Alternative to the Karatsuba Algorithm for Software Implementation of $GF(2^n)$ Multiplication

Haining Fan and M. Anwar Hasan

Abstract

In [11], a new approach to subquadratic space complexity multiplication for extended finite fields has recently been proposed for hardware implementation. In this article, we develop the corresponding algorithm for software implementation. Compared to the Karatsuba algorithm, the proposed algorithm has a lower theoretical time complexity when the size of the input is greater than a fixed integer. While its recursive implementation is as simple as that of the Karatsuba algorithm, it requires less memory to store the look-up table than the latter, e.g., 512 bytes vs. 128 kilobytes in our implementation. To the best of our knowledge, this is the first better alternative to the Karatsuba algorithm for software implementation dealing with "intermediate" sized finite fields.

May 18, 2006

Index Terms

Finite field, subquadratic time complexity multiplication algorithm, coordinate transformation, shifted polynomial basis, Toeplitz matrix.

I. Introduction

Ring $\mathbb{Z}/(m\mathbb{Z})$ and finite field $GF(2^n) = GF(2)[u]/(f(u))$ are two commonly used algebraic structures in cryptosystems, where m is a positive integer and $f(u)$ a degree n irreducible polynomial over $GF(2)$. Arithmetic operations in $\mathbb{Z}/(m\mathbb{Z})$ are addition and multiplication modulo

Haining Fan and M. Anwar Hasan are with the Department of Electrical and Computer Engineering University of Waterloo, Waterloo, Canada. emails: hfan@vlsi.uwaterloo.ca and ahasan@ece.uwaterloo.ca

m. When elements of $GF(2^n)$ are represented using a polynomial basis, they are polynomials of degree less than n and coefficients in $GF(2)$. Similarly, addition and multiplication operations in $GF(2^n)$ are defined as addition and multiplication of two such polynomials modulo $f(u)$. Since no carry occurs in $GF(2)$ addition, $GF(2^n)$ addition is straightforward.

For the software implementation, the polynomial basis $GF(2^n)$ multiplication operation may be performed by the following two steps: the conventional polynomial multiplication followed by the reduction modulo the field generating irreducible polynomial. The second step becomes simple if some special irreducible polynomials are chosen to generate $GF(2^n)$, e.g., irreducible trinomials or pentanomials. In fact, there is no known value of n for which an irreducible polynomial of degree n and weight $w < 6$ does not exist over $GF(2)$[1].

The elementary method to perform the first step, i.e., the conventional polynomial multiplication, is the school method which has an asymptotic complexity $O(n^2)$. Recursive application of the divide-and-conquer based Karatsuba algorithm leads to a running time of $O(n^{\log_2 3}) \approx O(n^{1.585})$ for $n = 2^i$ $(i > 0)$ [2]. The k-way ($k > 2$) generalization of the Karatsuba algorithm has a better asymptotic complexity [3]. For practical implementation, the crossover point between the school and the Karatsuba methods is reported to be $n = 576$ in [4]. Other more sophisticated algorithms, e.g., the FFT and Cantor's multiplication, have better asymptotic complexity and are suitable for large values of n [5]. The crossover point between these methods and the Karatsuba method is relatively large, e.g., the crossover point between the Karatsuba and the Cantor algorithms are 35840 and 16000 in [4] and [6], respectively. For software implementation of these algorithms, "a typical experience is that the school method is best for small inputs, Karatsuba's algorithm takes over for intermediate sizes, and a fast, for example, an FFT-based method, excels for large problems" [7].

In this article, we focus on the software implementation of the $GF(2^n)$ multiplication algorithm for cryptographic applications. In some of these applications, the value of n is often a few hundreds. For example, NIST has recommended the following five binary fields for the ECDSA (Elliptic Curve Digital Signature Algorithm) applications: $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$. According to the above suggestion, it seems that the Karatsuba algorithm is the best choice for intermediate sizes of n. In the following, we will combine two existing techniques: optimal Toeplitz matrix-vector product formulae [8] and the coordinate transformation technique of [9] and [10], and propose an alternative scheme for software implementation of the subquadratic $GF(2^n)$ multiplication. Theoretical analysis indicates that it is faster than the Karatsuba algorithm.

In order to speedup the implementation of the Karatsuba algorithm, in practice the multiplication of polynomials of degree less than 8 are often performed by the table-lookup technique [4]. The corresponding table uses 128 kilobytes. In our implementation of the proposed algorithm, the table-lookup technique is also adopted. Although the size of tables that we employ is relatively small, i.e., 512 bytes, experimental results show that the algorithm is still faster than the Karatsuba algorithm for $n = 2^i$, where $(6 < i < 18)$. Another advantage of the proposed algorithm is its simplicity, i.e., its recursive implementation is as simple as the Karatsuba algorithm. Therefore, the proposed method is a better alternative to the Karatsuba algorithm.

