Postprint of article in *Proceedings of the 1st Asia-Pacific Conference on Quality Software* (*APAQS '00*),
IEEE Computer Society, Los Alamitos, CA, 89–98 (2000)

Testing of Large Number Multiplication Functions in Cryptographic Systems [*†]

T.H. Tse [‡]
The University of Hong Kong
Pokfulam, Hong Kong

T.Y. Chen
Swinburne University of Technology
Melbourne, Australia

Zhi Quan Zhou
The University of Hong Kong
Pokfulam, Hong Kong

Abstract

Integer multiplication is one of the fundamental functions in cryptographic systems. Although much research has already been done on the testing of multiplication functions, most does not meet the need of cryptographic systems, where very large numbers are involved. Others provide only probabilistic algorithms. In this paper, we propose an efficient deterministic algorithm for verifying large number multiplications in cryptographic systems. A deterministic oracle for large integer multiplication functions will result. In addition, our method can also be used to verify selected segments of digits in the product of two numbers.

Keywords: Cryptographic systems, large number multiplication, software testing

1 Introduction

A testing oracle is some mechanism, either automatic or via a human tester, that specifies the expected outcome of a program on the testing data [2]. In software testing, it is generally believed that the correctness of a program can be verified by matching the program output with the expected outcome. This is known as the oracle assumption [11].

The oracle assumption, however, may not necessarily hold in every situation. This is known as the oracle problem. In numerical analysis, for example, it is not easy to predict the expected results [6] except for trivial cases. Weyuker [11] defined a program to be non-testable if the oracle does not exist or it is practically too difficult to be obtained.

The oracle problem is a major concern in cryptographic systems, because very large numbers are involved in the computations. Although it is theoretically possible to define an oracle for the computation, it is practically too difficult since the operands involved are too large. Among these computations, large number multiplication is the fundamental function for almost all kinds of public-key encryption/decryption systems such as the RSA algorithm, Diffie-Hellman key exchange, elliptic curve cryptography and primality testing [5, 12]. In addition, integer multiplication is also the basic function called by other number-theoretic functions such as division, modulo arithmetic, exponentiation and modular exponentiation [8]. Hence, the testing of large number multiplication functions is of particular importance in cryptographic software testing. In this paper, we shall present a method of testing large number multiplication functions in cryptographic systems.

Let us take a brief literature review first. One method for testing numerical functions when there is an oracle problem is to test the program on some simplified or specific data, for which the correctness of the results can be easily determined [11]. However, this approach cannot give us sufficient confidence on the correctness of the program for complex data, which are usually more error-prone. Another method is to use the inverse to verify the correctness of a function. The consecutive execution of a function followed by its inverse should result in the original input value. However, an inverse may not be found for every function.

[†] This research is supported in part by the Hong Kong Research Grants Council and the University of Hong Kong Committee on Research and Conference Grants.

[‡] All correspondence should be addressed to Prof. T.H. Tse, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: thtse@cs.hku.hk.

Despite the above problems, testing can still be carried out in most cases because the functions to be tested often have their own theoretical properties. Hence, if the output of the program does not match such properties, testers can immediately identify an error without actually knowing the answer. The concept of a program checker was presented by Blum *et al.* [1, 3, 4], to make use of properties of functions in software testing. The checker is a probabilistic program that checks the output of a computation. Given a program and an instance of its input data, the checker certifies whether the output of the program on that input data is correct with a high probability. In order to obtain the probabilistic assurance, however, the checker must repeatedly run the program being tested.

Although the program checker provides a probabilistic oracle for some functions, people are still facing the fundamental limitation of the inability to extrapolate from the correctness of the program on testing data to the correctness of the program on all elements of the input domain [11]. To cope with this challenge, the theory of program checker was extended to the theory of self-testing/correcting [3]. A self-tester T for a function f is a probabilistic program. For any program P supposedly computing f, T estimates the probability of error such that $P(x) \neq f(x)$ for a random input x. A self-corrector C for f is also a probabilistic program. If it is known that a program P has a sufficiently low probability of error, then for any input x, C can call P to compute $f(x)$ with a high probability of correctness. Blum *et al.* [3] presented general techniques for constructing self-testing/correcting pairs for a variety of numerical functions, including integer multiplication, modular multiplication, integer division, modular exponentiation, polynomial multiplication and matrix operations. Kaminski [7] also introduced methods to probabilistically verify the product of integers and polynomials. We found, however, that these methods have their limitations when implemented to cryptographic systems where very large numbers are involved.

The self-tester/corrector employs a black box strategy. It selects testing data on a "random" basis in order to achieve the probabilistic results. Hence, this technique is not suitable for white box testing, which requires the test cases to be selected according to the program structure. As a probabilistic program, the self-tester/corrector needs many repeated calls to the functions being tested to achieve the probabilistic assurance, bringing a relatively high time cost. Furthermore, specific techniques used in the algorithms are not suitable for cryptographic systems. For example, the self-tester/corrector for multiplication functions introduced in [3] assumes that, for a given integer y, $y \times 2^n$ can be computed by left-shifting y for n bits without calling the multiplication function being tested. This assumption is impractical in cryptographic systems, where the operand y is usually very large. In such systems, a large integer cannot be stored within one computer word, and hence it is usually stored digit by digit across several words. Thus, $y \times 2^n$ cannot be computed by a simple left-shifting of bits.

Kaminski [7] presented other probabilistic algorithms to check the product of integers, but they also have similar problems. For example, the relation $res(a, p) \times res(b, p) = res(c, p)$ in [7] is checked to verify the product $a \times b = c$, where a and b are n-bit integers, p is a relatively small prime number: $|p| = 2 \times \log_2 n$, where $|p|$ denotes the bit-length of p, and res is some residue function which can, for simplicity, be interpreted as the modulo function. In the testing the calculation $res(a, p) \times res(b, p)$ is assumed always to be correct. But this assumption is also not practical in large number operations.

