponentation (number of bits in the argument, exponent and modulo are equal , $b = 2^{32}$, on a 50 MHz 80486 based PC).

Basically in cryptography the number of bits in argument is more than 256. For this case it is more effective to use Montgomery's partial reduction.

5 Conclusion

The theoretical and experimental results show, that the proposed fast exponentation algorithm is faster than the h-ary one. It requires the same number of multiplications and less memory size compared with the algorithm given in [2]. In modular exponentation the proposed method with Mongomery's partial reduction is shown to be the best known.

References

1. D.E. Knuth. The Art of Computer Programming vol. 2 Fundamental Algorithm, Addision-Wasley Mass. 1968.

2. I.E.Bocharova, B.D.Kudryashov " Fast Exponentation based on Lempel-Ziv algorithm" Sixth Joint Swedish - Russian International Workshop on Information Theory pp 259 - 263, 1993.
3. J. Ziv, A. Lempel A Universal Algorithm for Sequential Data Compression IEEE Trans. Inform. Theory vol. IT-23 pp. 337-343, 1977.
4. I.E.Bocharova , B.D.Kudryashov "Fast Exponentation Based on Data Compression Algorithms" Seventh Joint Swedish-Russian International Workshop on Information Theory pp 36 -39, 1995.
5. Antoon Bosselaers, Rene Govaerts and Joos Vandewalle " Comparison of three modular reduction function" Advances in Cryptology ,Proc. Crypto '93 CRYPTO' 93 pp.176-186 .
6. "American National Standart for Programming Languages-C," ISO/IEC Standard 9899:1990, International Standart Organization, Geneva, 1990.
7. Y.Yacobi "Exponentation faster with addition chains" Proceeding of Eurocrypt'90