The remainder of this article is organized as follows: The algorithm is proposed in Section II, and the software implementation for irreducible trinomials is presented in Section III. Finally, concluding remarks are made in Section IV.

II. Alternative Subquadratic Multiplication Algorithm for Shifted Polynomial Basis

In this section, first we briefly review the subquadratic space complexity scheme of [11] and then present explicit formulae for general irreducible trinomials.

A. *Optimal Toeplitz Matrix-Vector Product Formula for* $n = 2^i$ $(i > 0)$

In this subsection, the noncommutative matrix-vector multiplication scheme for $n = 2^i$ $(i > 0)$ and its asymptotic time complexities are presented. Let $T = (m_{k,i})$ be an $n \times n$ Toeplitz matrix over $GF(2)$, where $0 \leq i, k \leq n-1$, i.e., $m_{k,i} = m_{k-1,i-1}$, where $1 \leq i, k \leq n-1$. It is clear that T is determined by the $2n-1$ elements of the first row and the first column. Let V be an $n \times 1$ column vector. Matrix T and vector V can be split as follows:

$$T = \begin{pmatrix} T_1 & T_0 \\ T_2 & T_1 \end{pmatrix} \text{ and } V = \begin{pmatrix} V_0 \\ V_1 \end{pmatrix},$$

where T_0, T_1 and T_2 are $(n/2) \times (n/2)$ matrices and are in the Toeplitz form, and V_0 and V_1 are $(n/2) \times 1$ column vectors. The following noncommutative formula compute the Toeplitz matrix-vector product TV [8]:

$$TV = \begin{pmatrix} T_1 & T_0 \\ T_2 & T_1 \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \end{pmatrix} = \begin{pmatrix} P_0 + P_2 \\ P_1 + P_2 \end{pmatrix} \tag{1}$$

where $P_0 = (T_0 + T_1)V_1, P_1 = (T_1 + T_2)V_0$ and $P_2 = T_1(V_0 + V_1)$. Please note that in (1) the product of an $n \times n$ Toeplitz matrix and an $n \times 1$ vector is primarily reduced to $three$ products of matrix and vector of sizes $(n/2) \times (n/2)$ and $(n/2) \times 1$.

Let $T_{mvp}(n)$ denote the time complexity for the computation of matrix-vector product TV. The following recurrence relations can be established when (1) is used recursively to compute TV.

$$\begin{cases} T_{mvp}(1) = e_{mvp}, \\ T_{mvp}(n) = 3T_{mvp}(n/2) + c_{mvp}n + d_{mvp}. \end{cases}$$

For the purpose of comparison, we also present the recurrence relation that describes the time complexity of the Karatsuba algorithm as follows.

$$\begin{cases} T_{kara}(1) = e_{kara}, \\ T_{kara}(n) = 3T_{kara}(n/2) + c_{kara}n + d_{kara}. \end{cases}$$

After solving these recurrence relations using Lemma 1 of [11], we have

$$T(n) = (2c + e + \frac{d}{2})n^{\log_2 3} - 2cn - \frac{d}{2}, \tag{2}$$

where subscripts (i.e., mvp and $kara$) are omitted.

Therefore, recursive application of (1) and the Karatsuba algorithm have the same asymptotic complexity, i.e., $O(n^{\log_2 3})$. In the next section we will estimate values of e, c and d, and show that coefficient $(2c + e + \frac{d}{2})$ of the proposed algorithm is smaller than that of the Karatsuba algorithm if formula (1) and the Karatsuba method are applied until n reaches 1.

Similar to the generalization of the Karatsuba method from $n = 2^i$ $(i > 0)$ to $n = p^i$ $(i > 0)$, where p is a small odd prime [3], we may also obtain matrix-vector product formulae for other small primes. The case of $n = 3^i$ $(i > 0)$ has been presented in [11]. In situations where none of these small primes divides n, one possible solution is to increase the size of the matrix and vector by 1 by padding zeroes, and then apply the existing complexity results to the case $n + 1$.

