Modular Multiplication using Redundant Digit Division

Ping Tak Peter Tang
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054
Peter.Tang@intel.com

Abstract

Most implementations of the modular exponentiation, $M^E \bmod N$, computation in cryptographic algorithms employ Montgomery multiplication, $ABR^{-1} \bmod N$, instead of modular multiplication, $AB \bmod N$, even the former requires some transformational overheads. This is so because a state-of-the-art Montgomery multiplication implementation has a performance advantage over direct modular multiplication based on the Barrett algorithm that more than compensates for the overhead. In this paper, we present a direct modular multiplication method that is comparable in speed to Montgomery multiplication. One consequence is that when the exponent in small, direct computation (which does not incur the transformational overhead) using the modular multiplication algorithm presented here results in practical performance gain. For the exponent 17, for instance, which requires five modular multiplication, a saving of up to 40% can be achieved.

1. Introduction

Computing a modular exponentiation, $M^E \bmod N$, is a key operation in cryptographic computation. Typically M, N, and E are large integers (for example each being 2048 bit long). In some special situations such as digital signature verification [3], only M and N are large and E is small (e.g. 2, 17, $2^{16}+1$, etc).

Practical algorithms for modular exponentiation are well known and they all employ the method of repeated squaring [3] or variations. For example,

$$M^5 \bmod N = (((M^2) \bmod N)^2 \bmod N) \cdot M \bmod N.$$

This can be called a direct method as it works directly with the input value M.

An alternative is that of a transformed method. Here, M is first transformed to $\tilde{M} = MR \bmod N$ for a special R. Then $M^5 \bmod N$ equals $T \cdot R^{-1} \bmod N$ where

$$T = ((\tilde{M}^2 R^{-1} \bmod N)^2 R^{-1} \bmod N) \cdot \tilde{M} R^{-1} \bmod N.$$

The computing of $\tilde{M}$ is an initial transformation, and the last multiplication by R^{-1} is a transformation to "undo" the initial transformation.

As described, we see that the main cost of a modular exponentiation is a sequence of operations of the form

$$AB \bmod N, \quad \text{or} \quad ABR^{-1} \bmod N.$$

The mod N operation in $AB \bmod N$ is deemed expensive as a division by N is present. A popular direct method is proposed by Barrett [1] where roughly speaking the division by N is replaced by a multiplication of $1/N$.

A popular transformed method is proposed by Montgomery [5] where R is chosen to be a power of two. The operation $ABR^{-1} \bmod N$ only requires large-integer multiplication followed by division by R. Exploiting this property appropriately, $ABR^{-1} \bmod N$ can be implemented more efficiently than $AB \bmod N$ using Barrett's method. Consequently, Montgomery's method is the method of choice in many software implementation of modular multiplication inside a modular exponentiation.

The main thesis of this paper is that direct modular multiplication $AB \bmod N$ can be carried out as economically as $ABR^{-1} \bmod N$. Thus, direct modular exponentiation is competitive in performance to transformed modular exponentiation. And in the case where the exponent is small, direct method will be faster as it does not incur the transformation overhead. Take the exponent 17 for example. A direct method requires 5 modular multiplications while a transform method based on Montgomery method requires 5 Montgomery multiplications and 2 modular multiplications. If the performance of a modular multiplication is the same as a Montgomery multiplication, the direct method is faster by $(7-5)/5 = 40\%$.

Barrett	Montgomery
Product Step:	
$P \leftarrow A \times B$	$P \leftarrow A \times B$
Cost: mul $L \times L \rightarrow 2L$	Cost: mul $L \times L \rightarrow 2L$
Get Quotient:	
$Q \leftarrow \left\lfloor \frac{\lfloor P/r^{L-1} \rfloor W}{r^{L+1}} \right\rfloor$	$Q \leftarrow PN' \bmod R$
Cost: mul $(L+1) \times L \rightarrow L_{\text{high}}$	Cost: mul $L \times L \rightarrow L_{\text{low}}$
Reduction:	
$P \leftarrow (P \bmod r^{L+1}) -$ $(QN \bmod r^{L+1})$	$P \leftarrow (P + QN)/R$
Cost: mul $L \times L \rightarrow (L+1)_{\text{low}}$	Cost: mul $L \times L \rightarrow L_{\text{high}}$
Fix Up:	
If $P < 0$, $P \leftarrow P + r^{L+1}$	If $P \geq N$,
While $P \geq N$, $P \leftarrow P - N$	$P \leftarrow P - N$

Table 1. High-Level Comparison of Barrett and Montgomery Methods

Montgomery	New Method
Product:	
$P \leftarrow 0$	$P \leftarrow a_{L-1}B$
Reduction:	
For $j = 0, 1, \ldots, L-1$	For $j = L-1, L-2, \ldots, 1$
$q_j \leftarrow (a_j b_0 + p_0)n' \bmod r$	Get q_j using P,a_j,B
$P \leftarrow (P + a_j B + q_j N)/r$	$P \leftarrow r(P - q_j N) + a_{j-1}B$
End	End
	Get q_0, $P \leftarrow P - q_0 N$
Fix up:	
If $P \geq N$, $P \leftarrow P - N$	If $P \geq N$, $P \leftarrow P - N$

