Parallel Multiplication in $GF(2^k)$ using Polynomial Residue Arithmetic *†

A. Halbutoğulları‡ and Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Abstract

We present a novel method of parallelization of the multiplication operation in $GF(2^k)$ for an arbitrary value of k and arbitrary irreducible polynomial $n(x)$ generating the field. The parallel algorithm is based on polynomial residue arithmetic, and requires that we find L pairwise relatively prime moduli $m_i(x)$ such that the degree of the product polynomial $M(x) = m_1(x)m_2(x)\cdots m_L(x)$ is at least $2k$. The parallel algorithm receives the residue representations of the input operands (elements of the field) and produces the result in its residue form, however, it is guaranteed that the degree of this polynomial is less than k and it is properly reduced by the generating polynomial $n(x)$, i.e., it is an element of the field. In order to perform the reductions, we also describe a new table lookup based polynomial reduction method.

Key Words: Finite field multiplication, residue arithmetic, parallel algorithm, cryptography.

1 Introduction

The arithmetic operations in the Galois field $GF(2^k)$ have several applications in coding theory, computer algebra and cryptography. We are especially interested in cryptographic applications where k is very large. The cryptographic applications include elliptic curve cryptosystems [7, 3, 6] and the Diffie-Hellman key exchange algorithm [1] based on the discrete exponentiation. In this paper, we describe a parallel multiplication algorithm for the field $GF(2^k)$ using Polynomial Residue Arithmetic (PRA). By proper selection of the modulus polynomials in the PRA, and the application of the Chinese Remainder Theorem (CRT), we obtain an efficient parallel algorithm for multiplying two elements of $GF(2^k)$ represented in the polynomial basis. The proposed parallel algorithm is more suitable for hardware implementations. It provides an alternative method of parallelization to the algorithm based on other types of bases. For example, the well-known Massey-Omura [8] algorithm uses the normal basis representation, where each element of result can be computed in parallel by replicating the same multiplication circuit k times and then by providing inputs to these circuits as the bit-level rotated versions of the input operands.

The parallel algorithm proposed in this paper is based on polynomial representation of the field elements with respect to an irreducible polynomial $n(x)$ of degree k over the field $GF(2)$. An element a of the field $GF(2^k)$ is represented using a polynomial $a(x)$ of degree $k-1$ (length k), whose coefficients are from the binary field $GF(2)$, i.e., $a(x) = (a_{k-1}a_{k-2}\cdots a_0)$, where $a_i \in \{0, 1\}$.

Designs, Codes and Cryptography, 20(2):155–173, June 2000.
†This research is supported in part by Intel Corporation and Secured Information Technology, Inc.
‡Present Address: i2 Technologies, 565 Technology Square, 9th Floor, Cambridge, MA 02139.

Furthermore, we assume that $sw \geq k > (s-1)w$, where w is the wordsize which depends on the implementation details and the computer. Usually, we take w as 8 or 16, which determine the sizes of the tables needed as well as the total computational time. We use the word-level representation of $a(x) = (A_{s-1}A_{s-2}\cdots A_0)$, where A_i is of length w for $i = 0, 1, \ldots, s-1$. In order to simplify the analysis, we will often take $k = sw$.

In order to multiply two elements a and b in $GF(2^k)$, we need the irreducible polynomial $n(x)$. The product $c = a \cdot b$ in $GF(2^k)$ is obtained by computing

$$c(x) = a(x)b(x) \bmod n(x) \ , \tag{1}$$

where $c(x)$ is a polynomial of length k, representing the element $c \in GF(2^k)$. Thus, the multiplication operation in the field $GF(2^k)$ is accomplished by first multiplying the input polynomials, and then performing a polynomial modular reduction using the generating polynomial $n(x)$.

2 Polynomial Residue Arithmetic

Let $m_1(x), m_2(x), \ldots, m_L(x)$ be a list of pairwise relatively prime polynomials such that the degree of $m_i(x)$ is equal to d_i for $i = 1, 2, \ldots, L$. We choose $m_i(x)$ such that each d_i is approximately equal to $2k/L$, and thus, the product polynomial $M(x) = \prod_{i=1}^{L} m_i(x)$ is of degree $d \geq 2k$. We represent a polynomial $p(x)$ using a list of remainders

$$\vec{p} = (p_1, p_2, \ldots, p_L) \ , \tag{2}$$

where $p_i(x) = p(x) \bmod m_i(x)$ for $i = 1, 2, \ldots, L$. For efficiency reasons, we select each $m_i(x)$ so that $p_i(x)$ is represented using at most w bits or 1 word. This implies that $\deg(p_i(x)) = d_i < w$. Furthermore, the polynomials $m_i(x)$ need to be pairwise relatively prime, i.e., $\gcd(m_i, m_j) = 1$ for $i \neq j$. Therefore, we construct a Polynomial Residue System (PRS) by finding L pairwise relatively prime polynomials $m_i(x)$, each of which is of degree w, such that the degree of $M(x)$ is $Lw \geq 2k$. Since $k = sw$, we have $L \geq 2s$. The reason for choosing the range of PRS as twice the size of the inputs is that we need to represent the product of two operands (or the square of one operand) uniquely.

Once the PRS is constructed by proper selection of the L such polynomials, we can perform polynomial residue arithmetic. The residue addition and multiplication operations in polynomial residue arithmetic are defined as follows:

- $\vec{c} := \vec{a} + \vec{b}$ represents the residue addition: $c_i := a_i + b_i \bmod m_i$ for $i = 1, 2, \ldots, n$.
- $\vec{c} := \vec{a} * \vec{b}$ represents the residue multiplication: $c_i := a_i \cdot b_i \bmod m_i$ for $i = 1, 2, \ldots, n$.

If the polynomial representation of an operand a has degree less than w, then we will assume that it is a vector with all entries equal to a as $(a, a, \ldots, a)$. We will use the notation $\vec{c} = a * \vec{b}$ to mean $(c_1, c_2, \ldots, c_L) = (a, a, \ldots, a) * (b_1, b_2, \ldots, b_L)$, or in other words, $c_i = a \cdot b_i \bmod m_i$.