B. Formulation Using Shifted Polynomial Basis for General Irreducible Trinomials

Shifted polynomial basis (SPB) is a shifted version of the standard polynomial basis. $GF(2^n)$ multiplication operation in the SPB is similar to that of the standard polynomial basis. $GF(2^n)$ hardware multipliers based on the SPB often have lower time complexity than the corresponding polynomial basis multiplier. Let x be a root of the irreducible polynomial $f(u)$ and $GF(2^n) = GF(2)[u]/(f(u))$. An SPB of $GF(2^n)$ over $GF(2)$ is defined as follows [12]:

Definition 1: Let v be an integer and the ordered set $M = \{x^i | 0 \leq i \leq n-1\}$ be a polynomial basis of $GF(2^n)$ over $GF(2)$. The ordered set $x^{-v}M := \{x^{i-v} | 0 \leq i \leq n-1\}$ is called a shifted polynomial basis with respect to M.

From now on, we assume that $f(u) = u^n + u^v + 1$ $(1 \leq v \leq n - 1)$ is an irreducible trinomial. Let $X = (x^{-v}, x^{-v+1}, \cdots, x^{n-v-1})^T$ be the column vector of SPB basis elements,

$A = (a_0, a_1, \cdots, a_{n-1})^T$ be the coordinate column vector of the field element $a = x^{-v}\sum_{i=0}^{n-1} a_i x^i$, and B, C and D are defined similarly.

Besides the two-step multiplication method mentioned in the introduction section, the other method to compute the product $c = ab$ is to form a binary $n \times n$ matrix Z first, which depends on b and $f(u)$, and then perform a matrix-vector product. Namely, the product $c = ab$ may be computed as follows:

$$\begin{aligned} c &= \sum_{i=0}^{n-1} a_i x^{i-v} b = (x^{-v}b, \cdots, x^{-1}b, b, xb, \cdots, x^{n-v-1}b)(a_0, a_1, \cdots, a_{n-1})^T \\ &= X^T (Z_0, \cdots, Z_{n-1}) A \\ &= X^T Z A, \end{aligned} \tag{3}$$

where Z_i is the coordinate column vector of $x^{i-v}b$ with respect to the SPB ($0 \le i \le n+1$), and Z is an $n \times n$ matrix.

This matrix-vector product method has been widely used to design $GF(2^n)$ multipliers using VLSI technologies. However, Z is not generally a Toeplitz matrix. Using coordinate transformation techniques presented in [9] and [10], one may first transform Z into a Toeplitz matrix T, i.e., $T = UZ$, where U is the transform matrix. Then compute the Toeplitz matrix-vector product $D = TA$. Finally, the result C is obtained by $C = U^{-1}D$. In [11], this idea has been used and the following simple transformation matrix U has been obtained:

$$U = \begin{pmatrix} 0 & I_{(n-v)\times(n-v)} \\ I_{v\times v} & 0 \end{pmatrix},$$

where $I_{v\times v}$ is the $v \times v$ identity matrix.

Since an $n \times n$ Toeplitz matrix is determined by the $2n-1$ elements of the first and last columns, we now determine the explicit formulae of these elements in T. Due to the property of the transformation matrix U, it is clear that T is obtained from Z by moving the lower $n-v$ rows to the upper $n-v$ rows. Therefore, we first determine the first and the last columns of Z.

From (3), we know that these two columns correspond to coordinate vectors of element $x^{-v}b$ and $x^{n-v-1}b$, respectively, which are given as follows:

$$\begin{aligned}
x^{n-v-1}b &= \sum_{i=0}^{n-1} b_i x^{i+n-2v-1} = \sum_{i=0}^{v} b_i x^{i+n-2v-1} + \sum_{i=v+1}^{n-1} b_i x^{i+n-2v-1} \\
&= \sum_{i=n-2v-1}^{n-v-1} b_{2v+1-n+i} x^i + \sum_{i=v+1}^{n-1} b_i (x^{i-v-1} + x^{i-2v-1}) \\
&= \sum_{i=n-2v-1}^{n-v-1} b_{2v+1-n+i} x^i + \sum_{i=0}^{n-v-2} b_{v+1+i} x^i + \sum_{i=-v}^{n-2v-2} b_{2v+1+i} x^i \\
&= \sum_{i=-v}^{-1} b_{2v+1+i} x^i + \sum_{i=0}^{n-2v-2} (b_{v+1+i} + b_{2v+1+i}) x^i \\
&\quad + \sum_{i=n-2v-1}^{n-v-2} (b_{2v+1-n+i} + b_{v+1+i}) x^i + b_v x^{n-v-1},
\end{aligned}$$

