AN EFFICIENT IMPLEMENTATION OF MULTI-PRIME RSA ON DSP PROCESSOR

Anand Krishnamurthy, Yiyan Tang, Cathy Xu and Yuke Wang

Department of Computer Science
University of Texas at Dallas, Richardson, TX, 75083 USA
{axk017510, yiyan, cathy, yuke} @ utdallas.edu

ABSTRACT

RSA is a popular cryptography algorithm widely used in signing and encrypting operations for security systems. Generally, the software implementations of RSA algorithm are based on 2-prime RSA. Recently multi-prime RSA has been proposed to speed up RSA implementations. Both 2-prime and multi-prime implementations require squaring reduction and multiplication reduction of multi-precision integers. Montgomery reduction algorithm is the most efficient way to do squaring and multiplication reductions. In this paper, we present a new method to implement the Montgomery squaring reduction, which speeds up squaring reduction by 10-15% for various key sizes. Furthermore, a multi-prime 1024-bit RSA signing operation is implemented on TI TMS320C6201 DSP processor with the new reduction method. As the result, signing operation can be finished within 6ms, which is about twice faster than the RSA implementation in [11] on the same DSP platform.

1 INTRODUCTION

RSA algorithm invented by Rivest, Shamir and Adleman in 1978 is the most popularly used security algorithm in public key cryptosystems [1]. It's widely used to secure network traffic, e-mail, e-commerce and e-business systems for applications in digital signatures and encryptions [2][3]. Since RSA is based on arithmetic modulo of large numbers, which requires large number of computations, fast implementation of RSA becomes vitally important for the performance of cryptosystems. Under this consideration, special-purpose hardware has been designed for RSA [4]-[7], in which high speed can be achieved but suffers from inflexibility. On the other hand, software solutions are inherently flexible for all kinds of emerging cryptosystems but comparatively slow. Hence it is necessary to develop efficient methods to implement RSA over software platforms.

The software implementations of RSA are generally based on 2-prime RSA. Recently multi-prime RSA has been proposed to speed up RSA implementations [8]. Both 2-prime and multi-prime implementations require two major operations: squaring reduction and multiplication reduction. Montgomery reduction algorithm is the most efficient way to perform square and multiplication reductions, on which most previous RSA implementations [4]-[7][9]-[13] are based.

In this paper we introduce a new method to implement the Montgomery squaring reduction. The new method restructures the loop bodies in the squaring reduction to make more room for software pipelining, hence speeds up the squaring reduction on multi-functional-unit DSP architectures by 10-15% for various key size comparing to [11]. Furthermore, an implementation of 1024-bit multi-prime RSA signing operation with the new reduction method is done on TI TMS320C6201 DSP processor for experimental purpose. The experimental result shows that the signing operation can be finished within 6ms, which is almost twice faster comparing to the implementation in [11] on the same DSP processor.

The rest of paper is organized as follows: Section 2 introduces the background of the basic RSA algorithm. The details of the new method to implement the Montgomery squaring reduction are described in Section 3. Section 4 presents the experimental results. Finally the conclusions are drawn in Section 5.

2 BASIC RSA ALGORITHM

The RSA algorithm defines a mechanism to secure the message exchanges in communication systems by providing two types of services: authentication and data integrity [9]. Authentication consists of signing and verifying operations to assure the identities of the message sender. In the signing operation the Sender takes the message *M*, his private signing key *D* and *N* from the public key *(E, N)* to compute the signature *S* by:

$$S = M^D \bmod N \qquad (1)$$

The signing key *D* is much larger than the verifying key *E*. Thus the performance of RSA relies on fast implementation of the signing operation $S = M^D \bmod N$.

2.1 Applying CRT to 2-Prime and Multi-Prime RSA

Based on the property of the RSA algorithm, the modulus *N* is the product of large prime numbers. Thus we can use Chinese Remainder Theorem (CRT) to accelerate the computation.

In 2-prime CRT and 2-prime RSA, the modulus $N = p \cdot q$, where *p, q* are large prime numbers. The CRT suggests that the computation can be separated into

$$S_1 = M^D \bmod p \quad \text{and} \quad S_2 = M^D \bmod q$$

By applying Fermat's theorem, we can obtain

$$S_1 = M^{D1} \bmod p \quad \text{and} \quad S_2 = M^{D2} \bmod q$$

where $D1 = D \bmod (p-1)$ and $D2 = D \bmod (q-1)$. Applying the CRT, we can compute the result *S* in (1) as:

$$S = (S_1 c_1 q + S_2 c_2 p) \bmod N \qquad (2)$$

where $c_1 = q^{-1} \bmod p$ and $c_2 = p^{-1} \bmod q$. The size of p and q is about half of N. Thus the size of the exponents is reduced to half of the original size in 2-prime RSA.

