Fast modular reduction for large wordlengths via One Linear and One Cyclic Convolution

Dhananjay S. Phatak and Tom Goff
Computer Science and Electrical Engineering Department
University of Maryland, Baltimore County
Baltimore, MD 21250
{phatak, tgoff1}@umbc.edu

***Abstract*— Modular reduction is a fundamental operation in cryptographic systems. Most well known modular reduction methods including Barrett's and Montgomery's algorithms leverage some-pre computations to avoid divisions so that the main complexity of these methods lies in a sequence of two long multiplications. For large wordlengths a multiplication which is tantamount to a linear convolution is performed via the Fast Fourier Transform (FFT) or other transform-based techniques as in the Schonhage-Strassen multiplication algorithm.**

We show a fundamental property (the separation principle): in a modular reduction based on long multiplications, the linear convolution required by one of the two long multiplications can be replaced by a cyclic convolution, and the halves can be separated using other information available due to the intrinsic redundancy of the operations. This reduces the number of operations by about 25%. We demonstrate that both Barrett's and Montgomery's methods can be sped up by using the aforementioned fundamental principle. It is shown that a direct application of this algorithm to modular exponentiation (either using Barrett's or Montgomery's methods) can be expected to yield about about 17% speedup.

***Index Terms*— fast modular reduction, large wordlength, elliptic-curve, cryptography, FFT multiply, number theoretic transforms, linear convolution, cyclic convolution, principle of separation**

I. Introduction

Modular reduction is a fundamental operation in cryptographic systems [1], [2]. In many cases such as in Elliptic Curve Cryptography, the modulus P (which is typically a large prime number) is fixed [1]. The fact that P is a constant makes it feasible to precompute some values ahead of time which typically results in
(i) avoiding divisions and replacing them by multiplications instead.
(ii) substantial speed-up of the modular reduction operation because divisions are avoided and pre-computation trades off storage for computation effort.

This work was supported in part by NSF grants ECS-9875705 and and ECS-0196362

For example, both Barrett's [3] and Montgomery's [4] methods need some pre-computations to obviate division (this is explained in detail in Section I-B and Section I-C below). The main complexity of these methods lies in a sequence of two long multiplications. For large wordlengths a multiplication which is tantamount to a linear convolution is performed via the Fast Fourier Transform (FFT) [5], [6] or other transform-based techniques [7]–[9] as in the Schonhage-Strassen [10] multiplication algorithm.

We show a fundamental property: in a modular reduction based on long multiplications, the linear convolution required by one of the two long multiplications can be replaced by a cyclic convolution, thereby reducing the total number of operations by about 25%. We demonstrate that both Barrett's and Montgomery's methods can be sped up by using the aforementioned principle. Furthermore, it is shown that a direct application of this algorithm to modular exponentiation (either using Barrett's or Montgomery's methods) can be expected to yield about about 17% speedup.

A. Problem Definition

The modular-reduction problem deals with generating the remainder R when a given dividend X is divided by a modulus P:

$$\begin{aligned} X &= Q\cdot P+R \quad \text{where} \quad 0<X<P^2,\ 0\le R<P \\ R &= X\%P \quad\quad \text{following C syntax} \end{aligned} \tag{1}$$

Modulus P is assumed to be n bits long. Typically it is also a prime number (Montgomery's method requires P to be odd, while Barrett's method imposes no restrictions on P as long as it is within the range specified by (1)). Our fundamental speedup-algorithm is independent of whether P is prime, even, odd.... (i.e., our algorithm works for any modulus, given some pre-computation). We also assume that

$$2^{n-1} < P < 2^n \tag{2}$$

i.e., in the unsigned (magnitude only) form, the nth bit of P (which has a weight of 2^{n-1}) is 1 and at least one more of the remaining bits of P is 1. Typically n is a power of 2 (for example, n = 1024, 2048 or higher). Dividend X is at most $2n$ bits long and is variable/arbitrary (dynamically available only at run-time). It is convenient to split X into an upper half $\mathcal{U}(X)$ (i.e., higher order n bits of X) and a lower half $\mathcal{L}(x)$ (i.e., lower order n bits of X):

$$\begin{aligned} X &= \mathcal{U}(X)\cdot 2^n + \mathcal{L}(X) = X_u \cdot 2^n + X_l \quad \text{where} \\ \mathcal{U}(Z) &= \lfloor \frac{Z}{2^n} \rfloor \quad \text{and} \\ \mathcal{L}(Z) &= Z\%2^n \end{aligned} \tag{3}$$

Next we outline Barrett's and Montgomery's methods for finding R.