and

$$\begin{aligned}
x^{-v}b &= \sum_{i=0}^{n-1} b_i x^{i-2v} = \sum_{i=0}^{v-1} b_i x^{i-2v} + \sum_{i=v}^{n-1} b_i x^{i-2v} \\
&= \sum_{i=0}^{v-1} b_i (x^{n+i-2v} + x^{i-v}) + \sum_{i=-v}^{n-2v-1} b_{2v+i} x^i \\
&= \sum_{i=n-2v}^{n-v-1} b_{2v-n+i} x^i + \sum_{i=-v}^{-1} b_{v+i} x^i + \sum_{i=-v}^{n-2v-1} b_{2v+i} x^i \\
&= \sum_{i=-v}^{-1} (b_{v+i} + b_{2v+i}) x^i + \sum_{i=0}^{n-2v-1} b_{2v+i} x^i + \sum_{i=n-2v}^{n-v-1} b_{2v-n+i} x^i.
\end{aligned}$$

Let $\widehat{x^{n-v-1}b}$ and $\widehat{x^{-v}b}$ be the two elements of $GF(2^n)$ whose SPB coordinates form the first and last columns of T. Applying the transformation U to Z, we may obtain explicit formulae of elements $\widehat{x^{n-v-1}b}$ and $\widehat{x^{-v}b}$ as follows:

$$\begin{aligned}
\widehat{x^{n-v-1}b} &= \left[\sum_{i=-v}^{-1} b_{2v+1+i}x^i\right] x^{n-v} + \left[\sum_{i=0}^{n-2v-2} (b_{v+1+i} + b_{2v+1+i})x^i \right.\\
&\quad \left. + \sum_{i=n-2v-1}^{n-v-2} (b_{2v+1-n+i} + b_{v+1+i})x^i + b_v x^{n-v-1}\right] x^{-v}\\
&= \sum_{i=-v}^{n-3v-2} (b_{2v+1+i} + b_{3v+1+i})x^i + \sum_{i=n-3v-1}^{n-2v-2} (b_{3v+1-n+i} + b_{2v+1+i})x^i\\
&\quad + b_v x^{n-2v-1} + \sum_{i=n-2v}^{n-v-1} b_{3v+1-n+i}x^i\\
&= \sum_{i=-v}^{n-2v-2} b_{2v+1+i}x^i + \left[\sum_{i=-v}^{n-3v-2} b_{3v+1+i}x^i + \sum_{i=n-3v-1}^{n-v-1} b_{3v+1-n+i}x^i\right]\\
&= \sum_{i=-v}^{n-2v-2} b_{2v+1+i}x^i + \sum_{i=-v}^{n-v-1} b_{<3v+1+i>}x^i, \qquad (4)
\end{aligned}$$

and

$$\begin{aligned}
\widehat{x^{-v}b} &= \left[\sum_{i=-v}^{-1} (b_{v+i} + b_{2v+i})x^i\right] x^{n-v} + \left[\sum_{i=0}^{n-2v-1} b_{2v+i}x^i + \sum_{i=n-2v}^{n-v-1} b_{2v-n+i}x^i\right] x^{-v}\\
&= \sum_{i=-v}^{n-3v-1} b_{3v+i}x^i + \sum_{i=n-3v}^{n-2v-1} b_{3v-n+i}x^i + \sum_{i=n-2v}^{n-v-1} (b_{2v-n+i} + b_{3v-n+i})x^i\\
&= \left[\sum_{i=-v}^{n-3v-1} b_{3v+i}x^i + \sum_{i=n-3v}^{n-v-1} b_{3v-n+i}x^i\right] + \sum_{i=n-2v}^{n-v-1} b_{2v-n+i}x^i\\
&= \sum_{i=-v}^{n-v-1} b_{<3v+i>}x^i + \sum_{i=n-2v}^{n-v-1} b_{2v-n+i}x^i, \qquad (5)
\end{aligned}$$

where $< j >$ denotes j mod n.

Note that explicit formulae of matrix T and U for irreducible pentanomials $g(u) = u^n + u^{k+1} + u^k + u^{k-1} + 1$ $(1 < k < n-1)$ have been presented in [11].

III. Software implementation

A. Data Representation and Computation of Inner-product

Let z be the full width of the data-path of the general-purpose processor on which the software will run, e.g., $z = 32$ for a Pentium processor. For the simplicity of introducing the data representation, we assume that n is a multiple of z in this paper. Since no irreducible binary trinomial exists for the case $8|n$, we assume that the multiplication operation is performed in the ring $GF(2)[u]/(f(u))$, where $f(u)$ is a trinomial and $n = 2^i$ $(i > 2)$, so that the following discussion is meaningful.

