Fast Modular Squaring Method for Public Key Cryptosystems

Chia-Long Wu
Department of Aviation & Communication Electronics, Chinese Air Force Institute of Technology, Kaohsiung 82042, Taiwan, R. O. C.
E-mail: chialongwu@seed.net.tw

Der-Chyuan Lou*, and Te-Jen Chang*
***Department of Electrical Engineering, Chung Cheng Institute of Technology, National Defense University, Tahsi, Taoyuan 33509, Taiwan, R. O. C.**
E-mail: dclou@ccit.edu.tw

Abstract

The squaring algorithm acts an important role in large integer arithmetic. The standard squaring algorithm is quite well-known, but there is an improper carry handling bug in it. The Guajardo-Paar's squaring algorithm fixes the carry handling bug, but generates error-indexing bug. In this paper, we propose a novel efficient squaring algorithm that not only avoids the bugs between the standard squaring algorithm and the Guajardo-Paar's squaring algorithm but also improves the performance in squaring computation for Yang-Hseih-Laih squaring algorithm. For base b, the products of $x_i * x_j$ can be pre-computed on-line, that is, 1*2, 1*3, …, (b-1)(b-1) are pre-computed. Some results will be determined and stored in a look-up table before the computation and we can speed up the performance of squaring algorithm. Our proposed algorithm is about 1.77 times faster in comparison with the Yang-Hseih-Laih's algorithm, and also faster than the standard squaring algorithm.

Keywords: Modular squaring, public key cryptosystem, algorithm, Look Up Table, shift.

摘要

平方演算法在大整數的運算，扮演很重要的角色。標準的平方演算法眾所週知，但有"錯誤進位"的疑慮發生。Guajardo 與 Paar 學者提出的平方演算法修正了這項缺點，但是又延生出"錯誤索引"的問題。在本篇論文中，我們提出一個有效的平方方法，不僅可以解決以上所述兩項問題，亦可改進 Yang、Hseih 與 Laih 三位學者所提出的演算法。對於基底 b 而言，$x_i * x_j$ 的乘積可以事先計算並儲存之，即 1*2, 1*3, …, (b-1)(b-1)可以在實際運算前，事先儲存之，進而加速平方演算法的執行效率。本文所提出的演算法與 Yang、Hseih 與 Laih 三位學者所提出的演算法相較之下，快了 1.77 倍，當然這個演算法比標準的平方法亦快的多。

關鍵詞：模平方運算、公開金鑰密碼系統、演算法、查表法、移位。

1. Introduction

Many applications of large integer arithmetic, such as public key cryptography [3, 8], integer squaring and multiplying are significant operations. Multiplication and squaring operations can be used to construct modular multiplication, modular exponentiation, inverse generation, and elliptic curve addition, etc. Some of the methods [2, 3] of squaring are discussed in this paper. The performance of large integer multiplication and squaring operations mainly depends on the number of single-precision multiplications (SPMs). Squaring is a special case of multiplying, yet squaring operations are much faster than multiplications [3].

Many exponentiation algorithms [4, 5] can reduce the number of multiplications, but they cannot reduce

the number of squaring operations. This shows that efficient squaring algorithms are very important in many public key cryptosystems. In this paper, we present some squaring algorithms, such as the standard squaring algorithm, the Guajardo-Paar squaring algorithm, and the Yang-Hseih-Laih algorithm [2, 4, 5]. In this paper, we propose a novel squaring algorithm which both improves the performance of these squaring algorithms and also avoids both carry handling bug and the error-indexing bug. The performance of the proposed squaring algorithm is nearly 1.65 times faster in comparison with squaring computation which uses the standard squaring algorithm and 1.77 times faster compared to the Yang-Hseih-Laih squaring algorithm.

The rest of this paper is organized as follows. In Section 2, we first review the squaring algorithm computation by using the standard squaring algorithm. In addition, we also introduce the Guajardo-Paar squaring algorithm and the Yang-Heih-Laih squaring algorithm. In Section 3, we present a novel squaring algorithm which can avoid the bugs mentioned in [2, 4]. The details of our analyses are described in Section 4. Finally, we conclude this paper in Section5.