The conversion from the PRS representation to the weighted polynomial representation is based on the extension of the Chinese Remainder Theorem to polynomials [5, 2]. Given the PRS representation of $p(x)$ as $\vec{p} = (p_1, p_2, \ldots, p_L)$, we use the Single Radix Conversion (SRC) algorithm to compute $p(x)$. Let $M_i(x)$ for $i = 1, 2, \ldots, L$ be defined as

$$M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x)m_2(x)\cdots m_{i-1}(x)m_{i+1}(x)\cdots m_L(x) \ . \tag{3}$$

The inverse of $M_i(x)$ modulo $m_i(x)$ exists since $\gcd(M_i, m_i) = 1$. We define the inverse $I_i(x)$ as

$$I_i = M_i^{-1} \pmod{m_i} \ . \tag{4}$$

The SRC algorithm computes the weighted polynomial $p(x)$ using the formula

$$p(x) = \sum_{i=1}^{L} (p_i \cdot I_i \mod m_i) \cdot M_i \ . \tag{5}$$

Unlike the integer case, the final reduction by the product polynomial $M(x)$ is not necessary in the case for polynomials over $GF(2)$. The degree of the sum is the less than or equal to the term $(p_i \cdot I_i \bmod m_i) \cdot M_i$, whose degree is at most $Lw-1$ since $\deg(p_i \cdot I_i) < w$ and $\deg(M_i) \leq (L-1)w$. We will assume that the coefficients M_i for $i = 1, 2, \ldots, L$ and the inverse vector

$$\vec{I} = (M_1^{-1} \bmod m_1, M_2^{-1} \bmod m_2, \ldots, M_L^{-1} \bmod m_L) \tag{6}$$

are precomputed and used in the SRC algorithm. Using these definitions, we give the steps of SRC algorithm which computes the polynomial $p(x)$ given its residue representation $\vec{p}$ as follows:

The SRC Algorithm	
Input:	$\vec{p} = (p_1, p_2, \ldots, p_L)$
Output:	$p(x) \bmod M(x)$
Auxiliary:	$M_1, M_2, \ldots, M_L$ and $\vec{I}$
Step 1.	$\vec{r} := \vec{p} * \vec{I}$
Step 2.	$p(x) := \sum_{i=1}^{L} r_i \cdot M_i$
Step 3.	return $p(x)$

If $s(x) = a(x) + b(x)$, then the degree of $s(x)$ is not larger than the maximum of the degrees $of a(x)$ and $b(x)$, thus, the polynomial residue arithmetic would yield the exact result, i.e., $\vec{s} = \vec{a} + \vec{b}$. However, in multiplication $p(x) = a(x)b(x)$, the degree of the resulting polynomial increases. The polynomial $p(x)$ needs to be reduced modulo the irreducible polynomial $n(x)$ in order to obtain the product $c = a \cdot b$ in $GF(2^k)$. Therefore, if we want to use polynomial residue arithmetic for multiplication modulo $n(x)$, we need to devise a method to reduce the resulting polynomial modulo $n(x)$. We will perform this reduction using the table lookup reduction algorithm which is described and analyzed in the following sections.

3 Table Lookup Reduction Algorithm

We propose a modular reduction method which uses a table of the multiples of the generating polynomial $n(x)$, and performs *word-level divisions*. We start with $n(x)$, which is a polynomial of degree k, and compute all multiples of $n(x)$ having degrees less than $k + w$. Consider the set Q_w of all polynomials over $GF(2)$ of length w and the set I_w of all w-bit integers as

$$\begin{aligned} Q_w &= \{0, 1, x, x+1, x^2, x^2+1, x^2+x, \ldots, x^{w-1} + x^{w-2} + \cdots + 1\} \ , \\ I_w &= \{0, 1, 2, \ldots, 2^w - 1\} \ . \end{aligned}$$

Let $i \in I_w$ and $q_i(x) \in Q_w$. The binary representation of the integer i is given as $(i_{w-1} i_{w-2} \cdots i_0)$ which determines $q_i(x)$ as $\sum_{j=0}^{w-1} i_j x^j$. We also define $v_i(x) = q_i(x)n(x)$ for $i \in I_w$. The polynomial $v_i(x)$ is of degree less than $k + w$, which we represent as an $(s+1)$-word number

$$v_i(x) = (V_{i,s} V_{i,s-1} \cdots V_{i,1} V_{i,0}) \ . \tag{7}$$

We then construct the table T containing 2^w rows, in which we store the polynomial $v_i(x)$ using its most significant word (w-bits) as the index, i.e.,

$$T(V_{i,s}) = (V_{i,s-1} \cdots V_{i,1} V_{i,0}) \ , \tag{8}$$

for $i \in I_w$. An important observation is that the most significant words $V_{i,s}$ for $i \in I_w$ span the set Q_w, in other words, they are all unique.

Proposition 1 *The most significant words $V_{i,s}$ are all unique for $i \in I_w$.*

Proof Assume $V_{i,s} = V_{j,s}$ for $i \neq j$. The polynomial $p(x) = v_i(x) + v_j(x)$ is of length at most k since

$$\begin{aligned} p(x) &= (V_{i,s}V_{i,s-1} \cdots V_{i,1}V_{i,0}) + (V_{j,s}V_{j,s-1} \cdots V_{j,1}V_{j,0}) \\ &= (0P_{s-1} \cdots P_1P_0) \ . \end{aligned}$$

Furthermore, $p(x)$ is divisible by $n(x)$ since

$$p(x) = v_i(x) + v_j(x) = q_i(x)n(x) + q_j(x)n(x) = (q_i(x) + q_j(x))n(x) \ ,$$

which means $p(x)$ can only be the zero polynomial, i.e., $i = j$. □

The table T is used for reducing polynomials modulo $n(x)$. Let $p(x)$ be a polynomial of length $sw + w$ denoted as $p(x) = (P_sP_{s-1} \cdots P_1P_0)$ which is to be reduced. The reduction algorithm computes $p(x) \bmod n(x)$, which is of length sw. In order to reduce $p(x) = (P_sP_{s-1} \cdots P_1P_0)$, we select the entry $(V_{s-1}V_{s-2} \cdots V_1V_0)$ from the table T using the index $P_s = V_s$. Since T was constructed so that the element $(P_sV_{s-1}V_{s-2} \cdots V_1V_0)$ resides in position P_s, we have

$$\begin{aligned} p(x) &:= (P_sP_{s-1} \cdots P_1P_0) + (P_sV_{s-1}V_{s-2} \cdots V_1V_0) \\ &:= (0P'_{s-1} \cdots P'_1P'_0) \ , \end{aligned}$$

where $P'_j = P_j + V_j$ for $j = 0, 1, \ldots, s-1$. We also discard the most significant w bits of the new $p(x)$, which are all zeros. Since we add a multiple of $n(x)$ to $p(x)$, and obtain a polynomial of length sw, we effectively compute $p(x) \bmod n(x)$, as required. We denote the above computation as

$$p(x) := p(x) + T(P_s) \ . \tag{9}$$

4 Multiplication using Table Lookup Reduction

The multiplication algorithm computes $c(x) = a(x)b(x) \bmod n(x)$ given $a(x)$, $b(x)$, and $n(x)$. In order to apply the table lookup reduction method, we first construct the table T using the generating polynomial $n(x)$. The algorithm then proceeds by multiplying one word of $a(x)$ by the entire $b(x)$, which is followed by a table lookup reduction to reduce the partial product. The steps of the table lookup multiplication algorithm are given below:

