Low-Weight Polynomial Form Integers for Efficient Modular Multiplication

Jaewook Chung and M. Anwar Hasan

February 9, 2006

Abstract

In 1999, Jerome Solinas introduced families of moduli called the generalized Mersenne numbers (GMNs), which are expressed in low-weight polynomial form, $p = f(t)$, where t is limited to a power of 2. GMNs are very useful in elliptic curve cryptosystems over prime fields, since only integer additions and subtractions are required in modular reductions. However, since there are not many GMNs and each GMN requires a dedicated implementation, GMNs are hardly useful for other cryptosystems. Here we modify GMN by removing restriction on the choice of t and restricting the coefficients of $f(t)$ to 0 and ± 1. We call such families of moduli low-weight polynomial form integers (LWPFIs). We show an efficient modular multiplication method using LWPFI moduli. LWPFIs allow general implementation and there exist many LWPFI moduli. One may consider LWPFIs as a trade-off between general integers and GMNs.

Index Terms

Cryptography, Mersenne numbers, modular multiplication, RSA, elliptic curve cryptosystems, Montgomery reduction, Barrett reduction.

I. Introduction

Modular multiplication is the main performance bottleneck in many cryptosystems, such as RSA [1], XTR [2] and the prime field based elliptic curve cryptosystems (ECC) [3], [4]. Hence, many algorithms have been proposed for implementing efficient modular multiplication. These algorithms can be classified into the following three categories:

Jaewook Chung and M. Anwar Hasan are with Department of Electrical and Computer Engineering, and with Centre for Applied Cryptographic Research at the University of Waterloo, Ontario, Canada.

1) Algorithms for general moduli: the classical algorithm [5], the Barrett algorithm [6] and the Montgomery algorithm [7].
2) Algorithms for special moduli: pseudo-Mersenne numbers [8] and generalized Mersenne numbers [9].
3) Look-up table methods: Kawamura, Takabayashi and Shimbo's method [10]; Hong, Oh and Yoon's method [11]; and Lim, Hwang and Lee's method [12].

Look-up table methods are normally faster than the generalized ones, but require a large amount of memory and are suitable only when some parameters are fixed. The modular multiplication method presented in this paper falls into the second category.

In 1644, Marin Mersenne conjectured that the numbers of the form $p = 2^k - 1$ are prime numbers for a certain set of integers $k \leq 257$. Although his conjecture turned out to be not entirely correct, the numbers of the form $p = 2^k - 1$ are now known as the *Mersenne numbers*. It is very easy to perform modular reduction using these numbers. However, these numbers are not attractive for cryptographic applications since there are very few Mersenne primes (e.g., if k is composite, Mersenne numbers are never primes) that are practically useful.

The moduli of the form $p = 2^k - c$, where c is a small integer, are known as *pseudo-Mersenne numbers* and they are patented by Richard Crandall [8]. Modular reduction using a pseudo-Mersenne number is also very efficient. However, because of security threats, these numbers are not recommended for cryptosystems that are based on the difficulty of integer factorization or discrete logarithm problem [13], [14].

In 1999, Jerome Solinas proposed generalized Mersenne numbers (GMNs). GMNs are expressed in low-weight polynomial form $p = f(t)$ where t is a power of 2 and the coefficients of low-degree polynomial $f(t)$ are very small compared to t. If the modulus is a GMN, the modular reduction requires simple integer additions and subtractions only. It is well known that all prime-field based elliptic curves recommended by NIST (National Institute of Standards and Technology) use GMNs [15], [16]. However, two significant shortcomings of GMNs are that there are not many useful GMNs and that each GMN requires dedicated implementation. Hence the use of GMN is currently limited to elliptic and hyper elliptic curve cryptosystems.

In this paper, a new family of integers, called the *low-weight polynomial form integers* (LWPFIs), is introduced. LWPFIs are similar to GMNs. However, for LWPFIs, t does not have to be a power of 2, and the coefficients of $f(t)$ are either 0 or ± 1. It will be shown in this paper that

an efficient modular multiplication based on LWPFI moduli can be implemented even though t is not a power of 2. Analysis and implementation results show that modular multiplication based on LWPFIs is asymptotically faster than any reduction algorithms for general moduli.

For software implementation, new modular multiplication based on LWPFI moduli can be implemented without using division instructions of the target processor. This feature is advantageous for processors whose division instruction is much slower than its multiplication instruction. Since LWPFI moduli are represented in polynomial form, their bit lengths are limited to a multiple of the degree of $f(t)$. However, extended LWPFIs make it possible to generate moduli of any bit length.

Since the publication of a preliminary version of this work at SAC 2003 [17], Bajard et al. have proposed two number systems called the adaptive modular number system (AMNS) [18] and the polynomial modular number systems (PMNS) [19]. These modular number systems have some similarities with LWPFIs in the sense that they use low-weight polynomial form moduli for efficient arithmetic and that numbers are represented in polynomial form. However, the representation of numbers and modular arithmetic in modular number systems are quite different from those in our modular multiplication using LWPFI moduli. In the modular number systems, an integer $x \in \mathbb{Z}_p$ is represented as a vector $(x_0, x_1, \ldots, x_{n-1})$, where $x = \sum_{i=0}^{n-1} x_i \gamma^i \bmod p$, $1 < \gamma < p$ and $x_i \in \{0, \ldots, \rho - 1\}$. Bajard et al. show that a careful choice of parameters, γ, ρ and p, makes arithmetic operations in the modular number systems efficient, and state that modular multiplication in AMNS is more efficient than Montgomery multiplication. However, the drawbacks of modular number systems are that the number of moduli for AMNS of practical use appears to be quite limited and that modular multiplications in PMNS require a large look-up table.

The remainder of this paper is organized as follows. First, we briefly review in Section II well known modular reduction algorithms and present a generalized version of the Barrett algorithm for integer division. In Section III, we describe how a modular multiplication using LWPFI moduli can be implemented efficiently. In Section IV, we discuss how to implement polynomial multiplication modulo $f(t)$ efficiently for some $f(t)$'s of small degrees. In Section V, we give detailed analysis of our LWPFI based modular multiplication scheme. We discuss practical considerations when using LWPFIs and show implementation results in Section VI. In Section VII, we show how our modular multiplication using LWPFIs can be improved.

Conclusions follow in Section VIII.

II. Overview of Modular Reduction Algorithms

In this section, we briefly discuss two well known modular reduction algorithms: the classical algorithm [5] and the Montgomery algorithm [7], [20], [21]. Then we present a generalization of the Barrett algorithm for modular reduction [6] [22]. Unlike the original Barrett algorithm, the generalized one does not have a limitation on the input size and can perform a multiple precision division for a fixed divisor.

Throughout this paper, we use the following notations:

- $b \geq 2$ is a radix for integer representation. In software implementation, $b = 2^w$ where w is the word-length in bits of the processor used.
- $x = (x_{n-1} \cdots x_1 x_0)_b$, where $0 \leq x_i < b$, is a radix-b representation of n-digit integer x.
- $\texttt{word}(x)$ denotes the number of words required to represent an integer x.

A. The Classical Algorithm

A good description and analysis of the classical algorithm for integer division (CAID) can be found in [23]; we have slightly modified this algorithm so that it accepts only normalized input, i.e., the most significant digit of the divisor $m_{k-1} \geq \lfloor b/2 \rfloor$. The resulting pseudo code is given in Algorithm 1.

Algorithm 1. Classical Algorithm for Integer Division (CAID)

INPUT: : Integers $x = (x_{n-1} \cdots x_1 x_0)_b$ and $m = (m_{k-1} \cdots m_1 m_0)_b$ with $n \geq k \geq 1$, $m_{k-1} \geq \lfloor b/2 \rfloor$.

OUTPUT: : The quotient $q = (q_{n-k} \cdots q_1 q_0)_b$ and the remainder $r = (r_{k-1} \cdots r_1 r_0)_b$ such that $x = qm + r$, $0 \leq r < m$.

1) For j from 0 to $(n-k)$ do: $q_j \leftarrow 0$.
2) If $x > mb^{n-k}$ then $q_{n-k} \leftarrow q_{n-k} + 1$, $r \leftarrow x - mb^{n-k}$; otherwise $r \leftarrow x$.
3) For i from $n-1$ down to k do the following:
   - 3.1 If $r_i = m_{k-1}$ then $q_{i-k} \leftarrow b - 1$; otherwise $q_{i-k} \leftarrow \lfloor (r_i b + r_{i-1})/m_{k-1} \rfloor$.
   - 3.2 While $(q_{i-k} m_{k-2} > (r_i b + r_{i-1} - q_{i-k} m_{k-1}) b + r_{i-2})$ do: $q_{i-k} \leftarrow q_{i-k} - 1$.
   - 3.3 $r \leftarrow r - q_{i-k} m b^{i-k}$.
   - 3.4 If $r < 0$ then $r \leftarrow r + mb^{i-k}$ and $q_{i-k} \leftarrow q_{i-k} - 1$.
4) Return q and r.

The input condition $m_{k-1} \geq \lfloor b/2 \rfloor$ guarantees that step 3.2 is repeated at most twice [5]. This condition can be met by left shifting x and m by a suitable number of bits. To obtain a correct result, we only need to shift the remainder r to the right by the same number of bits. Step 3.2 makes q_{i-k} to be at most one larger than the true value of quotient digit. The probability of $r < 0$ at step 3.4 is approximately $2/b$. Note that the values $q_{i-k}m_{k-1}$ and $q_{i-k}m_{k-2}$ in step 3.2 can be reused in step 3.3. Hence Algorithm 1 requires $k(n-k)$ single-precision multiplications and at most $(n-k)$ single-precision divisions.

B. The Montgomery Algorithm

The Montgomery algorithm performs modular reduction without using any division instruction of the underlying processor [7]. Let m be a modulus, and T be a positive integer which is to be reduced. We choose an integer R such that $R > m$, $\gcd(m, R) = 1$ and $0 \leq T < mR$. The Montgomery reduction is an operation which computes $TR^{-1} \mod m$. If R is chosen properly, then this modular reduction can be performed efficiently.

Let $m' = -m^{-1} \mod R$, and $U = Tm' \mod R$. Then $(T + Um)/R$ is an exact division. One can easily check that $(T + Um)/R \equiv TR^{-1} \pmod{m}$. Modular reduction and division by R can be done trivially if $R = b^k$ for some integer k. Since the condition $R > m$ must be met, k is usually chosen to be the number of digits in m.[1] Since $T < mR$ and $U < R$, it follows that $(T + Um)/R < 2m$. Hence a final subtraction may be needed depending on input T. The Montgomery algorithm for integer reduction (MAIR) is described in Algorithm 2.

Algorithm 2. The Montgomery Algorithm for Integer Reduction (MAIR)

INPUT: : $m = (m_{k-1} \cdots m_1 m_0)_b$ with $\gcd(m, b) = 1$, $R = b^k$, $m' = -m^{-1} \mod b$, and $T = (t_{2k-1} \cdots t_1 t_0)_b < mR$.

OUTPUT: : $A = TR^{-1} \mod m$.

1) $A = (a_{2k-1} a_{2k-2} \cdots a_0)_b \leftarrow T$.
2) For i from 0 to $(k-1)$ do the following:

   2.1 $u \leftarrow a_0 m' \mod b$.