2. Review of Previous Works

Among the standard squaring algorithm, the Guajardo-Paar algorithm, and the Yang-Hseih-Laih algorithm are important methods for speeding up the computation of squaring. These squaring algorithms will be introduced in the following.

2.1 The Standard Squaring Algorithm

Squaring is an easier operation than multiplication since half of the single-precision multiplications can be skipped. This is due to the fact that $s_{ij} = x_i * x_j = s_{ji}$. The standard squaring procedures are described in Algorithm 1 [2]. Thus, we can modify the standard multiplication procedure to take advantage of this property of the squaring operation.

However, the carry-sum pair produced by operation $(u, v) = s_{i+j} + 2 * x_j * x_i + v$ may be one bit longer than a single-precision number which requires *w* bits. Since

$$(2^w - 1) + 2 * (2^w - 1) * (2^w - 1) + (2^w - 1) = 2^{2w+1} - 2^{w+1}$$

and

$$2^{2w} - 1 < 2^{2w+1} - 2^{w+1} < 2^{2w+1} - 1,$$

the carry-sum pair requires 2*w*+1 bits instead of 2*w* bits for its representation. Thus, we need to accommodate this 'extra' bit during the operation execution in Algorithm 1. The resolution of this carry may depend on the way the carry bits are handled by the particular processor's architecture.

2.2 The Guajardo-Paar's Squaring Algorithm

The square of a multi-precision number uses an intermediate value of the form $(uv)_b$ where *b* is the base in which *uv* is represented and *u* is a double-precision number, i.e., *u* itself is the form of $(rs)_b$ where both *r* and *s* are single- precision quantities. This algorithm fixes the error carry handling bug in the standard squaring algorithm [4].

Algorithm 1 (The standard squaring algorithm)

Input: *x*

Output: $S = x * x$

begin

$S_i = 0$ ***for*** *i* = 1 ***to*** 2*n*

for *i* = 1 ***to*** *n*

$(u, v) = s_{i+i} + x_i * x_i$

for *j* = *i*+1 ***to*** *n*

$(u, v) = s_{i+j} + 2 * x_j * x_i + u$

$s_{i+j} = v$

$s_{i+n} = u$

return $S = (s_{2n}s_{2n-1}...s_1)_b$

end.

Let $X = (x_n x_{n-1}...x_2 x_1)_b$ be a multi-precision integer in base-*b* representation. Then $S = X^2 = (x_{2n}x_{2n-1}...x_2x_1)_b$ can be computed by using the following algorithm [4].

Algorithm 2 (The Guajardo-Parr's squaring algorithm)

Input: Integer $X = (x_n x_{n-1}...x_2 x_1)_b$

Output: Integer $S = X^2 = (s_{2n}s_{2n-1}...s_2s_1)_b$

begin

$s_i = 0$ ***for*** $i = 1$ ***to*** $2n$

for $i = 1$ ***to*** n

$(u, v)_b = s_{2i-1} + x_i^2$

$s_{2i-1} = v$, $d = u$, $e = 0$

for j=i+1 ***to*** n

$(p, q) = x_i * x_j$

$(u,v)_b = s_{i+j-1} + (p,q) + d$, $s_{i+j-1} = v$, $d = u$

$(u, v)_b = s_{i+j-1} + (p, q) + e$, $s_{i+j-1} = v$, $e = u$

$(u, v)_b = d + e$, $d = v$, $e = u$

$(u, v)_b = s_{i+n-1} + d$, $s_{i+n-1} = v$

$s_{i+n} = e + u$

return $S = (s_{2n}s_{2n-1}...s_2s_1)_b$.

end.

***Example*:**

Here let us assume $n = 3$ and $b = 10$.

The procedures performed by Algorithm 2 in squaring x=876 are shown in Table 1.

