More Generalized Mersenne Numbers
(Extended Abstract)

Jaewook Chung* and Anwar Hasan
j4chung@uwaterloo.ca
ahasan@ece.uwaterloo.ca

Centre for Applied Cryptographic Research
University of Waterloo
Ontario, Canada

Abstract. In 1999, Jerome Solinas introduced families of moduli called the generalized Mersenne numbers [8]. The generalized Mersenne numbers are expressed in a polynomial form, $p = f(t)$, where t is a power of 2. It is shown that such p's lead to fast modular reduction methods which use only a few integer additions and subtractions. We further generalize this idea by allowing any integer for t. We show that more generalized Mersenne numbers still lead to a significant improvement over well-known modular multiplication techniques. While each generalized Mersenne number requires a dedicated implementation, more generalized Mersenne numbers allow flexible implementations that work for more than one modulus. We also show that it is possible to perform long integer modular arithmetic without using multiple precision operations when t is chosen properly. Moreover, based on our results, we propose efficient arithmetic methods for XTR cryptosystem.

Keywords: Generalized Mersenne Numbers, RSA, XTR, Montgomery, Karatsuba-Ofman, Modular Reduction

1 Introduction

Many cryptosystems such as RSA, XTR, and elliptic curve cryptosystems (ECC) based on prime field involve significant use of modular multiplications and squaring operations. Hence it is very important to implement efficient modular multiplication and squaring operations in implementing such cryptosystems.

There are many well-known techniques for efficient implementation of modular arithmetic operations [5, 3]. The Montgomery algorithm is the

* This research was funded in part by Natural Sciences and Engineering Research Council of Canada (NSERC) through Post Graduate Scholarship – B (PGS-B) and in part by Canadian Wireless Telecommunications Association (CWTA) Scholarship.

most popular choice since it is very efficient and it works for any modulus. RSA cryptosystems are usually implemented using the Montgomery algorithm and ECC defined over prime field and XTR cryptosystems can also benefit from the Montgomery algorithm.

There are techniques that take advantage of special form of modulus. The Mersenne numbers $m = 2^k - 1$ [3] are well-known examples. However Mersenne numbers are never primes and thus they are not cryptographically useful. Due to Richard Crandall [2], the Mersenne numbers are generalized to the form $2^k - c$ where c is a small integer. Such families of integers are known as pseudo-Mersenne numbers. Modular reductions by pseudo-Mersenne numbers are very efficiently done using a few constant multiplications by a small integer c. However pseudo-Mersenne numbers are often avoided due to the patent issue [2].

In 1999, Jerome Solinas introduced families of moduli called the generalized Mersenne numbers [8]. The generalized Mersenne numbers are expressed in a polynomial form $p = f(t)$ where t is a power of 2. Such representations lead to fast modular reduction which uses only a few integer additions and subtractions. It is well-known that all 5 NIST-recommended curves [7, 6] defined over prime fields are based on the generalized Mersenne numbers.

However the generalized Mersenne numbers have a significant limitation: there could be only one number for a given bit length of p and a fixed $f(t)$. Hence, if $p = f(t)$ is not a desired number, we have to change either the bit length of p or $f(t)$. It follows that each generalized Mersenne number requires a dedicated implementation for that number. In some applications, this property makes the generalized Mersenne numbers less useful.

In this paper, we further extend the idea of generalized Mersenne numbers by removing the restriction that t must be a power of 2. Surprisingly, it will be shown that modular arithmetic based on such integers could lead to much faster modular arithmetic than the classical and the Montgomery modular arithmetic methods. We give criteria for choosing good $f(t)$'s and show further improvements are possible by choosing pseudo-Mersenne numbers for t or by choosing t whose bit length is a few bits shorter than processor's word size. The latter technique is quite useful in practice, since it makes possible to implement long integer modular arithmetic without using multiple precision operations. We also apply our methods and principles to the arithmetic operations required in XTR cryptosystems. We begin with a simple example.