Table 2. Montgomery Method and New Direct Method

2. Background

Let us set up the notation here to aid all subsequent discussions. A, B, N are multi-word integers:

$$\begin{aligned} A &= a_{L-1}r^{L-1} + a_{L-2}r^{L-2} + \cdots + a_1 r + a_0 \\ B &= b_{L-1}r^{L-1} + b_{L-2}r^{L-2} + \cdots + b_1 r + b_0 \\ N &= n_{L-1}r^{L-1} + n_{L-2}r^{L-2} + \cdots + n_1 r + n_0 \end{aligned}$$

Each of the words a_j, b_j, and n_j are "native" integers and r is the "radix." For example, a_j, b_j, and n_j can be 32-bit unsigned integers and A, B, N being 2048 bit long. This corresponds to $r = 2^{32}$ and $L = 64$. For a transformed method, R is typically r^L. We also assume that $0 < A, B < N$.

We first outline in Table 1 both the Barrett and Montgomery methods. The former computes $AB \bmod N$ and the latter $ABR^{-1} \bmod N$. Initialization in Barrett's method computes $W = \lfloor r^{2L}/N \rfloor$, and that in Montgomery's method, $N' = -N^{-1} \bmod R$.

In this form, the cost of the Barrett and Montgomery methods are quite comparable, roughly dominated by 2.5 full L-by-L word multiplication. The former is at a slight disadvantage as it requires a $(L+1)$ word by L word multiplication instead of a L by L, and that one may have to go through multiple times of adjustment (up to twice).

However, the Montgomery method admits an optimization for the "Get Quotient" step that allows the cost of computing Q at the stated L-by-L word multiplication, which costs $O(L^2)$ native integer multiplications, to that of L native integer multiplications. In addition to this saving of computational cost, the multiplication $P \leftarrow A \times B$ can be integrated into the reduction step, saving data movement cost as well as improving data locality. These optimizations are outlined in left column of Table 2. Here, n' is a native integer $n' = -{n_0}^{-1} \bmod r$. The key is that instead of trying to determine the Q completely before the reduction $(P + QN)/R$, it is more economical to compute $(P + QN)/R$ together with determining Q. One obtains the first digit q_0 of Q and immediately computes $P \leftarrow (P + q_0 N)/r$. The resulting P value will be used to obtain the next digit q_1 and the computation is repeated. That obtaining each q_j this way costs only one native integer multiplication results in savings stated above. This optimization method is well known (cf. [3, 4]). On the other hand, similar optimizations for the Barrett method is absent in standard texts or relevant literature ([3, 1]). Moreover, the overhead of having to compute $W = \lfloor r^{2L}/N \rfloor$ greatly diminishes the advantage of a direct method in the case of small exponent exponentiation.

In this paper, we will present a direct modular multiplication that is as economical as Montgomery's method. As a preview, our modular multiplication is outlined in Table 2 aside the Montgomery method. It is seen here that the costs of the two methods are comparable except possibly for that of the computation for q_j where we have omitted the details. That cost, though slightly higher than the corresponding Montgomery method, is only a small portion of the overall cost which is dominated by the updating of P. In subsequent sections, we present the algorithm in detail, analyze its correctness, and sketch one implementation.

3. Redundant Digit Division

The general idea we pursue here is that we compute a rather good approximation Q to the integer quotient AB/N

in a word-by-word manner starting at the most significant word. That is we compute integer values q_{L-1}, q_{L-2}, down to q_0, where

$$Q = q_{L-1}r^{L-1} + q_{L-2}r^{L-2} + \cdots + q_1 r + q_0.$$

This Q is a good approximation in the sense that

$$0 \le A\ B - Q\ N < 2N$$

and thus at most one more subtraction is needed after the remainder $A\ B - Q\ N$ is computed. This idea can be realized without too much effort if we consider the situation where $P = AB$ is computed before any reduction step. Given

$$P = p_{2L-1}r^{2L-1} + p_{2L-2}r^{2L-2} + \cdots + p_1 r + p_0$$

and

$$N = n_{L-1}r^{L-1} + n_{L-2}r^{L-2} + \cdots + n_1 r + n_0$$

the usual high-radix division method can be applied in general. For example, simply consider P and N as $2L$ digit floating point numbers and carry out the division to obtain L quotient digits and a L digit remainder. This kind of algorithm typically obtain one digit at a time based on approximation of the running (partial) remainder and the reciprocal of the divisor. A redundant digit system is required and $\{-r/2, \ldots, r/2\}$ or $\{-r+1, \ldots, r-1\}$ are common choices. These methods are mostly designed with hardware implementation in mind. Our situation is different as it is more natural to stay with unsigned integers of length $\log_2(r)$. So we will adapt to use a non-negative redundant digit system. In addition, integrating the product and the reduction steps are highly desirable [4]. This means that AB is accumulated one $a_j B$ at a time, while always in conjunction with a subtraction by some $q_k N$. But this integration requires more challenging analyses as the estimate of quotient digits are now based on an incomplete remainder (some parts are yet to be computed), which needs to be kept non-negative. Here is a sketch where superscripts are added to keep track of the iteration number.