The bug of Algorithm 1 is caused by the extra carry of $s_{i+j} + 2x_j * x_i + u$ [2]. Algorithm 2 fixes the carry handling bug in the Algorithm 1 by using many extra SPAs (single-precision additions) to record the carry state. The strategy of Algorithm 2 is to employ an extra addition to repair this bug and it needs many extra additions and four extra registers (*p*, *q*, *d*, *e*).

Table 1. The procedures performed by Algorithm 2 in squaring where *x*=876.

i	*j*	s_{2i} $+x_i^2$	s_{i+j} $+x_ix_j$ $+a$	*u*	*v*	s_1	s_2	s_3	s_4	s_5	s_6
1	-	0+36	-	3	6	6	0	0	0	0	0
	2	-	2+6*7 +3	4	7	6	7	0	0	0	0
	3	-	2+6*8 +4	5	4	6	7	4	0	0	0
				0	4	6	7	4	1	0	0
2	-	4+49	-	5	3	6	7	3	2	0	0
	2	-	2+7*8 +0	5	8	6	7	3	8	0	0
				0	9	6	7	3	9	0	0
3	-	1+64	-	6	5	6	7	3	7	5	6
				1	1	6	7	3	7	6	7

2.3 The Yang-Hseih-Laih Algorithm

The Yang-Hseih-Laih algorithm avoids both the improper carry handling bug [2] in the Algorithm 1 and the error-indexing bug [4] in the Algorithm 2. The Yang-Hseih-Laih algorithm is described as Algorithm 3 [5].

Yang, Hseih, and Laih separate the SPMs of Algorithm 1 into two parts. The first part is $x_i * x_j$ and the second part is $x_j * x_i$. Compute these two parts separately then combine these two parts. Because the second part should be doubled, if it is firstly computed then doubled together, the improper carry bug disappears. Therefore Yang, Hseih, and Laih firstly compute the multiplication $x_j * x_i$ ($i \neq j$), then double this part instep by shifting *w*. Finally Yang, Hseih, and Laih compute the first part, $x_i * x_j$.

All the SPSs (single-precision shifts) of Algorithm 1 are handled in Algorithm 3. Therefore, there is no carry propagation in Algorithm 3. This

strategy eliminates all the extra carry propagation of $2^{*x_j*x_i}$ without any extra register operation. Yang, Hseih, and Laih proposed this algorithm to fix these bugs as shown in Algorithm 2 and Algorithm 3. It needs $\frac{n^2+n}{2}$ SPMs, $2n^2+2n$ SPAs, and $2n^2+3n+4$ assignments in Algorithm 3.

Algorithm 3 (The Yang-Hseih-Laih squaring algorithm)

Input: Integer $X=(x_n x_{n-1}...x_2 x_1)_b$

Output: Integer $S=X^2=(s_{2n}s_{2n-1}...s_2 s_1)_b$

begin

$s_i = 0$ ***for*** i=1 ***to*** $2n$

for i=1 ***to*** n

$u = 0$

for $j = i+1$ ***to*** n

$(u,v) = s_{i+j-1} + x_j * x_i + u$

$s_{i+j} = v$

$s_{i+n} = u$

$s = 2s$ (Shift s left 1 bit)

$u = 0$

for $i = 1$ ***to*** n

$(u,v) = s_{2i} + x_i * x_i + u$, $s_{2i} = v$

$(u,v) = s_{2i+1} + u$, $s_{2i+1} = v$

return $S=(s_{2n}s_{2n-1}...s_2 s_1)_b$.

end.

3. The Proposed Squaring Algorithm

In this section, we propose a novel algorithm to avoid both the improper carry handling bug in Algorithm 1 [2] and error-indexing bug in Algorithm 2 [4]. The proposed squaring algorithm not only avoids the improper carry handling bug and error-indexing bug but also improves the performance of all the squaring algorithms in Section 2. The proposed squaring algorithm is described as Algorithm 4.

