A fast modular multiplication algorithm for calculating the product $AB$ modulo $N$

Chien-Yuan Chen $^{a,*}$, Chin-Chen Chang $^{b,1}$

$^a$ Department of Information Engineering, I-Shou University, Kaohsiung County, 84008 Taiwan
$^b$ Institute of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, 621 Taiwan

Received 1 June 1999; received in revised form 19 October 1999

Communicated by F.Y.L. Chin

Abstract

In this paper, we propose a fast iterative modular multiplication algorithm for calculating the product $AB$ modulo $N$, where $N$ is a large modulus in number-theoretic cryptosystems, such as RSA cryptosystems. Our algorithm requires $\frac{5k}{3} + \frac{1}{2}k - \frac{17}{16}k^2$ additions on average for an $n$-bit modulus if $k$ carry bits are dealt with in each loop. For a 512-bit modulus, the known fastest modular multiplication algorithm, Chen and Liu’s algorithm, requires 517 additions on average. However, compared to Chen and Liu’s algorithm, our algorithm reduces the number of additions by 26% for a 512-bit modulus. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Modular multiplication; RSA cryptosystem; Cryptography

1. Introduction

Modular multiplication plays an important role in a number of number-theoretic cryptosystems, such as RSA cryptosystems [4]. If modular multiplication with a large modulus is speeded up, then the speed of encryption/decryption will dramatically increase. So, a lot of research on this issue has been done. In 1983, Blakley [1] presented a traditional algorithm for calculating the product $AB$ modulo $N$, where $N$ is an $n$-bit integer. Assume that both addition and subtraction have the same complexity and the shift operation is negligible as compared with an addition operation. Blakley’s algorithm thus requires on average $2n$ additions. Improving Blakley’s algorithm, Chio and Yang [3] presented an iterative modular multiplication algorithm without magnitude comparison. Chiou and Yang’s algorithm only checks the carry bit of the partial product $P$ instead of comparing the partial product $P$ with the modulus $N$. A carry generated in computing the partial product $P$ means that the partial product $P$ with a carry is greater than or equal to $2^n$. If this is the case, the generated carry is discarded and the partial product $P$ is set to be $(P - 2^n + S)$, where $S = 2^n \mod N$ is precomputed. Thus, the value of the partial product $P$ is limited to be less than $2^n$ instead of $N$. Unfortunately, Su and Hwang [5] pointed out that there were errors in Chiou and Yang’s algorithm. When the partial product $P$ with a carry is greater than $2^n$, the fact that the new value $(P - 2^n + S)$ is less than $2^n$ cannot be guaranteed. Therefore, they computed the partial product $P$ as $(P - 2^n + S)$ until it had
no carry. According to the analysis of Su and Hwang, it required \( n + 11 \) additions on average to compute an \( n \)-bit modular multiplication. Thus, for a 512-bit modulus, it required 523 additions on average. In these algorithms, the computed values were not stored for use again. So, Su and Hwang spent much time to recompute these values. Chen and Liu used the Lempel–Ziv binary tree to store the computed values. According to [2], Chen and Liu’s algorithm only requires 517 additions on average for a 512-bit modulus. Furthermore, it only requires 992 additions on average for a 1024-bit modulus.

In this paper, improving the modified Chiou and Yang algorithm [5], we propose a new iterative algorithm dealing with two bits in each loop, instead of one bit. Then, the average number of additions is reduced to \( \left( \frac{77}{6}n + 22 \right) \frac{1}{4} \) with additional 21 storage values. For a 512-bit modulus, it only requires 433 additions on average. Obviously, our algorithm is more efficient than the presented algorithms. Furthermore, we propose a general iterative algorithm to deal with \( k \) carry bits in each loop. It only requires

\[
\left( \frac{5}{3} - \frac{1}{4^k} \right) \frac{n}{k} + \left( \frac{5}{3} - \frac{1}{3^k} - \frac{17}{6} \right)
\]

additions on average with additional \( 4^k + 2^{k+1} - 3 \) storage values. If we adopt a 512-bit sized number as the modulus in RSA cryptosystems, the average number of additions is 383 when \( k = 3 \). Compared to Chen and Liu’s algorithm, our algorithm reduces the number of additions by 26\%. Furthermore, for a 1024-bit RSA modulus, our algorithm requires 665 additions on average. Compared to Chen and Liu’s algorithm, our algorithm reduces the number of additions by 33\%.