In software implementation, the column vector $A = (a_0, a_1, \cdots, a_{n-1})^T$ is defined as a 1-dimensional unsigned integer array $V_A[0..(\frac{n}{z} - 1)]$, which is stored in $\frac{n}{z}$ successive computer's memory words, each having z consecutive coefficients (i.e., bits) are packed into one word. In this paper, we suppose that bit a_0 is stored in the least significant bit (LSB) of $V_A[0]$ and bit a_{n-1} the most significant bit (MSB) of $V_A[\frac{n}{z} - 1]$.

Since the detailed representation of T relies on the computation procedure of the matrix-vector product TA, we now introduce the latter. The usual computation procedure regards TA as an array of n inner-products to be computed one at a time in top-to-bottom order. In software implementation, the inner-products c_i $(0 \le i \le n-2)$ is calculated using two steps:

Step 1: Compute $GF(2)$ products $t_{i,j}a_j$ by processor's AND instruction, where $0 \le j \le n-1$.

Step 2: Compute the $GF(2)$ summation of the above n products using bit-wise XOR operations.

Due to the property of the Toeplitz matrix T, we know that it is sufficient to represent T by its first and last columns. Please note that elements in the first row are the same as those in the last column. Therefore, we may use a 1-dimensional array $V_T[0..(\frac{2n}{z} - 1)]$ of size $\frac{2n}{z}$ to store these two columns. Since bit a_{n-1} of vector A is stored in the MSB of $V_A[\frac{n}{z} - 1]$, we must also store bit $t_{0,n-1}$ in the MSB of $V_T[\frac{2n}{z} - 1]$ so that Step 1 may be performed by $\frac{n}{z}$ bitwise AND operations. Therefore, the i-th element $t_{0,i}$ in the first row of T will be stored in bit position (i mod z) of $V_T[\lfloor \frac{n+i}{z} \rfloor]$ $(0 \le i \le n-1)$.

Now we consider the representation of the first column of T. It is clear that element $t_{0,0}$ appears in both first row and first column of T. So we need only to store the rest $n-1$ bits, i.e., element $t_{i,0}$s ($1 \leq i \leq n-1$), in bit positions from $n-1$ downto 1 of V_T. The 0-th bit of V_T is unused. This results in the following bit organization of the 1-dimensional array $V_T[0..(\frac{2n}{z}-1)]$:

$*, t_{n-1,0}, t_{n-2,0}, \cdots, t_{2,0}, t_{1,0}, t_{0,0}, t_{0,1}, \cdots, t_{0,n-2}, t_{0,n-1}$, where * denotes the unused bit 0 in $V_T[0]$, which is the LSB of V_T.

Combining the results of (4) and (5), we have the following explicit formula of the i-th bit of V_T:

$$\begin{cases} * & i = 0, \\ b_{2v-i} + b_{v-i} & 1 \leq i \leq v, \\ b_{<2v-i>} & v+1 \leq i \leq n+v, \\ b_{<2v-i>} + b_{<v-i>} & n+v+1 \leq i \leq 2n-1. \end{cases}$$

This formula shows that the i-th bit of V_T is the summation of $b_{<2v-i>}$ and either 0 or $b_{<v-i>}$ for the case $1 \leq i \leq 2n-1$. But we know that the i-th bit of V_A is a_i, i.e., bit sequences $b_{<2v-i>}$ and a_i have different subscript orders: one is in ascending order and the other descending order. Since our goal is to compute the inner-product in the above Step 1, we should first reverse the bit order of the input vector B and then use this formula to form V_T. In our implementation, a small table is used to perform this bit-reverse operation. Its size is 256 bytes. The content of the i-th entry is the reserved eight bits of the binary representation of integer i ($0 \leq i \leq 255$).

Based on these representations, the following loop may be used to illustrate the computation of all inner-product c_is ($0 \leq i \leq n-1$).

TABLE I

COMPUTING INNER-PRODUCT c_iS ($0 \leq i \leq n-1$)

```
     for $i$=0 to $n-1$ {
S0       $n$-bit vector $tmp := V_T[\frac{n}{z}..(\frac{2n}{z}-1)]\ AND\ V_A[0..(\frac{n}{z}-1)]$;
S1       $c_i := GF(2)$ summation of all $n$ bits of $tmp$;
S2       Shift the $2n$-bit vector $V_T$ to the LSB once;}
```

Since the bit XOR operation ($GF(2)$ summation) is not directly supported in the high level programming language like C, we may perform step S1 via the table-lookup technique. In our implementation, we use a table of size 256 bytes. The content of the i-th entry is $(11111111)_2$ or $(00000000)_2$ if the binary representation of integer i contains an odd or even number of 1 bits ($0 \leq i \leq 255$), respectively. Please note that for the sake of software implementation on a typical processor, this table is allowed to have considerable amount of redundancy since only 256 bits are sufficient. Here, we also note that there is a parity bit in some processor's status register, e.g., Intel's x86 and MCS51 series. This bit is set if the least-significant byte of the result contains an even number of 1 bits, i.e., it is the $GF(2)$ summation of 1s in the least-significant 8 bits. Therefore, we may not only speed-up the computation but also save this small table if the proposed algorithm is coded using the assembly language of these processors.