$P^{(L-1)} \leftarrow a_{L-1}B$
For $j = L-1, L-2, \ldots, 1$ do
  Get q_j using approximate $P^{(j)}$, a_{j-1}, B
  $P^{(j-1)} \leftarrow r(P^{(j)} - q_j N) + a_{j-1}B$
End do
Get q_0 using approximate $P^{(0)}$
Compute $P \leftarrow P^{(0)} - q_0 N$
Compute if necessary $P \leftarrow P - N$

The crux of a practical algorithm lies in the ability to obtain the q_j at low cost while avoiding the q_j to be too wide. In order to obtain such q_j's, we must make use of more than

// $\texttt{FP}(\cdot)$ means rounds to nearest mode
Let $u \leftarrow \texttt{FP}(N \gg ((L-1)\ell_{\rm int} - e))$, $u \leftarrow u + 2$
// since $\ell_{\rm fp} \ge \ell_{\rm int} + e + 2$, $u = \lfloor 2^e N / r^{L-1} \rfloor + 2$ exactly
Let $u \leftarrow \texttt{FP}(r/u)$ // one FP division
$P^{(L-1)} \leftarrow a_{L-1}B$ // result in $L+1$ words
For $j = L-1, L-2, \ldots, 1$ do
  $T \leftarrow [(a_{j-1} \gg \ell_{\rm int}/2) \times (b_{L-1} \gg \ell_{\rm int}/2)] \gg (\ell_{\rm int} - e)$
  // $T = \lfloor 2^{-(\ell_{\rm int}-e)} \lfloor a_{j-1}/\sqrt{r} \rfloor \lfloor b_{L-1}/\sqrt{r} \rfloor \rfloor$ exactly.
  $W_j \leftarrow \texttt{FP}(P^{(j)} \gg (L\ell_{\rm int} - e)) + \texttt{FP}(T)$
  // will prove $W_j = \lfloor 2^e P^{(j)}/r^L \rfloor + T$ exactly.
  $q_j \leftarrow \lfloor \texttt{FP}(u \times W_j) \rfloor$ // can prove $0 \le q_j < 2r$
  // compute uW_j in FP, then trunc to int
  $P^{(j-1)} \leftarrow r(P^{(j)} - q_j N) + a_{j-1}B$
  // can prove $0 \le P^{(j-1)} < 2rN$
End do
$W_0 \leftarrow \texttt{FP}(P^{(0)} \gg (L\ell_{\rm int} - e))$
$q_0 \leftarrow \lfloor \texttt{FP}(u \times W_0) \rfloor$ // can prove $0 \le q_0 < 2r$
Compute $P \leftarrow P^{(0)} - q_0 N$ // can prove $0 \le P < 2N$
Compute if necessary $P \leftarrow P - N$

Figure 1. Redundant Digit Direct Modular Multiplication Algorithm

$\ell_{\rm int} = \log_2(r)$ leading bits of N and the partial remainder. We propose the use of a floating point type FP as a convenient way to facilitate computations involving more than $\ell_{\rm int}$ bits. Here are our parameterizations. $\ell_{\rm int} = \log_2(r)$ is the number of bits of the native integers a_j, b_j, n_j. We assume $\ell_{\rm int}$ to be even (which is realistic anyway) to ease our presentation. Let e denote the number of extra precision one chooses to use and $\ell_{\rm fp}$ denote the precision of FP. Our algorithm functions correctly as long as $6 \le e \le \ell_{\rm int}/2$ and $\ell_{\rm int} + e + 2 \le \ell_{\rm fp}$. The algorithm behaves better (to be explained later) for a larger e. For instance, in the case of $\ell_{\rm int} = 32$ (32-bit integer) and $\ell_{\rm fp} = 53$ (IEEE double precision), one can use $e = 16$. For $\ell_{\rm int} = 16$, we can use single precision FP and set $e = 6$. For $\ell_{\rm int} = 64$, one can use a (lazily) simulated FP type using two double precision type offering a type comparable to $\ell_{\rm fp} = 74$. (We will touch on this "lazy" type in Section 4). The algorithm maintains a partial remainder $0 \le P < 2rN$ which is represented by $(L+1)$ words and one bit. We use the notation $P \gg k$ to logical (unsigned) right shifting by k bits, which is equivalent to the mathematical operation of $\lfloor P/2^k \rfloor$. The outline of our algorithm is given in Figure 1. Except for u and W_j, all other variables are of unsigned integer type (either scalar or multi-word). Note that the superscripts in $P^{(j)}$ are merely for bookkeeping. In an actual implementation, they utilize the same integer array.

Conceptually, u and W_j approximate $1/(rN)$ and $rP^{(j)} + a_{j-1}B$ respectively. Thus $\lfloor uW_j \rfloor$ can be used as a quotient digit. The analysis to follow will establish that as long as u and W_j satisfy some accuracy requirements, the

algorithm maintains the following bounds:(1) $0 \le q_j < 2r$, (2) $0 \le P^{(j)} < 2rN$. These bounds are advantageous because

1. q_j is at most one bit more than one word
2. $P^{(j)}$ is at most one bit more than $L+1$ words
3. $P^{(j)}$ is never negative

Something stronger is true. The bound on $P^{(j)}$ is in the form $P^{(j)} < (1-\lambda)^{-1} rN$ for small λ (for $\ell_{\rm int} = 32$ and $\ell_{\rm fp} = 53$, we can have $0 \le \lambda < 2^{-11}$). Thus, for input value N that happens to satisfy $(1-\lambda)^{-1}N < r^L$, $P^{(j)}$ in fact always fit in $L+1$ words. Additionally, $q_j < r$ with probability no less than $1-\lambda$.