B. Barret Reduction [3]

The well known Barrett reduction method [3], [6] leverages the constancy of modulus P to reduce the problem to that of two multiplications. It is carried out in 3 steps as indicated below [3], [11] (Note that the precise implementation could slightly differ, the following is our rendering of the steps to bring out the main ideas)

<u>*Pre-computation*</u> : Given P, compute

$$P_{\text{inv}} = \left\lfloor \frac{2^{2n}}{P} \right\rfloor \tag{4}$$

<u>*Step 1*</u> : For an operand X to be reduced modulo-P, Barrett's method first determines quotient-estimate $\hat{Q}$:

$$\begin{aligned} \hat{Q} &= \text{close approximation to exact quotient} \\ Q &= \left\lfloor \frac{X}{P} \right\rfloor & (5) \\ \frac{X}{P} &= \frac{X}{2^n}\cdot\frac{2^{2n}}{P}\cdot\frac{1}{2^n} & (6) \\ \hat{Q} &= \left\lfloor \left(\left\lfloor \frac{X}{2^n} \right\rfloor \cdot \left\lfloor \frac{2^{2n}}{P} \right\rfloor \right) \cdot \frac{1}{2^n} \right\rfloor & (7) \\ \left\lfloor \frac{X}{2^n} \right\rfloor &= \mathcal{U}(X) = X_u \quad \text{: simply a right shift} \\ & \quad \text{yielding the upper half of } X & (8) \\ \left\lfloor \frac{2^{2n}}{P} \right\rfloor &= P_{\text{inv}} \quad \text{(precomputed)} & (9) \\ \hat{Q} &= \left\lfloor \frac{X_u \cdot P_{\text{inv}}}{2^n} \right\rfloor \quad \text{i.e., upper half of the} \\ & \quad \text{result of the multiplication } X_u \cdot P_{\text{inv}} & (10) \end{aligned}$$

<u>*Step 2*</u> : Carry out $\hat{Q} \times P$
The upper half of this product is guaranteed to closely match X_u, the upper half of X. Hence, generate only lower half:

$$L' = (\hat{Q}\cdot P)\%2^n \tag{11}$$

<u>*Step 3*</u> : Subtract and generate remainder estimate $\hat{R} = X_l - L'$. Correct remainder R can be obtained from $\hat{R}$ by subtracting P no more than 3 times.

The main complexity of Barret Reduction therefore lies in the two long multiply operations ($X_u \times P_{\text{inv}}$) and ($\hat{Q} \times P$).

C. Montgomery Reduction [4]

Let $R = 2^n > P$

<u>*Pre-computation*</u> : Compute R^{-1} and P' such that

$$RR^{-1} - P'P = 1 \quad \text{and} \quad 0 < R^{-1} < P \quad \text{and} \quad 0 < P' < R \tag{12}$$

This pre-computation is essentially the extended GCD algorithm.

Now given a T satisfying $0 \le T < RP$, Montgomery's method evaluates $TR^{-1} \mod P$ as follows.

<u>*Step 1*</u> :

$$\begin{aligned} m &= ((T \mod R)P') \mod R \\ \text{so that} & \quad 0 \le m \le R \end{aligned} \tag{13}$$

<u>*Step 2*</u> :

$$t = (T + mP)/R \tag{14}$$

Here, t is guaranteed to be an integer, so that the above division by R is exact.

<u>*Step 3*</u> : if $t \ge P$ return $(t-P)$ else return t

Once again the main complexity lies in the two long multiplications:
$[(T \mod R) \times P']$ in *Step 1* and
$[m \times P]$ in *Step 2*.

There is some post processing which is besides the point (not relevant to the matter at hand). Note that Like Barrett's method, Montgomery's method also requires only half of each product (either upper half or lower half).

D. Transform-based multiplication for large n

For large n, transform-based multiplication is the fastest/most efficient method [5]–[8], [12], [13]. The transform can be a complex valued FFT, or a number-theoretic transform as in Schonhage-Strassen method [9],

[10]. Since integers are polynomials (of the radix), integer multiplication can be realized via polynomial convolution [5], [6]. To evaluate $C = A \times B$, integers A, B are represented as strings of digits of some radix r (typically a power of 2). $\overline{A}$ then denotes the coefficients of a polynomial $A(x)$ where $A(x)|_{x=r} = A$, the integer value. Multiplication $A(x)B(x)$ produces a result polynomial $C(x)$ where
degree$[C(x)]$ = degree$[A(x)]$ + degree$[B(x)] - 1$. This multiplication is also referred to as a **linear convolution** [8], [13].

Step 1 :
create $2n$ long vectors $\overline{A}_{2n}$ and $\overline{B}_{2n}$ by padding 0's in the upper halves of A and B. Evaluate 2n-point FFTs of $\overline{A}_{2n}$ and $\overline{B}_{2n}$, denoted by $\overline{\mathcal{F}}_{2n}(\overline{A})$ and $\overline{\mathcal{F}}_{2n}(\overline{B})$, respectively.