   2.2 $A \leftarrow (A + um)/b$.

[1]However, in [24] and [25], to avoid the final subtraction, a value that is one or two more than the number of digits in m is proposed. Such techniques are useful in implementing cryptosystems, which resist timing attacks [26] better at a little cost in speed.

3) If $A \geq m$ then $A \leftarrow A - m$.

4) Return $A = (a_{k-1}a_{k-2}\cdots a_0)_b$.

Step 2.1 of MAIR requires one single-precision multiplication and step 2.2 requires k single-precision multiplications. Therefore, MAIR requires a total of $k(k+1)$ single-precision multiplications.

To use MAIR for computing modular multiplications, input values must be transformed to the so-called *Montgomery domain*. To transform an integer $x \in \mathbb{Z}_m$ to the Montgomery domain, we compute $(x \cdot R^2) \cdot R^{-1} \bmod m$ using a pre-computed value $R^2 \bmod m$. Hence one multiple-precision multiplication and one execution of MAIR are required.

To compute modular multiplication $x \cdot y \bmod m$ using MAIR, first we have to transform x and y to the Montgomery domain, $\bar{x} = x \cdot R \bmod m$, $\bar{y} = y \cdot R \bmod m$. Then we multiply them in the Montgomery domain, which results in $\bar{x}\bar{y} \equiv xy \cdot R^2 \pmod{m}$. One execution of MAIR will bring it back to an integer in the Montgomery domain; that is, $\bar{x}\bar{y} \cdot R^{-1} \equiv xy \cdot R \pmod{m}$. Another execution of MAIR on $xy \cdot R \pmod{m}$ will result in $xy \pmod{m}$. It is not at all efficient to use MAIR just to compute one modular multiplication. However, MAIR is very efficient when many modular multiplications are to be performed, since, for example, in modular exponentiation, one needs transformations only at the beginning and at the end.

C. The Barrett Algorithm

The Barrett algorithm [6] [22] is advantageous for applications in which a fixed modulus is used. It does not use any division instructions of the underlying processor, but it uses a small amount of pre-computation of size similar to that of the modulus. The description given in Algorithm 3 is a generalized version of the Barrett algorithm. We refer to it as GBAID (the generalized Barrett algorithm for integer division) since it has been modified such that a quotient is also computed. The original Barrett algorithm can reduce integers that are at most twice as long as a modulus. However, GBAID does not have such a limitation. Note that, Algorithm 3 becomes the original Barrett algorithm for integer reduction if we let $u = 2v$, remove "$q \leftarrow q+1$" in step 4, and change step 5 to "Return r".

Algorithm 3. The Generalized Barrett Algorithm for Integer Division (GBAID)

INPUT: Positive integers $x = (x_{u-1}\cdots x_1x_0)_b$, $m = (m_{v-1}\cdots m_1m_0)_b$ with $m_{v-1} \neq 0$, $u \geq v$ and a pre-computed

value $\mu = (\mu_{u-v} \cdots \mu_1 \mu_0)_b = \lfloor b^u/m \rfloor$.

OUTPUT: Integers q and r such that $x = qm + r$ where $r < m$.

1) $q \leftarrow \left\lfloor \sum_{u-v-1 \leq i+j} x_{i+v-1} \mu_j b^{i+j-u+v+1} / b^2 \right\rfloor (\approx \lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \rfloor)$.
2) $r_1 \leftarrow x \bmod b^{v+1}$, $r_2 \leftarrow \sum_{i+j<v+1} q_i m_j b^{i+j} \bmod b^{v+1} (= q \cdot m \bmod b^{v+1})$, $r \leftarrow r_1 - r_2$.
3) If $r < 0$ then $r \leftarrow r + b^{v+1}$.
4) While $r \geq m$ do: $r \leftarrow r - m$ and $q \leftarrow q + 1$.
5) Return q and r.

The computation of q in step 1 of Algorithm 3 is not exact, but it is quite accurate. Let $q' = \lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \rfloor$. Then q computed in step 1 is an approximation of q'. The approximation error $q' - q$ is at most 1 when $u - v \leq b$ [23], [27] and a proof is given in Appendix A. Let Q denote $\lfloor x/m \rfloor$. Then it can be easily shown that $Q - 2 \leq q' = \lfloor \lfloor x/b^{v-1} \rfloor \mu / b^{u-v+1} \rfloor \leq Q$.

$$
\begin{aligned}
q' &> \frac{1}{b^{u-v+1}} \cdot \left(\frac{b^u}{m} - 1 \right) \cdot \left(\frac{x}{b^{v-1}} - 1 \right) - 1, \\
&= \frac{x}{m} - \frac{b^{v-1}}{m} - \frac{x}{b^u} + \frac{1}{b^{u-v+1}} - 1, \\
&\geq Q - 2.
\end{aligned} \tag{1}
$$

Trivially, $q' \leq Q$.

Let $k = u - v$ for simplicity of description; then both μ and $\lfloor x/b^{v-1} \rfloor$ are at most $k + 1$ words long. It can be easily seen that step 1 requires at most $(k^2 + 5k + 2)/2$ single-precision multiplications. Note that q computed in step 1 is also at most $k + 1$ words long. One can easily verify that the number of single-precision multiplications required in step 2 is at most $(k + 1)v - k(k - 1)/2$ if $k \leq v$, $(v^2 + 3v)/2$ otherwise; therefore, the total number of single-precision multiplications required in Algorithm 3 is $uv + 3u - v^2 - 2v + 1$ if $u \leq 2v$, $(u^2 + 5u)/2 - uv + v^2 - v + 1$ if $u > 2v$.

D. Comments

In Table I, we show results of our analysis of CAID, MAIR and GBAID on the basis of number of single-precision multiplications and divisions. Note that while any of these three schemes can be used for modular reduction, only CAID and GBAID can perform a long integer division to output the quotient. As it can be seen in Table I, MAIR is more advantageous than CAID since a typical microprocessor's division instruction is slower than its multiplication instruction

TABLE I

ANALYSIS OF ALGORITHMS 1, 2 AND 3 ($\texttt{word}(m) = n$ AND $\texttt{word}(x) = 2n$)

Algorithm	# Mul	# Div	Pre-Computation
CAID	n^2	n	No
MAIR	$n^2 + n$	0	$R^2 \bmod m$ and $m' = -m^{-1} \bmod b$
GBAID	$n^2 + 4n + 1$	0	$\mu = \lfloor b^u/m \rfloor$

TABLE II

INSTRUCTION TIMING OF PENTIUM PROCESSORS

Processor	$\texttt{mul}$ (t_m)	$\texttt{div}$ (t_d)	t_d/t_m
Pentium II @ 350MHz	14.3ns	111.7ns	7.81
Pentium III Mobile @ 1.13GHz	4.6ns	35.7ns	7.76
Pentium IV @ 3.2GHz (Family 7, Model 4)	3.45ns	23.86ns	6.92

(see Table II for instruction timings on Pentium processors[2]). MAIR is more advantageous than GBAID also. In microprocessors where the ratio of division to multiplication instruction is approximately greater than 4, CAID uses more time for multiplication and division instructions. Note that the ratios shown in Table II are all greater than 4.

The above analysis is based on the number of multiplications and divisions only. The three algorithms being considered here, however, require other instructions and various implementation overheads. In order to have a more realistic comparison, we have implemented these algorithms and performed their timing analysis. Details of our implementation environment are given in Section VI-A, but timing results for varying bit sizes of moduli are presented in Figure 1. The figure shows that MAIR is the best reduction method among the three algorithms considered in this section. As a division algorithm, GBAID performs faster than CAID on Pentium 4 @ 3.2GHz. An important feature, not shown in Figure 1, of GBAID is that for a long integer division, GBAID is more advantageous than CAID if many divisions are to be performed for a fixed divisor. It is the case when our modular multiplication method presented in the following section is used in a modular exponentiation algorithm. We remark that Bosselaers et al. have

[2]The timing data in this table is the average time between the beginning of a corresponding multiplication or division instruction to the next dependent instruction over 2.5×10^9 executions using random operands.

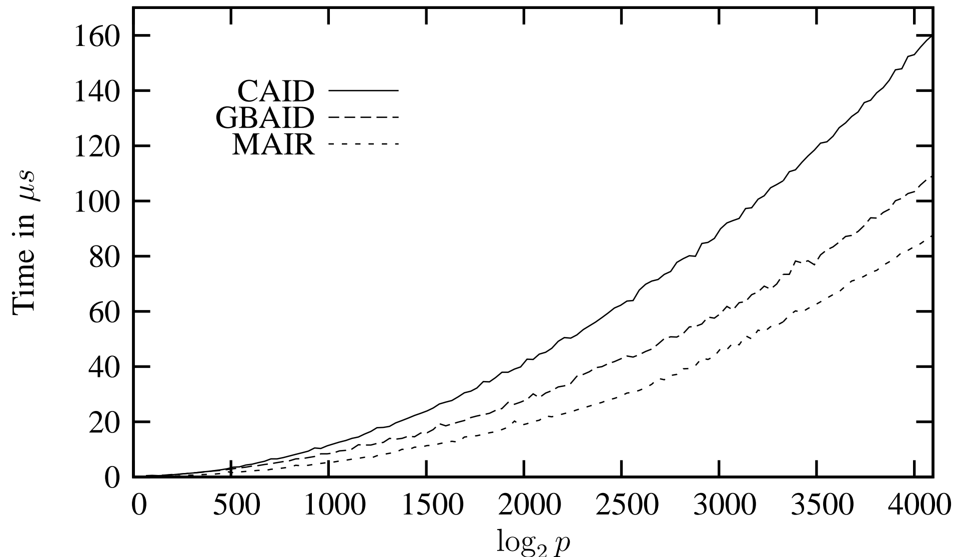

Fig. 1. Timing of Division Methods on Pentium 4 @ 3.2GHz

presented more detailed research on the three reduction methods in [28].

III. Modular Multiplication Using Low-Weight Polynomial Form Integers

In [9], Solinas has proposed generalized Mersenne numbers (GMNs) for efficient modular multiplication. A GMN is expressed as a low-weight polynomial $f(t)$, where t is a power of 2, and $f(t)$ is a small-degree polynomial. An LWPFI is also expressed as a polynomial $f(t)$, but t is not necessarily restricted to a power of 2, and the coefficients of $f(t)$ are limited to 0 and ± 1. Even though allowing only 0 and ± 1 for the coefficients of $f(t)$ leaves 3^l possible choices for $f(t)$, allowing any integer for t gives far more choices of integers than does GMN.

Definition 1 (LWPFI) *For a positive integer* t*, let* $f(t) = t^l - f_{l-1}t^{l-1} - f_{l-2}t^{l-2} - \cdots - f_1 t - f_0$ *be a monic polynomial of degree* l*. We call a positive integer* $p = f(t)$ *a* **low-weight polynomial form integer (LWPFI)** *if* $f_i \in \{-1, 0, 1\}$*,* $l \geq 2$ *and* $t > 2(2^{2l+1} - 1)(2^l - 1) \approx 2^{3l+2}$.

In Definition 1, the value $l = 1$ is excluded so that LWPFIs are different from the usual form of integers, and the reason for having the condition $t > 2(2^{2l+1} - 1)(2^l - 1)$ is explained in Section III-B. In practice, the condition $t > 2(2^{2l+1} - 1)(2^l - 1)$ is easily satisfied. For cryptographically useful values of $p = f(t)$, the degree l of $f(t)$ is a very small integer ($l =$

$2, 3, 4, \ldots$) and t is a large integer (at least $t > 2^w$, where w is the processor's word length in bits).

