Asymmetric Squaring Formulae[*]

Jaewook Chung[†]and M. Anwar Hasan
jaewook.chung@gmail.com and ahasan@secure.uwaterloo.ca
Department of Electrical and Computer Engineering,
University of Waterloo, Ontario, Canada

Abstract

We present efficient squaring formulae based on the Toom-Cook multiplication algorithm. The latter always requires at least one non-trivial constant division in the interpolation step. We show such non-trivial divisions are not needed in the case two operands are equal for three, four and five-way squarings. Our analysis shows that our 3-way squaring algorithms have much less overhead than the best known 3-way Toom-Cook algorithm. Our experimental results show that one of our new 3-way squaring methods performs faster than `mpz_mul()` *in GNU multiple precision library (GMP) for squaring integers of approximately 2400–6700 bits on Pentium IV Prescott 3.2GHz. For squaring in $\mathbb{Z}[x]$, our 3-way squaring algorithms are much superior to other known squaring algorithms for small input size. In addition, we present 4-way and 5-way squaring formulae which do not require any constant divisions by integers other than a power of 2. Under some reasonable assumptions, our 5-way squaring formula is faster than the recently proposed Montgomery's 5-way Karatsuba-like formulae.*

Keywords: Squaring, Karatsuba algorithm, Toom-Cook multiplication algorithm, Montgomery's Karatsuba-like formulae, multiple-precision arithmetic

1. Introduction

Multiplication is one of the most frequently used arithmetic operations in public key cryptography and the performance of a cryptosystem often depends mostly on the efficiency of a multiplication operation. Squaring is a special case of multiplication when two operands are identical and it is usually faster than multiplication, but not more than a constant factor.

Over the past four decades, many algorithms have been proposed to perform multiplication operation efficiently. Since Karatsuba discovered the first sub-quadratic multiplication algorithm [1], several innovations have been made on multiplication algorithms [2, 3, 4, 5]. Unfortunately, none of these sub-quadratic multiplication algorithms has been considerably specialized for squaring. In this work, we attempt to fill this gap in the literature. It is not possible to have a squaring algorithm that is asymptotically better than the fastest multiplication algorithm in a ring whose characteristic is greater than 2 [6]. However, there are possibilities of some optimization by exploiting the fact that two operands are identical. We present three 3-way squaring formulae that are based on the Toom-Cook multiplication algorithm. Detailed methods for obtaining such formulae are presented. Experimental results show that our algorithms are more efficient than other 3-way multiplication algorithms for certain range of operand sizes. We also present efficient 4-way and 5-way squaring formulae that are potentially useful in practice.

This paper is organized as follows. In Section 2, we briefly review known multiplication algorithms. Then we discuss details on the Toom-Cook multiplication algorithm and discuss its issues in Section 3. In Section 4, we show in detail the methods for obtaining potentially better squaring algorithms than the Toom-Cook multiplication algorithm and present our new 3-way squaring formulae. Analysis and implementation results of our squaring algorithms are given in Sections 5 and 6, respectively. We present asymmetric formulae for 4-way and 5-way squaring in Section 7 and conclusions follow in Section 8.

2. Review of Multiplication Algorithms

In this section, we briefly review some well-known multiplication algorithms. Since cryptographic computations must be exact and efficient, we focus only on the algorithms that compute such results using only integer arithmetic. Let

[*]This work was supported in part by Natural Science and Engineering Research Council of Canada (NSERC) discovery and strategic project grants awarded to Dr. Hasan.

[†]Jaewook Chung's research was funded by Bell 125th Anniversary Scholarship.

$A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ be in $\mathbb{Z}[x]$. The product of $A(x)$ and $B(x)$ is computed as follows:

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i = A(x) \cdot B(x), \tag{1}$$

where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$ for $0 \le i \le 2n-2$ and $a_j = 0$ and $b_j = 0$ for $j \ge n$ and $j < 0$. Let $L(\cdot)$ denote the set of all integral linear combinations of the coefficients of a polynomial. We call a computation of form "$a \cdot b$", where $a \in L(A)$ and $b \in L(B)$, a *coefficient multiplication*. The performance of multiplication algorithms are often analyzed in terms of the number of coefficient multiplications required to compute (1). The rest of the computational cost including the cost for computing the linear combinations $a \in L(A)$ and $b \in L(B)$ necessary to compute (1) is referred to as *overhead*. The multiplication $a \cdot b$ can be slower than computing $a_i \cdot b_j$, due to the carries occurring when computing the linear combinations a and b. We count the cost difference of two computations ($a \cdot b$ and $a_i \cdot b_j$) toward the overhead.

In order to compute (1) using paper and pencil, n^2 coefficient multiplications are required. Such a method is called the schoolbook multiplication method. When $A(x) = B(x)$, only $n(n+1)/2$ coefficient multiplications are required, since off-diagonal products (i.e., $a_i b_j$ where $i \neq j$) always occur twice and need to be computed only once. We call this squaring method the *schoolbook squaring method*.

The first multiplication algorithm that has sub-quadratic complexity was developed by Karatsuba in 1963 [1]. The Karatsuba algorithm (KA) performs the multiplication of two 2-term polynomials using only three coefficient multiplications as follows:

$$C(x) = a_1 b_1 x^2 + ((a_0+a_1)(b_0+b_1) - a_0 b_0 - a_1 b_1)x + a_0 b_0. \tag{2}$$

The time complexity of $O(n^{\log_2 3})$ can be achieved by recursively applying (2). The KA is asymptotically better than the schoolbook method since $\log_2 3 \approx 1.58 < 2$. However, in real world applications, KA is faster than the schoolbook method only when n is sufficiently large, due to the fact that a larger amount of overhead is required in the KA than in the schoolbook method.

There is a well-known 3-way multiplication method which is shown below [7].

$$\begin{aligned}(a_2x^2 + a_1x + a_0)&(b_2x^2 + b_1x + b_0)\\ &= D_2x^4 + (D_1 + D_2 - D_5)x^3\\ &+ (D_0 + D_1 + D_2 - D_4)x^2\\ &+ (D_0 + D_1 - D_3)x + D_0,\end{aligned} \tag{3}$$

where

$$\begin{aligned} D_0 &= a_0b_0, & D_3 &= (a_0 - a_1)(b_0 - b_1),\\ D_1 &= a_1b_1, & D_4 &= (a_0 - a_2)(b_0 - b_2),\\ D_2 &= a_2b_2, & D_5 &= (a_1 - a_2)(b_1 - b_2).\end{aligned}$$

This formula requires 6 coefficient multiplications. Recursive use of (3) results in $O(n^{\log_3 6})$ time complexity. This method is less efficient than KA in an asymptotic sense, since $\log_3 6 \approx 1.63 > \log_2 3$. We call (3) as 3-way KA-like formula.