2 An Example

We use the same polynomial $f(t) = t^3 - t + 1$ used in [8]. We generalize the ideas from [8] by removing the restriction that t must be a power of 2. Hence we assume that t could be any positive integer.

Let p be in the form $p = f(t) = t^3 - t + 1$ where t can be any positive integer. Note that any integers x and y, where $0 \le x, y < p$, can be expressed in the form of the second degree polynomial in $\mathbb{Z}[t]$,

$$\begin{aligned} x &= x_2 t^2 + x_1 t + x_0 \ , \\ y &= y_2 t^2 + y_1 t + y_0 \ , \end{aligned} \tag{1}$$

where $|x_i|$, $|y_i| < t$ for $i = 0, 1, 2$. Using simple polynomial multiplication and reduction, the modular multiplication of x and y is done as follows,

$$\begin{aligned} xy \equiv\ & (x_2y_0 + x_1y_1 + x_0y_2 + x_2y_2)t^2 \\ & + (x_1y_0 + x_0y_1 - x_2y_2 + x_2y_1 + x_1y_2)t \\ & + x_0y_0 - x_2y_1 - x_1y_2 \pmod{t^3 - t + 1} \ . \end{aligned} \tag{2}$$

If the school-book method is used in computing (2), 9 multiplications are required. However, when multiplying two second degree polynomials, it is advantageous to use the 3-segment Karatsuba-Ofman Algorithm (KOA) [1]. We first compute 6 intermediate values, D_0 through D_5, using 6 multiplications (or 6 squarings when $x = y$). Note that the normal application of KOA would require 7 multiplications (or 7 squarings when $x = y$).

$$\begin{aligned} D_0 &= x_0y_0 \ , \\ D_1 &= x_1y_1 \ , \\ D_2 &= x_2y_2 \ , \\ D_3 &= (x_0 + x_1)(y_0 + y_1) \ , \\ D_4 &= (x_0 + x_2)(y_0 + y_2) \ , \\ D_5 &= (x_1 + x_2)(y_1 + y_2) \ . \end{aligned} \tag{3}$$

Then the modular multiplication of x and y is done in $\mathbb{Z}[t]/f(t)$ as follows.

$$xy \equiv c_2t^2 + c_1t + c_0 \pmod{t^3 - t + 1} \ , \tag{4}$$

where,

$$\begin{aligned} c_2 &= D_4 - D_0 + D_1 \ , \\ c_1 &= D_3 - (D_0 - D_5) - 2(D_1 + D_2) \ , \\ c_0 &= (D_0 - D_5) + (D_1 + D_2) \ . \end{aligned} \tag{5}$$

Note that, at this stage, modular reduction is partially done by a polynomial reduction which uses simple integer additions and subtractions only. Since the bit lengths of c_i's are about twice as large as that of t, c_i's must be reduced before further modular multiplication or squaring operations are performed. We express c_i's in the form of the first degree polynomial in $\mathbb{Z}[t]$ as follows,

$$\begin{aligned} c_2 &= c_{21}t + c_{20} \ , \\ c_1 &= c_{11}t + c_{10} \ , \\ c_0 &= c_{01}t + c_{00} \ , \end{aligned} \tag{6}$$

where $|c_{i0}| < t$ for $0 \le i \le 2$. To compute (6), 3 integer divisions are required. Then it follows that,

$$xy \equiv (c_{11} + c_{20})t^2 + (c_{01} + c_{10} + c_{21})t + c_{00} - c_{21} \pmod{f(t)} \ . \tag{7}$$

We repeatedly apply this reduction process until the absolute values of the resulting coefficients are all less than t.

2.1 Analysis

We assume that the modular multiplication is implemented in software. Note that microprocessors usually deal with the units of data known as words. Hence we analyze our modular arithmetic method in terms of word length.