Step 2 :
Point-wise multiply to obtain 2n-FFT of C :

$$\overline{\mathcal{F}}_{2n}(\overline{C}) = \overline{\mathcal{F}}_{2n}(\overline{A}) \star \overline{\mathcal{F}}_{2n}(\overline{B}) \tag{15}$$

Step 3 :
Perform the 2n-point inverse FFT of $\overline{\mathcal{F}}_{2n}(\overline{C})$ to obtain a vector of coefficients.
Then convert the coefficients into binary number.

Such a transform based multiplication reduces the number of operations from $O(n^2)$ to $O(n \lg n)$. The main effort is in performing the forward and reverse transforms. It is well known that for a real valued n-long input vector $\overline{V}$, the amount of operations required to compute its n-point FFT is $\theta(\frac{n}{2}\lg(\frac{n}{2}))$ [14], [15]. The same holds for inverse transform when the resultant vector is guaranteed to be real (i.e., the IFFT complexity is also $\theta(\frac{n}{2}\lg(\frac{n}{2}))$).

We would like to point out that we have used multiplication based on the conventional FFT (using complex roots of unity) for the purpose of illustration. The fundamental speedup-mechanism we propose in the next section is valid irrespective of which specific transform-based multiplication method is employed. The exact amount of speedup could differ for different multiplication methods such as FFT based multiply versus the Schonhage-strassen algorithm, but a speedup is possible regardless of the specific method employed.

E. Complexity of Barrett's and Montgomery's methods

Ignoring pre and post computations, the complexity of both these methods is dominated by the two long multiplications. All the remaining operations are $O(n)$. Since the transforms of the precomputed values can also be precomputed, in Barrett's method we assume FFT of P as well as FFT of $(\lfloor \frac{2^{2n}}{P} \rfloor$ is precomputed and available. Likewise in Montgomery's method, we assume the FFTs of P and P' are precomputed. Then, the complexity of both methods essentially lies in 2 forward and 2 reverse transforms:

Barrett's method

Operation	Effort
2n-FFT of X_u	$n\lg n$
2n-IFFT of $[\overline{\mathcal{F}}(\overline{X_u}) \star \overline{\mathcal{F}}(\overline{P_{\text{inv}}})]$	$n\lg n$
2n-FFT of $\hat{Q}$	$n\lg n$
2n-IFFT to obtain $(P \cdot \hat{Q})$	$n\lg n$
total dominant effort	$4n\lg n$

Montgomery's method

Operation	Effort
2n-FFT of T_u	$n\lg n$
2n-IFFT $(T_u \cdot P')$	$n\lg n$
2n-FFT of m	$n\lg n$
2n-IFFT to obtain $(P \cdot m)$	$n\lg n$
total dominant effort	$4n\lg n$

II. The proposed speedup method

First note that in each multiplication in Barrett's as well as Montgomery's method only half the product output is needed: either the upper half or the lower half. Unfortunately this fact alone cannot be leveraged to reduce the complexity when a transform-based multiplication method is used. In other words, when product $C = AB$ is being generated by transform methods, even if only the upper or only lower half of C is required, the full $2n\lg 2n$ amount of work needs to be done. The discussion on the "half cyclic convolution" on page 220 of [16] basically says that there's no known way (that scales with n) of just getting one half without the other.

If however either the upper half or lower half or a sufficiently close approximation to one of the two halves is known then the effort required can be cut down to about half. We use the following properties of FFT to achieve the reduction:

Let $\overline{C} = [\overline{U}|\overline{L}]^T$ be the vector of length $2n$ (representing integer product $C = A \times B$) and let $\overline{U}$ and $\overline{L}$ be the upper and lower halves of $\overline{C}$. Then a cyclic polynomial convolution of $A(x)$ and $B(x)$ yields

$$\begin{aligned} C(x)\%(x^n-1) &= (U(x)x^n + L(x))\%(x^n-1) \\ &= [(U(x)\%(x^n-1)\cdot(x^n\%(x^n-1)) \\ &\quad + L(x)\%(x^n-1)]\%(x^n-1) \\ &= U(x) + L(x) \end{aligned} \tag{16}$$

which is the sum of upper halves $U(x)$ and $L(x)$ of full (linear) convolution $C(x)$. The **cyclic convolution** of A and B can be realized as

$$\overline{U} + \overline{L} = \overline{\mathcal{F}}\text{inv}_n(\overline{\mathcal{F}}_n(\overline{A}) \star \overline{\mathcal{F}}_n(\overline{B})) \tag{17}$$

Note that a cyclic convolution needs transforms of half the length but it produces $(U+L)$. Our speedup technique exploits the fact that for Barrett's method, the second product $\hat{Q}\cdot P$ must be such that the upper half of that product is almost identical to X_u the upper half of original operand X. So instead of performing a linear convolution to implement $\hat{Q}\times P$ we perform a circular convolution to obtain$(U+L)$, then subtract X_u and perform a small correction to obtain the lower half.