The Table Lookup Multiplication Algorithm

Input:	$a(x)$ and $b(x)$
Output:	$c(x)$
Auxiliary:	$n(x)$, T, and w
Step 1.	$c(x) := 0$
Step 2.	for $i = s-1$ downto 0 do
Step 3.	$\quad c(x) := x^w c(x) + A_i(x)b(x)$
Step 4.	$\quad c(x) := c(x) + T(C_s)$
Step 5.	return $c(x)$

The operation in Step 4 is performed by first discarding the sth (the most significant) word of $c(x) = (C_s C_{s-1} \cdots C_1 C_0)$, and then by adding the s-word number $T(C_s) = (M_{s-1} M_{s-2} \cdots M_1 M_0)$ to the partial product $c(x)$ as

$$\begin{array}{cccccc} & C_{s-1} & C_{i-2} & \cdots & C_1 & C_0 \\ + & M_{s-1} & M_{s-2} & \cdots & M_1 & M_0 \\ \hline \end{array}$$

Similarly, we perform the squaring operation using the table lookup reduction method. The steps of the squaring algorithm steps are given below. An important saving in this case is that the cross product terms disappear because the ground field is $GF(2)$. Since

$$a^2(x) = \sum_{i=0}^{k-1} a_i x^{2i} \;=\; a_{k-1} x^{2(k-1)} + a_{k-2} x^{2(k-2)} + \cdots + a_1 x^2 + a_0 \;, \tag{10}$$

the word-level multiplications (Step 3) can be skipped. The squaring algorithm starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$ given by

$$c(x) = (a_{k-1} \mathbf{0} a_{k-2} \mathbf{0} \cdots \mathbf{0} a_1 \mathbf{0} a_0) \;,$$

and then performs the reduction steps using the table T.

The Table Lookup Squaring Algorithm
Input: $a(x)$
Output: $c(x)$
Auxiliary: $n(x)$, T, and w
Step 1. $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2. for $i = 2s-1$ downto s do
Step 3. $\quad c(x) := c(x) + T(C_i)$
Step 4. return $c(x)$

We perform the operation in Step 3 by first discarding the ith (the most significant) word of $c(x) = (C_i C_{i-1} \cdots C_1 C_0)$, and then by adding the s-word number

$$T(C_i) = (V_{s-1} V_{s-2} \cdots V_1 V_0)$$

to the partial product $c(x)$ by aligning V_{s-1} with C_{i-1} from the left:

$$\begin{array}{cccccccccc} & C_{i-1} & C_{i-2} & \cdots & C_{i-s+1} & C_{i-s} & C_{i-s-1} & \cdots & C_1 & C_0 \\ + & V_{s-1} & V_{s-2} & \cdots & V_1 & V_0 & & & & \\ \hline \end{array}$$

Thus, each addition operation in Step 3 requires exactly s XOR operations.

5 An Example of Table Lookup Multiplication

We take the field $GF(2^8)$ to illustrate the construction of the table T, and also give an example of the table lookup multiplication method. We select the irreducible polynomial as

$$n(x) = x^8 + x^5 + x^3 + x^2 + 1 = (1\ 0010\ 1101) \;. \tag{11}$$

We also select $w = 4$, which gives $s = k/w = 8/4 = 2$. The table T is constructed by taking a polynomial $q(x)$ from Q_4, multiplying it by $n(x)$ to obtain $v(x) = q(x)n(x)$, and then placing the least significant $s = 2$ words of $v(x)$ to T using the most significant word as the index. The step-by-step construction of T is shown in Figure 1. The multiples of $n(x)$ do not necessarily come in an increasing order, however, we have a complete set of first words, and thus, we can use these values as their indices to store them in T. The table T is shown in Figure 1 in its unsorted form.

Figure 1: The construction of T for $n(x) = (0001\ 0010\ 1101)$.

$q(x)$	$v(x)$	i	$T(i)$
(0000)	(0000 0000 0000)	(0000)	(0000 0000)
(0001)	(0001 0010 1101)	(0001)	(0010 1101)
(0010)	(0010 0101 1010)	(0010)	(0101 1010)
(0011)	(0011 0111 0111)	(0011)	(0111 0111)
(0100)	(0100 1011 0100)	(0100)	(1011 0100)
(0101)	(0101 1001 1001)	(0101)	(1001 1001)
(0110)	(0110 1110 1110)	(0110)	(1110 1110)
(0111)	(0111 1100 0011)	(0111)	(1100 0011)
(1000)	(1001 0110 1000)	(1001)	(0110 1000)
(1001)	(1000 0100 0101)	(1000)	(0100 0101)
(1010)	(1011 0011 0010)	(1011)	(0011 0010)
(1011)	(1010 0001 1111)	(1010)	(0001 1111)
(1100)	(1101 1101 1100)	(1101)	(1101 1100)
(1101)	(1100 1111 0001)	(1100)	(1111 0001)
(1110)	(1111 1000 0110)	(1111)	(1000 0110)
(1111)	(1110 1010 1011)	(1110)	(1010 1011)

As an example, we take

$$\begin{aligned} a(x) &= x^7 + x^6 + x^4 + x^3 + x + 1\ , \\ b(x) &= x^7 + x^5 + x^3 + x^2 + x\ . \end{aligned}$$

We have $a = (A_1A_0) = (1101\ 1011)$ and $b = (B_1B_0) = (1010\ 1110)$. The algorithm starts with $c(x) = 0$ and then performs the following steps to find the result:

$$\begin{array}{lllll} i=1 & \text{Step 3:} & c(x) & := & c(x)x^4 + A_1(x)b(x) = (C_2C_1C_0) \\ & & & = & 0 + (1101)(1010\ 1110) = (0111\ 0110\ 0110) \\ & \text{Step 4:} & c(x) & := & c(x) + T(C_2) = (C_1C_0) + T(C_2) \\ & & & = & (0110\ 0110) + (1100\ 0011) = (1010\ 0101) \\ i=0 & \text{Step 3:} & c(x) & := & c(x)x^4 + A_0(x)b(x) = (C_2C_1C_0) \\ & & & = & (1010\ 0101\ 0000) + (1011)(1010\ 1110) = (1110\ 1101\ 0010) \\ & \text{Step 4:} & c(x) & := & c(x) + T(C_2) = (C_1C_0) + T(C_2) \\ & & & = & (1101\ 0010) + (1010\ 1011) = (0111\ 1001) \end{array}$$

Therefore, the result is found as $c(x) = (0111\ 1001) = x^6 + x^5 + x^4 + x^3 + 1$.