We now begin proving theorems to establish the correctness of the algorithm. We will assume N to be reasonably normalized: $(r^L/N) < \sqrt{r}$, that is, the upper half of n_{L-1} is non zero. Handling of unnormalized N is not difficult and we discuss that at the end of the paper.

4. Analysis of Algorithm

We will now perform an analysis on the algorithm in Figure 1. By design, both approximations u and the W_j are biased downwards. That is both u and W_j are no bigger than the actual value they aim at approximating. While the specific errors in W_j and the floating point related error in computing uW_j are different at each step. The design guarantees that the errors are all uniformly below some thresholds. We establish these facts by the first two theorems.

Theorem 1 *For $j = L-1, L-2, \ldots, 1$, $0 \le P^{(j)} < 2r^{L+1}$ implies*

$$W_j = \frac{2^e}{r^{L+1}}(rP^{(j)} + a_{j-1}B) - \delta_j,$$

and $0 \le P^{(0)} < 2r^{L+1}$ implies

$$W_0 = \frac{2^e}{r^L}P^{(0)} - \delta_0,$$

where all the δ_j are bounded uniformly $0 \le \delta_j < \delta_W$ for some fixed $\delta_W \le 4$.

Proof 1 *For $j = L-1, L-2, \ldots, 1$ we define:*

$$\hat{a}_{j-1} = \left\lfloor \frac{a_{j-1}}{\sqrt{r}} \right\rfloor,\ \hat{B} = \left\lfloor \frac{B}{r^{L-1/2}} \right\rfloor,\ \hat{P}^{(j)} = \left\lfloor \frac{2^e}{r^L}P^{(j)} \right\rfloor$$

From the algorithm,

$$T = \left\lfloor \frac{2^e}{r}\hat{a}_{j-1}\hat{B} \right\rfloor.$$

Since $\ell_{\rm fp} \ge \ell_{\rm int} + e + 2$, W_j is guaranteed to be equal to $\hat{P}^{(j)} + T$ exactly if $0 \le P^{(j)} < 2r^{L+1}$. Thus, clearly the four quantities satisfy

$$0 \le \left(\frac{a_{j-1}}{\sqrt{r}}\right) - \hat{a}_{j-1},\ \left(\frac{B}{r^{L-1/2}}\right) - \hat{B},$$
$$\left(\frac{2^e}{r^L}P^{(j)}\right) - \hat{P}^{(j)},\ \left(\frac{2^e}{r}\hat{a}_{j-1}\hat{B}\right) - T < 1.$$

Let us estimate

$$\frac{2^e}{r^{L+1}}a_{j-1}B - T$$

which is equal to

$$\left(\frac{2^e a_{j-1}B}{r^{L+1}} - \frac{2^e \hat{a}_{j-1}B}{r^{L+1/2}}\right) +$$
$$\left(\frac{2^e \hat{a}_{j-1}B}{r^{L+1/2}} - \frac{2^e}{r}\hat{a}_{j-1}\hat{B}\right) + \left(\frac{2^e}{r}\hat{a}_{j-1}\hat{B} - T\right)$$

which is

$$\frac{2^e}{r^{L+1/2}}B\left(\frac{a_{j-1}}{\sqrt{r}} - \hat{a}_{j-1}\right) +$$
$$\frac{2^e}{r}\hat{a}_{j-1}\left(\frac{B}{r^{L-1/2}} - \hat{B}\right) + \left(\frac{2^e}{r}\hat{a}_{j-1}\hat{B} - T\right)$$

This quantity clearly lies in the range $[0, 3)$. Thus

$$0 \le \frac{2^e}{r^{L+1}}\left(rP^{(j)} + a_{j-1}B\right) - W_j < 4.$$

For W_0, simply note that $P^{(0)}$ and $\hat{P}^{(0)}$ defined the same way gives

$$0 \le \frac{2^e}{r^L}P^{(0)} - \hat{P}^{(0)} < 1.$$

$0 \le P^{(0)} < 2r^{L+1}$ implies $W_0 = \hat{P}^{(0)}$ exactly. Thus

$$W_0 \le \frac{2^e}{r^L}P^{(0)} - \delta_0,$$

$0 \le \delta_0 < 1$ (so $\delta_0 < 4$). The proof is thus complete.

