Modular Reduction Methods

Sattar J. Aboud, Edmond Prakash

Abstract— Present public-key schemes are relied mostly on arithmetic operations such as multiplication and exponentiation of large integers, ranging of 128-2048 binary bits. Carrying out calculation of the large length with multiple precisions is not quick and not easy to employ. Most algorithms base on modular reduction methods to decrease a length and complexity to perform their public-key scheme executions more efficiently. In this paper, we concentrate on three recognized modular reduction methods employed to decrease the modular operations. These methods are the Classical, Barrett, and Montgomery. We study the application in an arithmetic exponentiation operation for every method. Results are drawn regarding the accuracy, operation and complexity efficiency of the methods relies on the results achieved.

Index Terms— Barrett modular reduction, Classical modular reduction, Montgomery modular reduction

1 INTRODUCTION

The public-key schemes need efficient algorithms of Carrying out multiplication and exponentiation in Z_p The efficiency of the specific scheme will based on a number of factors, such as parameter length, time memory tradeoffs, hardware and software optimization and arithmetical methods. This paper is mainly concerned the arithmetic methods for efficient performing these modular calculations. As modular reduction of large numbers is the essential operation in public-key schemes, efficient execution of this operation will allow software executions to run quicker than formerly achievable. The Classical, Barrett, and Montgomery methods are recognized modular reduction methods for large integers employed in public-key schemes. Every method has its own unique properties resulting in a certain area of application. Below are the illustrations of the methods with pseudo code.

2 THE EXISTING MODULA REDUCTION METHODS

There are three well-known algorithms for modular reduction of large integers numbers used in public-key schemes. The description of these methods is as follows:

2.1 Classical Modular Reduction

Suppose z is any integer, so $z \bmod p$ is a remainder in rang $[0, p-1]$, z divided by p is called a modular reduction of z with respect to modulus m. Therefore, both modular multiplication and multiple-precision are needed for carrying out modular reduction. The most direct algorithm for carrying out modular reduction is to calculate a reminder on division by p, using the multiple-precision division algorithm for example algorithm 3. This is denoted as a classical algorithm for carrying out modular multiplication. The following algorithms needed to perform a classical modular reduction which is as follows:

Algorithm 1: Classical modular multiplication

INPUT: two integers a, g and the modulus p, all in a radix b representation

OUTPUT: $ag \bmod p$

Find ag using algorithm 2

Find a remainder r if ag is divides by p using algorithm 3

return (r)

Algorithm 2: Multiple-precision multiplication

INPUT: two integers a, g having $n+1$ and $t+1$ base b digits respectively

OUTPUT: the product $ag = (w_{n+t+1}...w_1 w_0)_b$ in the radix b representation

for$(i = 0$ *to* $(n+t+1)$*do*

$w_i := 0$;

for$(i = 0$ *to* $t)$*do*

begin

$c := 0$;

for$(j = 0$ *to* $n)$*do*

begin

$(uv)_b = w_{i+j} + a_j g_i + c$;

$w_{i+j} := v$;

$c := u$;

end

$w_{i+n+1} := u$;

return$(w_{n+t+1}...w_1 w_0)$

Algorithm 3: Multiple-precision division

INPUT: two integers $a = (a_n...a_1 a_0)_b$; $g = (g_t...g_1 g_0)_b$ where $n \geq t \geq 1$; $g_t \neq 0$

OUTPUT: a quotient $q = (q_{n-t}...q_1 q_0)_b$ and remainder $r = (r_t...r_1 r_0)_b$ where $x = qg + r, 0 \leq r < g$

$a := qg + r$; $0 \leq r < g$

for$(j = 0$ *to* $n-t)$*do*

$q_j := 0$;

while$(a \geq gb^{n-t})$*do*

begin

$q_{n-t} := q_{n-t} + 1;$
$a := a - gb^{n-t};$
end ;
for$(i = n \ \ down \ \ to(t+1)do)$
begin
if$(a_i = g_t)then$
$q_{i-t-1} := b-1$
else
$q_{i-t-1} := \lfloor (a_i b + a_{i-1}) / g_t \rfloor;$
while$(q_{i-t-1}(g_t b + g_{t-1}) > a_i b^2 + a_{i-1} b + a_{i-2})do$
$q_{i-t-1} := q_{i-t-1} - 1;$
$a := a - q_{i-t-1} g b^{i-t-1};$
if$(a < 0)then$
begin
$a := a + gb^{i-t-1};$
$q_{i-t-1} := q_{i-t-1} - 1;$
end ;
end ;
$r := a;$
end ;
return$(q, r);$
end .