Let t_m and t_d respectively denote the time required for a microprocessor to compute word level multiplication and division operations. According to [5], the running times required to compute long integer multiplication, squaring and division are given as follows,

$$\begin{aligned} M(a) &= a^2 t_m \ , \\ S(a) &= \frac{a^2 + a}{2} t_m \ , \\ D(a,b) &= (a-b)(b+2)t_m + (a-b)t_d \ , \end{aligned} \tag{8}$$

where $M(a)$ and $S(a)$ are the running times for a long integer multiplication and a long integer squaring respectively, when multiplicands are both a words long. $D(a,b)$ is the running time for a long integer division when the dividend is a words long and the divisor is b words long.

Now, let n be the number of words required to store t. Suppose that t is fully occupying the n word blocks. Then the word length of $p = t^3 - t + 1$ should be about $3n$.

For simplicity, we do not consider the time required for additions and subtractions in our analysis. In (3), we need 6 integer multiplications ($6M(n)$). In (5), there are only additions and subtractions. For the first application of (7), we need 3 integer divisions ($3D(2n, n)$). Note that, after this first reduction, the size of coefficients should be about the same size as t. Therefore, we only need integer additions and subtractions in the subsequent applications of (7). Hence the total running time, $T_m(n)$ where n is the word length of t, of modular multiplication is,

$$\begin{aligned} T_m(n) &= 6M(n) + 3D(2n, n) \\ &= (9n^2 + 6n)t_m + 3nt_d \ . \end{aligned} \tag{9}$$

Similarly, the total running time, $T_s(n)$, for modular squaring is,

$$\begin{aligned} T_s(n) &= 6S(n) + 3D(2n, n) \\ &= (6n^2 + 9n)t_m + 3nt_d \ . \end{aligned} \tag{10}$$

In Table 1, we compare these results with the running times required for the classical methods and the Montgomery methods.

Table 1. Running Time Comparison

	Modular Multiplication	Modular Squaring
Classical	$(18n^2 + 6n)t_m + 3nt_d$	$(13.5n^2 + 7.5n)t_m + 3nt_d$
Montgomery	$(18n^2 + 6n)t_m$	$(13.5n^2 + 7.5n)t_m$
Our methods	$(9n^2 + 6n)t_m + 3nt_d$	$(6n^2 + 9n)t_m + 3nt_d$
Our methods (no KOA)	$(12n^2 + 6n)t_m + 3nt_d$	$(7.5n^2 + 10.5n)t_m + 3nt_d$

In Table 1, we see huge advantage of our modular multiplication and squaring methods over the classical algorithms. Our methods also should be usually faster than the Montgomery methods, since it is unlikely that there exists a microprocessor in which computing $9n^2$ multiplications is faster than computing $3n$ divisions.

In Table 1, we used the 3-segment KOA in the analyses of our methods, but we did not do the same in the analyses of the classical and the Montgomery algorithms. Note that typical software implementation of KOA leads to only 10%–20% improvement in the running time of long integer multiplications. Moreover such a noticeable improvement is expected only when the program is written very carefully and the operand size is very long (typically $>$ 32 blocks). However, in our method, it is easy to see that the running time for computing (4) is directly proportional to the required number of integer multiplications. For fair comparisons, in the last row in Table 1, we also provide the analysis results for our method assuming no 3-segment KOA is used in (3). Even without the KOA, our methods are still faster than the classical and the Montgomery algorithms.

3 The Ideas

In section 2, we have seen that efficient modular arithmetic is possible even though t is not a power of two. In this section, we discuss how the method shown in section 2 leads to fast modular arithmetic.

The proposed modular arithmetic is done in 3 steps as follows.

1. Polynomial multiplication (polynomials in $\mathbb{Z}[t]$)
2. Polynomial reduction by $f(t)$
3. Coefficient reduction (divisions by t)

It is clear that applying the KOA in polynomial multiplication step leads to a slight speed-up. Note that there could be similar *theoretical* improvements in the running times of the classical algorithms by applying the KOA. However, *in practice*, the KOA has significantly greater impact when it is applied to polynomial multiplications than when it is applied to integer multiplications.