Theorem 2 *For $j = L-1, L-2, \ldots, 0$, the quotient digits $q_j = \left\lfloor \frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j) \right\rfloor$, where $0 \le \epsilon_j < \epsilon_q$ for a fixed $\epsilon_q < (2(\frac{r^L}{N})+1)(2^e r)^{-1}$. This holds regardless of the sign of W_j. If $W_j \ge 0$, then $q_j \ge 0$ also.*

Proof 2 *Define $\hat{N} = \left\lfloor \frac{2^e}{r^{L-1}}N \right\rfloor + 2$. Thus*

$$\hat{N} = \frac{2^e}{r^{L-1}}N + \alpha, \qquad 1 < \alpha \le 2.$$

Consequently,

$$\hat{N} = \frac{2^e}{r^{L-1}}N(1+\beta), \qquad \beta = \left(\frac{r^L}{N}\right)\frac{\alpha}{2^e r}.$$

Thus

$$\beta_{\min}=\left(\frac{r^L}{N}\right)\frac{1}{2^e r}\le\beta\le\left(\frac{r^L}{N}\right)\frac{2}{2^e r}=\beta_{\max}$$

In particular β is always positive. Next, FP *in round to nearest mode gives*

$$u=\texttt{FP}(r/\hat{N})=\frac{r}{\hat{N}}(1+\gamma_1),$$

and

$$q_j=\left\lfloor W_j\frac{r}{\hat{N}}(1+\gamma_1)(1+\gamma_2)\right\rfloor$$

where $|\gamma_1|,|\gamma_2|\le 2^{-\ell_{\rm fp}}$. Hence,

$$\begin{aligned} q_j &= \left\lfloor \frac{r^L}{2^e}\frac{W_j}{N}(1+\gamma_1)(1+\gamma_2)(1+\beta)^{-1}\right\rfloor \\ &= \left\lfloor \frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j)\right\rfloor \end{aligned}$$

where

$$\epsilon_j=\frac{\beta-(\gamma_1+\gamma_2+\gamma_1\gamma_2)}{1+\beta}.$$

This is valid as long as $1+\beta\neq 0$ which is our case here as β is positive. The expression for ϵ_j when considered as a function of β is continuously differentiable for $\beta>-1$ and has derivative $(1+\gamma_1+\gamma_2+\gamma_1\gamma_2)/(1+\beta)^2$. This is positive as long as the gamma's are less than $1/4$ which is definitely the case for any realistic FP*. So, for any specific (fixed) γ_1 and γ_2, the expression for ϵ_j is smallest at $\beta_{\min}$ and largest at $\beta_{\max}$. Thus, a lower (or upper) bound for ϵ_j is obtained at $\beta_{\min}$ (or $\beta_{\max}$) and when $\gamma_1+\gamma_2+\gamma_1\gamma_2$ is at its maximum (or minimum). Denoting $2^{-\ell_{\rm fp}}$ by $\epsilon_{\rm fp}$, it is easy to see that $\gamma_1+\gamma_2+\gamma_1\gamma_2$ attains its minimum at $\gamma_1=\gamma_2=-\epsilon_{\rm fp}$ and its maximum at $\gamma_1=\gamma_2=\epsilon_{\rm fp}$. Thus, we have*

$$\frac{\beta_{\min}-2\epsilon_{\rm fp}-{\epsilon_{\rm fp}}^2}{1+\beta_{\min}}\le\epsilon_j\le\frac{\beta_{\max}+2\epsilon_{\rm fp}-{\epsilon_{\rm fp}}^2}{1+\beta_{\max}}.$$

Because $\beta_{\min}>(2^e r)^{-1}$, $\beta_{\min}-2\epsilon_{\rm fp}-{\epsilon_{\rm fp}}^2>0$ as long as $\ell_{\rm fp}\ge\ell_{\rm int}+e+2$. Hence $\epsilon_j>0$ always. (This is crucial.) As for an upper bound, we have $\epsilon_j<(2(r^L/N)+1)(2^e r)^{-1}$. Finally, since $u\ge 0$ and thus $W_j\ge 0$ guarantees $\texttt{FP}(uW_j)\ge 0$, implying $q_j\ge 0$ as well. This completes the proof.

We point out that the proof shows the purpose of the biasing by adding 2 to the truncated N is to maintain $\epsilon_j\ge 0$ always. With this bias, as long as the floating point errors cannot be big enough to offset $\beta_{\min}$, $\epsilon_j\ge 0$. Thus it is not even necessary to have IEEE floating point arithmetic. For example, we can easily simulate a FP arithmetic using two double precision number where the leading part only has 24 significant bits. This allows products be computed relatively easily (as the product of the leading portions are error free). This kind of arithmetic operation guarantees $|\gamma|\le 2^{-74}$ as no more than 3 bottom bits of the $24+53=77$ results are corrupted. This FP type is useful, for example, when $\ell_{\rm int}=64$.

Theorem 3 *For $0\le j\le L-1$, as long as $W_j\ge 0$,*

$$0\le\frac{r^{L+1}}{2^e}W_j-rq_jN<rN+\epsilon_q\frac{r^{L+1}}{2^e}W_j.$$

Proof 3 *Theorem 2 says:*

$$q_j=\left\lfloor\frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j)\right\rfloor,$$

where

$$0\le\epsilon_j<\epsilon_q<(2(\frac{r^L}{N})+1)(2^e r)^{-1}.$$

Moreover,

$$0\le\frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j)-q_j<1,$$

Thus, together with $W_j\ge 0$,

$$0\le\frac{r^{L+1}}{2^e}W_j-q_jrN<rN+\epsilon_q\frac{r^{L+1}}{2^e}W_j.$$

Theorem 4 *For $1\le j\le L-1$, $0\le P^{(j)}<2r^{L+1}$ implies*

$$P^{(j-1)}=\frac{r^{L+1}}{2^e}W_j-rq_jN+\frac{r^{L+1}}{2^e}\delta_j.$$

And $0\le P^{(0)}<2r^{L+1}$ implies

$$P=\frac{r^L}{2^e}W_0-q_0N+\frac{r^L}{2^e}\delta_0$$

where the δ_j are those in Theorem 1, that is, $0\le\delta_j<\delta_W$.