When computing modular arithmetic using LWPFI moduli, operands must be expressed in degree $l-1$ polynomial form. For an integer $x \in \mathbb{Z}_{p=f(t)}$, we use the following special form of redundant signed-digit representation,

$$x(t) = (x_{l-1} \cdots x_1 x_0)_{SD-(t,\psi)}, \tag{2}$$

where $|x_i| \leq \psi = (t + 2^{l+1} - 2)$ and

$$x \equiv x_{l-1}t^{l-1} + \cdots + x_1 t + x_0 \pmod{p = f(t)}. \tag{3}$$

For any $x \in \mathbb{Z}_p$, such x_i's for $i = 0, \ldots, l-1$ exist. For simplicity, we say a representation of an integer x is in ***SD-(t, ψ) form*** if it is in the above form. Note that we have chosen a slightly wider range $|x_i| \leq (t + 2^{l+1} - 2)$ than $|x_i| < t$, which is used in traditional signed-digit redundant representation [29]. The use of this wider range makes it possible to simplify our modular multiplication method using LWPFI moduli described later in this section. Given two input values in SD-(t, ψ) form, our modular multiplication method computes an output also in SD-(t, ψ) form.

Converting an integer in $\mathbb{Z}_{f(t)}$ into an SD-(t, ψ) form requires no more than $l-1$ integer divisions by t, where $l = \deg f(t)$. Usually this requirement is not an issue in cryptographic applications since, when the modular multiplication algorithm based on LWPFI moduli is used in exponentiation, conversions between the usual representation and an SD-(t, ψ) form is not significant compared to the entire exponentiation.

Let x, $y \in \mathbb{Z}_{p=f(t)}$ be represented in SD-(t, ψ) form as follows:

$$\begin{aligned} x(t) &= (x_{l-1} \cdots x_1 x_0)_{SD-(t,\psi)}, \\ y(t) &= (y_{l-1} \cdots y_1 y_0)_{SD-(t,\psi)}. \end{aligned} \tag{4}$$

In the following, we show an efficient way to perform modular multiplication of these two integers modulo an LWPFI $p = f(t)$; that is, $x(t) \cdot y(t) \bmod f(t)$. We call the proposed scheme the *LWPFI modular multiplication*.

The LWPFI modular multiplication is performed in two steps:

1) POLY-MULT-REDC: compute $\hat{z}(t) = x(t) \cdot y(t) \bmod f(t)$.

2) COEF-REDC: reduce coefficients of $\hat{z}(t)$, such that the resulting polynomial has coefficients that are at most ψ in magnitude.

A. POLY-MULT-REDC: Multiplication in $\mathbb{Z}[t]/f(t)$

Algorithm 4. Polynomial Multiplication & Reduction (POLY-MULT-REDC)

INPUT: $x(t)$ and $y(t)$ in SD-(t, ψ) form

OUTPUT: $\hat{z}(t) = x(t) \cdot y(t) \bmod f(t)$.

1) $z(t) = (z_{2l-2} \cdots z_1 z_0)_{SD-(t,*)} = x(t) \cdot y(t)$.
2) For i from $2l-2$ down to l do the following:
   2.1 $z(t) = z(t) + z_i \cdot (f_{l-1} f_{l-2} \cdots f_0)_{SD-(t,*)} \cdot t^{i-l}$.
3) $\hat{z}(t) = (z_{l-1} \cdots z_1, z_0)_{SD-(t,*)}$.
4) Return $\hat{z}(t)$.

Algorithm 4 is the most simple and general way to perform the POLY-MULT-REDC step. Step 1 is a multiplication of two l-term polynomials and can be computed in many different ways, requiring different amount of computation as discussed in Section V-A. Step 2 performs a polynomial reduction of a degree-$(2l-2)$ polynomial by $f(t)$. Note that it is only a general polynomial reduction method that works for any $f(t)$. For specific $f(t)$'s, one may find better ways to do this step.

Even though polynomial multiplication and polynomial reduction are separated in Algorithm 4, one can choose to combine them for better performance. In Section IV, we show how Algorithm 4 can be optimized by combining polynomial multiplication and polynomial reduction.

Proposition 1 *Suppose that the magnitude of the coefficients in $x(t)$ and $y(t)$ are bounded by a positive integer ψ. Then the coefficients of $\hat{z}(t)$ computed by Algorithm 4 are at most $(2^l - 1)\psi^2$ in magnitude.*

Proof: Let $z(t) = x(t) \cdot y(t)$. It is easily seen that $|z_i| \le (i+1)\psi^2$ for $i = 0, \ldots, l-1$ and $|z_i| \le (2l-1-i)\psi^2$ for $i = l, \ldots, 2l-2$. The magnitude of the coefficients in $\hat{z}(t) = z(t) \bmod f(t)$ is maximum when all the coefficients f_i's of $f(t)$ are either 1 or -1. In both cases, the maximum value of $|\hat{z}_i|$ is computed as $(2^l - 2^{l-i-1})\psi^2$. Therefore, $|\hat{z}_i| \le (2^l - 1)\psi^2$ for all $i = 0, \ldots, l-1$. ■

B. COEF-REDC: Coefficient Reduction

After POLY-MULT-REDC is completed, we obtain a degree $(l-1)$ polynomial $\hat{z}(t)$. As shown in Proposition 1, the bit lengths of $\hat{z}_i$'s could be more than twice as long as that of t. The coefficients $\hat{z}_i$'s must be reduced so that the result can be used as input to subsequent modular multiplications. Algorithm 5 shows an efficient way to reduce the coefficients of $\hat{z}(t)$, where we used $\lfloor\cdot\rceil$ to denote truncation toward zero, and $u \,\texttt{rem}\, v$ to denote $u - v\cdot\lfloor u/v\rceil$.

Algorithm 5. Coefficient Reduction (COEF-REDC)

INPUT: $\hat{z}(t) = (\hat{z}_{l-1}\cdots\hat{z}_1\hat{z}_0)_t$, where $|\hat{z}_i| \le (2^l-1)\psi^2$ for all $i = 0,\ldots,l-1$.

OUTPUT: $z'(t) = (z'_{l-1}\cdots z'_1 z'_0)_{SD-(t,\psi)}$.

1) $z'_l \leftarrow 0$, $z'(t) \leftarrow \hat{z}(t)$.
2) $a \leftarrow \lfloor z'_{l-1}/t\rceil$, $z'_{l-1} \leftarrow z'_{l-1} \,\texttt{rem}\, t$.
3) $z'(t) \leftarrow z'(t) + a\cdot(f_{l-1}f_{l-2}\cdots f_0)_{SD-(t,*)}$.
4) For i from 0 to $l-1$, do the following:
   - 4.1 $z_{i,1} \leftarrow \lfloor z'_i/t\rceil$ and $z_{i,0} \leftarrow z'_i \,\texttt{rem}\, t$.
   - 4.2 $z'_i \leftarrow z_{i,0}$, $z'_{i+1} \leftarrow z'_{i+1} + z_{i,1}$.
5) $z'(t) \leftarrow z'(t) + z'_l\cdot(f_{l-1}f_{l-2}\cdots f_0)_{SD-(t,*)}$.
6) Return $z'(t)$.

Below we show that Algorithm 5 results in SD-(t,ψ) form output.

Proposition 2 *Suppose that the coefficients of $\hat{z}(t)$ satisfy $|\hat{z}_i| \le (2^l-1)\psi^2$, where $\psi = t+2^{l+1}-2$. Given this input $\hat{z}(t)$, Algorithm 5 outputs $z'(t)$, whose coefficients are no greater than ψ in magnitude.*

Proof: Let $\theta = 2^{l+1}-2$. Then it follows that

$$(2^l-1)(\theta^2+4\theta+2) = 2(2^l-1)(2^{2l+1}-1) < t, \tag{5}$$

due to Definition 1. We use (5) throughout this proof.

In step 2, since $|z'_{l-1}| \le (2^l-1)(t+\theta)^2$, it is easy to see that

$$|a| \le \lfloor(2^l-1)(t+\theta)^2/t\rceil = (2^l-1)(t+2\theta), \qquad (\because (2^l-1)\theta^2 < t) \tag{6}$$

where $\lfloor\cdot\rceil$ is a truncation toward zero. After step 3,

$$\begin{aligned} |z'_i| &\le (2^l-1)[(t+\theta)^2+t+2\theta] \text{ for } i = 0,\ldots,l-2, \\ |z'_{l-1}| &\le (2^l-1)(t+2\theta)+t-1. \end{aligned} \tag{7}$$

In step 4, the maximum values of $z'_{i,1}$ are determined as follows:

$$|z'_{0,1}| \leq \left| \left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta]}{t} \right\rceil \right| \leq (2^l-1)(t+2\theta+1), \tag{8}$$

since $(2^l-1)(\theta^2+2\theta) < t$. We consider three cases where $l=2$, $l=3$ and $l>3$.

1) Case 1: if $l=2$,

$$\begin{aligned} |z'_2| = |z'_{1,1}| &= \left| \left\lfloor \frac{z'_1 + z'_{0,1}}{t} \right\rceil \right| \\ &\leq \left| \left\lfloor \frac{(2^l-1)(t+2\theta+t+2\theta+1)+t-1}{t} \right\rceil \right| \leq (2^{l+1}-1). \\ &(\because (2^l-1)(4\theta+1)-1 < t) \end{aligned} \tag{9}$$

2) Case 2: if $l=3$,

$$\begin{aligned} |z'_{1,1}| &\leq \left| \left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+1]}{t} \right\rceil \right| \leq (2^l-1)(t+2\theta+2), \\ |z'_3| = |z'_{2,1}| &\leq \left| \left\lfloor \frac{(2^l-1)(t+2\theta+t+2\theta+2)+t-1}{t} \right\rceil \right| \leq (2^{l+1}-1). \end{aligned} \tag{10}$$

3) Case 3: if $l>3$,

$$\begin{aligned} |z'_{1,1}| &\leq \left| \left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+1]}{t} \right\rceil \right| \leq (2^l-1)(t+2\theta+2), \\ |z'_{2,1}| &\leq \left| \left\lfloor \frac{(2^l-1)[(t+\theta)^2+t+2\theta+t+2\theta+2]}{t} \right\rceil \right| \leq (2^l-1)(t+2\theta+2), \\ &\vdots \\ |z'_l| = |z'_{l-1,1}| &\leq \left| \left\lfloor \frac{(2^l-1)(t+2\theta+t+2\theta+2)+t-1}{t} \right\rceil \right| \leq (2^{l+1}-1). \end{aligned} \tag{11}$$

Hence, $|z'_l| \leq (2^{l+1}-1)$ after step 4 of Algorithm 5 for all $l \geq 2$. Since $|z'_i| \leq t-1$ for $i=0,\ldots,l-1$ after step 4, the magnitudes of $|z'_i|$'s for $0 \leq i < l$ will be no greater than $\psi = t + 2^{l+1} - 2$ after the execution of step 5. Therefore, the output of Algorithm 5 is in SD-(t,ψ) form. ■

Algorithm 5 is much like the modular reduction algorithm using pseudo-Mersenne numbers [23]. However, Algorithm 5 is quite different from it since Algorithm 5 does not require a "while" loop, the reason being that the output of Algorithm 5 is reduced only to the point where the output meets the conditions for SD-(t,ψ) form. This feature makes Algorithm 5 behave in a completely deterministic way, that is, its performance is not random or input-value dependent.