The main source of speed-up is in the two reduction steps. Our method effectively splits one modular reduction into two steps: polynomial reduction step and coefficient reduction step. In polynomial reduction step, modular reduction is partially done using only a few integer additions and subtractions which are considered trivial. The rest of the reduction is done in coefficient reduction step. In that step, divisions are still unavoidable but many multiplications are saved.

The main idea of proposed modular arithmetic method is best described by Proposition 1.

Proposition 1. *Computing one long integer division takes more time than performing k short integer divisions where operands are shortened by the factor of k.*

Proof. We need to prove the following inequality.

$$D(a, b) > k \cdot D(\frac{a}{k}, \frac{b}{k}) \ , \tag{11}$$

where $a > b$ and $k > 1$. For simplicity, we assume that $k|a$ and $k|b$. Now it is easy to verify that,

$$D(a, b) - k \cdot D\left(\frac{a}{k}, \frac{b}{k}\right) = b(a-b)\frac{k-1}{k}t_m > 0 \ , \tag{12}$$

for $a > b$ and $k > 1$. □

What Proposition 1 says is that we can save about $b(a-b)(k-1)/k$ multiplications if we can break a long integer division into a number of short integer divisions. In practice, a should be about $2b$. Therefore the proposed modular arithmetic method saves about $b^2(k-1)/k$ multiplications, where k is the degree of $f(t)$ and b is the word length of p. In our example in section 2, the word length of p is $3n$ and $k = 3$. Hence, $6n^2$ multiplications are saved in reduction steps. Note that the number of saved multiplications increases as k increases. For large k, the number of saved multiplications gets close to b^2.

As shown in Proposition 1, now it appears to be clear that the proposed method is faster than the classical algorithms, since our method saves about $b^2(k-1)/k$ multiplications. Then, is the proposed method faster than the Montgomery algorithm? Note that the Montgomery algorithm uses the same number of multiplication as the classical algorithm but it does not use any division. If p is b words long, then the Montgomery algorithm saves b division instructions. Therefore, if $b^2(k-1)/k \cdot t_m > b \cdot t_d$, our method should be superior to the Montgomery algorithm. In our example in section 2, $b = 3n$ and $k = 3$. In such a case, if $2n$ multiplications take more time than single division, the proposed method should be faster than the Montgomery algorithm.

4 Desirable Properties of $f(t)$

We discuss which polynomials $f(t)$ are advantageous in our modular arithmetic method. The following is a list of desirable properties for good $f(t)$'s.

1. High degree polynomials.

   We have seen in section 3 that the efficiency of modular arithmetic increases as the degree of $f(t)$ increases. Hence it is advantageous to make the degree of $f(t)$ as high as possible. However using $f(t)$ of degree much higher than the word length of p should be avoided since it will increase the total number of multiplications and divisions. In fact, it is best to choose the degree of $f(t)$ such that the following conditions are satisfied.

$$\begin{aligned} \frac{\mathrm{word}(p)}{\deg(f)} &\approx l \cdot w \ , \\ \mathrm{word}(t) \cdot \deg(f) &\approx \mathrm{word}(p) \ , \end{aligned} \tag{13}$$

where word$(\cdot)$ denotes the word length of an integer, w is the processor's word size and $l > 0$ is an arbitrary positive integer. If the conditions (13) are not satisfied, the proposed modular arithmetic method could be slower than the classical algorithms.

2. Avoid using coefficients greater than 1.

   If any coefficient in $f(t)$ is greater than 1, it will introduce some constant multiplications by that coefficient value in polynomial reduction step.

3. Polynomials with small number of terms.

   It is easy to see that the number of terms in $f(t)$ has direct effect on the performance of polynomial reduction step. Suppose that degree k polynomial $f(t)$ has l terms. Then the number of additions or subtractions required to reduce polynomial of degree $2k-2$ is $(l-1)(k-1)$. Hence it is advantageous to choose $f(t)$ which has smaller number of terms.

4. Irreducible polynomials.

   For prime number generation, $f(t)$ should be chosen to be an irreducible polynomial. However, irreducible $f(t)$ does not guarantee the primality of $p = f(t)$.