Consider a typical 32-bit computer system. In order that $|res(a, p) \times res(b, p)| \leq 32$ for all cases, we must have $|(p-1) \times (p-1)| \leq 32$. Since p is a prime number, we must have $|p| \leq 16$. In other words, $2 \times \log_2 n \leq 16$, or $n \leq 2^8 = 256$. In this way, the lengths of operands in multiplications are limited to no more than 256 bits in a typical computer system. This is obviously too short for cryptographic systems.

In this paper, we shall present a method of testing large number multiplication functions in cryptographic systems. Our algorithm is deterministic rather than probabilistic, and does not have any restriction on the lengths of the operands. The result of our algorithm can be used as an oracle to verify the correctness of the product of integers. Note that the word "digit" in this paper means a digit of an integer under any numeration system, usually with a base larger than 2.

2 Exploring the relationships among selected segments of digits in the multiplicand, multiplier and their product

Most testing activities treat an input data as an integrated unit and the output data as another integrated unit when their relationship is analysed for the correctness verification. In our approach, however, we take a different view and explore the relationships among the digit components of the input and output data. The method is based on a method first introduced by Shi [9] for rapid mental calculations. It uses "comparison" to predict the carry in advance, thus enabling people to obtain the final result from the most significant digit to the least significant one. It is efficient for mental calculation because humans can do comparisons much faster than any other calculations.

We discover that Shi's method can be extended to provide an oracle for the testing of large number multiplication functions in cryptographic systems. Our testing method can also be used to verify the correctness of selected digits in the product of two numbers. Although in the following

i/k \ i \ k	1	2	3	4	5	6	7	8
2	0.5							
3	$0.\overline{3}$	$0.\overline{6}$	1					
4	0.25	0.5	0.75	1				
5	0.2	0.4	0.6	0.8	1			
6	$0.1\overline{6}$	$0.\overline{3}$	0.5	$0.\overline{6}$	$0.8\overline{3}$	1		
7	$0.\overline{142857}$	$0.\overline{285714}$	$0.\overline{428571}$	$0.\overline{571428}$	$0.\overline{714285}$	$0.\overline{857142}$	1	
8	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1
9	$0.\overline{1}$	$0.\overline{2}$	$0.\overline{3}$	$0.\overline{4}$	$0.\overline{5}$	$0.\overline{6}$	$0.\overline{7}$	$0.\overline{8}$

Table 1. Rules of carries

$$\begin{array}{rccccccc}
 & 0 & 4 & 2 & 8 & 6 & 5 & 7 \\
\times & & & & & & & 7 \\
\hline
 & & & & & & & \\
\hline \hline
\textbf{Digit No.} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7}
\end{array}$$

Figure 1. 428657 $\times$ 7

our discussion is often in the denary system (with base 10) for easy understanding, it is straightforward to implement our method in other numeration systems. The reference for Sections 2.1 and 2.2 is [10].

2.1 Multi-digit integer multiplied by one-digit integer

In this section, we consider the results of a multi-digit integer multiplied by a one-digit positive integer k, such as $1 \leq k \leq 9$ in the denary system. When the multiplier k is 0 or 1, there is no carry for whatever digit of the multiplicand. Even for the case when $k > 1$, the carries are surprisingly simple.

The fractions $1/k$, $2/k$, …, $(k-1)/k$ will be called the *carry points* for the multiplier k. These carry points divide the region [0, 1) into k sub-regions $[i/k, (i+1)/k)$, where $i = 0$, 1, …, $k-1$. For any pure decimal fraction l in the range $[i/k, (i+1)/k)$, we have $i \leq l \times k < i+1$. Hence, the carry to the integral part of the product $l \times k$ is i, irrespective of the value of l inside $[i/k, (i+1)/k)$. Table 1 shows the rules of carries for $k = 2$, 3, …, 9 in the denary system.

We note the following: The position of the decimal points in the multiplicand and multiplier will not influence the value of carry to the most significant digit of their product. Consider, for example, $0.46 \times 8 = 3.68$ and $46 \times 8 = 368$. The carries to the most significant digit of two products are both 3.

Let us illustrate the new multiplication method. Consider the conventional multiplication process in upright mode as shown in Figure 1. In the remaining parts of this paper, when multiplication is to be performed, an extra 0 will always be placed at the most significant digit of the multiplicand so that the multiplicand and the product can always have the same length. Using the new method, we can directly obtain the value of any digit of the product. Without loss of generality, let us look at digit 3: the corresponding value of the digit of the multiplicand is 2. By calculating (2×7) mod 10, we obtain 4. Now we deduce the carry that digit 3 will receive from its less significant digits 8657×7. Note that the carry of 8657×7 is the same as 0.8657×7. As discussed above, we have the following carry points for the multiplier 7:

$$\begin{aligned}
1/7 &= 0.142857\ 142857 \cdots = 0.\overline{142857} \\
2/7 &= 0.285714\ 285714 \cdots = 0.\overline{285714} \\
3/7 &= 0.428571\ 428571 \cdots = 0.\overline{428571} \\
4/7 &= 0.571428\ 571428 \cdots = 0.\overline{571428} \\
5/7 &= 0.714285\ 714285 \cdots = 0.\overline{714285} \\
6/7 &= 0.857142\ 857142 \cdots = 0.\overline{857142}
\end{aligned}$$

We see that $0.8657 > 0.\overline{857142} = 6/7$. Hence, the carry to the integer part in the product 0.8657×7 must be 6, which is also the carry of 8657×7, that is, digit 3 in Figure 1 receives the carry of 6 from its less significant digits. We already calculated (2×7) mod $10 = 4$ for digit 3. We can now obtain the final value for this digit, namely $(4+6)$ mod 10 $= 0$.