### 2. New algorithms

Let \( A, B, \) and \( N \) be three \( n \)-bit positive binary integers, where \( A, B < N \). Assume the binary representation of \( B \) is \( b_{n-1}b_{n-2} \ldots b_0 \). Let \( \text{carry}_j(P) \) denote the \( j \) most significant bits of \( P \). Based on the modified Chiou and Yang algorithm, a new iterative algorithm with two bits in each loop will be presented as Algorithm 1.

Firstly, consider the cost of precomputation. Computing \( S_j \), for \( j = 1, 2, \) and 3, needs one shift operations and four additions. Similarly, computing \( T_j \), for

---

**Algorithm 1**: (*Compute \( R = A \times B \mod N \).*)

**Input**: \( A, B, \) and \( N \).

**Output**: \( R \).

**Begin**

(* Let \( P \) be the result of the partial product, where the size of \( P \) is \( n + 2 \) bits.*)

\[
P = 0; S_1 = 1 \times 2^n \mod N; S_2 = 2 \times 2^n \mod N; S_3 = 3 \times 2^n \mod N;
T_1 = A; T_2 = 2 \times A \mod N; T_3 = 3 \times A \mod N;
U_1 = T_1; U_2 = T_2; U_3 = T_3;
U_4 = S_1; U_5 = S_1 + T_1 \mod N;
U_6 = S_1 + T_2 \mod N; U_7 = S_1 + T_3 \mod N;
U_8 = S_1; U_9 = S_1 + T_1 \mod N;
U_{10} = S_1 + T_2 \mod N; U_{11} = S_1 + T_3 \mod N;
U_{12} = S_1; U_{13} = S_1 + T_1 \mod N;
U_{14} = S_1 + T_2 \mod N; U_{15} = S_1 + T_3 \mod N;
\]

if \( n \) is odd then \( b_n = 0 \);

for \( i = \lceil \frac{n}{2} \rceil - 1 \) down to 0 do

begin

\[
P = 4 \times P;
\]

(* Shift-left the result of partial product twice.*)

\[
\text{case carry}_2(P) \text{ and } (b_{2i+1}, b_{2i}) \text{ of } (2.1)
\]

| 00 and 00: do nothing; (* No carry-out *) |
| 00 and 01: \( P = P + U_1; \) |
| 00 and 10: \( P = P + U_2; \) |
| 00 and 11: \( P = P + U_3; \) |
| 01 and 00: \( P = (P - 1 \times 2^n) + U_4; \) |
| (* Indicate the case \( P \geq 1 \times 2^n \). *) |
| 01 and 01: \( P = (P - 1 \times 2^n) + U_5; \) |
| 01 and 10: \( P = (P - 1 \times 2^n) + U_6; \) |
| 01 and 11: \( P = (P - 1 \times 2^n) + U_7; \) |
| 10 and 00: \( P = (P - 2 \times 2^n) + U_8; \) |
| (* Indicate the case \( P \geq 2 \times 2^n \). *) |
| 10 and 01: \( P = (P - 2 \times 2^n) + U_9; \) |
| 10 and 10: \( P = (P - 2 \times 2^n) + U_{10}; \) |
| 10 and 11: \( P = (P - 2 \times 2^n) + U_{11}; \) |
| 11 and 00: \( P = (P - 3 \times 2^n) + U_{12}; \) |
| (* Indicate the case \( P \geq 3 \times 2^n \). *) |
| 11 and 01: \( P = (P - 3 \times 2^n) + U_{13}; \) |
| 11 and 10: \( P = (P - 3 \times 2^n) + U_{14}; \) |
| 11 and 11: \( P = (P - 3 \times 2^n) + U_{15}; \) |

end case;

while \( \text{carry}_1(P) \) do

\[
P = P - 2^n + S_1; \quad (2.2)
\]

end;

if \( P \geq N \) then \( R = P - N \) else \( R = P; \) \quad (2.3)

end.
Algorithm 2: (* Compute $R = A \times B \mod N$. *)

Input: $A$, $B$, and $N$.
Output: $R$.