In the Montgomery's method, note that in the second step, $t=\frac{T+mP}{R}$ is guaranteed to be an integer. This in-turn implies that

$$\begin{aligned}&[\text{lower half of } \ (T+mP)]\%R=0\Rightarrow\\&\text{lower half of } \ mP=R-T_L=2^n-T_L\end{aligned}\quad (18)$$

Accordingly instead of performing linear convolution to evaluate $m\times P$, we perform a cyclic convolution (requiring half the effort) and do small corrections to obtain the upper half.

Algorithm 1: Faster Barrett Reduction

```
Given a $2n$ bit number $X=X_u\cdot 2^n+X_L$ and an
$n$ bit modulus $P>2^{(n-1)}$, find $X\%P$

  Let the integers be represented as
polynomials of radix $\beta=2^k$ and let the
degree of each polynomial be $N-1$, so that
each polynomial can be represented as a
vector of length $N$. Then $N=\frac{2^n}{2^k}$ so that $N$ is
a power of 2.

Pre-computation :
In our algorithm, the following are
precomputed:
(1) $P_{\text{inv}}=\left\lfloor\frac{2^{2n-1}}{P}\right\rfloor$  and its $2N$-length FFT.
/* In order to ensure that $X_u\times P_{\text{inv}}$ does not
exceed $2n$ bits, $P_{\text{inv}}$is set to
$P_{\text{inv}}=\left\lfloor\frac{2^{2n-1}}{P}\right\rfloor$ instead of $\left\lfloor\frac{2^{2n}}{P}\right\rfloor$ */
(2) $P'=2P\%(2^n-1)$ and $N$-length FFT of $P'$

/* Part 1 :
Find an estimate of Quotient $\frac{X_u 2^{n-1}}{P}$ by a
full linear convolution
Note that $\left\lfloor\frac{X_u 2^{n-1}}{P}\right\rfloor=\left\lfloor\frac{X_u 2^{2n-1}}{P\cdot 2^n}\right\rfloor=\left\lfloor\frac{X_u P_{\text{inv}}}{2^n}\right\rfloor$ */

 Step 1:   2N-length FFT of 0 padded $X_u$
 Step 2:   point-wise multiply:
           $\overline{\mathcal{F}}_{2N}(Q)=\overline{\mathcal{F}}_{2N}(X_u)\star\overline{\mathcal{F}}_{2N}(P_{\text{inv}})$
 Step 3:   IFFT: $\overline{Q}=$
           $\overline{\mathcal{F}}\text{inv}_{2N}(\overline{\mathcal{F}}_{2N}(X_u)\star\overline{\mathcal{F}}_{2N}(P_{\text{inv}}))$
 Step 4:   Convert $\overline{Q}$ into integer
 Step 5:   Truncate: $\hat{Q}=\left\lfloor\frac{Q}{2^n}\right\rfloor$

/* Part 2:
obtain the sum of upper and lower halves of
$\hat{Q}P'$ by a cyclic convolution */

 Step 6:   $N$-length FFT of $\hat{Q}$
 Step 7:   point-wise multiply: $\overline{\mathcal{F}}_N(\hat{Q})\star\overline{\mathcal{F}}_N(P')$
 Step 8:   IFFT: $\overline{Z}=\overline{U}+\overline{L}=\overline{\mathcal{F}}\text{inv}_N(\overline{\mathcal{F}}_N(\hat{Q})\star\overline{\mathcal{F}}_N(P'))$
 Step 9:   Convert $\overline{Z}$ into integer

/* Part 3:
Separate $U$ and $L$ */

  Step 10:   if (Length(Z) $>n$) then
                $Z=\left\lfloor\frac{Z}{2^n}\right\rfloor+Z\%2^n$
             /* cyclic convolution wraps around */
  Step 11:   $L=Z-X_u$
             if $(L<0)$ then $L=L+2^n-1$
  Step 12:   $L_0=$ LeastSignificantWord$(L)$
             $P_0=$ LeastSignificantWord$(P')$
             $\hat{Q}_0=$ LeastSignificantWord$(\hat{Q})$
             $T_0$ = LeastSignificantWord$(P_0Q_0)$
             $\delta=T_0-L_0$
  Step 13:   if $(\delta>0)$ then $L=L+\delta$
 Step 14:   $R=X_l-L$
            if $(R<0)$ then
                $R=R+2^n$
                $\delta=\delta-1$
            endif
 Step 15:   $R=R+2^n\delta$ , $R=R\%P$
```

The dominant complexity lies in forward and reverse transforms:

Step 1	$N\lg N$
Step 3	$N\lg N$
Step 6	$(N/2)\lg(N/2)$
Step 8	$(N/2)\lg(N/2)$
Overall	$\theta(3N\lg N)$

The above method has a complexity of $\theta(3n\lg n)$ which amounts to a reduction of 25% over prior known methods.