Let us calculate the whole product from the left to the right:

Digit 1: $0.\overline{428571}$ $(= 3/7)$ $\leq$ **0.428657** $<$ $0.\overline{571428}$ $(= 4/7)$. Hence, the carry of 428657×7 is 3. Thus, digit $1 = (0 \times 7 + 3)$ mod $10 = 3$.

	0 $\langle 1\rangle$	**3** $\langle 2\rangle$	**8** $\langle 3\rangle$	**9** $\langle 4\rangle$		
$\times$				**4** (4)	**3** (5)	**6** (6)
	1 $\langle 1\rangle(4)$	**5** $\langle 2\rangle(4)$	**5** $\langle 3\rangle(4)$	**6** $\langle 4\rangle(4)$		
		1 $\langle 1\rangle(5)$	**1** $\langle 2\rangle(5)$	**6** $\langle 3\rangle(5)$	**7** $\langle 4\rangle(5)$	
			2 $\langle 1\rangle(6)$	**3** $\langle 2\rangle(6)$	**3** $\langle 3\rangle(6)$	**4** $\langle 4\rangle(6)$
	1 [1]	**6** [2]	**9** [3]	**6** [4]	**0** [5]	**4** [6]

Figure 2. 389 $\times$ 436

Digit 2: $0.\overline{285714}$ $(= 2/7) \leq$ **0.28657** $<$ $0.\overline{428571}$ $(= 3/7)$. Hence, the carry of 28657×7 is 2. Thus, digit $2 = (4 \times 7 + 2) \bmod 10 = 0$.

$\ldots$

Digit 6: $0.\overline{571428}$ $(= 4/7) \leq$ **0.7** $<$ $0.\overline{714285}$ $(= 5/7)$. Hence, the carry of 7×7 is 4. Thus, digit $6 = (5 \times 7 + 4) \bmod 10 = 9$.

Digit 7 is the least significant digit. It receives no carry. Thus, digit $7 = (7 \times 7 + 0) \bmod 10 = 9$.

We can conclude, therefore, that $428657 \times 7 = 3000599$.

This method uses comparisons to obtain the carries in advance, so that calculations can be done from the left to the right. Since human beings are good at doing comparisons and they read numbers from the most significant digit to the least significant one, this method has been found to be very suitable for rapid mental calculations.

2.2 Multi-digit integer multiplied by multi-digit integer

Shi [9, 10] also studied the multiplication between two multi-digit integers. In the following, we shall first give a particular example and then discuss the general case.

Let x and y be the multiplicand and multiplier, respectively. Without loss of generality, suppose $x = 389$ and $y = 436$. After adding an extra "0" to the most significant digit of x, we write the calculation in the form of Figure 2. Please note the following notation throughout the paper: "$\langle 1\rangle\langle 2\rangle\langle 3\rangle\langle 4\rangle$" represents the digits of the multiplicand x, where $\langle 1\rangle = 0$ in the case of Figure 2. "(4)(5)(6)" represents the digits of the multiplier. "[1][2][3][4][5][6]" represents the digits of the product, where [1] may or may not be 0. Note also that "$\langle i\rangle(j)$" is *not* the value of "$\langle i\rangle \times (j)$", but represents the ith digit of the product "$\langle 1\rangle\langle 2\rangle\langle 3\rangle\langle 4\rangle \times (j)$". For example, $\langle 3\rangle(5) = 6$ in our example because $x \times (5) = 0389 \times 3 = 11\mathbf{6}7$.

The preceding example shows the following relationship between the final result and the intermediate results (without considering the carries generated by the addition operation):

digit [1] from $\langle 1\rangle(4)$
digit [2] from $\langle 2\rangle(4)$ + $\langle 1\rangle(5)$
digit [3] from $\langle 3\rangle(4)$ + $\langle 2\rangle(5)$ + $\langle 1\rangle(6)$
digit [4] from $\langle 4\rangle(4)$ + $\langle 3\rangle(5)$ + $\langle 2\rangle(6)$
digit [5] from $\langle 4\rangle(5)$ + $\langle 3\rangle(6)$
digit [6] from $\langle 4\rangle(6)$

Without loss of generality, let us take digit [3] of the product as an example to see how its value is obtained. Digit [3] resulted from the sum of three addends $\langle 3\rangle(4) + \langle 2\rangle(5) + \langle 1\rangle(6)$. The digit number of the "multiplicand" in each addend decreases from $\langle 3\rangle$ via $\langle 2\rangle$ to $\langle 1\rangle$, and hence the unit increases from 10 via 100 to 1000. The digit number of the "multiplier" in each addend increases from (4) via (5) to (6), and hence the unit decreases from 100 via 10 to 1. Thus, the unit of every addend in the addition $\langle 3\rangle(4) + \langle 2\rangle(5) + \langle 1\rangle(6)$ is 1000, so that the three elements can be added together.

Let us consider the general case. Note that when we calculate the product of two integers, we always assign the longer one to be the multiplicand and the shorter one to be the multiplier. Suppose the multiplicand consists of n digits and the multiplier consists of m digits, where $n \geq m$. It can be proved that the product will consist of $n+m$ or $n+m-1$ digits. For the simplicity of presentation, we shall assume that the product always consists of $n+m$ digits, such that its most significant digit may or may not be 0. This is also the reason why we always put an extra "0" at the most significant digit of the multiplicand.

We can now write the details of the multiplication in Table 2.

Note in Table 2 that "$\langle 1\rangle$" always equals 0, and "[1]" may or may not be 0. Table 2 is actually very similar to the traditional manual multiplication method. Before the final product is calculated, the intermediate result can be first obtained by directly summing up the elements in each column. Note that, in this intermediate result, the carries of the addition operation have not yet been sent to more significant digits. We use $\{i\}$ to denote digit i of the intermediate result and $[i]$ to denote digit i of the final product. Table 3 is an example illustrating Table 2.