IV. Optimization of POLY-MULT-REDC Step

In this section, we show that the POLY-MULT-REDC step can be implemented efficiently for some specific $f(t)$'s by combining polynomial multiplication and polynomial reduction by $f(t)$. We provide optimal $f(t)$'s for implementing the POLY-MULT-REDC step for $l = 2$ and 3. It will be shown in Section V that larger values of l lead to a better asymptotic bound; however, they introduce more overheads. We consider only small-degree $f(t)$'s that are useful in practice. It is straightforward however to extend this idea to larger degrees of $f(t)$.

The combining methods shown in this section are more efficient than the multiply-then-reduce method described in Algorithm 4. For $l = 2$, the combining method's performance is almost as good as that of polynomial multiplication only. Moreover, polynomial squaring in $\mathbb{Z}[t]/f(t)$ when $l = 2$ is asymptotically faster than polynomial multiplication for some $f(t)$'s. For $l = 3$, some $f(t)$'s make it possible that combined polynomial multiplication and polynomial reduction can be performed using the same number of operations as for polynomial multiplication only.

The methods shown in this section are to optimize Algorithm 4 for $l = 2$ and 3. The resulting output of the following methods will be identical to that of Algorithm 4 for the same input. Thus, the polynomials computed by the following methods will meet the input condition of Algorithm 5 too, provided that the input $x(t)$ and $y(t)$ in this section are also in SD-(t, ψ) form. However, computations in Algorithm 4 and the methods in this section do not depend on the fact that the input is in SD-(t, ψ).

We only consider irreducible $f(t)$'s. Reducible $f(t)$'s are guaranteed to generate composite numbers that are, in most cases, not useful for cryptography. When $f(t)$ is reducible there are better ways to perform polynomial multiplications in $\mathbb{Z}[t]/f(t)$. In particular, for $f(t) = \prod_{i=1}^{k} f_i(t)$, where $f_i(t)$'s are irreducible factors of $f(t)$, the minimum number of multiplications required to compute a polynomial multiplication in $\mathbb{Z}[t]/f(t)$ is $2 \cdot \deg f(t) - k$ [30].

A. *Case 1:* $l = 2$

We use KOA for 2-term polynomial multiplication. We consider two degree-2 polynomials $x(t) = (x_1 x_0)_{SD-(t,\psi)}$ and $y(t) = (y_1 y_0)_{SD-(t,\psi)}$. KOA computes $x(t) \cdot y(t)$ using only three multiplications.

$$x(t) \cdot y(t) = x_1 y_1 t^2 + ((x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0)t + x_0 y_0. \tag{12}$$

After polynomial reduction by $f(t)$, we have the following formula for polynomial multiplication and squaring in $\mathbb{Z}[t]/f(t)$.

$$x(t)\cdot y(t) \equiv ((x_0+x_1)(y_0+y_1)+(f_1-1)x_1y_1-x_0y_0)t+f_0x_1y_1+x_0y_0 \pmod{f(t)}. \quad (13)$$

$$x(t)^2 \equiv ((x_0+x_1)^2+(f_1-1)x_1^2-x_0^2)t+f_0x_1^2+x_0^2 \pmod{f(t)}. \quad (14)$$

Or, we can obtain alternative formulae by using the following version of KOA due to Knuth [5]:

$$x(t)\cdot y(t) = x_1y_1t^2+(x_1y_1+x_0y_0-(x_0-x_1)(y_0-y_1))t+x_0y_0. \quad (15)$$

The following formulae are obtained by taking modulo $f(t)$ of (15).

$$x(t)\cdot y(t) \equiv ((f_1+1)x_1y_1+x_0y_0-(x_0-x_1)(y_0-y_1))t+f_0x_1y_1+x_0y_0 \pmod{f(t)}. \quad (16)$$

$$x(t)^2 \equiv ((f_1+1)x_1^2+x_0^2-(x_0-x_1)^2)t+f_0x_1^2+x_0^2 \pmod{f(t)}. \quad (17)$$

Note that (13) and (14) are good when $f_1 = 1$ and (16) and (17) are good when $f_1 = -1$. Interestingly, when $f_0 = -1$, we can simplify (14) and (17) as follows:

$$x(t)^2 \equiv x_1(f_1x_1+2x_0)t+(x_0-x_1)(x_0+x_1) \pmod{f(t)}. \quad (18)$$

Formula (18) needs only two multiplications. Long integer squaring is usually faster than long integer multiplication. As long as integer squaring takes no less than 2/3 of multiplication time, (18) is faster than (14) and (17).

We find that $f(t) = t^2 \pm t + 1$ and $f(t) = t^2+1$ are the most attractive choices for $l = 2$. Note that $f_1 = 0$ reduces one addition in (18), but $f_1 = \pm 1$ reduces one double length addition in (13) and (16), respectively. However, modular multiplications occur less frequently than modular squaring in exponentiation algorithms. For example, the binary exponentiation algorithm requires twice more modular squarings than modular multiplications on average. So, both $f(t) = t^2 \pm t + 1$ and $f(t) = t^2+1$ result in the same speed up. For exponentiation methods that use less number of multiplications than the binary algorithm, $f(t) = t^2+1$ is preferred.

B. Case 2: $l = 3$

For 3-term polynomials, the following 3-way method requires six multiplications [31] as follows:

$$\begin{aligned} x(t) \cdot y(t) = & D_2 \cdot t^4 \\ & + (D_2 + D_1 - D_5) \cdot t^3 \\ & + (D_2 + D_1 + D_0 - D_4) \cdot t^2 \\ & + (D_1 + D_0 - D_3) \cdot t \\ & + D_0, \end{aligned} \tag{19}$$

where

$$\begin{aligned} D_0 &= x_0 y_0, & D_3 &= (x_0 - x_1)(y_0 - y_1), \\ D_1 &= x_1 y_1, & D_4 &= (x_0 - x_2)(y_0 - y_2), \\ D_2 &= x_2 y_2, & D_5 &= (x_1 - x_2)(y_1 - y_2). \end{aligned} \tag{20}$$

After polynomial reduction, we have the following result:

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv ([f_2(f_2+1) + (f_1+1)] \cdot D_2 + (f_2+1) \cdot D_1 + D_0 - f_2 \cdot D_5 - D_4) \cdot t^2 \\ & \qquad + ([f_1(f_2+1) + f_0] \cdot D_2 + (f_1+1) \cdot D_1 + D_0 - f_1 \cdot D_5 - D_3) \cdot t \\ & \qquad + (f_0(f_2+1) \cdot D_2 + f_0 \cdot D_1 + D_0 - f_0 \cdot D_5). \end{aligned} \tag{21}$$

Among all combinations of (f_2, f_1, f_0) that make $f(t)$ irreducible, $(f_2, f_1, f_0) = (-1, -1, 1)$ and $(0, -1, 1)$ put (21) into the simplest form.

For $f(t) = t^3 + t^2 + t - 1$,

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv (D_0 - D_4 + D_5) \cdot t^2 + (D_0 + D_2 - D_3 + D_5) \cdot t + (D_0 + D_1 - D_5). \end{aligned} \tag{22}$$

For $f(t) = t^3 + t - 1$,

$$\begin{aligned} & x(t) \cdot y(t) \pmod{f(t)} \\ & \qquad \equiv (D_0 + D_1 - D_4) \cdot t^2 + (D_0 - D_3 + D_5) \cdot t + (D_0 + D_1 + D_2 - D_5). \end{aligned} \tag{23}$$

It is interesting to observe that the computational cost of each of (22) and (23) is exactly the same as that of (19).

V. Analysis of LWPFI Modular Multiplication

In this section, the performance of LWPFI modular multiplication described in Section III is analyzed. In our analysis, we use n to denote the bit length used for $t + 2^{l+2} - 2$; that is, $t+2^{l+2}-2 < 2^n$. In practice, t will be quite larger than $2^{l+2}-2$, hence both t and $\psi = t+2^{l+1}-2$ are almost always n-bit integers. We use τ to denote the number of non-zero f_i's in $f(t)$. The following notations are used in our analysis of Algorithms 4 and 5.

- $T_m(u)$: time needed for multiplying two u-bit integers.
- $T_a(u)$: time needed for adding/subtracting u-bit integers.
- $T_d(u, v)$: time needed for dividing a u-bit integer by a v-bit integer.

We will use an assumption that adding a u-bit integer to a v-bit integer takes $T_a(\min(u, v))$ time. This is a reasonable assumption for most software implementations. A carry at the top most bit-position of the shorter integer may occur when adding two integers and it may increase the computation time slightly. However, the carry occurs with probability $\approx 1/2$ when adding two random integers and the probability that the carry will propagate more than one word is only $1/2^w$, where w is the bit size of a computer word.

A. *POLY-MULT-REDC step*

POLY-MULT-REDC takes two polynomials in SD-(t, ψ) form as input; that is, the coefficients of the two input polynomials are at most $\psi < t + 2^{l+2} - 2 < 2^n$ in magnitude.

There are many different ways to perform POLY-MULT-REDC step. Algorithm 4 is the most straightforward and general approach. If the schoolbook method is used for polynomial multiplication in step 1, l^2 multiplications and $(l-1)^2$ additions are required. The polynomial reduction, step 2, requires $\tau(l-1)$ additions.

Clearly, the multiplications among coefficients are all n-bit wide. For integer additions, the bit lengths of operands are not the same. However, regardless of the method used for polynomial multiplication and polynomial reduction, the output $\hat{z}(t)$ of Algorithm 4 will have coefficients that are at most $(2^l - 1)\psi^2$ in magnitude, which is at most an $(l + 2n)$-bit integer. Hence, for simplicity, we assume that all the integer additions are $(l + 2n)$ bits wide. As a result, we have the upper bound for the running time of Algorithm 4 as follows:

$$T(\text{POLY-MULT-REDC}) \le l^2 \cdot T_m(n) + (l + \tau - 1)(l - 1) \cdot T_a(l + 2n). \tag{24}$$

TABLE III

MODULAR MULTIPLICATION AND SQUARING COST IN $\mathbb{Z}[t]/f(t)$ FOR ALL IRREDUCIBLE DEGREE-2 $f(t)$'S

$f(t)$	Multiplication modulo $f(t)$	Squaring modulo $f(t)$
t^2+1	$3M+3A+2a$	$2M+2a+h$
t^2+t+1	$3M+2A+2a$	$2M+3a+h$
t^2-t+1	$3M+2A+2a$	$2M+3a+h$
t^2+t-1	$3M+2A+2a$	$3S+2A+a$
t^2-t-1	$3M+2A+2a$	$3S+2A+a$

Instead of the schoolbook method, other methods can be used for the multiplication of two l-term polynomials in POLY-MULT-REDC step. For example, at the expense of some overheads, KOA or KOA-like formulae [32] can reduce the factor l^2 associated with $T_m(n)$ in (24) to $M(l)$, where $M(l)$'s for some small l's are given as follows:

$$M(2) = 3, \quad M(3) = 6, \quad M(5) = 13, \quad M(6) = 17, \quad M(7) = 22. \tag{25}$$

Alternatively, one can use the Toom-Cook multiplication method [33]–[35] which requires only $(2l+1)$ multiplications at the expense of much more overheads, including exact divisions by fixed integers.