Finally, we indicate an optimization of the above inner-product computation. Using the method in Table I, it requires 32 shift operations of the 64-bit vector V_T to compute the matrix-vector product of size $n = 32$. If the minimal memory access unit of most computer is one byte, we may compute c_0, c_8, c_{16} and c_{24} before performing the first shift operation, then c_1, c_9, c_{17} and c_{25}, and so on. In general, this optimization can save 24 shift operations.

B. Proposed Algorithm and Theoretical Analysis

We now present the proposed algorithm in the C programming language. The recursive subprogram "**void mvp**$(w32 *c,\ w32 *a,\ w32 *b,\ int\ s)$" is called by the $GF(2^n)$ multiplication subprogram "**void multiplication**$(w32\ *c,\ w32\ *a,\ w32\ *b)$", where $w32$ denotes the 32-bit unsigned long data type and "$*c$" the pointer to the product of a and b. Integer s denotes the number of the computer words that used to store the vector c. For the purpose of comparison, we also present the Karatsuba algorithm in Table IV.

In Tables III and IV, the number of basic operations are also counted. Here, we assume that the C language statement $tdw1[j] = a[j+s/2]\hat{\ }a[j+s]$ requires two additions "+" ($j + s/2$ and

TABLE II

MAIN PROGRAM

```
     **void multiplication**($w32\ *c,\ w32\ *a,\ w32\ *b$)
     {$w32\ V_T[2n/z],\ V_A[2n/z]$;
S0     for ($i$=0; $i < n/z$; $i$++) $c[i] = 0$;
S1     form the $(2n-1)$-bit vector $V_T$ and $n$-bit vector $V_A$;
S2     **mvp**($c,\ V_T,\ V_A,\ n/z$);
S3     coordinate transformation: Rotate shift vector $c$ to the MSB $k$ times. }
```

$j+s$), two memory reads “R” ($a[j+s/2]$ and $a[j+s]$), one XOR “ˆ” and one memory write “W” ($tdw1[j] =$) operation. Since the value $s/2$ appears a few times in the program, we assume that it is calculated once and do not count it again.

For a rough estimation of the time complexity, we assume that both recursive subprograms are called by themselves until parameter s reaches 1 bit. Since the inner-product operation of two 1-bit vectors is simply the multiplication of the vectors, both subprograms have the same values of e in (2). The value of d depends on the number of the recursive calls, loops and pointer arithmetic operations, etc. We may assume that these two simple recursive subprograms have the same values of d since they have similar program structures. The above two tables show that the value of c in the matrix-vector product subprogram is lower than that in the Karatsuba subprogram, $17.5s$ vs. $22s$. Therefore, coefficient $(2c+e+\frac{d}{2})$ in formula (2) is smaller for the matrix-vector product algorithm, and it is clear that the matrix-vector product recursive subprogram is theoretically faster than the Karatsuba subprogram. Now we consider the other operations. Since both the reduction operation after calling the Karatsuba recursive subprogram and pre-/post-processings of the matrix-vector product algorithm, i.e., formation of V_T and the coordinate transformation, have the same asymptotic complexity $O(n)$ when the field generating irreducible polynomial is either a trinomial or a pentanomial, we may conclude that the matrix-vector product algorithm is expected faster than the Karatsuba algorithm.