In 1963, Toom developed an elegant idea to perform multiplication of two degree-$(n-1)$ polynomials using only $(2n-1)$ coefficient multiplications [2]. In 1966, Cook improved Toom's idea [3]. The multiplication method they developed is now called the Toom-Cook algorithm. The latter is based on a well-known result from linear algebra: any degree-n polynomial can be uniquely determined by its evaluation at $(n+1)$ distinct points. Algorithm 1 shows a general idea how the Toom-Cook multiplication algorithm works.

```
Algorithm 1 Toom-Cook Multiplication Algorithm
Require: Degree $n-1$ polynomials $A(x)$ and $B(x)$.
Ensure: $C(x) = A(x) \cdot B(x)$.
1: (Evaluation) $u_i = A(x_i)$ and $v_i = B(x_i)$ for $i = 1, \ldots, 2n-1$, where $x_i$'s are all distinct.
2: (Point-Wise Multiplication) $C(x_i) = u_i v_i$ for $i = 1, \ldots 2n-1$.
3: (Interpolation) given $C(x_i)$'s, uniquely determine $c_j$'s for $j = 0, \ldots, 2n-2$, where $C(x) = \sum_{j=0}^{2n-2} c_j x^j$.
```

3. Further Details on the Toom-Cook Multiplication Algorithm

In Section 2, we have reviewed various multiplication algorithms including the Toom-Cook multiplication algorithm. In this section, we look into details on the Toom-Cook algorithm, especially on its interpolation step.

By noticing that the interpolation step in Algorithm 1 solves a system of $(2n-1)$ linear equations with $(2n-1)$ unknown values (the coefficients of $C(x)$), we can construct the following linear system:

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^{2n-2} \\ 1 & x_2 & \cdots & x_2^{2n-2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{2n-1} & \cdots & x_{2n-1}^{2n-2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-2} \end{bmatrix} = \begin{bmatrix} C(x_1) \\ C(x_2) \\ \vdots \\ C(x_{2n-1}) \end{bmatrix}. \tag{4}$$

The $(2n-1) \times (2n-1)$ matrix on the left hand side of (4) is called the Vandermonde matrix. We denote it by V. A Vandermonde matrix has a known determinant,

$D = \prod_{1\le j<i\le 2n-1}(x_i - x_j)$. The system (4) is uniquely solvable, since x_i's are all distinct in Algorithm 1. Computing the inverse matrix can be pre-computed for a fixed set of x_i's. Therefore, the coefficients c_i's can be easily obtained by multiplying the inverse matrix to the right-hand side of (4). In the interpolation step, at least one division by an odd, nontrivial factor must occur if $n > 2$. We provide the following theorems without proofs.

Theorem 1 *Let V denote the $(2n-1)\times(2n-1)$ matrix on the left-hand side of (4). There is no set of distinct integers $\{x_1,\dots,x_s\}$'s such that $D = \det V$ is a power of 2; $D = \prod_{1\le j<i\le s}(x_i - x_j) = \pm 2^k$ for some positive integer k, if $s > 3$.*

Theorem 2 *In the Toom-Cook multiplication algorithm, at least one constant division by an integer which is not a power of 2 must occur for $n > 2$.*

There are heuristic approaches for small n to reduce the number of constant divisions and its sizes as much as possible. Such methods perform elementary row operations on both sides of (4) until the system is solved, rather than multiplying the inverse matrix of V. For instance, Paul Zimmermann's implementation in GNU multiple precision library (GMP) v4.2.1 uses only one constant division by 3 for the 3-way Toom-Cook multiplication algorithm as shown in Section 3.1. Currently, Bodrato and Zanoni's method is known to be the best 3-way Toom-Cook multiplication algorithm [8].

3.1. Bodrato-Zanoni's 3-Way Toom-Cook Multiplication

This method has been developed by Bodrato and Zanoni [8], and implemented in GMP library as subroutines of `mpz_mul()`. Bodrato and Zanoni use $\{0, 1, 2, -1, \infty\}$ for the set of evaluation points.

Let $A(x) = a_2x^2 + a_1x + a_0$, $B(x) = b_2x^2 + b_1x + b_0$ and $C(x) = A(x)B(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$. Evaluation of $A(x)$ and $B(x)$ at $x_i \in \{0, 1, 2, -1, \infty\}$ and point-wise multiplication of $A(x_i)$'s and $B(x_i)$'s result in the following system of equations:

$$\begin{bmatrix} S_1\\ S_2\\ S_3\\ S_4\\ S_5 \end{bmatrix} = \begin{bmatrix} a_0b_0\\ (a_2+a_1+a_0)(b_2+b_1+b_0)\\ (4a_2+2a_1+a_0)(4b_2+2b_1+b_0)\\ (a_2-a_1+a_0)(b_2-b_1+b_0)\\ a_2b_2 \end{bmatrix} \quad (5)$$
$$= \begin{bmatrix} 0&0&0&0&1\\ 1&1&1&1&1\\ 16&8&4&2&1\\ 1&-1&1&-1&1\\ 1&0&0&0&0 \end{bmatrix} \begin{bmatrix} c_4\\ c_3\\ c_2\\ c_1\\ c_0 \end{bmatrix}$$

Algorithm 2 Bodrato-Zanoni's 3-Way Interpolation

```
Require: $(S_1, S_2, S_3, S_4, S_5)$ as in (5).
Ensure: $C(x) = A(x) \cdot B(x)$.
1: $T_1 \leftarrow (S_3 - S_4)/3$.          $(= 5c_4 + 3c_3 + c_2 + c_1)$
2: $T_2 \leftarrow (S_2 - S_4)/2$.          $(= c_3 + c_1)$
3: $T_3 \leftarrow S_2 - S_1$.              $(= c_4 + c_3 + c_2 + c_1)$
4: $T_1 \leftarrow (T_1 - T_3)/2$.          $(= 2c_4 + c_3)$
5: $T_3 \leftarrow T_3 - T_2 - S_5$.        $(= c_2)$
6: $T_1 \leftarrow (T_1 - 2S_5)/2$.         $(= c_3)$
7: $T_2 \leftarrow T_2 - T_1$.              $(= c_1)$
8: return $C(x) = S_5x^4 + T_1x^3 + T_3x^2 + T_2x + S_1$.
```

Then the above linear system can be solved very efficiently using row operations as shown in Algorithm 2.