It seems that choosing binomial $f(t) = t^k + 1$ is a good choice for most applications. Note that, for prime number generation, we have to choose k such that $f(t) = t^k + 1$ is irreducible. For example, $f(t) = t^3 + 1$ should not be used for prime generation since $f(t)$ is not an irreducible polynomial. In case $f(t) = t^k + 1$ is not irreducible, either choose a small integer $c \neq 0, 1$ which makes $f(t) = t^k + c$ irreducible or use polynomials other than binomials.

It should be noted that t in $f(t) = t^k + 1$ must be limited to even numbers in prime generation, since otherwise $p = f(t)$ is never a prime. This restriction does not occur in polynomials with odd number of terms (trinomial, pentanomial, *etc.*), provided that all coefficients are either 0 or 1. Thus such polynomials have a better chance to generate prime numbers.

5 Improvements

5.1 Improvement by Using $t = 2^n - c$ for Small c

In our method, we require k divisions by t in the coefficient reduction step. Note that division is an expensive operation in most microprocessors. However, for some special form of t, division can be avoided.

It is well-known that pseudo-Mersenne numbers $t = 2^n - c$, where c is a small (preferably positive) integer, lead to fast modular reduction methods [2, 5]. Here we derive an efficient algorithm which computes both remainder and quotient when $t = 2^n - c$.

Suppose an integer x is to be divided by t. Define $q_0 = \lfloor x/2^n \rfloor$ and $r_0 = (x \bmod 2^n) + c \cdot q_0$. Define q_i's and r_i's for $i > 1$,

$$\begin{aligned} q_i &= \left\lfloor \frac{r_{i-1}}{2^n} \right\rfloor \ , \\ r_i &= (r_{i-1} \bmod 2^n) + c \cdot q_i \ . \end{aligned} \tag{14}$$

Then we can write x as follows,

$$\begin{aligned} x &= q_0 \cdot t + r_0 \\ &= q_0 \cdot t + q_1 \cdot t + r_1 \\ &\ \ \vdots \\ &= \left(\sum_{i=0}^{s} q_i \right) \cdot t + r_s \ . \end{aligned} \tag{15}$$

Observe that the inequality,

$$r_{i-1} = r_i + t \cdot q_i > r_i \ , \tag{16}$$

where $i > 0$, must hold since q_i's are non-negative integers. Since r_i is a strictly decreasing sequence, there exists an integer s such that $r_s < t$. If $r_s < t$, the quotient is $\sum_{i=0}^{s} q_i$ and the remainder is r_s. Algorithm 1 is a formal description of the division method when $t = 2^n - c$. Note that the size of c has no effect on the performance of Algorithm 1 as long as the word length of c is fixed. Therefore it is easy to observe that the running time of Algorithm 1 is $O(n)$, assuming that c is a small, fixed integer.

Algorithm 1. Integer Division by $t = 2^n - c$

INPUT: Dividend $x \geq 0$.
OUTPUT: q and rem, such that $x = q \cdot t + rem$ and $0 \leq rem < t$.

1. $rem \leftarrow x$, $q \leftarrow 0$.

2. While $rem > t$ do:
   2.1 $A \leftarrow \lfloor rem/2^n \rfloor$.
   2.2 $q \leftarrow q + A$.
   2.3 $rem \leftarrow rem \bmod 2^n + c \cdot A$.
3. Return q and rem.

5.2 Avoiding Multiple Precision Operations

We propose using t whose bit length is a few bits shorter than processor's word size. In such a case, we could avoid multiple precision operations in modular arithmetic.