The proposed squaring algorithm fixes the carry handling bug in Algorithm 1 and the error-indexing bug in Algorithm 2. At first, we pre-compute the products of 1*1, 1*2, …, 1*(*b*-1), 2*1,…, …, (*b*-1)(*b*-1), and store the corresponding products 1, 2, 3, …, $(b-1)^2$ in the initial LUT. The numbers of the products are at most $(b-1)^2$. If *b* is smaller than *n* far, the space used is very small and the time can be omitted. There are *n* assignments in Algorithm 4. In Algorithm 4, there are $2n^2$ SPAs and 2*n* assignments. The other analyses in detail are shown in Table 3. The execution processes are shown in Table 2.

Algorithm 4 (Proposed squaring algorithm):

Input: Integer $X=(x_n x_{n-1} x_{n-2}...x_2 x_1)_b$

Output: Integer $S=X^2=(S_{2n}S_{2n-1}S_{2n-2}...S_2 S_1)_b$

begin

compute the products of 1*2, …, 1*(*b*-1), 2*1,…, (*b*-1)(*b*-1), and store these results along with the corresponding products 1, 2, 3, …, $(b-1)^2$ in the initial LUT (Look-Up Table)

$S_i = 0$ ***for*** i=1 ***to*** $2n$

for i=1 ***to*** n

for j=1 ***to*** n

$x_i * x_j$ from LUT (Look-Up Table)

$(uv)_b = x_i * x_j$

$S_{i+j} = S_{i+j} + u$

$S_{i+j-1} = S_{i+j-1} + v$

if $S_{i+j-1} \geq b$

then $S_{i+j} = S_{i+j} + 1$

$S_{i+j-1} = S_{i+j-1} - b$

else $S_{i+j-1} = S_{i+j-1}$

if $S_{i+j} \geq b$

then $S_{i+j+1} = S_{i+j+1} + 1$

$S_{i+j} = S_{i+j} - b$

else $S_{i+j} = S_{i+j}$

return $S=(S_{2n}S_{2n-1}S_{2n-2}...S_2 S_1)_b$

end.

Table 2. The execution processes for proposed algorithm.

i	j	u	v	s_1	s_2	s_3	s_4	s_5	s_6
1	1	8	1	0 →1	0 →8				
	2	7	2		8 →0	0 →7 →11 →8			
	3	6	3			8 →1	6 →7		
2	1	7	2		0 →2	1 →8			
	2	6	4			12 →2	13 →1 4 →4	0 →1	
	3	5	6				10 →0	6 →7	
3	1	6	3			5	6		
	2	5	6				12 →2	12 →1 3 →3	1
	3	4	9					12 →2	5 →6

Table 3. The complexity comparisons of squaring algorithms.

Operation	Algorithms				Execution Time Weight
	1	2	3	4(Proposed)	
SPM	$\frac{n^2+n}{2}$	$\frac{n^2+n}{2}$	$\frac{n^2+n}{2}$	×	11
SPA	$2n^2$	$3n^2+4n$	$2n^2+2n$	$3n^2$	2
SPS	$\frac{n^2-n}{2}$	0	n	0	3
Assignment	$\frac{n^2+7n}{2}$	$7n^2+6n$	$2n^2+3n+4$	$\frac{n^2}{2}+3n$	1
Total estimated	$\frac{21n^2+15n}{2}$	$18.5n^2+19.5n$	$11.5n^2+15.5n+4$	$6.5n^2+3$	×
Extra Registers	2	6	2	2	×

4. Complexity Analyses

Table 3 shows the number of single precision operations of Algorithm 1, 2, 3, and 4. Because the SPM is the most time-consuming operation in these algorithms, we can replace SPM with LUT (Look-Up Table) to reduce the time complexity. So we only consider the effect of SPAs, SPSs, and assignments for performance estimation. Let us suppose that the execution-time weight of an assignment is 1, and then the execution-time weight can be estimated as 2, 3, and 11 for SPA, SPS, and SPM respectively. Here SPM is used only for Algorithm 1, 2, and 3. According to these estimations, we can estimate the overall weight of these algorithms in Table 3. If we assume the performance factor of Algorithm 1 is 1.00, then the performance factors of the Algorithm 1, 2, 3, and 4 will be those shown in Table 4. The results for Algorithm 1, 2, 3, and 4 are shown in Table 5.

