Efficient Modular Exponentiation Methods for RSA

Hatice Kübra Güner, Murat Cenk and Çağdaş Çalık

***Abstract*—RSA is a commonly used asymmetric key cryptosystem that is used in encrypting and signing messages. The efficiency of the implementation is an important factor in effectively using the system. The RSA algorithm heavily depends on the modular exponentiation operation on large integers. A drawback of this system is that it becomes inefficient so quickly when the parameters are adjusted to increase security. This situation causes the operations to be performed with large numbers. Therefore, implementations require the utilization of faster methods than the traditional ones. One popular modular exponentiation method is the *repeated squaring and multiplication algorithm*. In this study, we examine some of the modular exponentiation algorithms and implement them for comparison with the *repeated squaring and multiplication algorithm*. The results suggest that particular cases of studied methods provide at least 23% improvement over the *repeated squaring and multiplication algorithm*.**

***Index Terms*—RSA, modular exponentiation, m-ary method, efficiency, running time.**

I. Introduction

In RSA cryptosystem, encryption and decryption procedures are based on modular exponentiation operation. According to the RSA algorithm [**?**], the encryption process consists of first representing the plaintext as an integer *M* and then the ciphertext *C* is computed by

$$C = M^e \bmod n.$$

For decryption, the process is the same but with a different exponent, that is, the original message is obtained by

$$M = C^d \bmod n.$$

By looking at these procedures, it is obvious that modular exponentiation operation is at the core of the RSA cryptosystem and efficiency of the system on a particular platform heavily depend on how the modular exponentiation operation is implemented on that platform.

Over the years, the recommended key sizes for RSA have changed because of the increased computation power and the improved methods the cryptanalysts have [**?**]. Today, the minimum accepted RSA modulus length is 2048-bits.

The *repeated squaring and multiplication algorithm* was recommended by Rivest, Shamir and Adleman in [**?**]. Running time of the modular exponentiation increases rapidly when the key size is doubled. In TABLE I, running times of the RSA decryption operation for different modulus sizes are given. The decryption exponent *d* is calculated by choosing the encryption exponent *e* as 65537.

According to the table, if the modulus size is increased from 1024-bits to 2048-bits, running time of decryption increases by a factor of more than 7. If the key modulus size is increased to 3072-bits, the running time increases by a factor of almost 22 times compared to 1024-bits. When the modulus size is increased from 1024-bits to 4096-bits, the running time increases by a factor of 47. One of the way to overcome this fast increase is to find faster modular exponentiation methods . With this motivation, we examine some methods and compute required number of operations for each of them separately. Generally, the existing methods provide efficiency with using memory. So, if there was any memory usage in the method, required memory sizes are also tabulated. These methods are implemented using *MPIR* library with *Microsoft Visual Studio* on Intel Core i7 2.00 GHz., and the execution times were calculated for the *repeated squaring and multiplication algorithm*. In some cases, significant efficiency improvements were observed.

TABLE I
EXECUTION TIME OF RSA DECRYPTION OPERATION FOR VARIOUS MODULUS SIZES.

RSA modulus size	running time (ms)
1024-bits	4.52
2048-bits	32.55
3072-bits	99.46
4096-bits	214.96

II. Some Fast Modular Exponentiation Methods

A. The Repeated Squaring and Multiplication Algorithm

The *repeated squaring and multiplication algorithm* was examined by Knuth in [**?**]. This method has an old history [**?**]. There are two variants of the algorithm; the *left-to-right* method and the *right-to-left* method. Here, we only examine the *left-to-right* method in which the procedure starts from the most significant bit of binary expansion of the exponent. Let us represent the binary expansion of private exponent *d* as

$$d = (d_k d_{k-1} \dots d_1 d_0)_2 = \sum_{i=0}^{k} d_i \cdot 2^i \quad \text{,where } d_i \in \{0,1\}$$

In this algorithm, one squaring is performed in each step, and after that if the corresponding bit is 1, a successive multiplication is also carried out. The pseudo-code is presented in Algorithm 1.