4. New Squaring Formulae

To the best of our knowledge, no sub-quadratic multiplication algorithms reviewed in Section 2 have been considerably specialized for squaring. We attempt to fill in this gap in the literature. Of course, there is no squaring algorithm which is more than constant times faster than the fastest multiplication algorithm [6] and it is not a goal of this work to find such squaring algorithms.

In Section 3, we have seen that nontrivial constant divisions in the Toom-Cook algorithm are unavoidable. There are multiplication algorithms not requiring the constant division, but they use more than $(2n-1)$ coefficient multiplications [4]. Number theoretic transform-based multiplication algorithms do not require nontrivial constant divisions if the number of partitioning is a power of 2, but this means that N must be greater than $2n - 1$. However, this paper show that squarings can be performed without the nontrivial constant division using exactly $(2n-1)$ multiplications, at least for $n = 3, 4$ and 5.

All sub-quadratic multiplication algorithms we have reviewed in Section 2 are *symmetric* algorithms in the sense that all point-wise multiplications are squarings when $A(x) = B(x)$. On the other hand, our new squaring formulae are *asymmetric* algorithms, since they involve at least one point-wise multiplication of two different values.

4.1. Our Approach

To completely eliminate the constant divisions in the interpolation step, we take a different approach for constructing a linear system. Below we give detailed methods for obtaining linear equations on c_i's that cannot be derived by directly evaluating $C(x) = A(x)^2$. Our approach allows us to find linear equations of c_i's such that the corresponding linear system does not involve a Vandermonde matrix.

1. Taking modulo $(x^2 + ux + v^2)$, where u and v are some integers: By taking modulo $(x^2 + ux + v^2)$ on both sides of $C(x) = A(x)^2$, we obtain

   $$c'_1x + c'_0 \equiv (a'_1x + a'_0)^2 \pmod{(x^2 + ux + v^2)},$$

   where $a'_1x + a'_0 = A(x) \bmod (x^2 + ux + v^2)$ and $c'_1x + c'_0 = C(x) \bmod (x^2 + ux + v^2)$. Then it follows that

   $$c'_1x + c'_0 \equiv a'_1(2'a'_0 - ua'_1)x + (a'_0 - va'_1)(a'_0 + va'_1) \pmod{(x^2 + ux + v^2)}. \quad (6)$$

   It is interesting to see that computing both c'_0 and c'_1 requires only two coefficient multiplications. Hence, we obtain two useful linear equations for c'_is as follows:

   $$\begin{aligned} c'_1 &= a'_1(2'a'_0 - ua'_1), \\ c'_0 &= (a'_0 - va'_1)(a'_0 + va'_1). \end{aligned} \quad (7)$$

   Therefore, by choosing some small integers u and v, we can obtain useful linear equations on c_i's. Such equations cannot be obtained by simply evaluating $C(x) = A(x)^2$.
   We remark that a special case of this idea is known for efficient implementation of finite field squaring in $\mathbb{Z}_{p^2}[x]/f(x)$ where $f(x) = x^2 + x + 1$ [9].

2. Hermite interpolation: Interpolation using the evaluations of derivatives is known as Hermite interpolation. Interestingly, for squaring, each evaluation of the first derivative of $C(x)$ requires only one coefficient multiplication, since $C'(x) = 2A(x) \cdot A'(x)$.
   Evaluating the first derivative of $C(x)$ gives linear relations, some of which may not be obtained by evaluating $C(x) = A(x)^2$.

3. $A(x_i)^2 - A(x_j)^2 = (A(x_i) + A(x_j)) \cdot (A(x_i) - A(x_j))$ for $x_i \neq x_j$.
   Using this method, we can combine two linear equations from two distinct evaluations of $A(x)$ into one.

4. Duality: any function computing c_i can be used to compute c_{2n-1-i} with no changes. In other words, if $c_i = f(a_0, \ldots, a_{n-2}, a_{n-1})$, then $c_{2n-2-i} = f(a_{n-1}, \ldots, a_1, a_0)$ [10].
   Hence, we can safely substitute c_i to c_{2n-2-i} and a_j to a_{n-1-j} for all $0 \leq i \leq 2n-2$ and $0 \leq j \leq n-1$ in any linear equations on c_i's. This is a well-known fact and a similar argument holds for multiplications.

4.2. New 3-way Squaring

Let $\vec{C} = (c_4, c_3, c_2, c_1, c_0)$. To construct a 3-way squaring algorithm computing $C(x) = A(x)^2$ that requires only five coefficient multiplications, we need to find a set of five vectors five-tuple $(i_0, i_1, i_2, i_3, i_4)$, where the i_j's are all distinct, such that

- There exists a u_{i_j}, which is a product of two elements (not necessarily distinct) from $L(A)$, for each dimension-5 vector $\vec{L}_{i_j}$ such that $\vec{L}_{i_j} \circ \vec{C} = u_{i_j}$, where $\circ$ is a dot product.
- The set of vectors $\{\vec{L}_{i_0}, \ldots, \vec{L}_{i_3}, \vec{L}_{i_4}\}$ forms a basis in $\mathbb{Z}^5$.

Let $M = (\vec{L}_{i_4}, \ldots, \vec{L}_{i_1}, \vec{L}_{i_0})^T$. If we can find a five-tuple $(i_0, \ldots, i_3, i_4)$ which makes $\det M$ a power of 2, we get a squaring algorithm that require only 5 coefficient multiplications and no nontrivial constant divisions.

We have identified 24 potentially useful $\vec{L}_i$'s and u_i's by directly evaluating $C(x) = A(x)^2$ and by using our new construction methods given above, and show them in Table 1. Note that $\vec{L}_9$–$\vec{L}_{24}$ have been obtained using the methods given above and they cannot be obtained by simply evaluating $C(x) = A(x)^2$.

There are a total of $\binom{24}{5} = 42504$ possible combinations of $(i_0, i_1, i_2, i_3, i_4)$ and 34268 of them make $\{\vec{L}_{i_0}, \ldots, \vec{L}_{i_3}, \vec{L}_{i_4}\}$ a linearly independent set. We divide these 34268 combinations into the following three sets:

Set I: there are three or more i_j's such that $i_j \geq 9$.

Set II: there are only two i_j's such that $i_j \geq 9$.

Set III: there is only one i_j such that $i_j \geq 9$.

Set IV: there is no i_j such that $i_j \geq 9$.