In multi-prime CRT and RSA, we have $S = M^D \bmod (p \cdot q \cdot r)$. We can obtain $S_1 = M^{D1} \bmod p$, $S_2 = M^{D2} \bmod q$, and $S_3 = M^{D3} \bmod r$, where $D1 = D \bmod (p-1)$, $D2 = D \bmod (q-1)$ and $D3 = D \bmod (r-1)$. We can apply the CRT to retrieve S as

$$S = (S_1 c_1 qr + S_2 c_2 pr + S_3 c_3 pq) \bmod N \qquad (3)$$

where $c_1 = (rq)^{-1} \bmod p$, $c_2 = (pr)^{-1} \bmod q$, $c_3 = (qp)^{-1} \bmod r$. Hence the size of the exponents is further reduced to one third of the original. Based on above analysis, 1024-bit 2-prime and multi-prime RSA can be done with 512-bit and 341-bit exponents and modulus respectively.

2.2 Usage of m-ary Method

By applying the CRT, we can compute $M^{D1} \bmod p$ instead of $M^D \bmod N$, where the size of the exponent $D1$ is only one third of D. In this subsection, we explore the efficient implementation for $M^{D1} \bmod p$ based on the m-ary method, which can be done in the following five steps:

1) Group the k-bit exponent $D1$ into $s = k/\log_2 m$ groups. Each group has $\log_2 m$ bits and use $F_0, F_1, ..., F_{s-1}$ to denote the decimal equivalent values of each group.
2) Pre-compute $M^i \bmod p$ where $i = 2, 3, ..., m-1$.
3) Initialization: $tmp = M^{F_{S-1}} \bmod p$
4) Loop for all the groups:

```
for i = s − 2 to 0 {
    for j = 1 to log2 m {
        tmp = (tmp × tmp) mod p;          (4)
    }
    if F_i ≠ 0 then
        tmp = (tmp × M^{F_i}) mod p;      (5)
}
```

5) Save the result: $S_1 = tmp$;

In our implementation, m is equal to 16 so that $D1$ has 512 bits and s is 128 for 2-prime RSA and $D1$ has 352 bits and s is equal to 88 for multi-prime RSA.

2.3 The Montgomery Algorithm

In the above steps, (4) is mapped to Montgomery squaring reduction and (5) is mapped to Montgomery multiplication reduction. The Montgomery reduction algorithm can be briefly described as follows:

Given $(A \times B) \bmod N$ where A, B, N are all $s \times w$-bit wide integers. Each of them is divided into s words with w bits for each word. By defining $R = 2^{s \times w}$ and $A, B < N < R$ and by assuming $R \times R^{-1} = 1 \bmod N$ and $R \times R^{-1} - N \times N^{-1} = 1$ we have the Montgomery reduction function according to [11] as

Input: A, B, R and N;

Output: $T = (A \times B \times R^{-1}) \bmod N$.

```
MontReduction(A, B, N, R, T) {
    T = A × B;
    m = (T × N^{-1}) mod R;
    T = (T + m × N);
    T = T / R;
    if T ≥ N then return (T − N) else return (T);      (6)
}
```

Notice that the result of the above function is $(A \times B \times R^{-1}) \bmod N$ instead of $(A \times B) \bmod N$. In order to get $(A \times B) \bmod N$ from the Montgomery Reduction function, argument A has to be changed to $(A \times R) \bmod N$, which on the other hand is computed by $\text{MontReduction}(A, R^2 \bmod N, N, R, T)$. The value of $R^2 \bmod N$ can be pre-calculated and $m = (A \times B \times N^{-1}) \bmod R$ can be replaced by $m = (A \times B \times -n_0^{-1}) \bmod 2^w$ in multi-precision implementation, where n_0 is the least significant word of N [13].

Applying the Montgomery reduction method, $tmp = (tmp \times tmp) \bmod p$ (4) can be obtained by $\text{MontReduction}(\text{MontReduction}(tmp, R^2 \bmod p, p, R, tmp), tmp, p, R, tmp)$, where tmp and p are k-bit integers, $R = 2^k$.