The main point is that after evaluating $U+L$, we separate U and L by leveraging the fact that $U\approx X_u$. We show that

$$X_u-\mathcal{U}(\hat{Q}P')=\delta\leq 4 \quad (19)$$

In general let integers A,B satisfy $A>B>0$. Then

$$\left\lfloor\frac{A}{B}\right\rfloor\geq\frac{A-B}{B} \quad (20)$$

This is the main identity invoked in the proof.

$$\begin{aligned}\mathcal{U}(\hat{Q}P') &= \mathcal{U}(\hat{Q}[2P\%(2^n-1)])\\ &= \mathcal{U}(\hat{Q}2P)\%(2^n-1))\\ &= \mathcal{U}(\hat{Q}\cdot 2P)\end{aligned}\quad (21)$$

$$
\begin{aligned}
\hat{Q} &= \left\lfloor \left\lfloor \frac{2^{(2n-1)}}{P} \right\rfloor \cdot X_u \frac{1}{2^n} \right\rfloor \\
\hat{Q} &\geq \left\lfloor \left(\frac{2^{(2n-1)} - P}{P} \right) \cdot X_u \frac{1}{2^n} \right\rfloor \\
&= \left\lfloor \frac{2^{(n-1)} X_u}{P} - \frac{X_u}{2^n} \right\rfloor \\
&\geq \left\lfloor \frac{2^{(n-1)} X_u}{P} - 1 \right\rfloor \\
&\geq \left\lfloor \frac{2^{(n-1)} X_u}{P} \right\rfloor - 1 && (22) \\
\mathcal{U}(\hat{Q}2P) &= \left\lfloor \frac{\hat{Q}2P}{2^n} \right\rfloor \\
&\geq \left\lfloor \left(\left\lfloor \frac{2^{(n-1)} X_u}{P} \right\rfloor - 1 \right) \frac{2P}{2^n} \right\rfloor && (23) \\
&\geq \left\lfloor \left(\frac{2^{(n-1)} X_u - P}{P} - 1 \right) \frac{2P}{2^n} \right\rfloor \\
&\geq \left\lfloor \left(\frac{2^{(n-1)} X_u}{P} - 2 \right) \frac{2P}{2^n} \right\rfloor && (24)
\end{aligned}
$$

$$
\mathcal{U}(\hat{Q}2P) = \left\lfloor X_u - 2 \cdot \frac{P}{2^{n-1}} \right\rfloor \qquad (25)
$$

Since $2^n > P > 2^{n-1}$, $\quad 4 > 2 \cdot \frac{P}{2^{n-1}} > 2$ we get

$$
\begin{aligned}
\mathcal{U}(\hat{Q}2P) &\geq \lfloor (X_u - 4) \rfloor = X_u - 4, \text{ Hence} \\
\delta &= X_u - \mathcal{U}(\hat{Q}P') \leq 4 && (26)
\end{aligned}
$$

We find the actual value of δ as follows:

$$
\begin{aligned}
Z - X_u &= U + L - X_u = L - \delta \Rightarrow && (27) \\
\delta &= Z - X_u - L && (28)
\end{aligned}
$$

The key is to realize that since δ is a small value, only the least-significant words of Z, X_u and L suffice. Since L is the lower half of the product $\hat{Q}2P$, it takes only $O(1)$ work to find out the least significant word of L. Hence

$$
\begin{aligned}
\delta = \{&\text{LeastSignificantWord}(Z) - \\
&\text{LeastSignificantWord}(X_u) - \\
&\text{LeastSignificantWord}(\hat{Q}2P)\} \qquad (29)
\end{aligned}
$$

In fact 2 least significant bits would suffice but it is easier to subtract off words rather than access bits within the words. Next we briefly explain speeding up of Montgomery's method

Algorithm 2: Faster Montgomery Reduction

Given a $2n$ bit number $X = X_u \cdot 2^n + X_l$ and an n bit modulus $P > 2^{(n-1)}$ find $X\%P$.

Pre-computation : Let $R = 2^n$. Several entities are precomputed:
(1) Numbers R^{-1} and P' such that

$$
\begin{aligned}
RR^{-1} - P'P &= 1 \quad \text{and} \quad 0 < R^{-1} < P \\
&\text{and} \quad 0 < P' < R \qquad (30)
\end{aligned}
$$

This pre-computation typically involves GCD-like steps.
(2) Transform X to T, a number in the proper residue class

<u>*Part 1*</u> :
find $m = ((T \mod R)P') \mod R$
$T \mod R = T_u$, the upper half of T
/* main complexity lies in performing the product $T_u \times P'$ which needs
a full **linear convolution** requiring $2N\lg N$ work */

<u>*Part 2*</u> :
find $t = (T + mP)/R$
/* main complexity lies in finding the product $m \times P$.
Here since t is guaranteed to be an integer, lower half of mP satisfies

$$
(\mathcal{L}(mP) + \mathcal{L}(T))\%R = 0 \Rightarrow \mathcal{L}(mP) = \begin{cases} 0 \\ 2^n - T_l \end{cases} \qquad (31)
$$

Consequently we perform a **cyclic convolution** to obtain
$Z = \mathcal{U}(mP) + \mathcal{L}(mP)$ and then obtain the upper half. We show only the last few (correction) steps:

```
Last Step:   $t = Z + T_l$
             if $(t > 2^n)$  $t = t - 2^n + 1$
             $t = t + T_u$
              return   $t\%R$                    (32)
```

The complexity again is $\theta(3N\lg)$ an improvement of 25%.