2.2 Barrett Algorithm

The Barrett method for modular reduction of large integers is relied on a simple thought, like the way you make computation using the calculator. Barrett reduction (algorithm 4) finds $r = a \bmod p$ given a, and p. The algorithm needs the pre-computation of the amount $\mu = \lfloor b^{2k} / p \rfloor$. It is beneficial when many reductions are executed with a single modulus [1]. For instance, every *RSA* encryption for one individual needs reduction modulo that individual's public key modulus. The pre-computation takes a determined amount of work, which is small in comparison to modular exponentiation cost. Normally, a radix b is selected to be close to the word-length of a processor. Suppose $b > 3$ in algorithm 4.

Algorithm4: Barrett modular reduction

INPUT: two integers $a = (a_{2k-1} ... a_1 x_0)_b$, $p = (p_{k-1} ... p_1 p_0)_b$ (with $p_{k-1} \neq 0$), and $\mu = \lfloor b^{2k} / p \rfloor$

OUTPUT: $r = a \bmod p$

$q_1 := \lfloor a / b^{k-1} \rfloor;$
$q_2 := q_1 \mu;$
$q_3 := \lfloor q_2 / b^{k+1} \rfloor;$
$r_1 := a \bmod b^{k+1};$
$r_2 := q_3 p \bmod b^{k+1};$
$r := r_1 - r_2;$
if $r < 0$ *then*
$r := r + b^{k+1};$
while $r \geq p$ *do*
$r := r - p;$
return$(r);$

2.3 Montgomery Algorithm

Montgomery reduction is the method which allows efficient execution of modular multiplication without performing a classical modular reduction step. Suppose p, d and f are integers with $d > p$, $\gcd(p, d) = 1$ and $0 \leq f < pd$. The method for calculating $fd^{-1} \bmod p$ without using a classical method of Algorithm 1 is called the Montgomery reduction of f modulo p with respect to d. With an appropriate selection of d, the Montgomery reduction can be efficiently calculated. Let a, g are integers with $0 \leq a, g < p$. Let $a' = ad \bmod p$ where $g' = gd \bmod p$. A Montgomery reduction of $a'g'$ is $a'g'd^{-1} \bmod p = agd \bmod p$. This remark is used in Algorithm 5 to give an efficient algorithm for a Montgomery reduction.

Algorithm 5: Montgomery reduction

INPUT: integers $p = (p_{n-1} ... p_1 p_0)_b$ with $\gcd(m, b) = 1$, , $d = b^n$, $p' = -p^{-1} \bmod b$ and $f = (t_{2n-1} ... t_1 t_0)_b < pd$

OUTPUT: $fd^{-1} \bmod p$

$h := f;$
for$(i = 0 \ \ to(n-1))do$
begin
$u_i := a_i p' \bmod b;$
$h := h + u_i p b^i;$
end;
$h := h / b^n;$
if$(H \geq p)then$
$h := h - p;$
return$(h);$
end .

- *Sattar J. Aboud holds a PhD in Computing Systems and he is currently a visiting Professor at Bedfordshire University, UK. E-mail: sattar_aboud@yahoo.com*
- *Edmond Prakash holds a PhD in Computer engineering and he is currently a director, Institute for research in applicable Computing in University of Bedfordshire, UK, E-mail: Edmond.Prakash@beds.ac.uk*

3 APPROACH

The three algorithms are used to calculate $a \bmod p$ in terms of addition, subtraction, multiplication, and single precision division of both single and multiple precision integers.

3.1 The Program Language

Microsoft C++ was selected as a primary programming language. C++ was chosen because of its portability, high execution speed, and appropriateness for carrying out large amount of computational work. Development and time trials were done on the Pentium Intel(R) core(TM) processor, i5 CPU, with 500 GB hard drive, and Windows 7 operating systems.

3.2 The Input/output Specifications

INPUT: large integers read by variables from the batch file, there were 20 numbers in every of the following ranges (128, 256, 512, 1024, and 2048 bits). The input numbers are generated pseudo-random in the decimal format and changed to hexadecimal to ensure that the right number of bits was employed in the timing test. The input files hold an integer set that encounters the following input parameters for every of the methods. The Barrett algorithm needed a pre-calculation of $\mu = \lfloor b^{2k} / p \rfloor$ before a modular reduction. The time needed to achieve a calculation of μ was not included in a total time to carry out a modular reduction. The classical, Barrett, and Montgomery algorithms used the same input format $a = (a_{2k-1} \ldots a_1 a_0)_b$, $p = (p_{k-1} \ldots p_1 p_0)_b$, with $p_{k-1} \neq 0$.