The number of additions and subtractions in (24) can be reduced by combining polynomial multiplication and polynomial reduction as shown in Section IV. There are only five irreducible $f(t)$'s for $l = 2$ and we list all of them in Table III. The table also shows required cost for polynomial multiplication and squaring in $\mathbb{Z}[t]/f(t)$, where the notations M, S, A, a and h respectively mean multiplication, squaring, $(2n+l)$-bit addition, n-bit addition and bit-shift. For $l = 3$, there are twelve irreducible $f(t)$'s. The best performance is obtained when $f(t) = t^3+t^2+t-1$ or $f(t) = t^3+t-1$ is used. In these cases, the running time of the POLY-MULT-REDC step is $6T_m(n)+6T_a(n)+6T_a(2n+l)$. This computational cost is exactly the same as that for performing one 3-way multiplication as shown in (19).

In terms of the number of single-precision multiplications, there is little difference between multiplying two ln-bit long integers and multiplying two l-term polynomials whose coefficients are n-bits long. In fact, polynomial multiplication has a little less overhead since coefficients do not have to overlap, unlike the long integer multiplication. However, if implemented in software,

polynomial multiplication could be slower because microprocessors can deal only with units of data called word. For example, a 160-bit integer needs five words on 32-bit architecture, while the same integer in SD-(t,ψ) form with $l=2$ needs three 2-word coefficients, each filled with 80 bits, assuming $t+2^{l+1}-2<2^{80}$. Multiplying two 160-bit integers require only 15 multiplications using 2-way and 3-way KOA. However, multiplying two integers in SD-(t,ψ) form requires 18 multiplications using the same KOA methods.

B. COEFF-REDC step

Figure 2 shows how Algorithm 5, i.e. COEFF-REDC step, is performed; some input and intermediate values are labeled with circled numbers. We first determine the maximum possible bit lengths of these values. Note that $\hat{z}(t)=(\hat{z}_{l-1}\cdots\hat{z}_0)_{SD-(t,*)}$ is the output of Algorithm 4; that is, $|\hat{z}_i|\leq(2^l-1)\psi^2$ for $i=0,\ldots,l-1$.

① It is clear that $\hat{z}_i$'s are at most $(2n+l)$ bits long.

② $|a|\leq(2^l-1)(t+2^{l+2}-4)$ is at most $(n+l)$ bits long.

③ $|z'_{l-1}|<(2^l-1)(t+2^{l+2}-4)+t-1$ is at most $(n+l)$ bits long, since $|z'_{l-1}|<(2^l-1)(2^n-1)+2^n-1<2^{l+n}$.

④ $|z'_i|\leq(2^l-1)((t+2^{l+1}-2)^2+t+2^{l+2}-4)$ for $i<l-1$ are at most $(2n+l)$ bits long. Note that $t+2^{l+1}-2<2^n$ and $t+2^{l+2}-4<2^n$. It follows that $|z'_i|<(2^l-1)((2^n-1)^2+2^n-1)=(2^l-1)(2^{2n}-2^n)<2^{l+2n}$.

⑤ $|z'_{i,1}|\leq(2^l-1)(t+2^{l+2}-2)$ for $i\neq l-1$ is at most $(l+n)$ bits long.

⑥ $|z'_l|\leq(2^{l+1}-1)$ is at most $(l+1)$ bits long.

⑦ $|z'_i|\leq(t+2^{l+1}-2)$ is at most n bits long.

Note that we used the detailed calculations that have been already done in the proof of Proposition 2. Now it is easy to analyze Algorithm 5 using the above results.

- Step 2: one integer division for dividing a $(2n+l)$-bit integer by an n-bit integer is needed; that is, $T_d(2n+l,n)$.
- Step 3: τ additions of $(2n+l)$-bit integers and $(n+l)$-bit integers; that is, $\tau\cdot T_a(n+l)$.
- Step 4.1: for $i=0,\ldots,l-2$, a total of $(l-1)$ integer divisions for dividing $(2n+l)$-bit integer by an n-bit integer are required. For $i=l-1$, this division can be done by $(l+1)$ subtractions of up to $(l+n)$-bit integers from an $(l+n)$-bit integer. Thus, for step 4.1, the cost is $(l-1)\cdot T_d(2n+l,n)+(l+1)\cdot T_a(l+n)$.

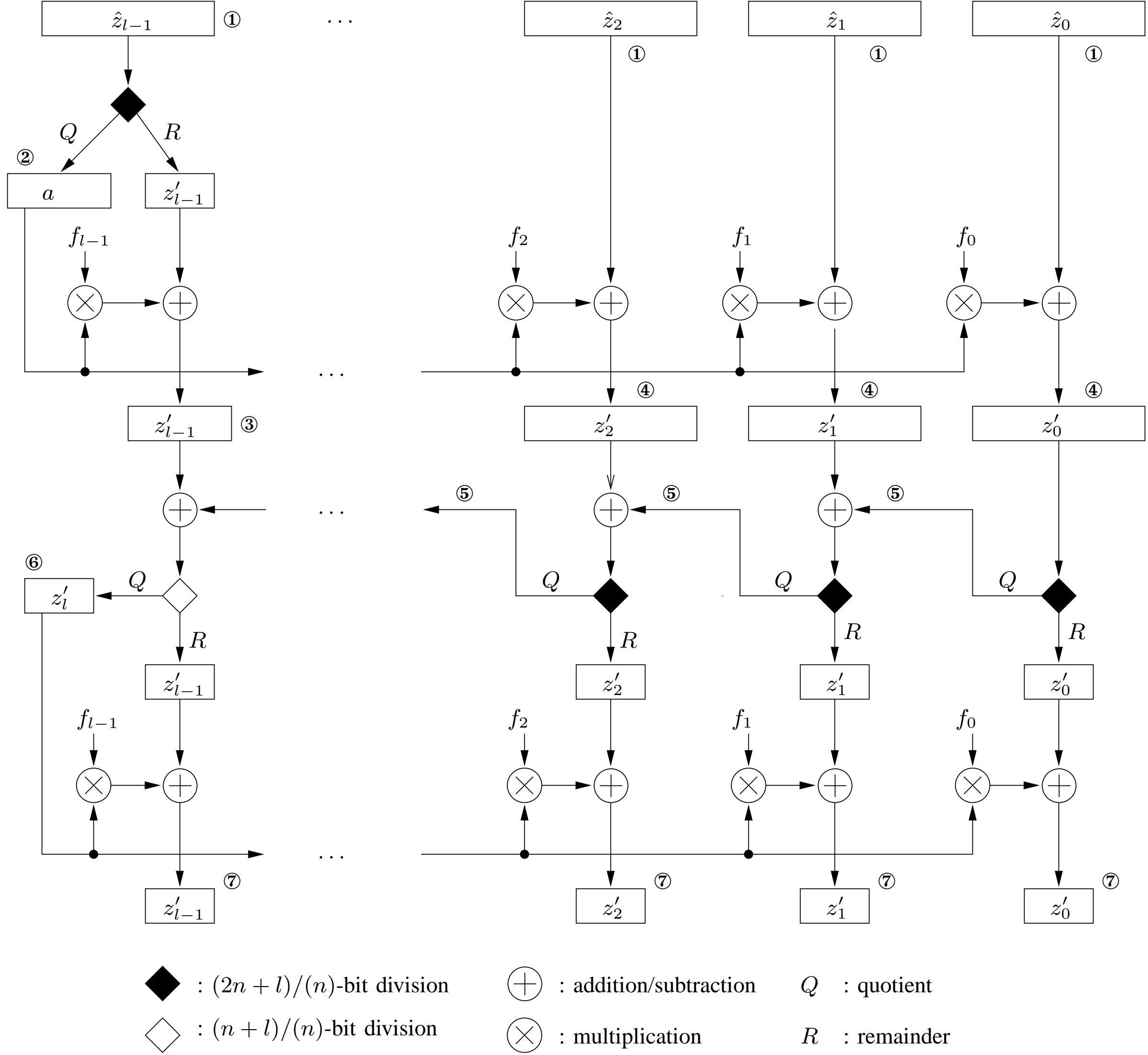

Fig. 2. Coefficient Reduction

- Step 4.2: for $i = 0, \dots l-3$, a total of $(l-2)$ additions of $(2n+l)$-bit and $(n+l)$-bit integers are required. For $i = l-2$, an addition of two $(n+l)$-bit integers is performed. For $i = l-1$, no operation is required since $z'_l = 0$. Thus, the cost of step 4.1 is $(l-1) \cdot T_a(n+l)$.
- Step 5: τ additions of an $(l+1)$-bit integer to n-bit integers are performed; that is, $\tau \cdot T_a(l+1)$.

In total, Algorithm 5 requires the following amount of time for reducing coefficients:

$$T(\text{COEFF_REDC}) = l \cdot T_d(2n+l, n) + (2l+\tau) \cdot T_a(n+l) + \tau \cdot T_a(l+1). \tag{26}$$

With regard to the time complexity related to the long integer division in (26), i.e., $l \cdot T_d(2n+l, n)$, note that the division algorithms, CAID and GBAID, shown in Section II take $O(n^2)$ time for one division, where n is the modulus size in bits. Hence, l executions of such algorithms for n-bit modulus take $O(ln^2)$ time. The overhead terms in (26), $(2l+\tau) \cdot T_a(n+l) + \tau \cdot T_a(l+1)$, take $O(nl)$ time.

C. Putting It Together

The main computational cost of an LWPFI modular multiplication is due to the following three, where L1 and L2 are performed in POLY-MULT-REDC step and L3 is in COEFF-REDC step.

L1: polynomial multiplication (e.g., using KOA)

L2: polynomial reduction

L3: coefficient reduction (e.g., using GBAID)

On the other hand, the main computational cost of a usual modular multiplication is due to the following two:

U1: integer multiplication (e.g., using KOA)

U2: modular reduction (e.g., using MAIR)

If L1 and U1 use the same algorithm (e.g., KOA), then they incur an equal amount of computation.

For an (nl)-bit modulus, assuming that GBAID is used for L3, the combined cost of L2 and L3 is

$$\tau(l-1) \cdot T_a(l+2n) + l \cdot T_d(2n+l, n) + (2l+\tau) \cdot T_a(n+l) + \tau \cdot T_a(l+1), \tag{27}$$

which is $O(ln^2)$ time. On the other hand, U2 using MAIR requires $O(l^2n^2)$ time. Therefore, the LWPFI modular multiplication has better asymptotic behavior than the usual modular multiplication. For example, the number of multiplication instructions required in GBAID for $(2n+l)$-bit dividend and n-bit divisor is expressed as follows:

$$\#\mathcal{M}_{GBAID} = \begin{cases} uv + 3u - v^2 - 2v + 1 & \text{if } u \leq 2v, \\ (u^2+5u)/2 - uv + v^2 - v + 1 & \text{if } u > 2v, \end{cases} \tag{28}$$

where $u = \lceil (2n+l)/w \rceil$, $v = \lceil n/w \rceil$, and w is the word length of a target architecture in bits. For $n = 512$ and $l = 2$, the COEFF-REDC step requires only $680 = 2 \cdot 340$ multiplication instructions, whereas MAIR for a similar size (i.e.,1024-bit) modulus requires 1056 multiplications.