Sets I, II, III and IV have 27254, 5946, 1012 and 56 combinations, respectively. In Set I, it is easily seen that combinations $(1, 2, 9, 10, 15)$, $(1, 2, 9, 10, 17)$, $(1, 2, 9, 10, 18)$, $(1, 2, 9, 10, 19)$ and $(1, 2, 9, 10, 20)$ lead to the simplest interpolation step. Note that $\vec{L}_1$, $\vec{L}_2$, $\vec{L}_9$, $\vec{L}_{10}$ immediately give the coefficients c_0, c_1, c_3 and c_4 of $C(x)$. The remaining coefficient c_2 can be obtained by at most two additions/subtractions. Among the five contenders, $(1, 2, 9, 10, 15)$ is the best choice, since computing u_{15} is easier than computing u_{17}, u_{18}, u_{19} and u_{20}.

In Set II, there are 124 combinations of $(i_0, \ldots, i_3, i_4)$ such that $|\det M| = 1$. To narrow down our search, we have considered only the combinations that lead to M such that the entries of M^{-1} are relatively small. Combinations $(1, 2, 3, 9, 10)$, $(1, 2, 4, 9, 10)$, $(1, 2, 4, 9, 22)$ and $(1, 2, 4, 10, 22)$ are the best, and they lead to the simplest form of M^{-1}. Combination $(1, 2, 4, 9, 10)$ is more advantageous than $(1, 2, 3, 9, 10)$, since computing u_4 is more efficient than computing u_3. Note that $(a_2 + a_1 + a_0)$ could be at most 1 bit longer than $(a_2 - a_1 + a_0)$.

Table 1. List of Candidate Vectors

i	$\vec{L}_i$	$u_i = \vec{L}_i \circ \vec{C}$	Comment
1	$(0,0,0,0,1)$	a_0^2	$C(0)$
2	$(1,0,0,0,0)$	a_2^2	$C(\infty)$
3	$(1,1,1,1,1)$	$(a_2+a_1+a_0)^2$	$C(1)$
4	$(1,-1,1,-1,1)$	$(a_2-a_1+a_0)^2$	$C(-1)$
5	$(16,8,4,2,1)$	$(4a_2+2a_1+a_0)^2$	$C(2)$
6	$(16,-8,4,-2,1)$	$(4a_2-2a_1+a_0)^2$	$C(-2)$
7	$(1,2,4,8,16)$	$(a_2+2a_1+4a_0)^2$	$2^4 \cdot C(1/2)$
8	$(1,-2,4,-8,16)$	$(a_2-2a_1+4a_0)^2$	$2^4 \cdot C(-1/2)$
9	$(0,0,0,1,0)$	$2a_0a_1$	$C'(0)$
10	$(0,1,0,0,0)$	$2a_1a_2$	Dual of 9
11	$(4,3,2,1,0)$	$2(a_2+a_1+a_0)(2a_2+a_1)$	$C'(1)$
12	$(-4,3,-2,1,0)$	$2(a_2-a_1+a_0)(-2a_2+a_1)$	$C'(-1)$
13	$(0,1,2,3,4)$	$2(a_2+a_1+a_0)(2a_0+a_1)$	Dual of 11
14	$(0,1,-2,3,-4)$	$2(a_2-a_1+a_0)(a_1-2a_0)$	Dual of 12
15	$(1,0,-1,0,1)$	$(a_0-a_2+a_1)(a_0-a_2-a_1)$	Constant term of $C(x) \bmod (x^2+1)$
16	$(0,-1,0,1,0)$	$2a_1(a_0-a_2)$	x's coefficient of $C(x) \bmod (x^2+1)$
17	$(-1,0,1,1,0)$	$(a_1-a_2+2a_0)(a_1+a_2)$	x's coefficient of $C(x) \bmod (x^2-x+1)$
18	$(0,-1,-1,0,1)$	$(a_0-a_1-2a_2)(a_0+a_1)$	Constant term of $C(x) \bmod (x^2-x+1)$
19	$(1,0,-1,1,0)$	$(a_2+a_1-2a_0)(a_2-a_1)$	x's coefficient of $C(x) \bmod (x^2+x+1)$
20	$(0,1,-1,0,1)$	$(a_0+a_1-2a_2)(a_0-a_1)$	Constant term of $C(x) \bmod (x^2+x+1)$
21	$(-1,0,0,0,1)$	$(a_0+a_2)(a_0-a_2)$	$A(0)^2-A(\infty)^2$
22	$(0,1,0,1,0)$	$2a_1(a_2+a_0)$	$(A(1)^2-A(-1)^2)/2$
23	$(0,4,0,1,0)$	$2a_1(4a_2+a_0)$	$(A(2)^2-A(-2)^2)/4$
24	$(0,1,0,4,0)$	$2a_1(4a_0+a_2)$	$4(A(1/2)^2-A(-1/2)^2)$

Moreover, $(1,2,4,9,10)$ is better than $(1,2,4,9,22)$ and $(1,2,4,10,22)$ since computing u_9 and u_{10} is faster than computing u_9 and u_{22} or computing u_{10} and u_{22}. We have also considered combinations that results in $|\det M| = 2,4,8$ and 16, but could not find a better combination than $(1,2,4,9,10)$.

In Set III, there is no combination that makes $|\det M| = 1$, but there are 26 combinations that makes $|\det M| = 2$. Among these 26 combinations, $(1,2,3,4,9)$ and $(1,2,3,4,10)$ lead to the most efficient squaring algorithm. We have also considered combinations that result in $|\det M| = 4,8$ and 16, but could not find a better combination than $(1,2,3,4,9)$ and $(1,2,3,4,10)$.

The Set IV only involves combinations of vectors that are derived from directly evaluating $C(x) = A(x)^2$ at rational points. The squaring formulae derived from these vectors must involve nontrivial constant divisions and are not considered further in this paper.

We have derived three new squaring methods from Set I, II and III.

1. Squaring Method 1 (SQR$_1$)

$$\begin{aligned} c_0 &= S_0 = a_0^2, \\ c_1 &= S_1 = 2a_1a_0, \\ S_2 &= (a_0-a_2+a_1)(a_0-a_2-a_1), \\ c_3 &= S_3 = 2a_1a_2, \\ c_4 &= S_4 = a_2^2, \\ c_2 &= S_0+S_4-S_2. \end{aligned} \tag{8}$$

The computation of S_i's requires 3 coefficient multiplications and 2 coefficient squarings.

2. Squaring Method 2 (SQR$_2$)

$$\begin{aligned} c_0 &= S_0 = a_0^2, \\ c_1 &= S_1 = 2a_1a_0, \\ S_2 &= (a_2-a_1+a_0)^2, \\ c_3 &= S_3 = 2a_1a_2, \\ c_4 &= S_4 = a_2^2, \\ c_2 &= S_2+S_1+S_3-S_0-S_4. \end{aligned}$$

The computation of S_i's requires 2 coefficient multiplications and 3 coefficient squarings.