6 Parallel Multiplication using Polynomial Residue Arithmetic

In this section, we present the parallel multiplication algorithm based on polynomial residue arithmetic and the table lookup reduction method. In order to utilize the table lookup multiplication algorithm, we compute the multiples of $n(x)$ in the PRS representation and store them similarly in the table $\vec{T}$, which has 2^w rows and L independent columns, where each column contains a 1-word number at each row. If the PRS representation of $(V_sV_{s-1}\ldots V_1V_0)$ is given as $\vec{v} = (v_1, v_2, \ldots, v_L)$, then the row V_s of the table $\vec{T}$ holds v_i in column i for $i = 1, 2, \ldots, L$, i.e., $\vec{T}[V_s] = (v_1, v_2, \ldots, v_L)$. The tables T and $\vec{T}$ are used to reduce a polynomial modulo the irreducible polynomial $n(x)$. The steps of the PRA-based multiplication algorithm are given below.

THE PRA-BASED MULTIPLICATION ALGORITHM

Input:	$\vec{a}$ and $\vec{b}$
Output:	$\vec{c}$ and $c(x)$
Auxiliary:	$M_1, M_2, \ldots, M_L$, $\vec{I}$, T, and $\vec{T}$
Step 1.	$\vec{c} := \vec{a} * \vec{b}$
Step 2.	$\vec{r} := \vec{c} * \vec{I}$
Step 3.	$c(x) := \sum_{i=1}^{L} r_i \cdot M_i$
Step 4.	for $i = 2s - 1$ downto s
Step 5.	$\quad \vec{c} := \vec{c} + \vec{T}[C_i] * x^{(i-s)w}$
Step 6.	$\quad c(x) := c(x) + T[C_i] \cdot x^{(i-s)w}$
Step 7.	return $\vec{c}$ and $c(x)$

Assuming the input polynomials $a(x)$ and $b(x)$ are of degree at most $k-1$, we have the product polynomial $c(x)$ in its residue representation at the end of Step 1, however, this polynomial is of degree at most $2(k-1)$. The representation is still unique, since the degree of $M(x)$ is at least $2k$, and thus, the SRC algorithm will yield a unique result. However, we cannot use the resulting polynomial $c(x)$ and its residue representation $\vec{c}$ as an input to another multiplication. We need to use the generating polynomial $n(x)$ to reduce $\vec{c}$ and $c(x)$ so that the result is again less than $n(x)$. Steps 3–6 accomplish this reduction. First we use the SRC algorithm to compute $c(x)$, and then in Steps 5 and 6, we use the table lookup reduction algorithm to reduce $c(x)$ so that it is of degree at most $k-1$. We perform the operation in Step 6 by first discarding the ith (the most significant) word C_i of $c(x) = (C_i C_{i-1} \cdots C_1 C_0)$, and then by adding the s-word number $T[C_i] = (V_{s-1} V_{s-2} \cdots V_1 V_0)$ to the partial product $c(x)$ by aligning V_{s-1} with C_{i-1} from the left:

$$\begin{array}{cccccccccc} & C_{i-1} & C_{i-2} & \cdots & C_{i-s+1} & C_{i-s} & C_{i-s-1} & \cdots & C_1 & C_0 \\ + & V_{s-1} & V_{s-2} & \cdots & V_1 & V_0 & & & & \end{array}$$

The addition of the most significant words $C_i + C_i = 0$ is not performed. Also the terms C_{i-s-1} down to C_0 are not involved in the addition either. Only the terms starting from V_{s-1} down to V_0 are added to the corresponding terms of $c(x)$ in order to reduce $c(x)$ modulo $n(x)$. Thus, the shift factor $x^{(i-s)w}$ is taken care of by this alignment process.

Furthermore, as we reduce $c(x)$ modulo $n(x)$ in Step 6, we also reduce its residue representation $\vec{c}$ using $\vec{n}$ and $\vec{T}$ in Step 5 by multiplying the residue numbers $\vec{T}[C_i]$ and $(x^{(i-s)w}, x^{(i-s)w}, \ldots, x^{(i-s)w})$, and then adding the result to $\vec{c}$.

7 An Example of the PRA-based Multiplication

We will illustrate the PRA-based multiplication algorithm for the field $GF(2^8)$ generated by the irreducible polynomial $n(x) = x^8 + x^5 + x^3 + x^2 + 1 = (1\ 0010\ 1101)$ which is the same polynomial (11) used in §5. Since $k = 8$, $w = 4$, and $s = k/w = 2$, we have $L = 2s = 4$, which implies that we need 4 pairwise relatively prime polynomials $m_i(x)$, each of which is of degree 4, to construct the PRS. We select the following:

$$\begin{aligned} m_1(x) &= x^4 + x + 1\ , \\ m_2(x) &= x^4 + x^3 + 1\ , \\ m_3(x) &= x^4 + x^3 + x^2 + 1\ , \\ m_4(x) &= x^4 + x^3 + x^2 + x + 1\ . \end{aligned}$$

This gives us $M(x) = m_1(x)m_2(x)m_3(x)m_4(x)$ as

$$M(x) = x^{16} + x^{15} + x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + 1 \ . \tag{12}$$

The following values are also easily computed:

$$\begin{aligned}
M_1 &= x^{12} + x^{11} + x^{10} + x^7 + x + 1 \ , \\
M_2 &= x^{12} + x^{10} + x^8 + x^3 + x^2 + 1 \ , \\
M_3 &= x^{12} + x^9 + x^6 + x^3 + 1 \ , \\
M_4 &= x^{12} + x^8 + x^5 + x^4 + x^3 + x^2 + x + 1 \ , \\
\vec{n} &= (x^3 + x^2 + x, x^3, x^3, x^2) \ , \\
\vec{I} &= (M_1^{-1}, M_2^{-1}, M_3^{-1}, M_4^{-1}) \ , \\
&= (x + 1, x^3 + x + 1, x^2 + x + 1, x^3 + x^2) \ .
\end{aligned}$$