We can see from Table 2 that digit $\{k\}$ of the intermediate result can be calculated as follows:

$(*)$ For $k \leq n$, $\{k\} = \langle k\rangle(n+1) + \langle k-1\rangle(n+2) + \cdots + \langle p\rangle(q)$, where $p = 1$ or $q = n+m$. For $k > n$, $\{k\} = \langle n+1\rangle(k) + \langle n\rangle(k+1) + \cdots + \langle k-m+1\rangle(n+m)$.

Multiplicand	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3\rangle$	...	$\langle n\rangle$	$\langle n+1\rangle$			
Multiplier						$(n+1)$	$(n+2)$	...	$(n+m)$
Middle Step	$\langle 1\rangle(n+1)$	$\langle 2\rangle(n+1)$	$\langle 3\rangle(n+1)$	...	$\langle n\rangle(n+1)$	$\langle n+1\rangle(n+1)$			
		$\langle 1\rangle(n+2)$	$\langle 2\rangle(n+2)$	...	$\langle n-1\rangle(n+2)$	$\langle n\rangle(n+2)$	$\langle n+1\rangle(n+2)$		
			$\langle 1\rangle(n+3)$	...	...	...	$\langle n\rangle(n+3)$	$\langle n+1\rangle(n+3)$	
				...	...	...	...	...	
				...	$\langle n-m+1\rangle(n+m)$	$\langle n-m+2\rangle(n+m)$	...	$\langle n\rangle(n+m)$	$\langle n+1\rangle(n+m)$
Intermediate Result	$\langle 1\rangle(n+1)$	$\langle 2\rangle(n+1)+$ $\langle 1\rangle(n+2)$	$\langle 3\rangle(n+1)+$ $\langle 2\rangle(n+2)+$ $\langle 1\rangle(n+3)$	...	$\langle n\rangle(n+1)+$ $\langle n-1\rangle(n+2)+...+$ $\langle n-m+1\rangle(n+m)$	$\langle n+1\rangle(n+1)+$ $\langle n\rangle(n+2)+...+$ $\langle n-m+2\rangle(n+m)$	...	...	$\langle n+1\rangle(n+m)$
Digits of the Intermediate Result	{1}	{2}	{3}	...	{n}	{n+1}	...	...	{n+m}
Digits of the Final Product	[1]	[2]	[3]	...	[n]	[n+1]	...	...	[n+m]

Table 2. n-digit integer multiplied by m-digit integer, where n $\geq$ m

Multiplicand	0	3	8	9		
Multiplier				4	3	6
Middle Step	1	5	5	6		
		1	1	6	7	
			2	3	3	4
Intermediate Result	1	6	8	15	10	4
Final Product	1	6	9	6	0	4
Digit No.	1	2	3	4	5	6

Table 3. Example of Table 2: 389 $\times$ 436

Note that, for $i = 1, 2, \ldots, n+1$ and $j = n+1, n+2, \ldots, n+m$, the value "$\langle i\rangle(j)$" can be directly calculated using the method introduced in the previous section.

Thus, we have a new multiplication algorithm that is very different from traditional ones: The algorithm can start the calculation from any digit instead of always starting from the least significant digit as is the practice for thousands of years. Note also that we need to obtain intermediate results first and then adjust them according to the carries generated in the addition operation before obtaining the final product in Tables 2 and 3.

3 Computer implementation and performance analysis

As we have stated earlier, Shi originally introduced his method for rapid mental calculation. Although a simple BASIC program was also included in [9, 10] to simulate the method, it served only the purpose of illustration and was not suitable for real-life computer applications.

We find, however, that Shi's manual calculation method has a feature suitable for cryptographic systems: its basic operation unit is a digit. In large number operations of cryptographic systems, the operands are also stored digit by digit, even though the base of the digits is not necessarily

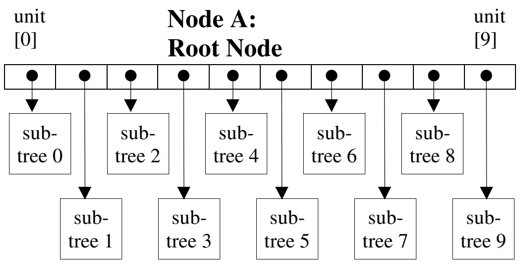

Figure 3. Search tree for multiplications in the denary system

10. In the following, we shall present our algorithm for large number multiplication based on the knowledge introduced in the previous sections. A program that implements our algorithm can be used as a deterministic checker to verify the products of integers.

3.1 Computer implementation

For conventional integer multiplication algorithms, which calculate the product starting from the least significant digit, the time cost is $O(n \times m)$, where n and m are the number of digits of the two operands [8]. For our algorithm, on the other hand, the critical part in the time cost lies in how we implement it to obtain the carries in advance. We propose to implement a search tree as shown in Figures 3 and 4.