3 NEW METHOD TO IMPLEMENT MONTGOMERY SQUARING REDUCTION ALGORITHM

Previous method to implement the signing operation $S = M^D \bmod N$ in [11] is based on the 2-prime CRT. our implementation is based on the multi-prime CRT. Furthermore we introduce a new method to speed up the Montgomery squaring reduction in the m-ary method. Montgomery squaring reductions are $\log_2 m$ times more frequent than the Montgomery multiplication reductions in the m-ary method. Thus it is more important to optimize the Montgomery squaring reduction. The target platform is TI TMS320C6201 DSP, which has eight parallel function units. Software pipelining is the most common and useful optimization technique to achieve better instruction level parallelism and higher performance. The key idea of the new method is to implement the Montgomery squaring reduction algorithm in the way in which fits the software pipelining on DSP processor the best.

In the following, we present different algorithms for implementing $\text{MontReduction}(A, A, N, R, T)$, where A, R and N are inputs and the output is $T = (A \times A \times R^{-1}) \bmod N$ in Fig. 1 to Fig. 5. We do not show the operation (6) since it is a common operation for all the algorithms. We use the following general notations: (X, Y) denotes a number which is the concatenation of a pair of two words X and Y, sw denotes the number of words in the multi-precision integers A and N, i.e., $A : (a_{sw-1} ... a_1 a_0)$, $N : (n_{sw-1} ... n_1 n_0)$, $N^{-1} : (n'_{sw-1} ... n'_1 n'_0)$; w denotes the number of

bits in one word; $R = (2^w)^{sw}$, s_{long} denotes a 40-bit integer due to the fact that long integers on fixed point DSP processors are 40-bit long, and finally the size of T varies from *sw*+1 words to $2sw-1$ words as according to the algorithm being used.

3.1 Previous Methods to Implement Squaring Reduction

We present two previous methods showing in Figure 1 from [11] and Figure 2 from [10]. It's easy to identify the redundancy in Figure 1since the same value of $a_j \times a_i$ and $a_i \times a_j$ has been computed twice. To address the redundancy problem, some efforts such as [10] make use of the repeated values (Fig. 2) to design a specialized Montgomery squaring reduction. However, both Fig. 1 and Fig. 2 have outer loops which cannot be efficiently software pipelined. Figure 2 also introduces an *ADD* function in the second outer loop to do the carry propagation, which contains the costly loop itself.

Fig.1 Previous Method in [11]

```
for i=0 to sw-1 {
   /* Outer Loop */
   (C, S) = a_i x a_0 + t_i;
   m = (n_0 x S) mod 2^w;
   (C_1, S) = m x n_0 + S;
   C_1 = 0;
   for j = 1 to sw-1 {
      /* Inner Loop */
      (C, S) = a_i x a_j + C + t_j;
      (C_1, t_j-1) = m x n_j + S + C_1;
   }
   (t_sw, t_sw-1) = t_sw + C + C_1;
}
```

Fig.2 Previous Method in [10]

```
for i = 0 to sw-1 {
   /* First Outer Loop */
   (C, t_i+i) = a_i x a_i + t_i+i;
   for j = i + 1 to sw-1 {
      /* Inner Loop */
      s_long = 2 x C_1 + C + t_i+j;
      (C_1, S) = a_i x a_j;
      (C, t_i+j) = s_long + 2 x S;
   }
   (t_i+sw+1, t_i+sw) = C + 2 x C_1;
} C = 0;
for i = 0 to sw-1 {
   /* Second Outer Loop */
   m = t_i x n_0' mod 2^w;
   for j = 0 to sw-1 {
      /* Inner Loop */
      (C, t_i+j) = t_i+j + m x n_j + C;
   }
   ADD(t_i+sw, C);
}
```

3.2 New Methods to Implement Squaring Reduction

Fig.3 Improved Squaring Reduction

```
/* Separate Loop for a_i x a_i */
for i=0 to sw-1 { (t_i+i+1, t_i+i) = a_i x a_i; }
for i=0 to sw-1 {
/* First Outer Loop */
   C=0; C_1=0;
   for j=i+1 to sw-1 {
   /* Inner Loop */
   /* Redundancy is removed */
      s_long = 2 x C_1 + C + t_i+j;
      (C_1, S) = a_i x a_j;
      (C, t_i+j) = s_long + 2 x S;
   }
   /* ADD function is replaced below */
   (prevcar, t_i+sw) = C + 2 x C_1
                     + t_i+sw + prevcar;
}
t_sw+sw = t_sw+sw + prevcar;
precvar = 0;
for i=0 to sw-1 {
   /* Second Outer Loop */
   m = t_i x n_0' mod 2^w;
   for j=0 to sw-1 {
      /* Inner Loop */
      (C, S) = t_i+j + m x n_j + C;
      t_i+j = S;
   }
   /* ADD function is replaced below */
   (prevcar, t_i+sw) = C + t_i+sw + prevcar;
}
t_sw+sw = t_sw+sw + prevcar;
```