Proof 4 *From Theorem 1,*

$$W_j=\frac{2^e}{r^{L+1}}(rP^{(j)}+a_{j-1}B)-\delta_j,\quad j=L-1,\ldots,1$$

for some δ_j where $0\le\delta_j<\delta_W$. Thus,

$$\begin{aligned} P^{(j-1)} &= r(P^{(j)}-q_jN)+a_{j-1}B \quad \textit{(from Algorithm)} \\ &= \frac{r^{L+1}}{2^e}W_j+\frac{r^{L+1}}{2^e}\delta_j-rq_jN. \end{aligned}$$

For P, applying Theorem 1 gives

$$\begin{aligned} P &= P^{(0)}-q_0N \\ &= \frac{r^L}{2^e}W_0-q_0N+\frac{r^L}{2^e}\delta_0. \end{aligned}$$

Theorem 5 *If for a certain j in the range $0 \le j \le L-1$ we have $0 \le P^{(j)} < \alpha r N$ for some α, $1/2 \le \alpha \le 2$, then the followings hold for some λ which satisfies the bound $0 \le \lambda < 2^{-e}(10(\frac{r^L}{N})+2)$:*

1. *$0 \le q_j < 1+\alpha r$ and $0 \le P^{(j-1)} < (1+\lambda\alpha)rN$ for $j \ge 1$.*
2. *$0 \le q_0 < \alpha r$, and $0 \le P < (1+\lambda\alpha)N$ for $j = 0$.*

Proof 5 *First note that $0 \le P^{(j)} < \alpha r N < 2r^{L+1}$ implies $W_j \ge 0$ and $q_j \ge 0$ (from Theorems1 and 2). Consider the case $j \ge 1$. By Theorem 1,*

$$\frac{r^{L+1}}{2^e} W_j = rP^{(j)} + a_{j-1}B - \frac{r^{L+1}}{2^e}\delta_j \le rP^{(j)} + a_{j-1}B$$

as $\delta_j \ge 0$. Now $P^{(j)} < \alpha r N$ by assumption and it is also obvious that $0 \le a_{j-1} < r$, $0 \le B < N$. So

$$0 \le \frac{r^{L+1}}{2^e} W_j \le rP^{(j)} + a_{j-1}B < (1+\alpha r)rN.$$

Since

$$q_j = \lfloor \frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j) \rfloor$$

we have $q_j < 1+\alpha r$. To establish the second part of (1), note that $W_j \ge 0$ and

$$q_j = \lfloor \frac{r^L}{2^e}\frac{W_j}{N}(1-\epsilon_j) \rfloor$$

shows $q_j \le \frac{r^L}{2^e}\frac{W_j}{N}$. In addition,

$$W_j \le \frac{2^e}{r^{L+1}}(rP^{(j)} + a_{j-1}B) \quad \textit{(Thm 1)}.$$

So,

$$P^{(j-1)} = rP^{(j)} + a_{j-1}B - rq_jN \ge 0.$$

On the upper bound:

$$\begin{aligned}
P^{(j-1)} &= \frac{r^{L+1}}{2^e}W_j + \frac{r^{L+1}}{2^e}\delta_j - rq_jN \quad \textit{(Thm 4)} \\
&< rN + \epsilon_q\frac{r^{L+1}}{2^e}W_j + \frac{r^{L+1}}{2^e}\delta_W \quad \textit{(Thm 3)} \\
&< rN + \epsilon_q(1+\alpha r)rN + \frac{r^{L+1}}{2^e}\delta_W \\
&= rN(1+\epsilon_q(1+\alpha r) + \frac{r^L}{N}\frac{\delta_W}{2^e}) \\
&= rN(1+\lambda\alpha)
\end{aligned}$$

where

$$\begin{aligned}
\lambda &= \frac{\epsilon_q}{\alpha} + \epsilon_q r + (\frac{r^L}{N})\frac{\delta_W}{2^e\alpha} \\
&\le \epsilon_q(2+r) + (\frac{r^L}{N})\frac{8}{2^e} \\
&< \left(2(\frac{r^L}{N})+1\right)\frac{2+r}{2^e r} + (\frac{r^L}{N})\frac{8}{2^e} \\
&< 2^{-e}\left(10(\frac{r^L}{N}) + 1 + \frac{2}{r} + \frac{4}{r}(\frac{r^L}{N})\right) \\
&< 2^{-e}\left(10(\frac{r^L}{N})+2\right).
\end{aligned}$$

by estimating $\frac{2}{r} + \frac{4}{r}(\frac{r^L}{N}) \le 1$. This holds if $r \ge 64$ and $(\frac{r^L}{N}) \le \sqrt{r}$. So (1) is established.