In Figure 3, node A is the root node of the search tree. It contains ten units. Each unit contains a pointer pointing to a sub-tree. There are ten units in every node of the search tree (in both the root node and the nodes of sub-trees) because the denary system is assumed and hence each single digit has ten possible values 0 to 9. Each sub-tree in

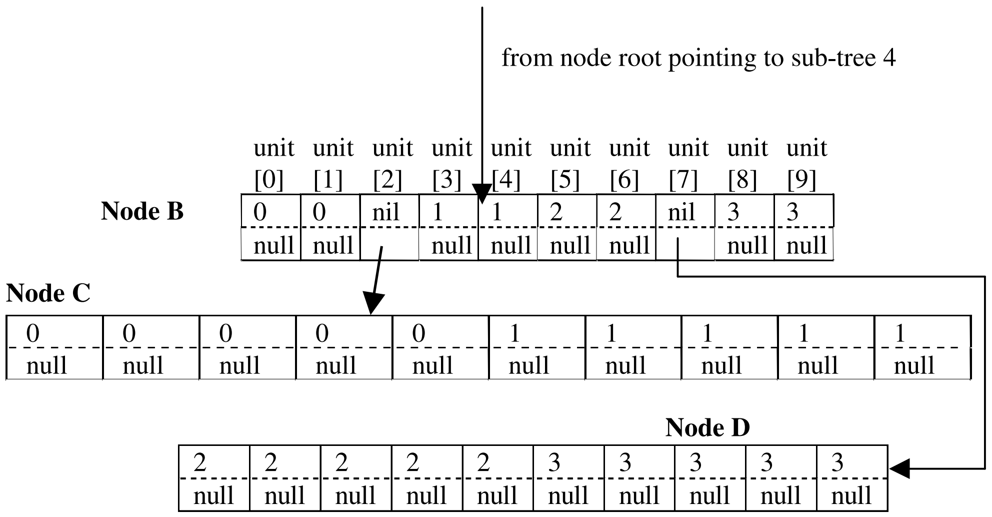
from node root pointing to sub-tree 4
unit [0] unit [1] unit [2] unit [3] unit [4] unit [5] unit [6] unit [7] unit [8] unit [9]
Node B
0 0 nil 1 1 2 2 nil 3 3
null null null null null null null null
Node C
0 0 0 0 0 1 1 1 1 1
null null null null null null null null null null
Node D
2 2 2 2 2 3 3 3 3 3
null null null null null null null null null null

Figure 4. The sub-tree 4

Figure 3 corresponds to a one-digit multiplier, namely sub-tree i corresponding to integer i, for $i = 0, 1, \ldots, 9$. Let us use the following example to illustrate how our algorithm works with the search tree.

Let A = "$A_nA_{n-1}\cdots A_1$" be a multiplicand and i be a single-digit multiplier. We search for the value of carry to the most significant digit (digit $n+1$) in the product $A \times i$. This is the fundamental operation in our algorithm. We employ the search tree as follows. First, following the pointer in unit [i] of the root node of the search tree (node A in Figure 3), we shall arrive at the root node of sub-tree i. For example, if $i = 4$, then we shall follow the pointer in unit [4] of node A to arrive at node B of Figure 4, which is the root node of the sub-tree 4.

Consider Figure 4. It shows the structure of a typical sub-tree. Each node of the sub-tree includes 10 units [0] to [9], corresponding to the values of the digits of the multiplicand. Each unit consists of two parts: the value part and the pointer part. The value part stores the value of the carry and the pointer part stores a pointer. These values are completely determined by $1/i$, $2/i$, ..., $(i-1)/i$. For instance, the sub-tree 4 in Figure 4 is constructed according to the value of $1/4 = 0.25$, $2/4 = 0.5$ and $3/4 = 0.75$.

There are 3 carry points (1/4, 2/4 and 3/4) for the multiplier 4. These 3 carry points divide the region [0, 1) into 4 sub-regions: When the multiplicand is within [0, 0.25), the carry is 0; within [0.25, 0.5), the carry is 1; within [0.5, 0.75), the carry is 2; and within [0.75, 1), the carry is 3. Hence, for the integer A = "$A_nA_{n-1}\cdots A_1$", the carry to digit $n+1$ in the product $A \times 4$ is determined by the first 1 or 2 digit(s) of A, that is, by A_n and A_{n-1}. If $A_n = 0$ or 1, then the carry will definitely be 0. Thus, unit [0] (corresponding to $A_n = 0$) and unit [1] (corresponding to $A_n = 1$) of node B are assigned the value "0", and their pointers are both assigned null, which means that the search for the carry can stop here. If $A_n = 2$, then the carry of $A \times 4$ may be 0 or 1. At this point we need to further look at A_{n-1} to make a decision. Thus, the value of unit [2] of node B (corresponding to $A_n = 2$) is "nil", which means "undecided". Its pointer indicates the lower level node C, which will provide us with the information for decision based on the value of digit A_{n-1}.

Now consider node C. When $A_n = 2$, the carry of $A \times 4$ can only have two possible values: 0 when $A_{n-1} < 5$, and 1 when $A_{n-1} \geq 5$. Hence, the value "0" is assigned to units [0] to [4], and "1" is assigned to units [5] to [9]. In any case, there will be no need for further exploring the subsequent digits A_{n-2}, A_{n-3}, ... Thus, all the pointers in node C are assigned "null" to indicate the end of the search.

Using the same method, we can assign the values to other units in node B and construct node D. Note that the units of node A, which is the root of the whole search tree, correspond to the multiplier, whereas the units of the nodes of the sub-trees correspond to the multiplicand.

Finally, let us consider a complete example that illustrates how the tree is used to search for the carry of 7498568×4. Since the multiplier is 4, we follow unit [4] of the root node A to arrive at node B, the root of the sub-tree 4. Then, according to the first digit "7" of the multiplicand, we compare the pointer in unit [7] with "null", find that it is not "null", and hence continue to follow the pointer to node D. According to the second digit "4" of the multiplicand, we compare the pointer in unit [4] with "null" and find that the pointer is "null", which means that we have found the value of the carry and the search can stop. Thus, the corresponding value in the value part of