Fig.4 SR with Merged Outer Loop

```
/* Separate Loop for a_i x a_i */
for i=0 to sw-1 { (t_i+i+1, t_i+i) = a_i x a_i; }
for i=0 to sw-1 {
/* Outer Loop */
   C=0; C_1=0;
   for j=i+1 to sw-1 {
   /* First Inner Loop */
   /* Redundancy is removed */
      s_long = 2 x C_1 + C + t_i+j;
      (C_1, S) = a_i x a_j;
      (C, t_i+j) = s_long + 2 x S;
   }
   /* ADD function is replaced below */
   (prevcar, t_i+sw) = C + 2 x C_1
                     + t_i+sw + prevcar;
   m = t_i x n_0' mod 2^w;
   C = 0;
   for j=0 to sw-1 {
      /* Second Inner Loop */
      (C, S) = t_i+j + m x n_j + C;
   }
   /* ADD function is replaced below */
   (prevcar1, t_i+sw) = C + t_i+sw + prevcar1;
}
t_sw+sw = t_sw+sw + prevcar + prevcar1;
```

The motivation of the proposed method to implement the Montgomery squaring reduction is to completely solve the problem of the redundancy and maximize the power of software pipelining. The key idea is instead of computing $a_j \times a_i$ for *i, j=1,…,sw,* as Figure 1 does, we compute only $a_j \times a_i$ for *i=1,…,sw,* and *j>i.* We achieve this goal by three steps:

1) Use a separate loop to compute $a_i \times a_i$
2) Remove the ADD function
3) Restructure the loop to remove the redundancy

Multiplication on TI TMS320C6201 DSP is a multi-cycle operation, which introduces delay slots to the program. To maximize the power of software pipelining, we need to fill up these delay slots as much as possible. Hence we move the computation $a_i \times a_i$ out of the first outer loop in Fig. 2. Moreover, to deal with the carry propagation loop of function *ADD*, we use a temporary variable *prevcar* to store the carry produced in the outer loops. Thus the carry propagation loop is consumed in the outer loops. By applying these two improvements, we can obtain an implementation in Fig. 3.

To optimize the implementation further, we need to restructure the loop structures in Fig.3. Since the software pipeline can be only performed on the inner loop, we can merge the outer loops together for better the software pipelining. It's easy to see that the computation of (C, t_{i+j}) in the inner loops shown in the Fig. 4 are overlapping with each other. We can maximize the power of software pipelining by remove the overlapping. Hence results the implementation with coupled loops in Fig. 5.

Fig. 5 Squaring Reduction with Coupled Inner Loop

```
/* Separate Loop for a_i x a_i */
for i=0 to sw-1 { (t_i+i+1, t_i+i) = a_i x a_i; }
for i=0 to sw-1 {
/* Outer Loop */
   m = t_i x n_0' mod 2^w;
   for j=0 to i {
   /* First Inner Loop */
      (C, t_i+j) = t_i+j + m x n_j + C;
   }
   C_1 = 0; C_2 = 0
   for j=i+1 to sw-1 {
      /* Second Inner Loop */
```

Continue on the left column…...

```
      /* Redundancy is removed */
         s_long = 2 x C_1 + C + t_i+j;
         (C_1, S) = a_i x a_j;
         (C, t_i+j) = s_long + 2 x S ;
         (C_2, t_i+j) = m x n_j + S + C_2;
      }
      /* ADD function is replaced below */
      (prevcar, t_i+sw) = C + 2 x C_1+ C_2
                        + t_i+sw + prevcar;
   }
   t_sw+sw = t_sw+sw + prevcar;
```

4 RESULTS

We have implemented the Montgomery squaring reduction with different methods mentioned in the last section with 1024-bit, 512-bit, and 352-bit operands respectively. The experimental results are listed in Table 1, in which we can observe about 13%, 11.5%, 11.1% maximum improvements in 352-bit, 512-bit and 1024-bit squaring reduction with the new methods respectively comparing to the method used in [11].