III. Implementation

The separation principle (viz., replacing a linear convolution with a cyclic convolution and separating the halves using other information available due to an intrinsic redundancy in the operations) was exhaustively verified for small wordlengths (upto $n = 8, 2n = 16$).

To demonstrate the speedup, we ran the algorithms (Algorithm 1 and Algorithm 2 above) for all wordlengths from $n = 2^6 = 1024$ to $n = 2^{21} = 2097152$. At $n = 2^{22}$ the floating point precision was observed to introduce errors (this is consistent with the findings reported in the literature [13], [17]. Following the established practices in literature we used radix $r = 2^{16}$ for the FFT computations, so that each n bit integer becomes a string of $N = \frac{n}{16}$ digits, with each digit being a 16 bit number. With this representation the an n point FFT on the original data is equivalent to an $N = \frac{n}{16}$ point FFT of the digit string.

For each wordlength, we implemented the following using the functions available in the GMP (GNU Multiple Precision library) and measured the execution delay (averaged over a large number of samples)

(1) Time for our enhanced Barrett reduction, i.e., Algorithm 1: $T_{\text{FFT\_BAR}}(n)$

(2) Time for an $n \times n$ multiply using complex floating point FFT: $T_{\text{FFTM}}(n)$

(3) Time for GMP's native multiply method: $T_{\text{GMPM}}(n)$
(they use different methods at increasing wordlengths, starting with brute-force $O(n^2)$ convolution, then Karatsuba method, then Toom's method $\cdots$. Finally for large wordlengths they use Schonhage-Strassen instead of complex FFT)

(4) Time for Barret reduction implemented using GMP's native multiply: $T_{\text{GMP\_BAR}}(n)$

(5) Time for GMP's native remaindering: $T_R(n)$

(6) Time for our enhanced Montgomery reduction, i.e., Algorithm 2: $T_{\text{FFT\_MTG}}(n)$

(7) Time for conventional Montgomery reduction (without our enhancements) implemented using GMP functions: $T_{\text{GMP\_MTG}}(n)$

Now more details of the implementation: for each word length, for each of the above cases, at most 100,000 randomly generated pairs (X,P) are run. For each P value, 10 X values are run (so that the total number of random P values used is 10,000). Furthermore we arbitrarily set a time limit of 5 hours for any single case (so in the worst case the run for one wordlength cannot exceed 30 hours). For such large wordlengths, covering a significant fraction of the input space is impossible. So we decided to select reasonable number of cases to run to ensure that the data could be gathered within about a week or so. The only time counted is the reduction itself. Pre and post processing times are ignored. All the cases were run on a 2.5 GHz Pentium IV box with 1 GB RAM, running FreeBSD 4.7, GMP library version 4.1.3 and FFTW version 3.0.1 [18]. The machine was kept in a single user mode, isolated from the network for the entire duration of the runs. It did not run anything (like X windows ....) except the bare minimum number of core processes needed to be able to carry out the experiments.

Since the main complexity in all the algorithms of interest lies in multiply operations, the proper comparison is to normalize everything by the average time delay of the multiply operation used in the respective remaindering methods.

It is expected that the ratio

$$\Delta_{\text{FFT\_BAR}} = \frac{T_{\text{FFT\_BAR}}(n)}{T_{\text{FFTM}}(n)} \approx 1.5 \quad \text{for} \quad n \geq n_t \qquad (33)$$

where n_t is the threshold above which transform based multiplication is the most efficient (as opposed to Karatsuba or other methods). This is because Algorithm1 performs one full multiply (linear convolution) and a half multiply (cyclic convolution).
In contrast the ratio

$$\Delta_{\text{GMP\_BAR}} = \frac{T_{\text{GMP\_BAR}}}{T_{\text{GMPM}}} \approx 2 \qquad (34)$$

for $n \geq n_t$ since the prior known methods would do two linear convolutions.