Multiplicand	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle 5\rangle$	...	$\langle n\rangle$	$\langle n{+}1\rangle$				
Multiplier								(n+1)	(n+2)	...	(n+m)	
Middle Step	base−2 (8)	base−1 (9)	base−1 (9)	base−1 (9)	base−1 (9)	...	base−1 (9)	base−1 (9)				**Row 1**
		base−2 (8)	base−1 (9)	base−1 (9)	base−1 (9)	base−1 (9)	...	base−1 (9)	base−1 (9)			**Row 2**
			base−2 (8)	base−1 (9)	base−1 (9)	base−1 (9)	base−1 (9)	...	base−1 (9)	base−1 (9)		**Row 3**
			...	...	...	...	...	...	...	...		**...**
				base−2 (8)	base−1 (9)	...	base−1 (9)	base−1 (9)	...	base−1 (9)	base−1 (9)	**Row *m***
Intermediate Result	{1}	{2}	...	{m}	...	...	{n}	{n+1}	...	...	{n+m}	
Final Result	[1]	[2]	...	[m]	...	...	[n]	[n+1]	...	...	[n+m]	
Digit No.	1	2	...	m ($\leq n$)	...	...	n	n+1	...	...	n+m	

Table 4. Integer multiplication, n $\geq$ m

unit [4], is returned. In other words, 2 is the carry of the product 7498568×4. In this example, the time cost is two computer-word comparisons.

A slightly longer process will result when we deal with recurring decimals. For example, when a multiplier 7 is involved, all the units in sub-tree 7 contain recurring decimals. However, it will not be difficult to use flags to handle these cases.

3.2 Time cost analysis

For a j-digit integer A = "$A_jA_{j-1}\cdots A_1$" multiplied by a single digit k, it will take j computer-word comparisons in the worst case to obtain the carry to digit $j+1$ using the search tree. This worst scenario happens in the rare case when the digits of "$A_jA_{j-1}\cdots A_1$" are exactly the same as those in the expansion of i/k for some $i < k$. Even in such situations, we may make use of the recursive property of the expansion of i/k, if any. When A = "$A_jA_{j-1}\cdots A_1$" = $33\cdots 3$ and $k = 3$, for instance, it does not take $j-1$ comparisons to obtain the carry to digit j of the product. Since we do the multiplication from the left to the right, a control mechanism can be added into the search program to record the recurring values, so that we can directly copy the previous result when the pattern is repeated.

On average, each search for a carry takes a constant time. Hence, the time cost of calculating the product $A \times k$ using the search tree is $O(n)$, where A is an n-digit integer and k is a single-digit integer. Thus, the overall time cost of our algorithm in calculating the product $A \times B$ is $O(nm)$ if B contains m digits. This time cost is of the same order as that of the conventional multiplication algorithm [8].

Although the time cost of the algorithm is in the same order as that of the conventional algorithm when a search tree is implemented, it is relatively slower than complex multiplication algorithms introduced in [8]. In practice, we can use fast and complex algorithms in the implementation of multiplication functions in cryptographic systems while applying our algorithm as an oracle for testing.

4 Checking selected segments of digits in the product of two integers

In the preceding sections, we introduced a deterministic oracle for large number multiplication functions. In this section, we shall present one more feature of our testing method: not only can the whole multiplication product be checked, but the correctness of a selected segment of digits in the product can also be verified without having to calculate other digits.

Theorem 1 *Suppose the multiplicand is an n-digit integer and the multiplier is an m-digit integer in a numeration system with base > 2, such that $n \geq m$. Suppose the results of multiplications are as shown in Table 2, where $\{i\}$ denotes digit i of the intermediate result, $[i]$ denotes digit i of the final product, and digit $\langle 1\rangle = 0$. Let $C[i]$ denote the carry generated in digit i by the addition operation when calculating the final product, that is, $[i] = (\{i\} + C[i+1])$ mod base, where $i = 1, 2, \ldots, n+m$ and $C[n+m+1] = 0$. Let $max\{C[i]\}$ denote the upper bound of the value of $C[i]$. Then*

$$max\{C[i]\} = n+m-i \text{ when } n+1 \leq i \leq n+m$$
$$max\{C[i]\} = m-1 \text{ when } m \leq i < n+1$$
$$max\{C[i]\} = i-1 \text{ when } 1 \leq i < m$$

Proof In order to obtain an upper bound of $C[i]$, we assign the maximum possible values to every element in the "middle step" of Table 2 and redraw Table 2 as Table 4. In the latter table, the multiplicand and multiplier has n and m digits respectively. Since the multiplier has m digits, there are m rows in the middle step. Each digit of the intermediate result is calculated by directly adding up all the elements in its column. We assign the maximum possible value $base-1$ (such as 9 in the denary system) to every element in the middle step except the first digit of each row, whose maximum value is $base-2$ (such as 8 in the denary system).

First, consider the case when $n+1 \le i \le n+m$.

(a) When $i = n+m$, there is only one element in column i, so that $\max\{C[i]\} = 0 = n+m-i$.

(b) Suppose that, when $i = k$ for some $n+1 < k \le n+m$, we have $\max\{C[k]\} = n+m-k$. Consider the situation when $i = k-1$: Since the number of elements for addition in column j is $n+m-j+1$ for $n < j \le n+m$, the number of elements for addition in column $k-1$ is $n+m-(k-1)+1 = n+m-k+2$. Since the maximum carry that column $k-1$ can receive from less significant digits is $\max\{C[k]\} = n+m-k$, the maximum carry digit $k-1$ can generate is $\max\{C[k-1]\} = \lfloor((n+m-k+2)\times(base-1)+n+m-k) \;/\; base\rfloor = n+m-(k-1)$.

Next, consider the situation when $m \le i < n+1$.

(a) When $i = n$, the carry that digit i receives from less significant digits is no more than $\max\{C[n+1]\} = n+m-(n+1) = m-1$. Since $n+1 > m$, there are m elements in column i for addition. Thus, the maximum value of the carry generated in digit n is $\max\{C[n]\} = \lfloor(m\times(base-1)+(m-1)) \;/\; base\rfloor = m-1$.