We then compute the multiples of $n(x)$ and $\vec{n}$ and store them in tables T and $\vec{T}$. The construction of the table T was already shown in §5 in Figure 1. In order to construct $\vec{T}$, we take $q_i(x)n(x)$, where $q_i(x)$ is an element of Q_w, and compute the residues of this polynomial with respect to each of the modulus as $q_i(x)n(x) \pmod{m_j(x)}$ for $j = 1, 2, 3, 4$. For example, $\vec{T}(0010)$ is computed by taking $q_2(x)n(x) = (x)(x^8 + x^5 + x^3 + x^2 + 1) = x^9 + x^6 + x^4 + x^3 + x$, and then reducing it modulo $m_i(x)$ for $i = 1, 2, 3, 4$ as

$$\begin{aligned}
x^9 + x^6 + x^4 + x^3 + x \quad &(\text{mod } x^4 + x + 1) &&= 1111 \ , \\
x^9 + x^6 + x^4 + x^3 + x \quad &(\text{mod } x^4 + x^3 + 1) &&= 1000 \ , \\
x^9 + x^6 + x^4 + x^3 + x \quad &(\text{mod } x^4 + x^3 + x^2 + 1) &&= 1000 \ , \\
x^9 + x^6 + x^4 + x^3 + x \quad &(\text{mod } x^4 + x^3 + x^2 + x + 1) &&= 0100 \ ,
\end{aligned}$$

which gives $\vec{T}(0010) = (1111, 1000, 1000, 0100)$. The final forms of the lookup tables T and $\vec{T}$ are given in Figure 2.

Figure 2: The lookup tables T and $\vec{T}$.

i	$T(i)$	$\vec{T}(i)$
(0000)	(0000 0000)	(0000, 0000, 0000, 0000)
(0001)	(0010 1101)	(1110, 1000, 1000, 0100)
(0010)	(0101 1010)	(1111, 1001, 1101, 1000)
(0011)	(0111 0111)	(0001, 0001, 0101, 1100)
(0100)	(1011 0100)	(1101, 1011, 0111, 1111)
(0101)	(1001 1001)	(0011, 0011, 1111, 1011)
(0110)	(1110 1110)	(0010, 0010, 1010, 0111)
(0111)	(1100 0011)	(1100, 1010, 0010, 0011)
(1000)	(0100 0101)	(0111, 0111, 0110, 0101)
(1001)	(0110 1000)	(1001, 1111, 1110, 0001)
(1010)	(0001 1111)	(1000, 1110, 1011, 1101)
(1011)	(0011 0010)	(0110, 0110, 0011, 1001)
(1100)	(1111 0001)	(1010, 1100, 0001, 1010)
(1101)	(1101 1100)	(0100, 0100, 1001, 1110)
(1110)	(1010 1011)	(0101, 0101, 1100, 0010)
(1111)	(1000 0110)	(1011, 1101, 0100, 0110)

We now illustrate the steps of the PRA-based finite field multiplication algorithm for computing $c(x) = a(x)b(x) \bmod n(x)$, where $a, b \in GF(2^8)$ are given as

$$\begin{array}{rcccccc} a(x) & = & x^7 + x^6 + x^4 + x^3 + x + 1 & = & (1101\ 1011) & = & (A_1 A_0)\ , \\ b(x) & = & x^7 + x^5 + x^3 + x^2 + x & = & (1010\ 1110) & = & (B_1 B_0)\ . \end{array}$$

The PRS representations of a and b are found as

$$\begin{array}{rcl} \vec{a} & = & (1111, 1010, 1001, 0010)\ , \\ \vec{b} & = & (0011, 0010, 1000, 1011)\ . \end{array}$$

The PRA-based multiplication algorithm starts with $c = 0$ and performs the following steps to find the result $c(x) = x^6 + x^5 + x^4 + x^3 + 1 = (0111\ 1001)$ as follows:

$$\begin{array}{lllcl} & \text{Step 1:} & \vec{c} & = & \vec{a} * \vec{b} = (1111, 1010, 1001, 0010) * (0011, 0010, 1000, 1011) \\ & & & = & (0010, 1101, 0110, 1001) \\ & \text{Step 2:} & \vec{r} & = & \vec{c} * \vec{I} = (0010, 1101, 0110, 1001) * (0011, 1011, 0111, 1100) \\ & & & = & (0110, 0010, 1111, 1111) \\ & \text{Step 3:} & c(x) & = & \sum_{i=1}^{4} r_i \cdot M_i \\ & & & = & (0110) \cdot (0001\ 1100\ 1000\ 0011) + (0010) \cdot (0001\ 0101\ 0000\ 1101) + \\ & & & & (1111) \cdot (0001\ 0010\ 0100\ 1001) + (1111) \cdot (0001\ 0001\ 0011\ 1111) \\ & & & = & (0111\ 0010\ 1110\ 0010) = (C_3 C_2 C_1 C_0) \\ (i = 3) & \text{Step 5:} & \vec{c} & = & \vec{c} + \vec{T}[C_3] * x^4 = \vec{c} + \vec{T}[0111] * x^4 \\ & & & = & (0010, 1101, 0110, 1001) + (1100, 1010, 0010, 0011) * x^4 \\ & & & = & (0101, 0001, 0001, 0111) \\ & \text{Step 6:} & c(x) & = & c(x) + T[C_3] \cdot x^4 = (0010\ 1110\ 0010) + T[0111] \cdot x^4 \\ & & & = & (1110\ 1101\ 0010) = (C_2 C_1 C_0) \\ (i = 2) & \text{Step 5:} & \vec{c} & = & \vec{c} + \vec{T}[C_2] * x^0 = \vec{c} + \vec{T}[1110] \\ & & & = & (0101, 0001, 0001, 0111) + (0101, 0101, 1100, 0010) \\ & & & = & (0000, 0100, 1101, 0101) \\ & \text{Step 6:} & c(x) & = & c(x) + T[C_2] \cdot x^0 = (1101\ 0010) + T[1110] \\ & & & = & (1101\ 0010) + (1010\ 1011)\ =\ (0111\ 1001) = (C_1 C_0) \end{array}$$

The last vector $\vec{c}$ in Step 5 or the last polynomial $c(x)$ in Step 6 yields the result. The result $\vec{c}$ in Step 5 is reduced modulo $n(x)$, and thus, it can be used as an input to another multiplication.

8 Improving the PRA-based Multiplication Algorithm

In this section, we give an improved PRA-based multiplication algorithm, which saves computational time and space as compared to the algorithm given in §6 and exemplified in §7. The improved algorithm is based on the following observations:

- We compute the polynomial representation of $c(x)$ along with its residue representation $\vec{c}$ only because we need the words of $c(x)$ to reduce $\vec{c}$ modulo $n(x)$.
- We only use the most significant s words of $c(x)$ starting from $2s - 1$ down to s to perform this reduction, as seen in Step 4.