Suppose t is only a few bits shorter than a processor's word size. Then, when operands are expressed as polynomials in $\mathbb{Z}[t]$, all the coefficients are one word long. Therefore, in polynomial multiplication step, we only require processor's multiplication instructions, not the multiple precision multiplications. Since the bit length of t is slightly less than the word size, the results of the coefficient multiplications are stored in at most two words. Note that this does not cause problems, since many currently available processors support double word registers and instructions for simple arithmetic between double word operands. The results of coefficient multiplications should have enough bit margin in double word registers to prevent overflows and underflows during polynomial multiplication and polynomial reduction steps. The divisions in coefficient reduction step can also be implemented efficiently, since division instructions in most processors support dividing a double word dividend by a single word divisor.

We give a practical example. Suppose modular arithmetic operations are implemented on a processor where the word size is 32 bits long and the modulus p is about 160 bits long. Suppose 2-bit margin for t is required to prevent overflows and underflows during the modular arithmetic. Then t should be at most 30 bits long. It follows that the degree of $f(t)$ should be $\lceil 160/30 \rceil = 6$. Therefore we have to choose $\lceil 160/6 \rceil = 27$ bit t. In such a case, $p = f(t)$ will be about 162 bits long.

One drawback of this method is that it may increase the number of words required to store operands. In the above example, our method requires 6 word blocks, but originally only 5 words are sufficient. The increased word length may result in performance degradation. However, if the increment is not too large, this is not a serious issue. Usually there is a lot of unavoidable redundancy in the implementations of multiple precision arithmetic operations. In some cases when operand size is not too large, such redundancy overwhelms the running time of multiple precision operations. Hence, if the operand sizes are not too large, it is better to avoid multiple precision arithmetic, even though the increased word length may cause a slight drop in performance.

6 Advantages of Our Proposed Method

Note that one big issue for the generalized Mersenne numbers proposed in [8] is that there could be only one p for a given bit length and a polynomial $f(t)$. If $f(t)$ for some $t = 2^n$ is not useful, then we have no choice but to choose other polynomial $f(t)$ or change the bit length n of t. However our methods do not impose this kind of constraints. For a fixed $f(t)$, one can find many prime numbers $p = f(t)$ of the same bit length.

This makes significant difference in implementation. Note that there could be no general implementation for the generalized Mersenne numbers. In other words, each generalized Mersenne number requires a dedicated implementation. On the other hand, general implementation is possible in our case. If one implements modular arithmetic operations based on a pre-determined $f(t)$, then that implementation will work for any value of t. Even though $f(t)$ is not pre-determined, it is still easy to implement a software that works for any $f(t)$.

Since parameter generation is not simple, it is a usual practice in ECC to share a set of pre-determined parameters among domain users. In this case, one implementation will benefit the entire domain which shares common parameters, and hence general implementation may not be needed. In fact, there are only 5 NIST (National Institute of Standards and Technology)-recommended curves based on prime fields and they all use the generalized Mersenne primes [7, 6].

However every user has to generate their own parameters in RSA cryptosystem and it is recommended that users generate their own parameters in XTR cryptosystem [4]. The generalized Mersenne numbers are not advantageous in such cases. On the other hand, the proposed method given in this paper is highly suitable for RSA and XTR cryptosystems, since it is easy to implement a software that works for all possible choices of $f(t)$ and t. Note that the more generalized Mersenne numbers are characterized by only $f(t)$ and t. Hence we can reduce the storage requirement for the modulus $p = f(t)$.

Our more generalized Mersenne numbers are also advantageous due to the efficiency of modular arithmetic. As shown in section 2, our modular multiplication and squaring methods are much faster than the classical and the Montgomery arithmetic methods. Moreover, further speed-up is possible by using special form of $t = 2^n - c$ for small integer c or using t whose bit length is a few bits shorter than the word size.

7 Efficient Arithmetic for XTR Cryptosystem

In this section, we apply our methods to the modular arithmetic operations required in XTR cryptosystems. Algorithm 2 is the simplest and fastest algorithm for generating two essential parameters p and q for use in XTR cryptosystem.

Algorithm 2. Selection of q and p [4]

OUTPUT: XTR parameters q and p such that both are primes and $q|(p^2 - p + 1)$.
1. $r \in_R \mathbb{Z}$ such that $q = r^2 - r + 1$ is a prime.
2. $k \in_R \mathbb{Z}$ such that $p = r + kq = kr^2 + (1 - k)r + k$ is a prime and is $2 \pmod 3$.