3. Squaring Method 3 (SQR$_3$)

$$\begin{aligned}
c_0 &= S_0 = a_0^2, \\
S_1 &= (a_2 + a_1 + a_0)^2, \\
S_2 &= (a_2 - a_1 + a_0)^2, \\
c_3 &= S_3 = 2a_1a_2, \\
c_4 &= S_4 = a_2^2, \\
T_1 &= (S_1 + S_2)/2, \\
c_1 &= S_1 - T_1 - S_3, \\
c_2 &= T_1 - S_4 - S_0.
\end{aligned}$$

The computation of S_i's requires 1 coefficient multiplication and 4 coefficient squarings.

5. Analysis

In this section, we analyze the squaring algorithms SQR$_1$, SQR$_2$ and SQR$_3$ presented in Section 4. The analysis may differ depending on how the various squaring algorithms are used in specific applications (e.g., long integer squaring, squaring in extension field $GF(p^m)$, polynomial squaring in $\mathbb{Z}[x]$,...). In this section, we assume that our squaring formulae are applied to the arithmetic in $\mathbb{Z}[x]$. However, the results shown in this section are relevant to other applications. For applications in $GF(p^m)$, after the polynomial squaring has been completed, one needs to perform reduction by an irreducible polynomial for $GF(p^m)$ and then reduce each coefficient modulo p. These reduction operations are not dependent on the algorithm used for the polynomial squaring. For long integer squaring, an integer is interpreted as a polynomial and a polynomial squaring is performed. Then, one needs to overlap the coefficients and perform carry propagations. The overlapping and carry propagation is also not related to the algorithm used for the polynomial squaring.

We compare our algorithms with other known 3-way squaring algorithms: schoolbook squaring algorithm, 3-way KA-like formula and Bodrato-Zanoni's 3-way Toom-Cook algorithms.

We denote the radix of the representation by b. Addition or subtraction of two u-digit integers requires $\mathcal{A}(u)$ time. Multiplication and squaring of two u-digit integers require $\mathcal{M}(u)$ and $\mathcal{S}(u)$ times, respectively. Bit shift of u-digit integers require $\mathcal{B}(u)$ time. During evaluation and interpolation step, there are cases when the operands to addition/subtraction and shift are a few bits larger than u or $2u$ digits, where the coefficients of $A(x)$ are at most u digits long. For simplicity, we ignore this overhead caused by carries and borrows. However, we do not ignore the overhead involved in multiplying two integers that are slightly longer than u digits. For example, assume integers s and t are only 1-bit longer than u digits. Then we can write $s = s_hb^u + s_l$ and $t = t_hb^u + t_l$, where $|s_h|, |t_h| \leq 1$ and $0 \leq s_l, t_l < b^u$. For simplicity, we ignore the cost for multiplying carries, i.e., s_h and t_h.[1] Then the time required to compute $s \cdot t$ is at most $\mathcal{M}(u) + 2\mathcal{A}(u)$. If $|s_h|$ and $|t_h|$ are greater than 1, then constant multiplications $s_h \cdot t_l$ and $t_h \cdot s_l$ are performed using shifts and additions. For example, if $|s_h| \leq 3$ and $|t_h| \leq 1$, then $s_h \cdot t_l$ can be computed at most $\mathcal{A}(u) + \mathcal{B}(u)$. Therefore the total time requred to compute $s \cdot t$ is at most $\mathcal{M}(u) + 2\mathcal{A}(u) + \mathcal{B}(u)$. The time required to compute s^2 is $\mathcal{S}(u) + \mathcal{B}(u) + \mathcal{A}(u)$ in the worst case. When computing a product $2a_ia_j$, we always compute a_ia_j first and then perform the bit shift later. It is reasonable to assume that $\mathcal{A}(\cdot)$ and $\mathcal{B}(\cdot)$ are linear functions; $\mathcal{A}(fu+gv) = f\mathcal{A}(u)+g\mathcal{A}(v)$ and $\mathcal{B}(fu + gv) = f\mathcal{B}(u) + g\mathcal{B}(v)$. The exact division by 3 of an u-digit integer used in the 3-way Toom-Cook algorithm shown in Section 3.1 is denoted by $\mathcal{D}_3(u)$.

We assume that $A(x) = a_2x^2 + a_1x + a_0$ is the input, where a_i's are u digits long. Table 2 shows our analysis results. Table 3 shows the conditions for which our squaring algorithms are superior to the other algorithms.

Table 3 shows that there is apparently no single algorithm that is absolutely superior to the others. Without considering the actual values $\mathcal{S}(u)$, $\mathcal{M}(u)$, $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$, which are very platform specific, it is not possible to decide which algorithm is faster than the rest. However, one thing that is clear from Table 3 is that the 3-way Toom-Cook algorithm becomes the best for squaring polynomials

Table 3. Conditions for Which SQR$_i$'s Are Faster Than Other 3-way Algorithms

i	SQR$_i$ vs. 3-way Toom-Cook
1	$3\mathcal{M}(u) < 3\mathcal{S}(u) + 7\mathcal{B}(u) + 16\mathcal{A}(u) + \mathcal{D}_3(2u)$
2	$2\mathcal{M}(u) < 2\mathcal{S}(u) + 7\mathcal{B}(u) + 14\mathcal{A}(u) + \mathcal{D}_3(2u)$
3	$\mathcal{M}(u) < \mathcal{S}(u) + 6\mathcal{B}(u) + 10\mathcal{A}(u) + \mathcal{D}_3(2u)$
i	SQR$_i$ vs. Schoolbook sqr.
1	$7\mathcal{A}(u) < \mathcal{S}(u) + \mathcal{B}(u)$
2	$9\mathcal{A}(u) < \mathcal{M}(u) + \mathcal{B}(u)$
3	$\mathcal{S}(u) + 13\mathcal{A}(u) < 2\mathcal{M}(u)$
i	SQR$_i$ vs. 3-way KA-like
1	$3\mathcal{M}(u) + 2\mathcal{B}(u) < 4\mathcal{S}(u) + 11\mathcal{A}(u)$
2	$2\mathcal{M}(u) + 2\mathcal{B}(u) < 3\mathcal{S}(u) + 9\mathcal{A}(u)$
3	$\mathcal{M}(u) + 3\mathcal{B}(u) < 2\mathcal{S}(u) + 5\mathcal{A}(u)$

[1] Note, however, that the 3-way Toom-Cook multiplication algorithm in GMP v4.2.1 stores carries and borrows in the most significant digit place instead of handling them separately with extra variables. This method has a trade-off. Using extra digit reduces the number of additions and subtractions, but coefficient multiplications and squarings become slower. We have tested both methods and found that it is better to use extra variables for carries and borrows on architectures on which we have performed our experiments.