Begin

(* Let $P$ be the result of the partial product, where the size of $P$ is $n + k$ bits. *)

$P = 0$; $S_0 = 0$; $S_i = i \times 2^n \mod N$, $i = 1, 2, \ldots, 2^k - 1$;
$T_0 = 0$; $T_i = (i \times A) \mod N$, $i = 1, 2, \ldots, 2^k - 1$;
$U_j = T_j \mod N$, $j = 1, 2, \ldots, 2^k - 1$;
$U_j = S_i + T_g \mod N$, $i = 1, 2, \ldots, 2^k - 1$, $g = 1, 2, \ldots, 2^k - 1$ and $j = i \times 2^k + g$;

(* $j = 2^k, 2^k + 1, \ldots, 4^k - 1$; *)

for $i = k \times \lfloor \frac{n}{k} \rfloor - 1$ down to $0$ do

begin

$P = 2^k \times P$; (* Shift-left the result of partial product $k$ times. *)

case carry$\gamma_i(P)$ and $(b_{k+i+k-1} \ldots b_{k+i+k-1})$ of

0$k-1$0$k-2$0100 and 0$k-1$0$k-2$0100: do nothing;
0$k-1$0$k-2$0100 and 0$k-1$0$k-2$0101: $P = P + U_1$;
0$k-1$0$k-2$0100 and 0$k-1$0$k-2$1100: $P = P + U_2$;

$\vdots$
0$k-1$0$k-2$0100 and 1$k-1$1$k-2$1110: $P = P + U_{2^{k-1}}$;
0$k-1$0$k-2$0110 and 0$k-1$0$k-2$0100: $P = (P - 1 \times 2^n) + U_{2^{k-1}}$;

$\vdots$
1$k-1$1$k-2$1110 and 1$k-1$1$k-2$1110: $P = (P - (2^k - 1) \times 2^n) + U_{2^{k-1}}$;

end case;

case carry$\gamma_i(P)$ do $P = P - 2^n + S_1$;

end;

if $P \geq N$ then $R = P - N$ else $R = P$;

end.

\( j = 2 \text{ and } 3 \), requires one shift operations and three additions. Besides, we must consider the cost of computing $U_j$. Because $U_j = S_i + T_g \mod N$, for some integers $i$ and $g$, includes two steps: $U_j = S_i + T_g$ and while carry$\gamma_i(U_j)$ do $U_j = U_j - 2^n + S_i$. From [2], the second step requires $\frac{j}{4}$ additions on average. Thus, computing $U_j$, for $j = 1, 2, \ldots, 15$, needs $3 \times 3 \times (1 + \frac{j}{4})$ additions on average. Therefore, the required total number of precomputed additions is 22. Next, in each loop, the probability of performing one addition in cases (2.1) is $\frac{15}{16}$. The probability of performing one addition in expression (2.2) is $\frac{7}{8}$. So, on average, this algorithm requires

\[
\left( \frac{15}{16} + \frac{2}{3} \right) \frac{n}{2} + 22 + \frac{1}{2} = \left( \frac{77}{96}n + 22 \frac{1}{2} \right)
\]

differences, where $\frac{1}{2}$ means the cost of performing expression (2.3). For a 512-bit modulus, it only requires 433 additions on average. Obviously, our algorithm is more efficient than the presented algorithms. In addition to the time of the algorithm, we also consider additional space of the algorithm. It needs to store $S_j$, $T_j$ and $U_j$. Thus, the number of storage values is $3 + 3 + 15 = 21$.

Next, we generalize Algorithm 1 to give the iterative Algorithm 2 with $k$ carry bits in each loop.