The prime $p = kr^2+(1-k)r+k$, generated by Algorithm 2, has special structure. In [4], it is stated that such p results in fast GF(p) arithmetic. However they did not show details on how to achieve efficient arithmetic using such p. Here we show detailed methods and analysis for efficient arithmetic in GF(p).

7.1 Efficient Arithmetic in GF(p) When $k = 1$

When $k = 1$, p takes its simplest form, $p = r^2 + 1$. We express elements x, $y \in$ GF(p) as polynomials in $\mathbb{Z}[r]$.

$$\begin{aligned} x &\equiv x_1 r + x_0 \pmod{r^2 + 1} \ , \\ y &\equiv y_1 r + y_0 \pmod{r^2 + 1} \ , \end{aligned} \tag{17}$$

where $|x_i|$, $|y_i| < r$ for $i = 0, 1$.

Then the modular multiplication is computed as follows,

$$xy \equiv z_1 r + z_0 \pmod{r^2 + 1} \ , \tag{18}$$

where,

$$\begin{aligned} z_1 &= C - A - B \ , \\ z_0 &= A - B \ , \\ A &= x_0 y_0 \ , \\ B &= x_1 y_1 \ , \\ C &= (x_0 + x_1)(y_0 + y_1) \ . \end{aligned} \tag{19}$$

Since the bit lengths of x_i's and y_i's are about half the bit length of p, we need 3 half-length integer multiplications to compute (18). However the bit lengths of two resulting coefficients in (18), z_1 and z_0, can be about twice the bit length of t. They must be reduced before further multiplication or squaring operations in GF(p) are performed, since otherwise the size of the multiplication result will keep growing. Using two divisions by r, we represent z_1 and z_0 in the following form,

$$z_1 = z_{11}r + z_{10} \ , \quad z_0 = z_{01}r + z_{00} \ , \tag{20}$$

where $|z_{10}|$, $|z_{00}| < r$. Then it follows that,

$$\begin{aligned} xy &\equiv (z_{11}r + z_{10})r + z_{01}r + z_{00} \pmod{r^2+1} \\ &\equiv (z_{10} + z_{01})r + z_{00} - z_{11} \pmod{r^2+1} \ . \end{aligned} \tag{21}$$

Note that $|z_{10}+z_{01}|$ and $|z_{00}-z_{11}|$ could be greater than r. In such a case, we repeatedly apply (21) until the desired results are obtained. However after the first application of (21), the resulting coefficients have almost equal lengths with r. Thus, we only need additions and subtractions instead of expensive integer divisions.

7.2 Efficient Arithmetic in GF(p) When $k \neq 1$

When $k \neq 1$, $p = kr^2 + (1-k)r + k$. Note that the coefficients are not all 1. Then we have to express x and y as polynomials in $\mathbb{Z}[kr]$.

$$\begin{aligned} x &\equiv x_1(kr) + x_0 \pmod{kr^2 + (1-k)r + k} \ , \\ y &\equiv y_1(kr) + y_0 \pmod{kr^2 + (1-k)r + k} \ , \end{aligned} \tag{22}$$

where $|x_i|$, $|y_i| < kr$ for $i = 0$, 1. The modular multiplication is computed as follows,

$$xy \equiv z_1 \cdot (kr) + z_0 \pmod{f(r)} \ , \tag{23}$$

where,

$$\begin{aligned} z_1 &= C - A - 2B + D \ , \\ z_0 &= A - E \ , \\ A &= x_0 y_0 \ , \\ B &= x_1 y_1 \ , \\ C &= (x_1 + x_0)(y_1 + y_0) \ , \\ D &= Bk \ , \\ E &= Dk \ . \end{aligned} \tag{24}$$

Since z_1 and z_0 are not reduced, we express them as follows using two divisions,

$$z_1 = z_{11} \cdot (kr) + z_{10} \ , \quad z_0 = z_{01} \cdot (kr) + z_{00} \ , \tag{25}$$

where $|z_{10}|$, $|z_{00}| < r$. Then it follows that,

$$xy \equiv (z_{10} - z_{11} + z_{01} + kz_{11}) \cdot (kr) - k^2 z_{11} + z_{00} \pmod{f(r)} \ . \tag{26}$$

We repeatedly apply (26) until the desired result is obtained.