Table 2. Analysis Results of Various Squaring Algorithms

Algorithm	$\mathcal{S}\&\mathcal{M}$	Overhead
3-way Toom-Cook	$5\mathcal{S}(u)$	$12\mathcal{B}(u)+25\mathcal{A}(u)+\mathcal{D}_3(2u)$
Schoolbook sqr.	$3\mathcal{S}(u)+3\mathcal{M}(u)$	$6\mathcal{B}(u)+2\mathcal{A}(u)$
3-way KA-like	$6\mathcal{S}(u)$	$3\mathcal{B}(u)+20\mathcal{A}(u)$
SQR$_1$	$2\mathcal{S}(u)+3\mathcal{M}(u)$	$5\mathcal{B}(u)+9\mathcal{A}(u)$
SQR$_2$	$3\mathcal{S}(u)+2\mathcal{M}(u)$	$5\mathcal{B}(u)+11\mathcal{A}(u)$
SQR$_3$	$4\mathcal{S}(u)+1\mathcal{M}(u)$	$6\mathcal{B}(u)+15\mathcal{A}(u)$

as u increases. The timings $\mathcal{B}(u)$, $\mathcal{A}(u)$ and $\mathcal{D}_3(2u)$ grow linearly with u, but timings of multiplication ($\mathcal{M}(u)$) and squaring ($\mathcal{S}(u)$) grow quadratically or sub-quadratically depending on the methods used for point-wise multiplications. It is obvious that, for large u, the effect of reduced overhead in our algorithms will be overwhelmed by the timing difference in multiplication and squaring.

However, SQR$_i$'s have very little amount of overhead compared to the 3-way Toom-Cook multiplication algorithm. Hence, it is possible that, for some small u, the timing difference of multiplication and squaring is small enough to satisfy some of the conditions in Table 3. In fact, our implementation results given in Section 6 confirm that there is a range of u where some conditions of Table 3 are satisfied.

6. Implementation Results

To verify the practical usefulness of our algorithms, we have implemented in software the functions for large integer squaring, and the functions for degree-2 polynomial squaring in $\mathbb{Z}[x]$ using SQR$_1$, SQR$_2$ and SQR$_3$ presented in Section 4. Our experiments have been performed on Linux (kernel version 2.6.15.26) running on Intel Pentium IV Prescott 3.2GHz. We have used GCC 4.0.3 to compile all programs. We have compiled GMP library v4.2.1 in two passes. Between the first and the second passes of compilations, we have performed GMP's tuneup program (with an option '-p 100000000' for better precision than default) so that GMP uses the optimal threshold values between multiplication algorithms. We compiled all our source codes using the same compiler options used for compiling the GMP library. We have ensured that our program does not link with the shared library of GMP, since shared libraries have a performance penalty due to the runtime address resolution. The testing program has been run at the highest possible priority to minimize the risk of interference by other running processes.

6.1. Application to Large Integer Squaring

We have compared our implementation of SQR$_i$'s with GMP's squaring function. For fair comparison, all of our algorithms have been written so that they have the same functionality as `mpn_toom3_sqr_n()` function in GMP. Note that `mpn_toom3_sqr_n()` is a low level implementation of the squaring algorithm.

Our squaring algorithms have been written using the same coding style that Harley used for implementing `mpn_toom3_sqr_n()` in GMP 4.1.4. The implementations of the two algorithms in GMP v4.1.4 and v4.2.1 use different coding styles. Harley stores the carries in separate variables in GMP 4.1.4, but Zimmermann stores them in the most significant digit place in GMP 4.2.1. Zimmermann's method increases the digit length of input to coefficient multiplications, additions and subtractions. However, such a method reduces the number of function calls to additions and subtractions. We implmented Bodrato-Zanoni's 3-way Toom-Cook algorithm using Harley's coding style and compared it with the one available at `http://bodrato.it/software/mul_n.11-gen-2007.c`. We have determined that our implementation is faster on the processor (Pentium IV 3.2GHz) we used for our timing experiments. Therefore, we used our own implementation of Bodrato-Zanoni's algorithm in all our experiments.

Figure 1 shows the timing ratio of `mpz_mul()` and the 3-way Toom-Cook multiplication algorithm to SQR$_3$. On Pentium IV 3.2GHz, `mpz_mul()` uses the schoolbook squaring algorithm for small operands, KA for input longer than 1984 bits, the 3-way Toom-Cook algorithm for input longer than 3744 bits. In our experiments, we have found that SQR$_1$ and SQR$_2$ are slower than `mpz_mul()` for all sizes of input. Our experiments show that SQR$_3$ outperforms `mul_mul()` for operands that are approximately 2400–6700 bits long.

6.2. Application to Polynomial Squaring in $\mathbb{Z}[x]$

We have applied our squaring algorithm for performing polynomial multiplication in $\mathbb{Z}[x]$. We have implemented functions for squaring degree-2 polynomials in $\mathbb{Z}[x]$. The

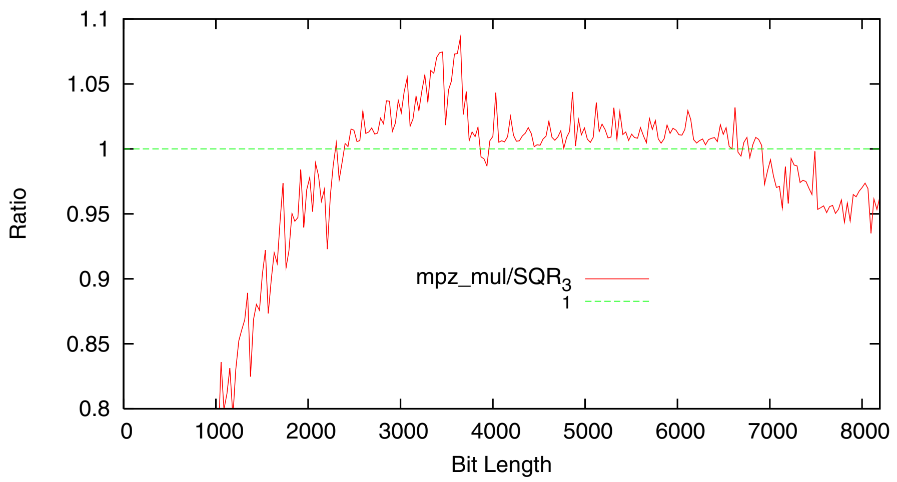

Figure 1. Timing Ratio of SQR$_3$ vs. Other Algorithms on Pentium IV 3.2GHz

implementation uses the functions from the GMP library. The timing results on Pentium IV 3.2GHz are shown in Table 4. The first column in Table 4 shows the sizes of coefficients in bits. In the table, the best timing for each bit length is indicated in bold. SQR$_1$ is the most efficient squaring algorithm for squaring polynomials having small coefficients of up to 576 bits. For polynomials with coefficients up to 1216 bits, SQR$_3$ is the most efficient. However, the 3-way Toom-Cook algorithm becomes the fastest algorithm for squaring degree-2 polynomials whose coefficients are at least 1216 bits long.