Therefore, the SRC algorithm for computing $c(x)$ can be modified so that we only compute the most significant s words of $c(x)$, and thus, save space and time in the PRA-based multiplication algorithm. In Step 3 of the PRA-based multiplication algorithm, we compute $c(x) = \sum_{i=1}^{L} r_i \cdot M_i$

and then perform a modulo $M(x)$ reduction. Since $L = 2s$, and each one of M_i is of length (at most) $2s-1$ words, the above computation can be written as

$$c(x) = \sum_{i=1}^{L} r_i \cdot (M_{i,2s-2}M_{i,2s-3}\cdots M_{i,1}M_{i,0}) \ . \tag{13}$$

Since we are only interested in computing the most significant s words of $c(x)$, the above sum must be divided by x^{sw}. To avoid unnecessary computation, we first divide M_i by $x^{(s-1)w}$, then perform the multiplications and the summation, and finally divide the result by x^w. This way we first discard the part of M_i which does not contribute to the final result, and then, we perform another division to get only the necessary part of $c(x)$. More explicitly, we truncate M_i by ignoring the words indexed from 0 up to $s-2$, and only keep the ones from $s-1$ up to $2s-2$. This implies that we compute

$$\sum_{i=1}^{L} r_i \cdot (M_{i,2s-2}M_{i,2s-3}\cdots M_{i,s}M_{i,s-1}00\cdots 0) \ . \tag{14}$$

The least significant $s-1$ words of zeros can be ignored by computing

$$\sum_{i=1}^{L} r_i \cdot (M_{i,2s-2}M_{i,2s-3}\cdots M_{i,s}M_{i,s-1}) = \sum_{i=1}^{L} r_i \cdot \frac{M_i}{x^{(s-1)w}} \ . \tag{15}$$

The multiplication operation $r_i \cdot (M_{i,2s-2}M_{i,2s-3}\cdots M_{i,s}M_{i,s-1})$ produces an $(s+1)$-word number; the sum of L such numbers is still of length $s+1$ words. Since we only need the most significant s words, we ignore the least significant word of the result as

$$c'(x) = \left(\sum_{i=1}^{L} r_i \cdot \frac{M_i}{x^{(s-1)w}}\right) \cdot \frac{1}{x^w} \ = \ (C'_{s-1}C'_{s-2}\cdots C'_1C_0) \ . \tag{16}$$

The values $M'_i(x) = M_i(x)/x^{(s-1)w}$ can be precomputed. The most significant s words of $c(x)$ is computed using the modified SRC algorithm as follows:

The Modified SRC Algorithm

Input:	$\vec{p} = (p_1, p_2, \ldots, p_L)$
Output:	$p'(x)$: The most significant s words of $p(x)$
Auxiliary:	$M'_1, M'_2, \ldots, M'_L$ and $\vec{I}$
Step 1.	$\vec{r} := \vec{p} * \vec{I}$
Step 2.	$p'(x) := \left(\sum_{i=1}^{L} r_i \cdot M'_i\right) \cdot x^{-w}$
Step 3.	return $p'(x)$

The improved PRA-based multiplication algorithm uses the modified SRC algorithm given above to compute $c'(x)$ in order to reduce $\vec{c}$.

The Improved PRA-based Multiplication Algorithm

Input:	$\vec{a}$ and $\vec{b}$
Output:	$\vec{c}$
Auxiliary:	$M'_1, M'_2, \ldots, M'_L$, $\vec{I}$, T, and $\vec{T}$
Step 1.	$\vec{c} := \vec{a} * \vec{b}$
Step 2.	$\vec{r} := \vec{c} * \vec{I}$
Step 3.	$c'(x) := \left(\sum_{i=1}^{L} r_i \cdot M'_i\right) \cdot x^{-w}$

Step 4. for $i = s-1$ downto 0
Step 5. $\vec{c} := \vec{c} + \vec{T}[C'_i] * x^{iw}$
Step 6. $c'(x) := (\ c'(x) \bmod x^{iw}\ ) + T[C'_i] \cdot x^{-(s-i)w}$
Step 7. return $\vec{c}$

We have only the most significant s words of $c(x)$. The entire $c(x)$ can be written as

$$c(x) = (C'_{s-1}C'_{s-2} \cdot C'_1C'_0C_{s-1}C_{s-2} \cdot C_1C_0)\ .$$

In order to reduce $c(x)$ in step for $i = s-1$, we need to take the most significant word C'_{s-1} and obtain the table entry $T[C'_{s-1}] = (V_{s-1}V_{s-2} \cdots V_1V_0)$, and add it to $c(x)$ by ignoring the addition of the most significant word C'_{s-1}, as follows:

$$\begin{array}{cccccccccc} & C'_{s-2} & \cdots & C'_1 & C'_0 & C_{s-1} & C_{s-2} & \cdots & C_1 & C_0 \\ + & V_{s-1} & \cdots & V_2 & V_1 & V_0 & & & & \\ \hline \end{array}$$

In general, in the ith step, we need to take the least significant i words of $c'(x)$ and add $(s-i)$ words right-shifted version of $T[C'_i]$ to $c'(x)$ in Step 6. Similarly, we perform the reduction on $\vec{c}$ in Step 5. The most significant words of $c(x)$ are completely zeroed during the reduction process in Step 6. We do not provide $c'(x)$ as an output, and the improved PRA-based multiplication algorithm returns $\vec{c}$ only.

9 An Example for the Improved Algorithm

We illustrate the steps of the improved PRA-based multiplication algorithm using the same example as the one in §7. The precomputation part and the tables remain the same. Additionally, we need to compute $M'_i(x)$ for $i = 1, 2, \ldots, L$, which are obtained using $M'_i = M_i/x^{(s-1)w} = M_i/x^4$ as

$$\begin{aligned} M'_1 &= (0001\ 1100\ 1000\ 0011)/x^4 = (0001\ 1100\ 1000)\ , \\ M'_2 &= (0001\ 0101\ 0000\ 1101)/x^4 = (0001\ 0101\ 0000)\ , \\ M'_3 &= (0001\ 0010\ 0100\ 1001)/x^4 = (0001\ 0010\ 0100)\ , \\ M'_4 &= (0001\ 0001\ 0011\ 1111)/x^4 = (0001\ 0001\ 0011)\ , \end{aligned}$$

The algorithm takes the same $\vec{a}$ and $\vec{b}$ as input operands, and performs the following steps to find the result $\vec{c} = (0000, 0100, 1101, 0101)$.