TABLE III

MATRIX-VECTOR PRODUCT BASED ALGORITHM IN THE C PROGRAMMING LANGUAGE

	Statements	ˆ	R	W	+
	void mvp($w32\ *c,\ w32\ *a,\ w32\ *b,\ int\ s$)				
	{$w32\ tdw1[s],\ tdw2[s/2]$;				
	if (s==4) {				
S0	perform basic computation;}				
	return;}				
	if (s%2 == 0) { /* if 2 divides s */				
	for (j=0; j<s; j++) $tdw1[j] = a[j+s/2]$ˆ$a[j+s]$;	s	$2s$	s	$2s$
	mvp(c, $tdw1$, $b+s/2$, $s/2$); /* compute P0 */				
	for (j=0; j<s; j++) $tdw1[j] = a[j]$ˆ$a[j+s/2]$;	s	$2s$	s	s
	mvp($c+s/2$, $tdw1$, b, $s/2$); /* compute P1 */				
	for (j=0; j<$s/2$; j++) {$tdw1[j] = b[j+s/2]$ˆ$b[j]$;	$s/2$	s	$s/2$	$s/2$
	$tdw2[j] = 0$;}			$s/2$	
	mvp($tdw2$, $a+s/2$, $tdw1$, $s/2$); /* compute P2 */				
	for (j=0; j<$s/2$; j++) {$c[j]\hat{} = tdw2$[j];	$s/2$	$s/2$	$s/2$	
	$c[j+s/2]\hat{} = tdw2[j]$;}	$s/2$	$s/2$	$s/2$	$s/2$
	}				
	}				
	Total=$17.5s$	$7s/2$	$6s$	$4s$	$4s$

For practical implementation of the Karatsuba algorithm on a 32-bit computer, actual computations, i.e., step S0 in Table IV is performed when the size of the operand reaches 1 computer word (32 bits). In [4], it is shown that the best way to perform the multiplication of polynomials of degree less than 32 is by applying the Karatsuba method twice, which yields 9 multiplications of 8-bit blocks, and then obtaining these 9 16-bit products via the table-lookup technique (the corresponding table uses 128 kilobytes). We also follow this approach in our implementation.

For the implementation of the matrix-vector product algorithm, actual computation step S0 in Table III is performed when the size of the operand reaches 4. The reason is that the minimal operation unit of the AND operation is 1 computer word, and we also want to apply the matrix-vector product formula (1) twice in step S0 so that both subprograms have the same number of

TABLE IV

THE KARATSUBA ALGORITHM IN THE C PROGRAMMING LANGUAGE

	Statements	ˆ	R	W	+
	void Kara$(w32\ *c,\ w32\ *a,\ w32\ *b,\ int\ s)$				
	$\{w32\ tdw1[s],\ tdw2[s/2];$				
	if (s==1) {				
S0	perform basic computation;}				
	return;}				
	if (s%2 == 0) { /* if 2 divides s */				
	for (j=0; $j<s$; j++) $tdw1[j] = 0$;			s	
	Kara$(tdw1,\ a,\ b,\ s/2)$;				
	for (j=0; j<s; j++){$c[j]\hat{} = tdw1[j]$;	s	s	s	
	$c[j+s/2]\hat{} = tdw1[j]$;}	s	s	s	s
	for (j=0; $j<s$; j++) $tdw1[j] = 0$;			s	
	Kara$(tdw1,\ a+s/2,\ b+s/2,\ s/2)$;				
	for (j=0; $j<s$; j++){$c[s+j]\hat{} = tdw1[j]$;	s	s	s	s
	$c[j+s/2]\hat{} = tdw1[j]$;}	s	s	s	s
	for (j=0; $j<s/2$; j++) {$tdw1[j] = a[j]\hat{}a[j+s/2]$;	$s/2$	s	$s/2$	$s/2$
	$tdw2[j] = b[j]\hat{}b[j+s/2]$;}	$s/2$	s	$s/2$	$s/2$
	Kara$(c+s/2,\ tdw1,\ tdw2,\ s/2)$;				
	}				
	}				
	Total=22s	$5s$	$6s$	$7s$	$4s$

recursive callings.

C. Timing Results

In order to compare the actual performance of these two algorithms, we have implemented them in ANSI C and tested on the following two computers:

1. A PC compatible desktop computer with a 3.0 GHz Pentium 4 processor (2M L2 Cache) and 1G memory running Linux 2.6.15, gcc 4.03 complier.

2. A PC compatible desktop computer with a 3.0 GHz Pentium 4 processor (512k L2 Cache) and 1G memory running Windows XP Professional, Visual C++ 6.0 complier.

Table V summarizes the timings for $n = 2^i$ $(6 < i < 18)$ and $f(u) = u^n + u^{15} + 1$. These timing and relative $speed\text{-}up = \frac{T_{Kara} - T_{mvp}}{T_{Kara}}$ values are the average values over several thousand (resp. hundred) executions for n less (resp. greater) than 65536. These $ratio$ values show that the matrix-vector product algorithm achieves a considerable improvement on the time complexity.