7. 4-way and 5-way Squaring Formulae

Let $A(x) = a_3x^3+a_2x^2+a_1x+a_0$. To compute $C(x) = \sum_{i=0}^{6} c_ix^i = A(x)^2$, we first compute S_i's as shown below:

$$\begin{aligned}
S_1 &= a_0^2 = A(0)^2, \\
S_2 &= 2a_0a_1 = 2A(0)A'(0), \\
S_3 &= (a_0+a_1-a_2-a_3)(a_0-a_1-a_2+a_3) \\
&= Re(A(i)^2), \qquad (9)\\
S_4 &= (a_0+a_1+a_2+a_3)^2 = A(1)^2, \\
S_5 &= 2(a_0-a_2)(a_1-a_3) = Im(A(i)^2), \\
S_6 &= 2a_3a_2 = 2A(\infty)A'(\infty), \\
S_7 &= a_3^2 = A(\infty)^2.
\end{aligned}$$

The linear combinations of a_i's in (9) can be computed using the following:

$$U_1 = a_0 - a_2,\ U_2 = a_1 - a_3,$$
$$U_3 = U_1 + U_2,\ U_4 = U_1 - U_2,$$
$$U_5 = a_0 + a_1 + a_2 + a_3.$$

Algorithm 3 New 4-Way Toom-Cook Interpolation for Squaring

```
Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$ as in (9).
Ensure: $C(x) = A(x) \cdot B(x)$.
1: $T_1 \leftarrow S_3 + S_4$.            $(= c_5 + 2c_4 + c_3 + c_1 + 2c_0)$
2: $T_2 \leftarrow (T_1 + S_5)/2$.        $(= c_5 + c_4 + c_1 + c_0)$
3: $T_3 \leftarrow S_2 + S_6$.            $(= c_5 + c_1)$
4: $T_4 \leftarrow T_2 - T_3$.            $(= c_4 + c_0)$
5: $T_5 \leftarrow T_3 - S_5$.            $(= c_3)$
6: $T_6 \leftarrow T_4 - S_3$.            $(= c_6 + c_2)$
7: $T_7 \leftarrow T_4 - S_1$.            $(= c_4)$
8: $T_8 \leftarrow T_6 - S_7$.            $(= c_2)$
9: return $C(x) = S_7x^6 + S_6x^5 + T_7x^4 + T_5x^3 + T_8x^2 + S_2x + S_1$.
```

This method uses 3 coefficient squarings and 4 coefficient multiplications. Note that KA requires 9 coefficient squarings for squaring a polynomial using 4-way split. The interpolation method is given in Algorithm (3) and it requires 8 additions/subtractions and 1 bit shift.

Using the same analysis methods in Section 5, we obtain that our 4-way squaring algorithm requires $3\mathcal{S}(u) + 4\mathcal{M}(u) + 28\mathcal{A}(u) + 13\mathcal{B}(u)$.

7.1. New 5-way Squaring Method

Let $A(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. To compute $C(x) = \sum_{i=0}^{8} c_ix^i = A(x)^2$, we first compute S_i's as

Table 4. Timing Results of Polynomial Squaring on Pentium IV 3.2GHz (unit= μs.)

Word length	SQR$_1$	SQR$_2$	SQR$_3$	3-way Toom-Cook	3-way KA-like	Schoolbook sqr.
8	**1.12**	1.42	1.33	1.67	1.80	1.26
18	**3.43**	3.96	3.59	3.94	4.79	4.15
19	3.89	4.25	**3.85**	4.26	5.25	4.42
32	8.37	9.65	**8.17**	8.25	11.12	10.72
38	11.28	12.67	**10.73**	10.48	14.17	14.38
39	12.84	14.11	11.51	**10.88**	15.21	16.29
47	17.15	18.94	15.42	**14.47**	19.85	22.02

shown below:

$$\begin{aligned}
S_1 &= a_0^2 = A(0),\\
S_2 &= a_4^2 = A(\infty),\\
S_3 &= (a_0+a_1+a_2+a_3+a_4)^2 = A(1)^2,\\
S_4 &= (a_0-a_1+a_2-a_3+a_4)^2 = A(-1)^2,\\
S_5 &= 2(a_0-a_2+a_4)(a_1-a_3) = Im(A(i)^2),\\
S_6 &= (a_0+a_1-a_2-a_3+a_4) \qquad (10)\\
&\times\ (a_0-a_1-a_2+a_3+a_4) = Re(A(i)^2),\\
S_7 &= (a_1+a_2-a_4)(a_1-a_2-a_4+2(a_0-a_3))\\
&= (A(x)^2 \mod (x^2-x+1))',\\
S_8 &= 2a_0a_1 = 2A(0)A'(0),\\
S_9 &= 2a_3a_4 = 2A(\infty)A'(\infty).
\end{aligned}$$

The above system needs 4 squarings and 5 multiplications. The linear combinations of a_i's can be computed as follows using 14 additions or subtractions and 1 shift:

$$\begin{aligned}
U_1 &= a_0+a_4,\\
U_2 &= a_1+a_3,\\
U_3 &= a_1-a_4,\\
U_4 &= \mathbf{a_1}-\mathbf{a_3},\\
U_5 &= U_1+a_2 = a_0+a_2+a_4,\\
U_6 &= U_1-a_2 = \mathbf{a_0}-\mathbf{a_2}+\mathbf{a_4},\\
U_7 &= U_5+U_2 = \mathbf{a_0}+\mathbf{a_1}+\mathbf{a_2}+\mathbf{a_3}+\mathbf{a_4}.\\
U_8 &= U_5-U_2 = \mathbf{a_0}-\mathbf{a_1}+\mathbf{a_2}-\mathbf{a_3}+\mathbf{a_4},\\
U_9 &= U_6+U_4 = \mathbf{a_0}+\mathbf{a_1}-\mathbf{a_2}-\mathbf{a_3}+\mathbf{a_4},\\
U_{10} &= U_6-U_4 = \mathbf{a_0}-\mathbf{a_1}-\mathbf{a_2}+\mathbf{a_3}+\mathbf{a_4},\\
U_{11} &= U_3+a_2 = \mathbf{a_1}+\mathbf{a_2}-\mathbf{a_4},\\
U_{12} &= U_3-a_2 = a_1-a_2-a_4,\\
U_{13} &= U_{12}+2(a_0-a_3) = \mathbf{a_1}-\mathbf{a_2}-\mathbf{a_4}+\mathbf{2}(\mathbf{a_0}-\mathbf{a_3}).
\end{aligned}$$