Based on the above discussion, we see that the main advantage of LWPFI modular multiplication compared to usual modular multiplication is not due to the POLY-MULT-REDC step. Rather, the main performance gain for using LWPFI modular multiplication comes from the reduced complexity in the COEFF-REDC step.

D. Comments

The reduced complexity of LWPFI modular multiplication does not come for free. In fact, LWPFI modular multiplication introduces overhead mainly resulting from additions and subtractions. Such overhead due to additions and subtractions needs to be carefully considered. On some microprocessors, the time difference between multiplication and addition/subtraction is relatively not that significant. For example, on Pentium 4 3.2GHz processor (Family 7, Model 4), the latency of multiplication instruction (`mul`) is 11 clock cycles, and that of add-with-carry (`adc`) and subtract-with-borrow (`sbb`) instructions, the most frequently used ones for long integer additions and subtractions, is 10 clock cycles [36]. On the other hand, on Freescale ColdFire 5307, timing ratio of multiplication to addition is 5 when operands are in registers, and the ratio is only 2 when the operands are in memory [37].

In addition, overheads may result from factors pertaining to the implementation environment, and can potentially affect the performance of the modular multiplication algorithms. For example, for software implementation using general purpose processors, these factors would include the size and the number of the registers, cache size and speed, features of the data-path including pipe-lining, multiple execution units, etc. A detailed analysis of the effect of such factors on the performance of the modular multiplication algorithms is not simple. However, to give a good indication on how the LWPFI based algorithm compares with its counterparts we will consider timing results based on actual implementations. This is presented in the following section.

VI. Implementation Results and Practical Considerations

In this section, first we present timing results of modular multiplications. Then we discuss some general practical considerations for LWPFIs.

TABLE IV

FUNCTIONS USED FOR IMPLEMENTING LWPFI MODULAR MULTIPLICATIONS

Long Integer Operation	GMP
Multiplication	`mpz_mul()`
Addition	`mpz_add()`, `mpz_add_ui()`
Subtraction	`mpz_sub()`, `mpz_sub_ui()`
Bit-Shift	`mpz_mul_2exp()`

A. *Our Platform and Software Routines*

We have implemented LWPFI modular multiplications based on $f(t) = t^2 + 1$ and $f(t) = t^3 + t - 1$, and an LWPFI modular squaring based on $f(t) = t^2 + 1$. Our implementation uses GNU multiple precision (GMP) library v4.1.4 (`http://www.swox.com/gmp`). We implemented GBAID, which is not provided in GMP, using the C programming language. Since our implementation of GBAID uses only the C programming language, we have disabled all assembly routines in GMP library. We used Microsoft Visual Studio 2005 to compile all programs, and performed timing measurements on Intel Pentium 4 3.20GHz (Family 7, Model 4). To compile GMP with Visual Studio, we used Visual Studio project file for GMP v4.1.4 downloaded from `http://fp.gladman.plus.com/computing/gmp4win.htm`.

Our implementation of LWPFI modular multiplication is based on high level functions of GMP library. Table IV lists GMP functions that we used for implementing LWPFI modular multiplications. We used our GBAID routine for divisions in COEFF-REDC step, since our GBAID routine is much faster than the division function in GMP (`mpz_tdiv_r()`). The timing results shown in this section could be improved by using low level functions (`mpn_*()` functions) that have less redundancy than high level functions.

Our GBAID routine turned out to be faster than MAIR routines in GMP. Thus, we have written our own MAIR routine using the same coding style and optimization that we used when writing GBAID. Our MAIR performs better than our GBAID for all input lengths. The timing results in the following subsection are based on our own Montgomery reduction routine, not on `redc()` in GMP library.

TABLE V

DETAILED ANALYSIS OF LWPFI MODULAR MULTIPLICATION ($n = \log_2 p$)

	T(POLY-MULT-REDC) $= T_1 + T_2$	
$f(t)$	T_1	T_2
$f(t) = t^2 + 1$	$3 \cdot T_m(n/2)$	$2 \cdot T_a(n+2) + 2 \cdot T_a(n/2)$
$f(t) = t^3 + t - 1$	$6 \cdot T_m(n/3)$	$6 \cdot T_a(2n/3+2) + 6 \cdot T_a(n/3)$
	T(COEFF-REDC) $= T_3 + T_4$	
$f(t)$	T_3	T_4
$f(t) = t^2 + 1$	$2 \cdot T_B(n+2, n)$	$5 \cdot T_a(n/2+2) + T_a(3)$
$f(t) = t^3 + t - 1$	$3 \cdot T_B(2n/3+3, n)$	$8 \cdot T_a(n/3+3) + 2 \cdot T_a(4)$

B. Component-wise Breakdown of Timing

Table V shows detailed analyses of LWPFI modular multiplication methods for the two $f(t)$'s that we used in our implementation. The notations $T_m(n)$, $T_a(n)$ and $T_B(u, v)$ respectively refer to the running time for long integer multiplication of two n-bit integers, long integer addition of two n-bit integers and GBAID for u-bit dividend and v-bit divisor. T(POLY-MULT-REDC) and T(COEFF-REDC) refer to the time required for POLY-MULT-REDC and COEFF-REDC steps, respectively.

We experimentally measured T_1, T_2, T_3 and T_4, as defined in Table V, for varying bit sizes of p and plotted the results in Figures 3 and 4. In the figures, we use $T_i(l)$ to denote T_i for the l-th degree $f(t)$ in Table V. In Figure 4, $T_M(u, v)$ denotes the timing for Montgomery reduction when the input integer is u bits long and the modulus is v bits long. In Figure 4, we present $T_B(2n, n)$ to show how much amount of time COEFF-REDC saves by breaking up a full $(2nl)$-bit by (nl)-bit division into l short divisions for $(2n+l)$-bit dividend and n-bit divisor and some overheads. The $T_M(u, v)$ is shown as a reference timing of the best modular reduction algorithm considered in this paper.

In Figures 3 and 4, we see that the overheads resulting from additions/subtractions (T_2's and T_4's) are not significant in both POLY-MULT-REDC and COEFF-REDC steps. Especially in Figure 4, the overhead timings, $T_4(i)$ for $i = 2$ and 3, are very small compared to the reduction timings and they both are plotted close to the x-axis of the graph. The figures confirm our analytical conclusion in Section V that the efficient modular multiplication using LWPFI moduli

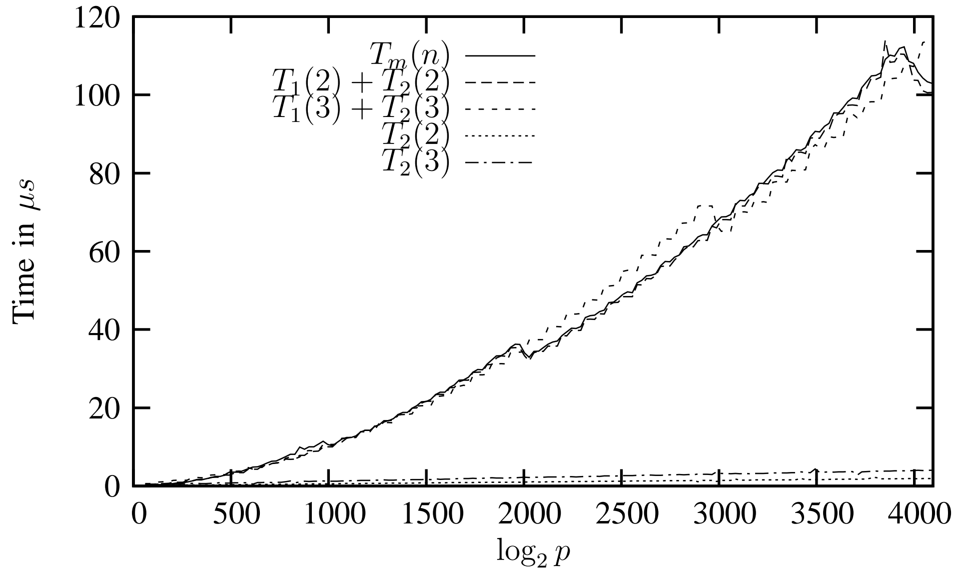

Fig. 3. Timing Results for POLY-MULT-REDC step on Pentium 4 @ 3.2GHz

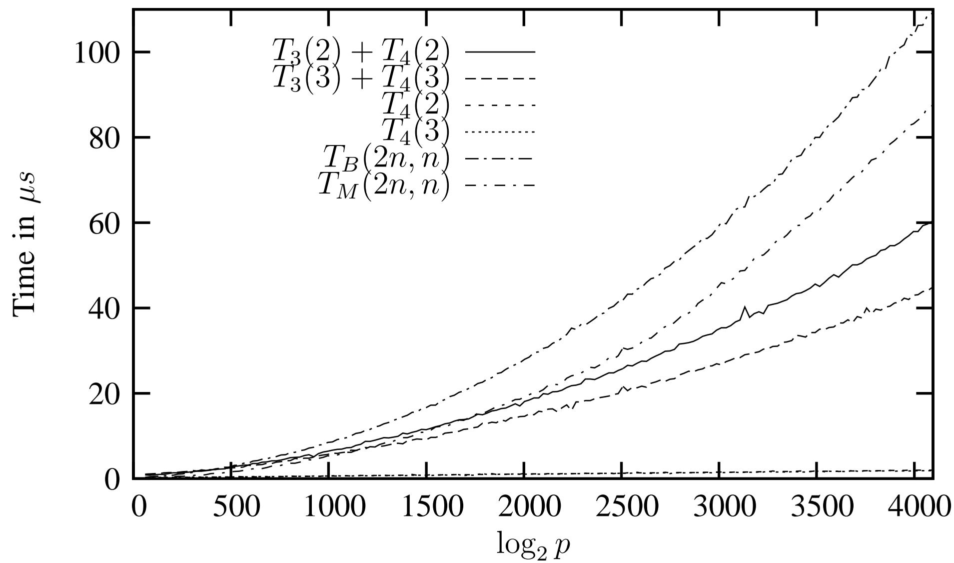

Fig. 4. Timing Results for COEFF-REDC step on Pentium 4 @ 3.2GHz

is not due to the POLY-MULT-REDC step where KOA has been used, but due to the COEFF-REDC step.

Note that GMP's `mpz_mul()` routine switches from classical algorithm to KOA when the operand size is more than 32 words long (i.e., 1024 bits for $w = 32$), and hence there is are sudden

changes in $T_m(n)$ whenever $\log_2 p = 1024 \cdot 2^i$ for $i \geq 0$. The performance of POLY-MULT-REDC for $l = 2$ is very close to that of `mpz_mul()` ($T_m(n)$ in Figure 3) for $\log_2 p \geq 1024$, since they both have the same asymptotic speed-up due to KOA. However, the timing results of POLY-MULT-REDC for $l = 3$ shows sudden changes in timing whenever $\log_2 p = 3 \cdot 1024 \cdot 2^i$ for $i \geq 0$, but it does not become similar to $T_m(n)$, since the 3-way KOA presented in (19) does not lead to the same asymptotic speed-up as the original 2-way KOA.