OUTPUT: findings are written to a text file and included for every input integer, the first integer in hexadecimal notation, its length, the remainder $x \bmod p$, the calculation time delta, and the number of steps accepted during the process.

3.3 The Test Program

We use the same input for every of the three algorithms, while the input was kept in different ways. We checked the outcome for three algorithms versus the results of *CBigInt* and NTL C++ libraries. Both libraries are able of working with random size integer mathematics operations on large numbers. We verify the performance of every method by measuring a time needed for each run. The computational time is recorded as an elapsed time between a start time and end time, using Win32API function *GetTickCount*(). The Win32 API *GetTickCount*() was chosen due to ease of execution and overall accuracy to $10ms$.

4 RESULTS

The following three tables show the results of the Classical algorithm, Montgomery algorithm, and Barrett algorithm, regarding the time needed for every run in $m$sec

Table 1: Time needed for every run ($m$sec) using Classical algorithm.

	128	256	512	1024	2048
1	5.00	49.00	59.00	96.00	99.00
2	4.00	46.00	63.00	92.00	159.00
3	6.00	41.00	51.00	79.00	163.00
4	5.00	37.00	61.00	104.00	170.00
5	7.00	32.00	61.00	97.00	161.00
6	9.00	27.00	59.00	97.00	170.00
7	4.00	50.00	63.00	93.00	165.00
8	5.00	55.00	60.00	91.00	180.00
9	9.00	41.00	47.00	93.00	143.00
10	6.00	40.00	52.00	102.00	162.00
11	7.00	48.00	59.00	96.00	163.00
12	9.00	39.00	72.00	94.00	169.00
13	7.00	54.00	50.00	99.00	163.00
14	8.00	53.00	67.00	97.00	170.00
15	5.00	38.00	64.00	91.00	161.00
16	4.00	22.00	58.00	84.00	162.00
17	8.00	26.00	50.00	81.00	166.00
18	7.00	41.00	64.00	86.00	178.00
19	7.00	52.00	55.00	97.00	93.00
20	5.00	33.00	58.00	101.00	141.00
Average	**6.20**	**41.00**	**58.65**	**93.50**	**156.90**

Table 2: Time needed for every run ($m$sec) using Barrett algorithm

	128	256	512	1024	2048
1	0.00	10.00	10.00	50.00	180.0
2	0.00	10.00	20.00	50.00	160.0
3	0.00	0.00	10.00	50.00	170.0
4	0.00	0.00	20.00	50.00	160.0
5	0.00	0.00	10.00	40.00	180.0
6	10.00	0.00	20.00	50.00	180.0
7	0.00	10.00	10.00	60.00	170.0
8	0.00	10.00	20.00	40.00	170.0
9	10.00	0.00	20.00	50.00	170.0
10	0.00	0.00	10.00	60.00	160.0
11	0.00	10.00	20.00	50.00	170.0
12	10.00	0.00	10.00	50.00	170.0
13	0.00	10.00	20.00	50.00	180.0
14	0.00	10.00	10.00	40.00	160.0
15	0.00	0.00	20.00	50.00	170.0
16	0.00	0.00	10.00	50.00	180.0
17	10.00	0.00	10.00	40.00	170.0
18	10.00	0.00	20.00	40.00	170.0
19	0.00	10.00	20.00	50.00	100.0
20	0.00	0.00	20.00	50.00	160.0
Average	**2.50**	**4.00**	**15.50**	**48.50**	**168.5**

Table 3: Time needed for every run ($m$sec) using Montgomery algorithm

	128	256	512	1024	2048
1	67.00	73.00	62.00	62.00	86.00
2	79.00	74.00	75.00	73.00	62.00
3	81.00	76.00	73.00	80.00	64.00
4	67.00	61.00	61.00	81.00	73.00
5	65.00	83.00	79.00	61.00	82.00
6	64.00	80.00	60.00	80.00	81.00
7	61.00	63.00	78.00	60.00	80.00

8	67.00	81.00	80.00	72.00	71.00
9	79.00	81.00	74.00	81.00	73.00
10	76.00	60.00	81.00	80.00	74.00
11	81.00	82.00	63.00	61.00	84.00
12	78.00	80.00	74.00	73.00	85.00
13	79.00	80.00	62.00	81.00	84.00
14	77.00	80.00	72.00	74.00	82.00
15	82.00	83.00	80.00	63.00	81.00
16	63.00	74.00	81.00	62.00	70.00
17	72.00	61.00	62.00	73.00	72.00
18	75.00	72.00	73.00	80.00	84.00
19	78.00	73.00	74.00	80.00	81.00
20	71.00	62.00	75.00	73.00	82.00
Average	**73.10**	**74.00**	**71.55**	**72.50**	**76.00**