7.3 Analysis

We assume that r is stored in n word blocks. Then $p = kr^2 + (1-k)r + k$ should be about $2n$ words long. Let t_c denote the time required to compute the constant multiplication by k. We use the same analysis technique from section 2 and the results are shown in Table 2. Table 2 also compares our methods with the classical and the Montgomery algorithms.

Table 2. Running Time Comparison of Modular Arithmetic Methods

	Modular Multiplication	Modular Squaring
Our Method ($k = 1$)	$(5n^2 + 4n)t_m + 2nt_d$	$(4n^2 + 4n)t_m + 2nt_d$
Our Method ($k \neq 1$)	$(5n^2 + 4n)t_m + 2nt_d + 4t_c$	$(4n^2 + 4n)t_m + 2nt_d + 3t_c$
Classical	$(8n^2 + 4n)t_m + 2nt_d$	$(6n^2 + 5n)t_m + 2nt_d$
Montgomery	$(8n^2 + 4n)t_m$	$(6n^2 + 5n)t_m$

It is clearly seen in Table 2 that our methods save $3n^2$ and $2n^2$ multiplications for modular multiplication and modular squaring, respectively. Hence our methods are faster than the classical algorithms. Note that constant multiplications can be done efficiently if k is a very small integer or it has a low Hamming weight. Even though our methods use additional $2n$ divisions over the Montgomery algorithm, our modular multiplication methods should be faster if $3n^2$ multiplications are more expensive than $2n$ divisions. Similarly, if $2n^2$ multiplications are more expensive than $2n$ divisions, our modular squaring methods should be faster than the Montgomery algorithm.

8 Conclusions

In this paper, we further extended the generalized Mersenne numbers proposed in [8]. The generalized Mersenne numbers are expressed in polynomial form, $p = f(t)$ where t is a power of 2. However our generalization allows any integer for t. We have shown that efficient modular arithmetic is still possible even though t is not a power of 2. In our analysis, we have shown that our methods are much faster than the classical algorithm. We have also shown strong evidences that our methods are faster than the Montgomery algorithm. We have presented some criteria for good $f(t)$'s which lead to fast modular arithmetic.

Moreover, we have provided techniques for improving the modular arithmetic methods based on more generalized Mersenne numbers. One is using special form of $t = 2^n - c$ where c is a small integer. If such t is used, we can perform coefficient reduction steps using only constant multiplications. Another technique is using t whose bit length is slightly shorter than word size. In this case, the long integer modular arithmetic does not require multiple precision operations.

We have also shown efficient techniques for modular operations in XTR cryptosystem based on our results. Our analysis results show that the proposed methods should be faster than the classical and the Montgomery algorithms.

References

1. Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.
2. Richard E. Crandall. Method and apparatus for public key exchange in a cryptographic system (oct. 27, 1992). U.S. Paent # 5,159,632.
3. Donald E. Knuth. *Seminumerical Algorithms*. Addison-Wesley, 1981.
4. Arjen K. Lenstra and Eric R. Verheul. The XTR public key system. In *Advances in Cryptology - CRYPTO 2000*, LNCS 1880, pages 1–19. Springer-Verlag, 2000.
5. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
6. National Institute of Standards and Technology. Digital signature standard (DSS). FIPS Publication 186-2, February, 2000.
7. National Institute of Standards and Technology. Recommended elliptic curves for federal government use, July, 1999.
8. Jerome A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo, 1999. http://cacr.uwaterloo.ca/techreports/1999/corr99-39.ps.