Methods	1024-bit	512-bit	352-bit
Prev. method in [11]	6262	1838	1001
Prev. method in [10]	6974	2090	1146
Improved (SR)	6220	1828	980
SR with merged outer loop	6026	1733	942
SR with coupled inner loops	5561	1625	867

Table 1 Execution speed for various squaring reductions on the DSP

The overall performance of the RSA with different methods to implement the squaring reduction is reported in Table 2, from

which we can see that the combination of the new methods proposed in this paper and the multi-prime RSA results in significant improvement over the same application in [11] on the same DSP platform in terms of execution time by almost twice faster.

Methods	Non-CRT	2-prime	Multi-prime
Prev. method in [11]	40.07 ms	11.76 ms	n/a
SR with coupled inner loops	36.5 ms	10.6 ms	6.06 ms

Table 2 Execution time for different implementations of the signing operation

5 CONCLUSIONS

We have presented a new method to implement the Montgomery squaring reduction on the DSP platform. The new method removes the redundancies existing in the previous implementations and specifically optimized for the DSP architecture by restructuring the loop body. A multi-prime 1024-bit RSA algorithm is implemented on a TI TMS320C6201 DSP processor with the new method. The experimental results show that the squaring reduction is improved by 10-15% for various key sizes and by 10% for 2-prime RSA comparing to the results reported in [11]. Also by combining the new method with 1024-bit multi-prime RSA, a signing operation can be finished within 6 ms, which is about twice faster than the RSA implementation in [11] on the DSP platform.

REFERENCES

[1] R. L. Rivest, A. Shamir and L. Adleman, "A Method of obtaining Digital Signatures and Public Key Cryptosystems," *Comm. of ACM,* vol. 21, no.2, pp. 120-126, Feb 1978.

[2] T. Unkasevie, M. Markovic and G. Dordevic, "Optimization of RSA Algorithm Implementation on TI TMS320C54x Signal Processors," *Proc. of TELSIKS 2001,* pp. 603-606, Sept. 2001.

[3] D. Boneh and H. Shacham, "Fast variants of RSA," *CryptoBytes*, vol.5, no.1, pp. 1-9, 2002.

[4] S. E. Eldridge and C. D. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Trans. Comput.,* vol. 42, pp. 693-699, June 1993.

[5] C. C. Yang, T. S. Chang, and C. W. Jen, "A new RSA cryptosystem hardware design based on Montgomery's algorithm," *IEEE Trans. Circuit and Systems II: Analog and Digital Signal Processing,* vol. 45, pp. 908-913, July 1998.

[6] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," *Proc. 14th IEEE Symp. On Computer Arithmetic,* pp. 70-77, 1999.

[7] T-W. Kwon, C-S. You, W-S. Heo, Y-K. Kang and J-R. Choi, "Two Implementation Methods of a 1024-bit RSA Cryptoprocessor Based on Modified Montgomery Algorithm," *Proc. of ISCAS 2001,* vol. 4, p.650-653, 2001.

[8] "PKCS #1 v2.0 Amendment 1: Multi-Prime RSA," *RSA Laboratories*, July 20, 2000.

[9] C. K. Koc, "High-Speed RSA Implementation," *RSA Publications*, ver. 2 1994.

[10] C. K. Koc, T. Acar, B.S. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro,* vol. 16, No. 3, pp. 26-33, June 1996.

[11] K. Itoh, M. Takenaka, N. Torii, S. Temma, Y. Kurihara, "Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201," *Proc. of CHES' 99,* pp. 61-71, 1999.

[12] P. Barrett, "Implementing the Rivest, Shamir, and Adleman Public-Key Encryption Algorithm on a Standard Digital Signal Processor," *Advances in Cryptology CRYPTO'86,* vol. 263, pp. 311-323, 1987.

[13] S. R. Dusse, B. S. Kaliski, "A Cryptographic Library for the Motorola DSP56000," *Advances in Cryptology-Eurocrypt'90,* pp. 230-244, 1990.

[14] A. Selby and C. Mitchell, "Algorithms for software implementations of RSA," *IEEE PROCEEDINGS,* vol. 136, No. 3, pp.166-170, May 1989.

[15] D. E. Knuth, "The Art of Computer Programming: Seminumberical Algorithms," *vol. 2. Reading,* Sept. 1993.

[16] D. M. Gordon, "A Survey of Fast Exponentiation Methods," *Journal of Algorithms 27,* pp. 129-146, 1998.

[17] J. Bos and M. Coster, "Addition Chain Heuristics," *Proc. of CRYPTO 89,* No. 435, pp. 368-370, 1989.

[18] P.L. Montgomery, "Modular Multiplication without Trial division," *Mathematics of Computation,* vol. 44, pp. 519-524, 1985.