C. Overall Timing Results and Comparisons

Figures 5 shows timing results of our implementations of the following three modular multiplication methods:

1) "LWPFI mul." where KOA is used for polynomial multiplication and GBAID is used for COEFF-REDC (as discussed in this article). For this method, we show three plots corresponding to $f(t) = t^2 + 1$, $f(t) = t^3 + t - 1$ and $f(t) = t^4 - t^2 - 1$ as well.
2) "Mul. + MAIR" where KOA is used for long integer multiplications and MAIR is for modular reduction.
3) "Mul. + GBAID" where KOA is used for long integer multiplications and GBAID is for modular reduction.

In method 3), instead of GBAID, one can use the original Barrett reduction algorithm, which does not generate a quotient as output, However, as discussed in Section II, the difference between the computational costs of these two schemes is negligible. Also note that in method 3), the divisor of GBAID is the modulus (say (nl) bits long). On the other hand, for the same size moduli, the size of the divisor in the GBAID used in method 1) is n bits only. However, in method 1), the GBAID routine is used l times, whereas in method 3), the GBAID is used only once for each modular multiplication.

Figure 6 shows timing results for modular squaring operations using the same three methods, and in the case of LWPFI, only one graph for $l = 2$ is shown.

We clearly observe in the figures that LWPFI modular multiplications become more efficient than GBAID and MAIR based modular multiplications as the modulus size increases. We also observe that the asymptotic behavior of LWPFI modular multiplication improves as l increases, and that the LWPFI squaring for $l = 2$ indeed performs better than modular squaring methods using GBAID and MAIR.

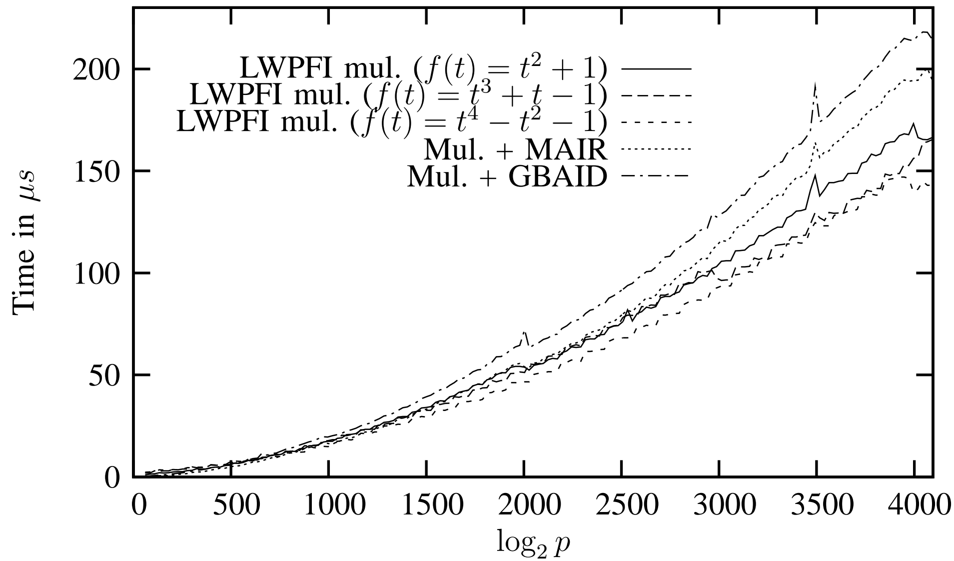

Fig. 5. Modular Multiplication Algorithms on Pentium 4 @ 3.2GHz

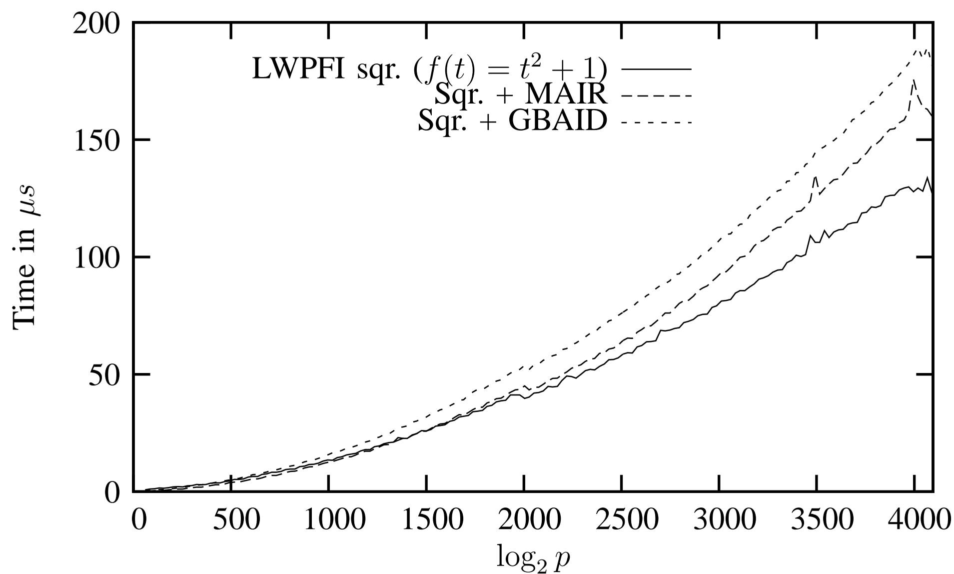

Fig. 6. Modular Squaring Algorithms on Pentium 4 @ 3.2GHz

D. Practical Considerations

- General implementation is possible using LWPFI proposed in this work. For a given bit length, we can find many useful moduli by varying the value of t, even for a fixed $f(t)$. On the other hand, the most limiting part of GMNs proposed in [9] is that there can be

only one GMN for a given bit length and a polynomial $f(t)$. This fact makes generalized implementation infeasible, and each GMN requires a dedicated implementation. This is not a problem in ECC and HECC, since it is the usual practice to set up domain parameters for use by many users in such cryptosystems. There are even pre-defined sets of recommended parameters for ECC [15], [16]. However, in RSA cryptosystems, every user has to generate his or her own parameters, and in XTR cryptosystems [2], every user is advised to do so. Hence, these cryptosystems do not benefit from the fast modular multiplication that GMNs provide.

- LWPFI modular multiplication makes it easy for parallel implementation. In POLY-MULT-REDC, KOA involves several multiple-precision multiplications. These multiplications are independent of each other and they can be computed in a parallel manner. For example, if $l = \deg(f(t))$ is 3, then six multiplications (D_0 through D_5 in (19)) can be computed by two, three or six processors or multipliers simultaneously. Moreover, l divisions by t in coefficient reduction step can be parallelized, even though not explicitly shown in this paper.
- Cryptographic computations usually require operations involving large operands. However, implementing operations which deal with very long operand sizes is challenging in restricted environments, such as smart cards and certain embedded systems. LWPFI modular multiplications make it possible to reduce the operand sizes by $1/l$, where $l = \deg(f(t))$.
- There are security concerns when moduli take a special form. Mersenne numbers are avoided in cryptosystems, since they are easier to be factored using the special number field sieve (SNFS) [13], [14]. A similar technique known as special function field sieve (SFFS) can be used for solving discrete logarithm problems based on special form of moduli [38]. However, SNFS and SFFS are applicable only to integers of the form $p = b^s - c$, where b and c are very small (e.g., Fermat's numbers, Mersenne numbers, *etc.*). SNFS and SFFS are not applicable to LWPFI, since LWPFI moduli are not in such a form of integers i.e., $p = t^l - (f_{l-1}t^{l-1} + \cdots + f_0)$ in which t and $(f_{l-1}t^{l-1} + \cdots + f_0)$ are both very large. However, currently there is no guarantee that cryptographic applications are secure when LWPFI is used. It is an open question whether LWPFI makes factoring and discrete logarithm easier.

VII. Enhancing the LWPFI Modular Multiplication

In this section, we show some methods for enhancing LWPFI modular multiplication.

A. *Using Pseudo-Mersenne Numbers for t ($t = 2^k - c$)*

If t is chosen to be a pseudo-Mersenne number, such that $t = 2^k - c$ for some k and small c, the performance of the LWPFI modular multiplication can be further improved. For a pseudo-Mersenne number t, there exists an efficient modular reduction algorithm due to Crandall [8]. The original Crandall's algorithm is used only for computing the remainder. Algorithm 6 shows the modified Crandall's algorithm which also computes the quotient.

Algorithm 6. Modified Crandall's Algorithm

INPUT: positive integers $x \geq t$ and $t = 2^k - c$.

OUTPUT: q and r, such that $x = q \cdot t + r$ and $0 \leq r < t$.

1) $q_0 \leftarrow \lfloor x/2^k \rfloor$, $r_0 \leftarrow x \bmod 2^k$
2) $q \leftarrow q_0$, $r \leftarrow r_0$, $i \leftarrow 0$.
3) While $q_i > 0$ do:
   - 3.1 $q_{i+1} \leftarrow \lfloor q_i \cdot c/2^k \rfloor$, $r_{i+1} \leftarrow q_i \cdot c \bmod 2^k$.
   - 3.2 $i \leftarrow i + 1$, $q \leftarrow q + q_i$, $r \leftarrow r + r_i$.
4) While $r \geq t$, do: $r \leftarrow r - t$, $q \leftarrow q + 1$.
5) Return q and r.

The correctness of Algorithm 6 can be easily derived from the one shown in [23] for the original Crandall's algorithm. The main difference between the original Crandall's algorithm and Algorithm 6 is that Algorithm 6 accumulates q_i's also, while original Crandall's algorithm accumulates only r_i's.

The multiplication by a constant c is required only twice if the size of c is at most half the size of t. In general, if $l = (s-2)k/(s-1)$ where l is the bit length of c, then step 3.1 is executed s times [23].

B. *Using LWPFI for t*

When LWPFIs are used for t, coefficient reduction could be done trivially. We show that dividing an integer in SD-(t, ψ) form by an LWPFI can be done very efficiently. Suppose $f(t)$ is a monic polynomial of degree l:

$$f(t) = t^l - f_{l-1}t^{l-1} - \cdots - f_1 t - f_0. \tag{29}$$

Let $x(t)$ be a degree $2l-2$ polynomial:

$$x(t) = x_{2l-2}t^{2l-2} + \cdots + x_1 t + x_0. \tag{30}$$

Define a polynomial $q(t)$ of degree $l-2$ such that

$$\begin{aligned} q(t) &= q_{l-2}t^{l-2} + \cdots + q_1 t + q_0, \\ q_i &= x_{l+i} + \sum_{j=i+1}^{l-2} q_j f_{j+1}.. \end{aligned} \tag{31}$$

Then it follows that $q(t)$ satisfies the following:

$$x(t) = q(t)f(t) + r(t) \qquad (\deg(r(t)) < \deg(f(t))). \tag{32}$$

Therefore, we have a formula for the quotient polynomial $q(t)$. The formula for $r(t)$ can be easily obtained by a similar method as shown in Section III-A. Since the quotient polynomial $q(t)$ can be obtained while computing the remainder polynomial $r(t)$, the above method requires at most $\tau(l-1)$ additions/subtractions, where τ is the number of non-zero f_i's in $f(t)$.