5 Analysis

The comparison of Classical, Barrett, and Montgomery methods is given below. Table 4 show the execution time needed for all algorithms (m sec)

	128	**256**	**512**	**1024**	**2048**
Classical	1.5	5.3	14.4	35.6	159.1
Barrett	4.7	33	45.2	82.1	145.5
Montgomery	74.8	73.7	70.6	71.6	76.2

- When a pre-computation, argument transformation, and post-computation time is neglected, Barrett reduction algorithm is a fastest, followed by classical and Montgomery, for size lesser than 1024 bits. For size longer than 1024 bits, Montgomery is about twice as quick as the Barrett method and to some extent quicker than the classical.
- The Montgomery reduction method is almost the constant line, demonstrating no dependency between an execution time and a size. This is because of the truth that the Montgomery reduction requires the modular multiplication of d^{-1} despite the value of an argument [2].
- As showed in the outcome above, the three algorithms have input by which they execute faster than an average. As these inputs are dissimilar for every method, no algorithm has the best carrying out for all inputs of the given size.

6 Applications

The standard algorithm for performing the z^e mod p modular exponentiation is by using a recognized repeated square and multiply algorithm [3]. The Left-to-Right type of this algorithm includes repeated squaring and multiplying the result by the determined value of z . If z has a particular formation, the multiplication by the determined value is surely easier than a multiplication of two random numbers. The computation of z^e mod p can use $m-ary$ the Binary Square and multiply algorithm [4]. It is reflected that for $m=16$ this decreases an average number of modular multiplications to around $1/5$ the bit numbers of e compared to Binary Square and multiply method is ½ [5]. Every of three reduction methods are used in this execution, resulting in three modular exponentiation methods. The pace difference between the reductions methods are shown in pace differences between an exponentiation functions. The performance of the reduction functions with respect to the length of an argument will be shown in the performance of an exponentiation functions. For the complete size exponentiation the Montgomery typed exponentiation is somewhat quicker than the Barrett typed exponentiation, in turn being somewhat quicker than a classical one [6]. But, for small sufficient arguments, a Barrett-typed exponentiation is fastest of the three algorithms [7].

7 Conclusion

Three methods for modular reduction of large numbers were illustrated and evaluated according to their accuracy, calculation operation and efficiency. When the time for pre-and post-calculation and for m-residue alteration for Montgomery is ignore, the Barrett algorithm is the best for case smaller than 1024 bits, where Montgomery is the best for case larger than 1024 bits. Every algorithm has its own characteristics proper for the specific area of application. No single method gives the great solution to encounter all demands; based on the milieu by which calculation are to be executed, one algorithm can be preferable over another. For single modular reductions, the classical algorithm appears to be the best selection, as a pre- and post-computations only contain very quick and direct calculation. For small cases, classical and Barrett algorithms are about equally quick, with minor better execution for Barrett. For modular exponentiation, the exponentiation relies on Montgomery's method has the best execution.

Acknowledgment

The authors wish to thank the University of Bedfordshire, department of Computer science and Technolgy for its support us financially.

References

[1] Amiel F., Feix B., Tunstall M., Whelan C., and Marnane W., "Distinguishing Multiplications from Squaring Operations", Selected Areas in Cryptography, LNCS 5381, pp. 346-360, 2008.

[2] Amiel F., Feix B., and Villegas K., "Power Analysis for Secret Recovering and Reverse Engineering of Public Key Algorithms", Selected Areas in Cryptography, LNCS 4876, pp. 110-125, 2007.

[3] Schmidt J., Tunstall M., Avanzi R., Kizhvatov I., and Oswald D., "Combined Implementation Attack Resistant Exponentiation", LATINCRYPT 2010, LNCS 6212, pp. 305-322, 2010.

[4] Miroslav Kne!zevi c, Frederik Vercauteren, and Ingrid Verbauwhede, "Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods", IEEE Transaction on Computers, Volume 59, No. 12, December, pp. 1715-1721, 2010.

[5] Cao ZJ and Wu XJ, "An improvement of the Barrett modular reduction algorithm", International Journal of Computer Mathematics, Taylor & Francis, 2013

[6] Dupaquis V., and Venelli A., "Redundant Modular Reduction Algorithms", Proceeding of CARDIS 2011, LNCS, 7079, Springer, pp. 102-114, 2011.
[7] Zhengjun Cao1, Ruizhong Wei, and Xiaodong Lin, "A Fast Modular Reduction Method", eprint.iacr.org/2014/040.pdf