Consider $j = 0$. $\frac{r^L}{2^e}W_0 \le P^{(0)} < \alpha r N$ and

$$q_0 = \left\lfloor \frac{r^L}{2^e}\frac{W_0}{N}(1-\epsilon_0) \right\rfloor \Rightarrow q_0 < \alpha r$$

Applying to P the previous analysis on $P^{(0)}$ shows

$$P < N\left(1 + \epsilon_q(1+\alpha r) + \frac{r^L}{N}\frac{\delta_W}{2^e}\right)$$

giving the same expression for $P^{(0)}$ but less a factor of r. The proof is complete.

The following is the main theorem that shows the correctness property of the complete algorithm.

Theorem 6 *Let λ be the value in Theorem 5. As long as $\lambda < 1/2$, the following hold:*

1. *$0 \le P^{(j)} < (1-\lambda)^{-1}rN$ for $j \ge 0$, and $0 \le P < (1-\lambda)^{-1}N$.*
2. *$0 \le q_j < 1 + (1-\lambda)^{-1}r$ for $j \ge 1$, and $0 \le q_0 < (1-\lambda)^{-1}r$.*

The algorithm maintains $0 \le q_j < 1+2r$, $0 \le P^{(j)} < 2rN$, $0 \le P < 2N$. If in addition $\lambda < 1/2 - (4r-2)^{-1}$, $0 \le q_j < 2r$ is maintained.

Proof 6 *First, $P^{(L-1)} = a_{L-1}B$. Since $a_{L-1} \le r-1$ and $B < N$, we have $P^{(L-1)} < (r-1)N < rN$. We now apply Theorem 5 repeatedly.*

$$\begin{array}{ccccc}
0 & \le & P^{(L-1)} & < & rN \\
0 & \le & P^{(L-2)} & < & (1+\lambda)rN \\
\cdots & & \cdots & & \cdots \\
0 & \le & P^{(0)} & < & \left(\sum_{i=0}^{L-1}\lambda^i\right)rN \\
0 & \le & P & < & \left(\sum_{i=0}^{L}\lambda^i\right)rN
\end{array}$$

Similarly, using the bounds on $P^{(j)}$ just established and Theorem 5,

$$\begin{array}{ccccc}
0 & \le & q_{L-1} & < & 1+r \\
0 & \le & q_{L-2} & < & 1+(1+\lambda)r \\
\cdots & & \cdots & & \cdots \\
0 & \le & q_1 & < & 1+\left(\sum_{i=0}^{L-1}\lambda^i\right)r \\
0 & \le & q_0 & < & \left(\sum_{i=0}^{L}\lambda^i\right)r
\end{array}$$

Summing to infinity gives the bounds as stated in the theorem. In addition, $\lambda < (1/2) - (4r-2)^{-1}$ implies $q_j < 2r$ and $P^{(j)} < 2rN$.

We comment that when N is normalized, $(r^L/N) < 2$, then $e \geq 6$ guarantees $\lambda < 22/64$. And for a normalized N, $\ell_{\rm int} = 32$, $\ell_{\rm fp} = 52$, $e = 16$, the resulting λ is less than 2^{-11}. If N is too unnormalized, a convenient way to handle the modular exponentiation is to compute $(2^k M^E \bmod 2^k N)/2^k$ where $2^k N$ is N normalized. This computation will first compute $R = M^E \bmod 2^k N$, followed by $2^k R \bmod 2^k N$. The result is than right shifted k bits.

5. Implementation

The key inner loop is the computation of

$$P^{(j-1)} \leftarrow r(P^{(j)} - q_j N) + a_{j-1} B.$$

There are a number of considerations relevant to implementation.

In practice, the variables $P^{(j+1)}$ and $P^{(j)}$ share the same storage. The theorems show that $P^{(j)} < 2r^{L+1}$. Hence it suffices to compute $P^{(j)} \bmod 2r^{L+1}$ for all j and $P \bmod 2r^L$. That is, the key computation is

$$r(P^{(j)} - q_j N) + a_{j-1} B \bmod 2r^{L+1}.$$

Second, it is more convenient to compute addition rather than subtraction of unsigned multi-word integers. We therefore use the complement of N

$$\tilde{N} \leftarrow r^L - N$$

thus

$$r(P^{(j)} - q_j N) + a_{j-1} B \bmod 2r^{L+1} = r(P^{(j)} + q_j \tilde{N}) + a_{j-1} B - q_j r^{L+1} \bmod 2r^{L+1}$$

The fact that q_j can be as large as 1 bit more than a word means that in general $q_j = \sigma_j r + \hat{q}_j$ where $\hat{q}_j = q_j \bmod r$ and σ_j is either 0 or 1. Thus the inner loop in general computes

$$r(P^{(j)} + \hat{q}_j \tilde{N} + \sigma_j r \tilde{N}) + a_{j-1} B - q_j r^{L+1} \bmod 2r^{L+1}$$

Note there are two one-word-by-multi word product. The term $\sigma_j r \tilde{N}$ does not involve multiplication as σ_j is just one bit. Moreover, because of the high probability that $\sigma_j = 0$, our code has a branch without this term altogether. The computation of $-q_j r^{L+1} \bmod 2r^{L+1}$ involves a simple "XOR"ing of the lsb of q_j.