$$\begin{array}{lllll} & \text{Step 1:} & \vec{c} & = & \vec{a} * \vec{b} = (1111, 1010, 1001, 0010) * (0011, 0010, 1000, 1011) \\ & & & = & (0010, 1101, 0110, 1001) \\ & \text{Step 2:} & \vec{r} & = & \vec{c} * \vec{I} = (0010, 1101, 0110, 1001) * (0011, 1011, 0111, 1100) \\ & & & = & (0110, 0010, 1111, 1111) \\ & \text{Step 3:} & c'(x) & = & (\sum_{i=1}^{4} r_i \cdot M'_i) \cdot x^{-4} \\ & & & = & [(0110) \cdot (0001\ 1100\ 1000) + (0010) \cdot (0001\ 0101\ 0000) + \\ & & & & \ (1111) \cdot (0001\ 0010\ 0100) + (1111) \cdot (0001\ 0001\ 0011)] \cdot x^{-4} \\ & & & = & (0111\ 0010) \\ (i=1) & \text{Step 5:} & \vec{c} & = & \vec{c} + \vec{T}[C'_1] * x^4 = \vec{c} + \vec{T}[0111] * x^4 \\ & & & = & (0010, 1101, 0110, 1001) + (1100, 1010, 0010, 0011) * x^4 \\ & & & = & (0101, 0001, 0001, 0111) \\ & \text{Step 6:} & c'(x) & = & (\ c'(x) \bmod x^4\ ) + T[C'_1] \cdot x^{-4} = (0010) + T[0111] \cdot x^{-4} \\ & & & = & (0010) + (1100) = (1110) \end{array}$$

$$
\begin{array}{llrl}
(i=0) & \text{Step 5:} & \vec{c} & = \vec{c} + \vec{T}[C_0] * x^0 = \vec{c} + \vec{T}[1110] \\
 & & & = (0101, 0001, 0001, 0111) + (0101, 0101, 1100, 0010) \\
 & & & = (0000, 0100, 1101, 0101) \\
 & \text{Step 6:} & c'(x) & = (\ c'(x) \bmod x^0\ ) + T[C'_0] \cdot x^0 = (1110) + T[1110] \ = \ (0000)
\end{array}
$$

Since $c'(x)$ after Step 6 for $i = 0$ is not needed, this computation may be skipped.

10 Analysis of the Algorithms

In this section, we analyze the table lookup based reduction and multiplication algorithms, and then calculate the time and processor requirements of the improved PRA-based multiplication algorithm. We calculate the size of the lookup tables and count the total number of the table read, the word-level $GF(2)$ addition, and the word-level $GF(2)$ multiplication operations.

First we start the word-level $GF(2)$ operations. The word-level addition is simply the bit-wise `XOR` operation on a pair of 1-word binary numbers, which is a readily available instruction on most general purpose microprocessors and signal processors. On the other hand, the word-level multiplication operation receives two 1-word (w-bit) polynomials $A(x)$ and $B(x)$ defined over $GF(2)$, and computes the 2-word polynomial $C(x) = A(x)B(x)$. The degree of the product polynomial $C(x)$ is $2(w-1)$. For example, given $A = (1101)$ and $B = (1010)$, this operation computes C as

$$
A(x)B(x) \ = \ (x^3 + x^2 + 1)(x^3 + x) \ = \ x^6 + x^5 + x^4 + x \ = \ (0111\ 0010)\ .
$$

The implementation of this operation, which we call `MULGF2` as in [4], can be performed in three different ways. From the fastest to the slowest, these methods are:

- An instruction implemented on the processor.
- The table lookup method.
- The emulation using the `XOR` and `SHIFT` operations.

The details of the analysis of these methods can be found in [4]. In this paper, we will simply count the number of `MULGF2` operations, and assume that they are implemented using any of the above methods. A simple method for implementing the table lookup approach is to use two tables, one for computing the higher (`H`) and the other for computing the lower (`L`) bits of the product. We store the values `H` and `L` in two table reads. The tables are addressed using the bits of the operands, and thus, the total size of the tables `H` and `L` would be of size $2 \times 2^w \times 2^w \times w$ bits. For $w = 4$, $w = 8$, and $w = 10$, these amount to 256 bytes, 131,072 bytes, and 2,621,440 bytes, respectively. The size grows quite excessively, and thus, limiting the implementation for $w \geq 16$. Other approaches are also possible, for example, a hybrid approach was suggested in [4].

The other operation to consider is the table read operation from T. We will denote this operation using `TREAD`, and count the total number of `TREAD` operations. In regard to the size of the table, we note that the table T has 2^w rows, each of which contains a polynomial of length k. This implies that the size of the tables is $2^w \times k$ bits. The space requirements for the tables T (or T_2) for performing the `TREAD` operation are exemplified in Figure 3.

Figure 3: The size of the table T in bytes

k	$w = 4$	$w = 8$	$w = 10$	$w = 16$
160	320	5,120	20,480	1,310,720
256	512	8,192	32,768	2,097,152
512	1,024	16,384	65,536	4,194,304
1024	2,048	32,768	131,072	8,388,608

For example, if $w = 8$ and $k = 160$, the size of the table is $2^8 \times 20 = 5,120$ bytes, which is quite reasonable. However, the table size becomes excessive as we increase the wordsize. For a fixed field size k, we can decide about the wordsize w given the memory capacity of the computer system.

Now, we give the steps of the table lookup based multiplication algorithm in detail in Figure 4, together with the number of TREAD, MULGF2, and XOR operations.

Figure 4: The operation counts for the table lookup based multiplication algorithm.

	TREAD	MULGF2	XOR
for i=0 to s do	-	-	-
C[i]:=0	-	-	-
for i=s-1 downto 0 do	-	-	-
P:=0	-	-	-
for j=s-1 downto 0 do	-	-	-
(H,L):=MULGF2(A[i],B[j])	-	s^2	-
C[j+1]:=C[j] XOR H XOR P	-	-	$2s^2$
P:=L	-	-	-
C[0]:=P	-	-	-
for j=0 to s-1 do	-	-	-
C[j]:=C[j] XOR T[C[s]][j]	s	-	s^2

The total number of TREAD, MULGF2, and XOR operations for the multiplication algorithm is found as s, s^2, and $3s^2$, respectively. In Figure 5, we compare the two table lookup based multiplication method to the algorithm in [4], which does not require table lookup. The operation counts indicate that the table lookup based algorithms are computationally more efficient since we trade off computational time for space for the tables.

Figure 5: The operation counts for the algorithms.