(b) Suppose that, when $i = k$ for some $m < k \le n$, we have $\max\{C[k]\} = m-1$. Consider the situation when $i = k-1$. Since column j $(m \le j < n+1)$ includes m elements for addition, column $k-1$ also includes m elements for addition, and hence $\max\{C[k-1]\} = \lfloor(m\times(base-1)+\max\{C[k]\}) \;/\; base\rfloor = m-1$.

Finally, when $1 \le i < m$, since $m \le n$, column i contains i elements for addition. This includes one element whose maximum value is $base-2$, which is the first element of each row.

(a) When $i = m-1$, $\max\{C[i]\} = \max\{C[m-1]\} = \lfloor((m-2)\times(base-1) + (base-2) + \max\{C[m]\}) \;/\; base\rfloor = m-2 = i-1$.

(b) Suppose that, when $i = k$, $\max\{C[i]\} = i-1$. Then, when $i = k-1$, we have $\max\{C[i]\} = \max\{C[k-1]\} = \lfloor((k-2)\times(base-1) + (base-2) + \max\{C[k]\}) \;/\; base\rfloor = k-2 = i-1$. ■

Here is an example for Theorem 1: Let A and B be 15-digit and 10-digit integers, respectively, in any numeration system. If we write the calculation $A\times B$ in the form of Table 3, then the carries for digits 1 to 25, generated by the addition operation of the middle step, are no more than 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 9, 9, 9, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1 and 0, respectively. We can see a clear symmetric pattern of the carries, namely that the digits in the middle part generate or receive the most and the two ends generate or receive the least.

Let $[i..j]$ denote the integer corresponding to the digit string "$[i][i+1]\cdots[j]$", where $1 \le i \le j \le n+m$. In Table 3, for instance, $[2..4] = 696$. Using our multiplication algorithm and employing statement $(*)$, we can directly obtain a selected segment of digits $\{i\}$, $\{i+1\}$, …, $\{j\}$ in the intermediate result. Then, it is straightforward to obtain a lower bound of $[i..j]$ of the intermediate result. Let $\lfloor i..j\rfloor$ denote this lower bound. We have $\lfloor i..j\rfloor = (\sum_{k=0}^{j-i}\{j-k\}\times base^{k}) \bmod base^{j-i+1}$, where $\{j-k\}$ denotes the value of digit $j-k$ in the intermediate result. We use the notation $\lfloor i\rfloor$, $\lfloor i+1\rfloor$, …, $\lfloor j\rfloor$ to denote the digits of this lower bound. In Table 3, for example, $\lfloor 2..4\rfloor = 695$, $\lfloor 2\rfloor = 6$, $\lfloor 3\rfloor = 9$ and $\lfloor 4\rfloor = 5$. By employing Theorem 1, we can easily obtain the maximum value of the carry to digit j, denoted by $\max\{C[j+1]\}$. In the preceding example in Table 3, $\max\{C[j+1]\} = \max\{C[4+1]\} = \max\{C[5]\} = n+m-5 = 1$. Then, it is also straightforward to obtain the upper bound of $[i..j]$. Let $\lceil i..j\rceil$ denote this upper bound. We have $\lceil i..j\rceil = (\lfloor i..j\rfloor + \max\{C[j+1]\}) \bmod base^{j-i+1}$. We use the notation $\lceil i\rceil$, $\lceil i+1\rceil$, …, $\lceil j\rceil$ to denote the digits of this upper bound. In Table 3, for instance, $\lceil 2..4\rceil = (\lfloor 2..4\rfloor + \max\{C[5]\}) \bmod 10^3 = 696$. In this way, without calculating other digits, we can obtain a narrow range of the possible values of a selected segment of digits in the final product, namely $\lfloor i..j\rfloor \le [i..j] \le \lceil i..j\rceil$. Theorem 2 proves the remarkable accuracy of this approach.

Theorem 2 *Let Table 4 be the integer multiplication in a numeration system with the base* > 2. *Let* "$\lfloor i\rfloor$, $\lfloor i+1\rfloor$, …, $\lfloor j-d\rfloor$, …, $\lfloor j\rfloor$" *and* "$\lceil i\rceil$, $\lceil i+1\rceil$, …, $\lceil j-d\rceil$, …, $\lceil j\rceil$" *be the lower and upper bounds of the segment of digits in the final product* "$[i]$, $[i+1]$, …, $[j-d]$, …, $[j]$", *respectively, where* $d = \lfloor\log_{base} max\{C[j+1]\}\rfloor + 1$ *when* $max\{C[j+1]\} \ne 0$ *and* $d = 1$ *when* $max\{C[j+1]\} = 0$, *that is, d is the number of digits of* $max\{C[j+1]\}$. *Then, one of the following two identities holds:*

(a) $\lceil i..(j-d)\rceil = \lfloor i..(j-d)\rfloor$

(b) $\lceil i..(j-d)\rceil = (\lfloor i..(j-d)\rfloor + 1)$ *mod* $base^{j-d-i+1}$

Thus, one of the following two identities holds:

(c) $[i..(j-d)] = \lfloor i..(j-d)\rfloor$

(d) $[i..(j-d)] = \lceil i..(j-d)\rceil = (\lfloor i..(j-d)\rfloor + 1)$ *mod* $base^{j-d-i+1}$

where $[i..(j-d)]$ *is the integer corresponding to the string from digit* $[i]$ *to digit* $[j-d]$ *of the final product.*

$\lfloor i..j \rfloor$	$\lfloor i \rfloor$	$\lfloor i+1 \rfloor$	$\cdots$	$\lfloor j-d \rfloor$	$\lfloor j-d+1 \rfloor$	$\cdots$	$\cdots$	$\lfloor j \rfloor$
$\max\{C[j+1]\}$					C_1	C_2	$\cdots$	C_d
$\lceil i..j \rceil$	$\lceil i \rceil$	$\lceil i+1 \rceil$	$\cdots$	$\lceil j-d \rceil$	$\lceil j-d+1 \rceil$	$\cdots$	$\cdots$	$\lceil j \rceil$