For Montgomery's method we ignore the pre-and post processing since that method is mainly used for modular exponentiation wherein the main complexity is in repeated squaring and remaindering, not in the pre and post processing. As a result, the same values can be expected for Montgomery, viz.,

$$\Delta_{\text{FFT\_MTG}} = \frac{T_{\text{FFT\_MTG}}}{T_{\text{FFTM}}} \approx 1.5 \qquad (35)$$

$$\Delta_{\text{GMP\_MTG}} = \frac{T_{\text{GMP\_MTG}}}{T_{\text{GMPM}}} \approx 2 \qquad (36)$$

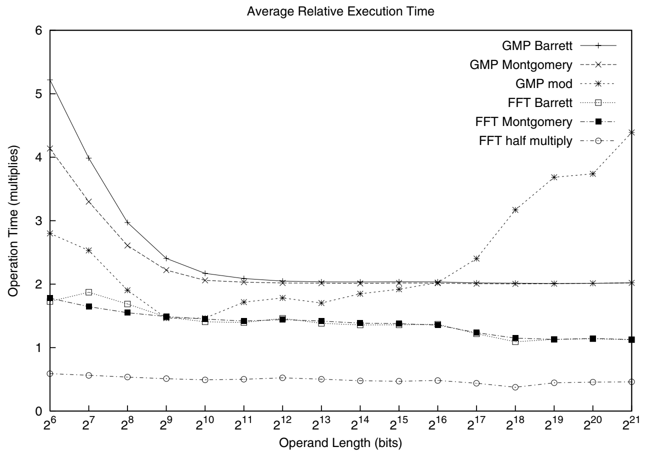

FIG. 1. AVERAGE NORMALIZED RUN-TIMES OF ALL ALGORITHMS

Figure 1 shows the average execution times of all algorithms normalized (divided) by the average delay of multiplication used in the respective methods. The correspondence between plot label and algorithm is indicated in the table below

Plot label	$\longleftrightarrow$	Algorithm
GMP-Barrett	$\longleftrightarrow$	Barret reduction
	using	GMP's native multiply
GMP-Montgomery	$\longleftrightarrow$	Montgomery reduction
	using	GMP's native multiply
GMP-mod	$\longleftrightarrow$	GMP's native remaindering
FFT_Barrett	$\longleftrightarrow$	Algorithm 1
FFT-Montgomery	$\longleftrightarrow$	Algorithm 2

The plots demonstrate that the ratios come out as expected.

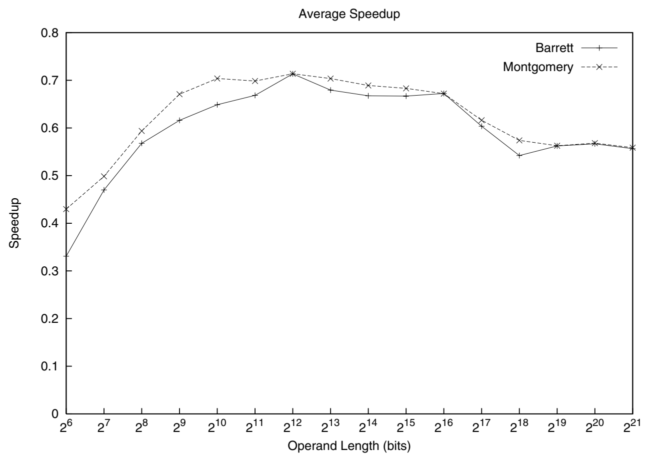

FIG. 2. SPEEDUP RATIOS

Figure 2 shows the speedup ratios

$$S_{\text{FFT\_BAR}} = \frac{\Delta_{\text{FFT\_BAR}}}{\Delta_{\text{GMP\_BAR}}} \approx \frac{1.5}{2} = 0.75 \quad (37)$$

$$S_{\text{FFT\_MTG}} = \frac{\Delta_{\text{FFT\_MTG}}}{\Delta_{\text{GMP\_MTG}}} \approx \frac{1.5}{2} = 0.75 \quad (38)$$

The above equations define the speedup ratios and indicate their approximate expected values.

Again, Figure 2 shows that the speedup ratios come out as expected. It illustrates that our faster Barrett and Montgomery methods (Algorithms 1 and 2) take about 70% of the time needed by the respective baseline methods (i.e., Barret and Montgomery without our enhancements). This is a 30% speedup, which is slightly better than the theoretically predicted 25% improvement. We conjecture that the extra gain reflects improvements in the coefficients of the non-dominant terms. Let the total computational effort be expressed as a function of the wordlength:

$$\text{Effort} \;= C_1 \cdot (n \lg n) + C_2 \cdot (n) + C_3 \cdot (\lg n) + C_4 \quad (39)$$

Our speedup ratios evaluate only the improvement in the coefficient of the dominant term (i.e., C_1). The other coefficients are likely to improve as well, which leads to a gain better than that predicted on the basis of C_1 alone.