Table 4. The performance comparisons of squaring algorithms.

Digits	Algorithms			
	1	2	3	4(Proposed)
1	1	0.474	0.581	1.89
2	1	0.504	0.704	1.78
4	1	0.529	0.792	1.678
8	1	0.546	0.847	1.664
16	1	0.556	0.879	1.64
32	1	0.562	0.895	1.628
64	1	0.565	0.904	1.622
128	1	0.566	0.909	1.619
∞	1	0.568	0.913	1.615

The proposed squaring algorithm not only avoids the bugs among the standard squaring algorithm, the Guajardo-Paar squaring algorithm, and Yang-Hseih-Laih squaring algorithm but also improves the performance in squaring computation.

So the proposed method is superior to the Algorithm 1, 2, and 3.

Table 5. The comparison results for the analyses of squaring algorithms in Table 3.

Digits	Algorithms			
	1	2	3	4 (Proposed)
1	18	38	31	9.5
2	57	113	81	32
4	198	374	250	118
8	732	1,340	864	440
16	2,808	5,048	3,196	1,712
32	10,992	19,568	12,276	6,752
64	43,488	77,024	48,100	26,816
128	172,992	305,600	190,404	106,880
∞	10.5	18.5	11.5	6.5

Now there are still many more novel methods [1, 2] issued in computer security journals and reports for computer arithmetic operations analyses. In the future, we can incorporate modular arithmetic and some novel methods to reduce the number of multiplications or squarings for modern cryptographic applications.

Table 3 shows the analyses of squaring algorithms. The weight estimation is not very good. The execution time of the single precision operations depends on the operands. For example, the clocks of SPA are 1, 2, or 3 clocks for register-to-register, memory-to-register, and register-to-memory respectively [5].

5. Conclusions

In this paper, we propose a novel squaring algorithm. The performance of our proposed squaring algorithm is 1.65 and 1.77 times faster than squaring computation by using the standard squaring algorithm and the Yang-Hseih-Laih squaring algorithm. Moreover, this algorithm can apply to fast modular arithmetic in public key cryptosystems such as RSA cryptosystem.

References

[1] B. Cao, T. Srikanthan, and C.-H. Chang, "A New Design Method to Modulo 2^n-1 Squaring," Proceedings of IEEE International Symposium on Circuits and Systems 2005, Vol. 1, May 2005, pp. 664-667.

[2] C. K. Koc, Tech. Notes, High-Speed RSA Implementation, RSA Labs. Tech. Note TR 201, Available in http://www.rsasecurity.com /rsalabs/tech-notes, 1994.

[3] D. Zuras, "More on Squaring and Multiplying Large Integers," IEEE Transactions on Computers, Vol. 43, No. 8, pp. 899-908, August 1994.

[4] J. Guajardo and C. Paar, "Modified Squaring Algorithm," Available from http://www.crypto.ruhr-uni-bochum.de/guajardo/cv.html#pubs.

[5] P.-Y. Hseih and C.-S. Laih, An Exception Handling Model and Its Application to the Multiple-Precision Integer Library, Master Thesis, June 2003.

[6] S. Yasuyuki and S. Kouichi, "Simple Power Analysis on Fast Modular Reduction with Generalized Mersenne Prime for Elliptic Curve Cryptosystems," Side Channel Analysis A: on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E89-A, No.1, pp.231-237, Jan. 2006.

[7] W. Choi and T. K. Sarkar, "Minimum Norm Property for the Sum of the Adaptive Weights for a Direct Data Domain Least Squares Algorithm," IEEE Transactions on Antennas and Propagation, Vol. 54, No. 3, pp. 1045-1050, Mar. 2006.

[8] W.-C. Yang, P.-Y. Hseih, and C.-S. Laih, "Efficient Squaring of Large integers," IEICE Transactions on Fundamentals, Vol. E87-A, No. 5, May 2004.