Interpolation can be performed as shown in Algorithm 4, which requires 16 additions/subtractions and 3 shifts. Algorithm 4 has been proposed in [8] by Bodrato and Zanoni. It is an improved algorithm than the one presented in [11], which requires 18 additions and subtractions, 6 shifts and no division by constant.

Algorithm 4 Bodrato-Zanoni's 5-Way Toom-Cook Interpolation for Squaring

```
Require: $(S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9)$ as in (10).
Ensure: $C(x) = A(x) \cdot B(x)$.
 1: $T_1 \leftarrow (S_3 + S_4)/2$        $(= c_8 + c_6 + c_4 + c_2 + c_0)$
 2: $T_2 \leftarrow S_7 - S_2 - S_8 - S_9$        $(= -c_5 - c_4 + c_2)$
 3: $T_3 \leftarrow (S_6 + T_1)/2$        $(= c_8 + c_4 + c_0)$
 4: $T_4 \leftarrow S_3 - T_1$        $(= c_7 + c_5 + c_3 + c_1)$
 5: $T_5 \leftarrow (T_4 - S_5)/2$        $(= c_7 + c_3)$
 6: $T_6 \leftarrow T_4 - S_8$        $(= c_7 + c_5 + c_3)$
 7: $T_7 \leftarrow T_1 - T_3$        $(= c_6 + c_2)$
 8: $T_8 \leftarrow T_6 - T_5$        $(= c_5)$
 9: $T_9 \leftarrow T_3 - S_0 - S_1$        $(= c_4)$
10: $T_{10} \leftarrow T_5 - S_9$        $(= c_3)$
11: $T_{11} \leftarrow T_2 + T_8 + T_9$        $(= c_2)$
12: $T_{12} \leftarrow T_7 - T_{11}$        $(= c_6)$
13: return $C(x) = S_2x^8 + S_9x_7 + T_{12}x^6 + T_8x^5 + T_9x^4 + T_{10}x^3 + T_{11}x^2 + S_8x + S_1$.
```

Note that Montgomery's 5-way formulae [12] requires 13 squarings. Using the same analysis technique and assuming that each coefficient a_i is a u-digit integer, we obtain that our 5-way squaring algorithm requires at most $4\mathcal{S}(u) + 5\mathcal{M}(u) + 56\mathcal{A}(u) + 20\mathcal{B}(u)$. Montgomery's 5-way algorithm requires at most $13\mathcal{S}(u)+65\mathcal{A}(u)+10\mathcal{B}(u)$ when two operands are identical. Therefore, if $5\mathcal{M}(u) + 10\mathcal{B}(u) < 9\mathcal{S}(u) + 9\mathcal{A}(u)$, then our algorithm is superior. Ignoring the overhead terms ($\mathcal{A}$ and $\mathcal{B}$), our algorithm is superior if squaring/multiplication ratio is more than $5/9 \approx 0.56$. This condition appears to be easily satisfied in practice. The GMP's squaring/multiplication timing ratio is between 0.6–0.8 for operand sizes larger than 500 bits on Pentium IV Prescott 3.2GHz.

8. Conclusions

In this paper, we have presented new 3, 4 and 5-way polynomial squaring formulae. Our new formulae are based on the Toom-Cook multiplication algorithm and they require the same number of coefficient multiplications used in the Toom-Cook multiplication algorithm. However, our approach eliminates the need for nontrivial constant divisions that are always required in the n-way Toom-Cook multiplication algorithms for $n \geq 3$. Our experimental results confirm that one of our 3-way formulae is slightly faster than GMP's squaring routine for squaring integers of size approximately 2300–6700 bits on Pentium IV 3.2GHz. Moreover, according to our implementation results, our methods are the best among all known 3-way squaring formulae for squaring degree-2 polynomials whose coefficients are shorter than approximately 1200 bits on Pentium IV 3.2GHz. However, symmetric squaring algorithms are advantageous for squaring very large size operands, since our asymmetric squaring algorithms use at least one point-wise multiplication that cannot be computed by squaring.

Acknowledgement We thank Duho Kang from POSTECH (Pohang University of Science and Technology, Korea) for implementing the Bodrato-Zanoni's 3-way Toom-Cook squaring algorithm.

References

[1] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady (English translation)*, vol. 7, no. 7, pp. 595–596, 1963.

[2] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Math*, vol. 3, pp. 714–716, 1963.

[3] S. A. Cook, *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, May 1966.

[4] S. Winograd, *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics 33, Society for Industrial and Applied Mathematics, 1980.

[5] A. Schönhage and V. Strassen, "Schnelle mutiplikation grosser zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[6] D. Zuras, "More on squaring and multiplying large integers," *IEEE Transactions on Computers*, vol. 43, pp. 899–908, August 1994.

[7] D. V. Bailey and C. Paar, "Efficient arithmetic in finite field extensions with application in elliptic curve cryptography," *Journal of Cryptology*, vol. 14, no. 3, pp. 153–176, 2001.

[8] M. Bodrato and A. Zanoni, "What about Toom-Cook matrices optimality?," 2006. Available at `http://bodrato.it/papers/WhatAboutToomCookMatricesOptimality.pdf`.

[9] M. Stam and A. K. Lenstra, "Speeding up XTR," in *Advances in Cryptology - ASIACRYPT 2001*, LNCS 2248, pp. 125–143, Springer-Verlag, 2001.

[10] B. Sunar, "A generalized method for constructing subquadratic complexity $GF(2^k)$ multipliers," *IEEE Transactions on Computers*, vol. 53, pp. 1097–1105, 2004.

[11] J. Chung and M. A. Hasan, "Asymmetric squaring formulae," 2006. Available at `http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-24.pdf`.

[12] P. L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," *IEEE Transaction on Computers*, vol. 54, no. 3, pp. 362–369, 2005.