IV. DISCUSSION AND CONCLUSIONS

A. Application to Modular Exponentiation

Here the problem is to evaluate $X^Y\%P$. If $X\%P = R_0$, repeated squaring and reducing yields $X^2\%P,\ X^4\%P,\ X^{16}\%P,\ \cdots X^{2^k}\%P$.
The pseudo-code to find the result is:

```
Let $Y = (y_{k-1}y_{n-2}\cdots y_0)$ in binary.
initialize: $R_{\text{pow}} = X\%P,\ R = 1$
for (i = 0; i < k; i++){
    if $(y_i == 1)\ R = (R \cdot R_{\text{pow}})\%P$
    if $(i < k-1)\ R_{\text{pow}} = (R_{\text{pow}} \cdot R_{\text{pow}})\%P$
}
```

Both the reductions inside the loop can be done using either Either Barret or Montgomery. Each of the two reductions in the loop involves two steps:
(1) First compute $(R \cdot R_{\text{pow}})$ or $(R_{\text{pow}})^2$ which yields a $2n$ bit value V.
(2) Reduce $V\%P$.

It is clear that the second step can be sped up by 25% using our algorithms. Since R_{pow} is a common operand in both multiplications, each iteration of the loop can be done with
$[(2n\lg n + 3n\lg n)] + [2n\lg n + 3n\lg n]$ effort using our methods
(instead of $[(2n\lg n + 4n\lg n)] + [2n\lg n + 4n\lg n]$ effort for prior known methods)
This results in a ratio of $\frac{5}{6}$ or an improvement of about 17%. We are exploring methods to further speed up modular exponentiation.

Note that the constancy of modulus P is less important for modular exponentiation. Since the repeated-squaring loop is iterated (relatively) large number of times, the initial setup time to compute $P_{\text{inv}}$for Barrett reduction is not a dominant contributor to the complexity. Likewise, the pre and post-processing required by Montgomery's method is not the main contributor to the complexity. Hence the speedup mechanisms we proposed are always applicable to modular exponentiation independent of whether the modulus P is known ahead of time.

In [19]. it is shown how to do n-bit modular multiplication in time $(3 + 1/3)n \cdot L(n)$ after a certain modulus precomputation; where $n.L(n)$ is the time for multiplication modulo the special modulus $(2^n + 1)$. Our results are consistent with those in [19].

B. Conclusions

We demonstrated a fundamental separation principle: in a modular reduction based on long multiplications, the linear convolution required by one of the two long multiplications can be replaced by a cyclic convolution and the halves can be separated using other information available due to the intrinsic redundancy of the operations. This reduces the total number of operations by about 25%. We demonstrated that both Barrett's and Montgomery's methods can be sped up by using the aforementioned fundamental principle. The theory was experimentally validated via comprehensive simulations. It was shown that a direct application of this algorithm to modular exponentiation (either using Barrett's or Montgomery's methods) can be expected to yield about about 17% speedup.

Acknowledgment The authors would like to thank Dr. David Fu from the National Security Agency (NSA) for his feedback during the course of this project.

References

[1] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996. http://www.cacr.math.uwaterloo.ca/hac/.

[2] D. Naccache and D. M'Raïhi, "Cryptographic smart cards," *IEEE Micro*, pp. 14–24, June 1996.

[3] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances In Cryptology – CRYPTO '86 (LNCS 263)* (A. M. Odlyzko, ed.), pp. 311–323, 1987.

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

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2nd ed., 2001.

[6] D. E. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, vol. 2. Addison-Wesley, 3rd ed., 1998.

[7] D. J. Bernstein, "Multidigit multiplication for mathematicians," *Advances in Applied Mathematics*, Aug. 12 2001.

[8] Kerry Bloodworth's web-site http://careybloodworth.tripod.com/index.htm.

[9] D. G. Cantor and E. Kaltofen, "On fast multiplication of polynomials over arbitrary algebras," *Acta Informatica*, vol. 28, no. 7, pp. 693–701, 1991.

[10] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen. (German) [Fast multiplication of large numbers]," *Computing*, vol. 7, no. 3–4, pp. 281–292, 1971.

[11] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Advances In Cryptology – CRYPTO '93 (LNCS 773)* (D. R. Stinson, ed.), pp. 175–186, 1994.

[12] GNU Multi Precision Library Reference Manual http://www.gnu.org/software/gmp/manual/.

[13] R. Crandall and B. Fagin, "Discrete weighted transforms and large-integer arithmetic," *Mathematics of Computation*, vol. 62, pp. 305–324, Jan. 1994.

[14] FFT De-mystified http://http://www.eptools.com/tn/T0001/INDEX.HTM.

[15] Dan Bernstein's web-site, http://cr.yp.to.

[16] see the discussion on the half cyclic convolution on page 220 of a recent textbook draft (2004) http://www.jjj.de/fxt/fxtbook.pdf.

[17] C. Percival, "Rapid multiplication modulo the sum and difference of highly composite numbers," *Mathematics of Computation*, vol. 72, no. 241, pp. Pages 387–395, 2002.

[18] http://www.fftw.org.

[19] A. Schönhage, "Fast algorithms: a multitape turing machine implementation," 1994.