C. *Generating $p = f(t)$ of Any Bit Length*

Given the definition in Section III, we see that the bit length of an LWPFI can be only about a multiple of $l = \deg(f(t))$. We can remove this constraint by introducing the *extended low-weight polynomial form numbers (ELWPFIs)*. ELWPFIs are almost the same as LWPFIs, except that $f(t)$ is not necessarily a monic polynomial; for example $f(t) = f_l t^l - f_{l-1}t^{l-1} - \cdots - f_1 t - f_0$, where $f_l > 1$ and $f_i \in \{0, \pm 1\}$ for $i < l$.

With appropriately chosen f_l, we can generate moduli of any bit length. For example, if we choose h-bit integer for f_l and a third degree polynomial $f(t)$, we can generate p's that are about $(ln + h)$ bits long, where n is the bit length of t.

However, a polynomial reduction by an ELWPFI is not simple since the reduction formula involves many divisions by f_l. Even though f_l is chosen to be a power of 2, the dividends are not necessarily divisible by f_l. This issue can be resolved easily by using the following representation:

$$x(t) \equiv x_{l-1}(f_l t)^{l-1} + \cdots + x_1(f_l t) + x_0 \pmod{f(t)}. \tag{33}$$

Then it becomes straightforward to check that a polynomial reduction by non-monic polynomial $f(t)$ will not have divisions by f_l. Note that the introduction of $f_l > 1$ will result in constant

multiplications by f_l when performing LWPFI modular multiplication. Therefore, f_l must be either a very small integer or a power of 2 to avoid actual multiplication instructions in the computation of LWPFI modular multiplication.

VIII. Conclusions

In this paper we have considered low-weight polynomial form integers (LWPFIs) to devise an efficient modular multiplication method. LWPFI is a family of integers expressed in polynomial form $p = f(t)$, and they further generalize Mersenne numbers by allowing any integer for t. Our analysis shows that LWPFI modular multiplication has better asymptotic behavior than other general modular reduction methods. Our implementation results show that LWPFI modular multiplication is faster than Montgomery reduction for moduli of large sizes. We have shown techniques that can speed up LWPFI modular multiplication. GMN or pseudo-Mersenne number based modular multiplication would be faster than LWPFI based one, however there are not that many GMNs and pseudo-Mersenne numbers. LWPFI has its advantage that the implementation does not have to be specific to a single modulus and that LWPFI provides a considerably large choices of moduli. Hence, one may consider LWPFI as a trade-off between general integer and other special type of moduli such as GMNs and pseudo-Mersenne numbers.

Acknowledgment

The authors would like to thank the anonymous reviewers for carefully reading the manuscript and providing useful comments to improve its quality. Jaewook Chung's research was funded by Natural Science and Engineering Research Council of Canada (NSERC) through Post Graduate Scholarship – B (PGS–B) and in part by Canadian Wireless Telecommunications Association (CWTA) Scholarship. This work was also supported in part by NSERC discovery and strategic project grants awarded to Dr. Hasan.

Appendix A
Proof for the Generalized Barrett Algorithm for Integer Division

Proposition 3 *Computation of q in step 1 of Algorithm 3 is not exact. The error in q is at most 1, if $b \geq u - v$.*

Proof: For simplicity, we let $k = u-v$, $\gamma = (\gamma_k \cdots \gamma_1\gamma_0)_b = (x_{u-1} \cdots x_v x_{v-1})_b = \lfloor x/b^{v-1} \rfloor$. Note that μ is also at most $k+1 = u-v+1$ words long, since $\mu = \lfloor b^u/m \rfloor$ where m is v words long. Let $q' = \lfloor \gamma\mu/b^{u-v+1} \rfloor$ be the value computed by the following full multiplication:

$$q' = \left\lfloor \frac{\gamma \cdot \mu}{b^{k+1}} \right\rfloor = \left\lfloor \frac{\sum_{i+j} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor . \tag{34}$$

The computation of q in step 1 of Algorithm 3 is done by the following partial multiplication:

$$q = \left\lfloor \frac{\sum_{k-1 \le i+j} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor . \tag{35}$$

Observe that q' and q have a common part:

$$\begin{aligned} q' &= \left\lfloor \frac{\sum_{i+j \le k} \gamma_i \mu_j b^{i+j} + \sum_{k+1 \le i+j} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor \\ &= \sum_{k+1 \le i+j} \gamma_i \mu_j b^{i+j-k-1} + \left\lfloor \frac{\sum_{i+j \le k} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor \\ q &= \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1} + \sum_{k+1 \le i+j} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor \\ &= \sum_{k+1 \le i+j} \gamma_i \mu_j b^{i+j-k-1} + \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor . \end{aligned} \tag{36}$$

Let ϵ be the difference between q' and q, i.e., $\epsilon = q' - q$.

$$\begin{aligned} \epsilon &= \left\lfloor \frac{\sum_{i+j \le k} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor - \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor \\ &= \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j} + \sum_{i+j \le k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor - \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor \\ &= \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1}}{b^2} + \frac{\sum_{i+j \le k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor \\ &\quad - \left\lfloor \frac{\sum_{k-1 \le i+j \le k} \gamma_i \mu_j b^{i+j-k+1}}{b^2} \right\rfloor \le \left\lfloor \frac{\sum_{i+j \le k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor + 1. \\ &\quad (\because \lfloor A+B \rfloor \le \lfloor A \rfloor + \lfloor B \rfloor + 1) \end{aligned} \tag{37}$$

Since $\gamma_i,\ \mu_j < b$,

$$\left\lfloor \frac{\sum_{i+j \le k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor < \left\lfloor \frac{\sum_{i+j \le k-2} b^{i+j}}{b^{k-1}} \right\rfloor . \tag{38}$$

Then we can see that for $b \geq k$,

$$\sum_{i+j \leq k-2} b^{i+j} = (k-1)b^{k-2} + (k-2)b^{k-3} + \cdots + 2b + 1 < b^{k-1}. \tag{39}$$

Therefore,

$$\left\lfloor \frac{\sum_{i+j \leq k-2} \gamma_i \mu_j b^{i+j}}{b^{k+1}} \right\rfloor = 0, \tag{40}$$

and

$$\epsilon \leq 1. \tag{41}$$

■

REFERENCES

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] A. K. Lenstra and E. R. Verheul, "The XTR public key system," in *Advances in Cryptology - CRYPTO 2000*, ser. LNCS 1880. Springer-Verlag, 2000, pp. 1–19.

[3] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology - CRYPTO '85*, ser. LNCS 218. Springer-Verlag, 1986, pp. 417–426.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comp.*, vol. 48, pp. 203–209, Jan. 1987.

[5] D. Knuth, *The Art of Comuter Programming, Vol. 2, Seminumerical Algorithms*, 2nd ed. Addison-Wesley, 1981.

[6] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86*, ser. LNCS 263. Springer-Verlag, 1987, pp. 311–323.

[7] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[8] R. E. Crandall, "Method and apparatus for public key exchange in a cryptographic system (oct. 27, 1992)," U.S. Patent # 5,159,632.

[9] J. A. Solinas, "Generalized Mersenne numbers," Centre for Applied Cryptographic Research, University of Waterloo, Tech. Rep. CORR 99-39, 1999, `http://cacr.uwaterloo.ca/techreports/1999/corr99-39.ps`.

[10] S. Kawamura, K. Takabayashi, and A. Shimbo, "A fast modular exponentiation algorithm," *IEICE Transactions*, vol. E-74, no. 8, pp. 2136–2142, August 1991.

[11] S.-M. Hong, S.-Y. Oh, and H. Yoon, "New modular multiplication algorithms for fast modular exponentiation," in *Lecture Notes in Computer Science*, ser. LNCS 1070. Springer-Verlag, 1996, pp. 166–177.

[12] C. H. Lim, H. S. Hwang, and P. J. Lee, "Fast modular reduction with precomputation," in *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC '97)*, Seoul, 1997, pp. 65–79.

[13] A. K. Lenstra and H. Lenstra Jr, "The development of the number field sieve," in *Lecture Notes in Mathematics*, ser. 1554, 1993, pp. 11–42.

[14] A. K. Lenstra, H. Lenstra Jr, M. Manasse, and J. Pollard, "The factorization of the ninth Fermat number," *Mathematics of Computation*, vol. 61, no. 203, pp. 319–349, 1993.

[15] National Institute of Standards and Technology, "Recommended elliptic curves for federal government use," July, 1999.

[16] N. I. of Standards and Technology, "Digital signature standard (DSS)," FIPS Publication 186-2, February, 2000.

[17] J. Chung and A. Hasan, "More generalized Mersenne numbers," in *Selected Areas in Cryptography - SAC 2003*, ser. LNCS 3006. Springer-Verlag, 2003, pp. 335–347.

[18] J.-C. Bajard, L. Imbert, and T. Plantard, "Modular number systems: Beyond the Mersenne family," in *Selected Areas in Cryptography 2004*, ser. LNCS 3357. Springer-Verlag, 2004, pp. 159–169.

[19] ——, "Arithmetic operations in the polynomial modular number system," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ser. ARITH'05, 2005, pp. 206–213.

[20] S. R. Dussé and B. S. Kaliski Jr., "A cryptographic library for the Motorola DSP56000," in *Advances in Cryptology - EUROCRYPT '90*, ser. LNCS 473. Springer-Verlag, 1991, pp. 230–244.

[21] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, June 1996.

[22] J.-F. Dhem, "Efficient modular reduction algorithm in $\mathbb{F}_q[x]$ and its application to "left to right" modular multiplication in $\mathbb{F}_2[x]$," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, ser. LNCS 2779, 2003, pp. 203–213.

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[24] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, 1999.

[25] G. Hachez and J.-J. Quisquater, "Montgomery exponentiation with no final subtractions: Improved results," in *Cryptographic Hardware and Embedded Systems - CHES 2000*, ser. LNCS 1965. Springer-Verlag, 2000, pp. 293–301.

[26] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology - CRYPTO '96*, ser. LNCS 1109. Springer-Verlag, 1996, pp. 104–113.

[27] D. Naccache and H. M'Silti, "A new modulo computation algorithm," *Recherche Opérationnelle - Operations Research (RAIRO-OR)*, vol. 24, pp. 307–313, 1990.

[28] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Advances in Cryptology - CRYPTO '93*, ser. LNCS 773. Springer-Verlag, 1994, pp. 175–186.

[29] A. Avizienis, "Signed-digit number representation for fast parallel arithmetic," *IRE Transaction on Computers*, vol. EC-10, pp. 389–400, 1961.

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

[31] A. Weimerskirch and C. Paar, "Generalization of the Karatsuba algorithm for efficient implementations," Ruhr-Universität Bochum, Gemany, Tech. Rep., 2003, available at http://www.crypto.ruhr-uni-bochum.de/en_publications.html.

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

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

[34] S. A. Cook, "On the minimum computation time of functions," Ph.D. dissertation, Havard University, May 1966.

[35] D. Zuras, "More on squaring and multiplying large integers," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 899–908, August 1994.

[36] T. Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," 2005, available at http://swox.com/doc/x86-timing.pdf.

[37] Freescale Semiconductor, Inc., "MCF5307 ColdFire, integrated microprocessor user's manual," 2005, available at http://www.freescale.com/files/soft_dev_tools/doc/ref_manual/MCF5307BUM%.pdf.

[38] O. Shirokauer, "The special function field sieve," *SIAM Journal on Discrete Mathematics*, vol. 16, no. 1, pp. 81–98, 2002.