Firstly, consider the cost of precomputation. Computing $S_j$, for $j$ is even, needs one shift operation and one addition operation. On the other hand, computing $S_j$, for $j$ is odd excluding 1, requires two additions. So, computing $S_j$ requires $(2^k - 1) +
and \(g\).

larly, computing \(T_j\) requires \(2^k + 2^{k-1} - 3\) additions. Besides, computing \(U_j\), for \(j = 1, 2, \ldots, 4^k - 1\), needs \((2^k - 1)(2^k - 1)(1 + \frac{1}{2})\) additions on average. The required total number of precomputed additions is

\[
\frac{5}{3} \frac{4^k}{3} - \frac{10}{3}. \tag{2.7}
\]

In each loop, the probability of performing one addition in case (2.4) is

\[
\left(\frac{4^k - 1}{4^k}\right). \tag{2.8}
\]

additions, where \(\frac{1}{2}\) means the cost of performing expression (2.6). Therefore, Algorithm 2 including the cost of precomputation requires on average

\[
\left(\frac{5}{3} \frac{1}{4^k}\right) n \frac{5}{k} + \frac{5}{4^k} - \frac{1}{3} 3^k - \frac{17}{6}. \tag{2.9}
\]

Next, we consider the additional space of Algorithm 2. It needs to store \(S_j, T_j\) and \(U_j\). Therefore, the number of storage values is \((2^k - 1) + (2^k - 1) + (4^k - 1) = 4^k + 2^k - 3\).

Given \(n\), let \(g_1(k)\) and \(g_2(k)\) be expressions (2.7) and (2.8), respectively. Then, the derivatives of \(g_1(k)\) and \(g_2(k)\) are

\[
g'_1(k) = \frac{5}{3} 4^k \ln 4 - \frac{1}{3} 3^k \ln 2 \quad \text{and} \quad g'_2(k) = \frac{n(3 + 3k \ln 4 - 5 \times 4^k)}{3k^2 4^k}.
\]

Because \(g'_1(k) > 0\) as \(k > 0\), \(g_1(k)\) is an increasing function over positive integers. However, \(g_2(k)\) is a decreasing function over positive integers because \(g'_2(k) < 0\) as \(k > 0\). What is the optimal value \(k\) to minimize the expression (2.9) given an \(n\)-bit modulus? To answer this problem, we find that if \(k\) is an optimal value, the expression (2.9) in terms of \(k\) is smaller than that in terms of \((k - 1)\) and \((k + 1)\). When the expression (2.9) in terms of \(k\) is equal to that in terms of \((k + 1)\), we can find the maximum of \(n\) whose optimal value is \(k\). Therefore, if \(k\) is an optimal value, the maximum of \(n\) is

\[
k(k + 1) 4^{k+1}(15 \times 4^k - 2^k) \quad \frac{20 \times 4^k - 9k - 12}{n}. \tag{2.10}
\]

The probability of performing one addition in expression (2.5) is \(\frac{2}{3}\). So, on average, it requires

\[
\left(\frac{5}{3} \frac{1}{4^k}\right) n \frac{5}{k} + \frac{5}{4^k} - \frac{1}{3} 3^k - \frac{17}{6}. \tag{2.9}
\]

Table 1 shows the optimal value \(k\) given \(n \leq 90000\). For a 512-bit modulus, the average number of additions is 383 as \(k = 3\). Furthermore, for a 1024-bit modulus, the average number of additions is 665 as \(k = 3\).

3. Conclusions

A new iterative modular multiplication algorithm based on the modified Chiou and Yang algorithm has been presented. Compared to the modified Chiou and Yang algorithm, our algorithm deals with \(k\) carry bits, instead of one bit, in each loop. For a 512-bit RSA modulus, our algorithm only requires 383 additions on average at the cost of 77 stored values, while the known fastest algorithm requires 517 additions. Obviously, our algorithm reduces the number of additions by 26%. Furthermore, for a 1024-bit RSA modulus, our algorithm requires 665 additions on average, while the known fastest algorithm requires 992 additions. Obviously, our algorithm reduces the number of additions by 33%.

<table>
<thead>
<tr>
<th>The range of given (n)</th>
<th>[32, 312]</th>
<th>[313, 2356]</th>
<th>[2357, 15440]</th>
<th>[15441, 90000]</th>
</tr>
</thead>
<tbody>
<tr>
<td>The optimal value (k)</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>The average number of required additions</td>
<td>1.50(n)~0.87(n)</td>
<td>0.87(n)~0.59(n)</td>
<td>0.59(n)~0.44(n)</td>
<td>0.44(n)~0.35(n)</td>
</tr>
<tr>
<td>The number of stored values</td>
<td>21</td>
<td>77</td>
<td>285</td>
<td>1085</td>
</tr>
</tbody>
</table>
References