Table 5. $\lceil i..j \rceil = (\lfloor i..j \rfloor + \max\{C[j+1]\})$ **mod** $base^{j-i+1}$

A	0	8	6	9	4	9	8	6	5	2	9	4	0	7	3	4									
B																3	6	8	7	4	8	9	8	9	5
Middle Step	2	6	0	8	4	9																			
		5	2	1	6	9																			
			6	9	5	5																			
				6	0	8																			
					3	4																			
						6																			
Lower Bound	$\bar{3}$	$\bar{2}$	$\bar{0}$	$\bar{6}$	$\bar{2}$	1																			
Max{*C*[7]}						6																			
Upper Bound	$\bar{3}$	$\bar{2}$	$\bar{0}$	$\bar{6}$	$\bar{2}$	7																			
Assured Digits	$\bar{3}$	$\bar{2}$	$\bar{0}$	$\bar{6}$	$\bar{2}$																				
Real Result	$\bar{3}$	$\bar{2}$	$\bar{0}$	$\bar{6}$	$\bar{2}$	6	7	4	9	6	4	3	5	0	6	8	6	5	8	8	8	2	9	3	0
Digit No.	1	2	3	4	5	6	7	...																	

Table 6. Checking the first 6 digits of the product 869498652940734 $\times$ 3687489895

Proof By definition, $\lceil i..j \rceil = (\lfloor i..j \rfloor + \max\{C[j+1]\})$ mod $base^{j-i+1}$. Since $\max\{C[j+1]\}$ has d digits, we can write the addition operation in the form of Table 5, where C_r denotes the rth digit of $\max\{C[j+1]\}$, $r = 1, 2, \ldots, d$, and C_1 is the most significant digit. In the table, digit $i-1$ is omitted, which is equivalent to the operation "mod $base^{j-i+1}$". In any numeration system, in the addition calculation shown in Table 5, the value of carry that digit $(j-d)$ can receive from its lower part is no more than 1. Hence, $\lceil i..(j-d) \rceil = \lfloor i..(j-d) \rfloor + 0$, or $\lceil i..(j-d) \rceil = (\lfloor i..(j-d) \rfloor + 1)$ mod $base^{j-d-i+1}$. Thus, $[i..(j-d)] = \lfloor i..(j-d) \rfloor$, or $[i..(j-d)] = \lceil i..(j-d) \rceil = (\lfloor i..(j-d) \rfloor + 1)$ mod $base^{j-d-i+1}$. ■

Table 6 illustrates how our method is applied to verify the correctness of a selected segment of digits in integer multiplication. The multiplicand is $A = 869498652940734$ with $n = 15$ digits, and the multiplier is $B = 3687489895$ with $m = 10$ digits. We would like to check the first 6 digits of the product. The numbers in the "middle step" can be directly obtained using our multiplication algorithm. By employing statement $(*)$, we obtain a lower bound for the segment of digits 1 to 6, namely 320621. Then, according to Theorem 1, we can easily obtain the maximum value of the carry that digit 6 may receive from less significant digits, namely $\max\{C[7]\} = 7 - 1 = 6$. By adding $\max\{C[7]\}$ to digit 6 of the lower bound, we obtain an upper bound for the segment of digits, namely 320627. We can see that the first 5 digits of the lower and upper bounds are exactly the same, and hence these 5 digits can be assured, and digit 6 of the final product should be in the range [1, 7].

5 Conclusion

In this paper, we have introduced a deterministic algorithm that provides an oracle for the testing of large number multiplication functions in cryptographic systems. Our method can be used to verify the correctness of either the whole product of two integers, or a selected segment of digits in the product. The approach is based on Shi's mental calculation method.

The method we presented in this paper is deterministic, efficient and suitable for any type of testing strategy, especially the testing of large number multiplication in cryptographic systems. In addition, it can also be applied to verify significant digits of the products of decimal fractions. Table 7 shows a comparison of our method with others.

References

[1] L.M. Adleman, M.-D. Huang and K. Kompella. Efficient checkers for number-theoretic computations. *Information and Computation*, 121: 93–102, 1995.

	Effectiveness	Testing Data Selection	Suitable Testing Strategy
Other Methods	probabilistic (<100%)	randomly	black box testing
Our Method	deterministic (100%)	no requirement	both black and white box
	Efficiency	**Suitability for Cryptographic Systems**	**Can Check Selected Parts of Products**
Other Methods	repeated executions require high time cost	unsuitable	no
Our Method	only one execution, efficient	suitable	yes

Table 7. Comparison with other multiplication testing methods

[2] B. Beizer. *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, 1990.

[3] M. Blum, M. Luby and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47: 549–595, 1993.

[4] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42 (1): 269–291, 1995.

[5] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[6] M.-C. Gaudel. Testing can be formal, too. In *Proceedings of 6th International Joint CAAP/FASE Conference on Theory and Practice of Software Development* (*TAPSOFT '95*), volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, Berlin, Germany, 1995.

[7] M. Kaminski. A note on probabilistically verifying integer and polynomial products. *Journal of the ACM*, 36 (1): 142–149, 1989.

[8] D.E. Knuth. *The Art of Computer Programming*, volume 2, Addison Wesley, Reading, MA, 1998.

[9] F. Shi. *A Rapid Calculation Method* (in Chinese), Anhui Science and Technology, Hefei, China, 1979.

[10] F. Shi. *A Marvel of Intelligence Development: Shi Fengshou Rapid Calculation Method that Challenges Computers* (in Chinese), Joint Publishing Co., Hong Kong, 1990.

[11] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4): 465–470, 1982.

[12] W. Stallings. *Cryptography and Network Security: Principles and Practice*, Prentice Hall, Upper Saddle River, NJ, 1999.