	Method in [4]	Table Lookup Method
TREAD	0	s
MULGF2	s^2	s^2
XOR	$3s^2(w/2+1) + sw/2$	$3s^2$
SHIFT	$2s^2(w+1) + s(w+1)$	0

We now give a detailed analysis of the improved PRA-based multiplication algorithm. Following the assumption made in §2, we select $\deg(m_i) = w$, and perform modulo $m_i(x)$ multiplications using the table lookup method. Since there are L different moduli, we assume that we have $L = 2s$ processors each of which performs its arithmetic (addition and multiplication) operations with respect to its selected modulus. We also use a processor which we call the 'server' to perform a few table lookup operations. The server can be one of the processors. We ignore the server operations and the communication overhead in our analysis.

The analysis of the improved PRA-based multiplication algorithm is given below. The results are summarized in Figure 6.

Step 1: This step requires a single MULGF2 operation by each of $2s$ processors.

Step 2: This step requires a single MULGF2 operation by each of $2s$ processors.

Step 3: Let $M'_i = (M'_{i,s-1} M'_{i,s-2} \cdots M'_{i,0})$. This value is already precomputed and saved. During Step 3, the ith processor multiplies r_i by M'_i using MULGF2 operation and obtains the $(s+1)$-word result. Each word multiplication for $j = 0, 1, \ldots, s-1$ produces a 2-word result $r_j \cdot M'_{i,j} =$

(H_jL_j). These parts must then be added to obtain the final result $r_i \cdot M'_i$ as follows:

		$M'_{i,s-1}$	$M'_{i,s-2}$	$\cdots$	$M'_{i,2}$	$M'_{i,1}$	$M'_{i,0}$
$\times$							r_i
		L_{s-1}	L_{s-2}		L_2	L_1	L_0
$+$	H_{s-1}	H_{s-2}	H_{s-3}		H_1	H_0	
	S_s	S_{s-1}	S_{s-2}	$\cdots$	S_2	S_1	S_0

The above operation is performed using s `MULGF2` operations and $s-1$ `XOR` operations by the ith processor for all processors $i = 1, 2, \ldots, 2s$.

Then the 0th word is discarded and the remaining s-word polynomials are summed by all $2s$ processors using the binary tree algorithm. This operation takes $\log_2(2s)$ steps, where at each step two s-word polynomials are added. Therefore, $s\log_2(2s)$ `XOR` operations are required.

The resulting s-word polynomial $c'(x)$ is communicated to the server and to the first s processors among all $2s$ processors. This polynomial is needed in Step 6.

Step 5: The values $x^{iw} \pmod{m_j}$ are precomputed and stored for all $i = 1, 2, \ldots, s-1$ and $j = 1, 2, \ldots, 2s$. The server performs a lookup operation for $x^{iw} \pmod{m_j}$ and also for $\vec{T}[C'_i]$, and sends them to the jth processor for all $j = 1, 2, \ldots, 2s$. Each processor then performs a single `MULGF2` operation and a single `XOR` operation to obtain $\vec{c} := \vec{c} + \vec{T}[C'_i]$ for a single vector entry.

Step 6: In this step, we add the i-word shifted polynomial $T[C'_i]$ to $c'(x)$. Since $c'(x)$ is available in the first s processors, each of one of these processors performs a single `XOR` operation. The updated polynomial $c'(x)$ is then communicated to the server.

Figure 6: Operation counts for the PRA-based multiplication.

Steps	`MULGF2`	`XOR`
Step 1	1	
Step 2	1	
Step 3	s	$s - 1 + s\log_2(2s)$
Step 5 (s times)	1	1
Step 6 (s times)		1
Total	$2s + 2$	$3s - 1 + s\log_2(2s)$

11 Applications of the PRA-based Multiplication

The improved algorithm takes two polynomials in their residue representation $\vec{a}$ and $\vec{b}$ such that the degree of each of $a(x)$ and $b(x)$ is less than k, and produces the product polynomial in its residue representation $\vec{c}$. The squaring algorithm can be given using the similar construction method, however, there may be certain optimizations.

The PRA-based multiplication algorithm for the field $GF(2^k)$ finds its applications in cryptography where the range of the operands is large, usually $160 \leq k \leq 1024$, therefore, it is justifiable to use parallel polynomial arithmetic. We give an exponentiation algorithm for computing g^e where $g \in GF(2^k)$ and e is an r-bit integer $e = (e_{r-1}e_{r-2}\cdots e_1e_0)$ below.

The PRA-based Exponentiation Algorithm
Input: $g(x)$, e, and $n(x)$
Output: $c(x) = g^e$

Auxiliary:	$M'_1, M'_2, \ldots, M'_L$, $\vec{I}$, T, and $\vec{T}$
Step 1.	Compute $\vec{g}$ and $\vec{c} := (1, 1, \ldots, 1)$
Step 2.	for $i = r - 1$ downto 0
Step 3.	$\vec{c} :=$ Multiply$(\vec{c}, \vec{c})$
Step 4.	if $e_i = 1$ then $\vec{c} :=$ Multiply$(\vec{g}, \vec{c})$
Step 5.	$c(x) :=$ SRC$(\vec{c})$
Step 6.	return $c(x)$

Here the multiplication algorithm is the improved PRA-based multiplication method given in §8, which uses the modified SRC algorithm within. Since we need the entire $c(x)$ as the output of the exponentiation operation, the original SRC algorithm is used in Step 6.

12 Conclusions

The proposed parallel algorithm requires $O(s \log s)$ arithmetic operations using $2s$ processors. Additionally there are some table lookup operations performed by the server, and there is also the communication overhead which we ignored in our analysis. The proposed parallel algorithm is not suitable for implementation on a general purpose parallel computer, however, it is highly suitable for hardware implementation. For example, by selecting $w = 8$ and $L = 40$, we can implement $GF(2^{160})$ arithmetic, which is desired several applications of elliptic curve cryptography [6]. Furthermore, by selecting special irreducible polynomials, for example, trinomials or all-one-polynomials, we may not have a need for constructing the tables T and $\vec{T}$ and the reduction operation can be simplified. We are currently working on such improvements and hardware implementation of the proposed PRA-based multiplication algorithm.

References

[1] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[2] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Reading, MA: Addison-Wesley, Third edition, 1998.

[3] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[4] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Design, Codes and Cryptography*, 14(1):57–69, April 1998.

[5] J. H. McClellan and C. M. Rader. *Number Theory in Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[6] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Boston, MA: Kluwer Academic Publishers, 1993.

[7] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. New York, NY: Springer-Verlag, 1985.

[8] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. U.S. Patent Number 4,587,627, May 1986.