Another important observation is that not only the fix up step is seldom needed, but that one can often recognize this without comparing P to N. Here is why. The fix up is not needed if in fact $q_0 = \lfloor P^{(0)}/N \rfloor$, which is equivalent to $|(P^{(0)}/N) - q_0| < 1$. Now

$$\left| \frac{P^{(0)}}{N} - q_0 \right| \leq \left| \frac{P^{(0)}}{N} - \texttt{FP}(uW_0) \right| + |\texttt{FP}(uW_0) - q_0|$$

Hence as long as

$$|\texttt{FP}(uW_0) - q_0| < 1 - |(P^{(0)}/N) - \texttt{FP}(uW_0)|,$$

no fix up is necessary. Using Theorems 1 and 2, we can derive a constant $\Delta \geq |(P^{(0)}/N) - \texttt{FP}(uW_0)|$. In fact λ works (but we can have a tighter number still). Thus if $|\texttt{FP}(uW_0) - q_0| < 1 - \Delta$, we by pass the fix up. Only when this test fails that we compare P against N to determine if the subtraction $P - N$ is needed. This saving is worthwhile as comparing P against N involves a multiword addition.

Indeed, in our tests of roughly 1200 cases, only 4 out of the 1200×64 q_j needs one more bit. Moreover, not only none of the cases require a fix up step, but that none require a comparison with N as the fast test allowed us to bypass them all.

We implemented in a similar manner Montgomery multiplication as outline in Table 2. Both implementations are in generic C and does not make use of vectorization such as SSE2. The inner loops for both are similar in structure and their fragments are shown here.

```
//inner loops
//--------------------------------------//
//Modular Multiplication
for ( k = L; k > 1; k-- )
{
  a_x_B = (dword)AA[j-1] * (dword)BB[k-1];
  q_x_Ncomp =(dword)q * (dword)Ncomp[k-2];
  PP64[k] = Xtmp_64 + PP64[k-1] +
            HIGH_WORD( a_x_B )  +
            HIGH_WORD( q_x_Ncomp );
  Xtmp_64 = LOW_WORD( a_x_B ) +
            LOW_WORD( q_x_Ncomp );
}
//--------------------------------------//
//Montgomery Multiplication
for ( k = 1; k < L; k++ )
{
  a_x_b = (dword)AA[j] * (dword)BB[k];
  q_x_n = q * (dword)NN[k];
  PP64[k-1]  = PP64[k] + TMP +
               LOW_WORD(a_x_b) +
               LOW_WORD(q_x_n);
  TMP = HIGH_WORD(a_x_b) +
        HIGH_WORD(q_x_n);
}
```

We ran 1200 test cases on a 1.86 GHz Intel Pentium M machine under Windows XP using a gcc 3.4.4 compiler. The modular multiplication took on the average 158

thousand cycles and the Montgomery multiplication took 161 thousand cycles. The quotient digit generations constitute 7% and 3% of the time, respectively. While further optimization can be done for specific situations that may favor one or the other algorithms, it is fair to conclude that the two are comparable. Hence the direct method has an advantage for small exponent exponentiation. In the case of the exponent 17, the ratio of the latency of Montgomery method to that of the direct method is $(5T_{\mathrm{Mont}} + 2T_{\mathrm{Modmul}})/(5T_{\mathrm{Modmul}})$ which is roughly 1.4 using the timing just obtained.

We also point out that out of the 1200 tests in the Montgomery multiplication, about 200 cases require fix ups. This is consistent with common knowledge and that there are side-channel attacks that are based on the need for fix up steps (see [2, 6, 7] for example). The direct method presented here might offer some advantages in this aspect.

References

[1] P. Barrett, Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, *Advances in Cryptology, Proc. Crypto'86*, LNCS, volume 263, A.M. Odlyzko, editor, Springer-Verlag, 1987, pp. 311–323.

[2] J. F. Dhem, F. Koeune, P. A. Leroux, P. Mestre, J. J. Quisquater and J. L. Willems, A practical implementation of the timing attack, *Proc. CARDIS 1998*, LNCS, volume 1820, Springer-Verlag, 2000, pp. 175–190.

[3] A. J. Menezes, P. C. van Oorschut and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996

[4] Ç.K. Koç, T. Acar, and B. S. Kaliski, Jr., Analyzing and comparing Montgomery multiplication algorithms, *IEEE Micro*, Volume 16, Issue 3, June 1996, pp. 26–33.

[5] P. L. Montgomery, Modular multiplication without trial division, *Mathematics of Computation*, volume 44, 1985, pp. 519–521.

[6] W. Schindler, A timing attack against RSA with Chinese Remainder Theorem, *Cryptographic Hardware and Embedded Systems (CHES) 2000*, Christof Paar and Çetin Koç, editors, LNCS, volume 1965, Springer-Verlag

[7] C. D. Walter and S. Thompson, Distinguishing exponent digits by observing modular subtractions, *Progress in Cryptology, CT-RSA, 2001*, LNCS, volume 2020, Springer-Verlag, 2001.