TABLE V

TIMING RESULTS (s)

n	Linux			Windows XP		
	mvp	Kara	*speed-up*	mvp	Kara	*speed-up*
128	9.48×10^{-7}	1.22×10^{-6}	21.9%	1.18×10^{-6}	1.30×10^{-6}	9.3%
256	2.92×10^{-6}	3.77×10^{-6}	22.7%	3.00×10^{-6}	4.02×10^{-6}	25.4%
512	8.82×10^{-6}	1.20×10^{-5}	26.3%	7.56×10^{-6}	1.31×10^{-5}	42.1%
1024	2.67×10^{-5}	3.60×10^{-5}	25.8%	2.55×10^{-5}	4.01×10^{-5}	36.4%
2048	7.95×10^{-5}	1.10×10^{-4}	27.4%	9.00×10^{-5}	1.18×10^{-4}	23.9%
4096	2.38×10^{-4}	3.33×10^{-4}	28.6%	2.45×10^{-4}	3.59×10^{-4}	31.8%
8192	7.16×10^{-4}	9.78×10^{-4}	26.8%	7.78×10^{-4}	1.08×10^{-3}	27.7%
16384	2.15×10^{-3}	3.04×10^{-3}	29.4%	2.03×10^{-3}	3.36×10^{-3}	39.6%
32768	6.45×10^{-3}	8.81×10^{-3}	26.8%	6.80×10^{-3}	1.01×10^{-2}	32.5%
65536	1.94×10^{-2}	2.70×10^{-2}	28.2%	1.94×10^{-2}	3.17×10^{-2}	38.8%
131072	5.80×10^{-2}	8.07×10^{-2}	28.1%	6.56×10^{-2}	1.07×10^{-1}	38.5%

IV. CONCLUSIONS

When compared with the Karatsuba algorithm, the algorithm presented here has the same level of simplicity for its implementation in software. It also has the following two main advantages: smaller look-up table and higher speed performance. As a result, it appears to be a better alternative to the Karatsuba algorithm for software implementation of $GF(2^n)$ multiplications when n is of intermediate sizes.

Although only the SPB is considered in this paper, we note that when the field elements are represented in polynomial, dual, weakly dual and triangular bases, the product $c = ab$ may also be written as a Toeplitz matrix-vector product. Thus, the method is still applicable for these bases.

Acknowledgment

The authors would like to thank Abdulaziz Alkhoraidly and Bijan Ansari for their help with obtaining timing results on Linux. The work was supported in part by NSERC through grants awarded to Dr. Hasan.

References

[1] G. Seroussi, "Table of Low-Weight Binary Irreducible Polynomials," *Technical Report HPL*-98-135, *Hewlett-Packard Laboratories*, Palo Alto, Calif., Aug. 1998, Available at http://www.hpl.hp.com/techreports/98/HPL-98-135.html.

[2] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics-Doklady (English translation)*, vol. 7, no. 7, pp. 595-596, 1963.

[3] P. L. Montgomery, "Five, Six, and Seven-Term Karatsuba-Like Formulae," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 362-369, 2005.

[4] J. V. Z. Gathen and J. Gerhard, "Polynomial Factorization over F_2," *Mathematics of Computation*, vol. 71, no. 240, pp. 1677-1698, 2002

[5] D. G. Cantor, "On arithmetical Algorithms over Finite Fields", *J. Combin. Theory Ser. A*, vol. 50, pp. 285-300, 1989.

[6] P. Roelse, "Factoring High-degree Polynomials over F_2 with Niederreiter's Algorithm on the IBM SP2", *Mathematics of Computation*, vol. 68, no. 226, pp. 869-880, 1999

[7] J. V. Z. Gathen and J. Gerhard, *Modern computer algebra*, Cambridge University Press, Cambridge, UK, 2nd. Edition, 2003.

[8] S. Winograd, *Arithmetic Complexity of Computations*, SIAM, 1980.

[9] M. A. Hasan and V. K. Bhargava, "Division and Bit-serial Multiplication over $GF(q^m)$," *IEE Proceedings-E*, vol. 139, no. 3, pp. 230-236, May. 1992.

[10] M. A. Hasan and V. K. Bhargava, "Architecture for Low Complexity Rate-Adaptive Reed-Solomon Encoder," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 938-942, 1995.

[11] H. Fan and M.A. Hasan, "A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields," *Technical Report CACR 2006-02*, University of Waterloo, Jan., 2006. Available at: http://www.cacr.math.uwaterloo.ca

[12] H. Fan and Y. Dai, "Fast bit parallel $GF(2^n)$ Multiplier for All Trinomials," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 485-490, 2005.