The required number of operations [**?**] is

$$(\lceil log(d) \rceil - 1)S + (H(d) - 1)M$$

where *H(d)* is the Hamming weight of *d*, S represents the squaring operation and M represents the multiplication operation. The cost of exponentiation for different scenarios are given as:

- Worst case: $(\lceil log(d) \rceil - 1)$(S + M)

H. K. Güner, M. Cenk and Ç. Çalık are with the Cryptography Program, Institute of Applied Mathematics, METU, Ankara, Turkey. e-mail: kubra.guner@metu.edu.tr, mcenk@metu.edu.tr, ccalik@metu.edu.tr

Algorithm 1 *Left-to-right* method of the *repeated squaring and multiplication algorithm*

Input: $M,\ n,\ (d_k d_{k-1} \dots d_1 d_0)_2$
Output: $C = M^d \bmod n$
1: **if** $d_k = 1$ **then**
2: $C \leftarrow M$
3: **else**
4: $C \leftarrow 1$
5: **end if**
6: **for** i **from** $k-1$ **to** 0 **do**
7: $C \leftarrow C^2 \bmod n$
8: **if** $d_i = 1$ **then**
9: $C \leftarrow C \cdot M \bmod n$
10: **end if**
11: **end for**
12: **return** C

- Average case: $(\lceil log(d) \rceil - 1)$(S + $\frac{1}{2}$M)
- Best case: $(\lceil log(d) \rceil - 1)$S

The number of multiplications and squarings are calculated separately because the squaring operation can be implemented more efficiently than the multiplication operation [**?**].

B. The m-ary Method

The *m-ary* method is explained in [**?**]. Idea of the method is the same with the *left-to-right* method of the *repeated squaring and multiplication algorithm* with addition of a look-up table. A look-up table is prepared prior to the exponentiation to keep powers of *M* such as $M^i \bmod n$ where $i \in \{2, 3, \cdots, m-1\}$. In this method, m-th power of the partial value is computed and if the related digit value of the expansion is different from 0, a subsequent multiplication is also applied in every step. Here, multiplier is obtained from the look-up table by looking at value of the corresponding digit. Base *m* representation of private key d is

$$d = (d_l d_{l-1} \dots d_1 d_0)_m = \sum_{i=0}^{l} d_i \cdot m^i,$$

where $d_i \in \{0, 1, \dots, m-1\}$. The pseudo-code is given as Algorithm 2.

The method can be used for any *m* values. But, the more efficient results are obtained when *m* is chosen as power of 2 [**?**]. Here we only focus on the values of *m* in the form $m = 2^r$ and assume that d is a k-bit integer. The required number of operations for this method is [**?**]

- For precomputation: S + $(m-3)$M
- For powering: $(\frac{k}{r} - 1) \cdot r$S
- For multiplication: $\frac{m-1}{m} \cdot (\frac{k}{r} - 1)$M

For each RSA modulus size, there is a particular value of *m* for which the method requires the minimum multiplications and squarings [**?**]. In TABLE II, the number of operations are calculated, and optimal *m* values are defined.

By these results, *m*=32 provides the least number of operations for 1024-bits modulus size. For 2048-bits modulus size, 64 is the best option. Taking *m*=128 gives the most efficient results for 3072-bits and 4096-bits modulus sizes.

Algorithm 2 *m-ary* method

Input: $M,\ n,\ (d_l d_{l-1} \dots d_1 d_0)_m$
Output: $M^d \bmod n$
1: $pc[0] \leftarrow M^2 \bmod n$
2: **for** i **from** 1 **to** $m-3$ **do**
3: $pc[i] \leftarrow pc[i-1] \cdot M \bmod n$ { for look-up table}
4: **end for**
5: **if** $d_l = 0$ **then**
6: $C \leftarrow 1$
7: **else**
8: **if** $d_l = 1$ **then**
9: $C \leftarrow M$
10: **else**
11: $C \leftarrow pc[d_l - 2]$
12: **end if**
13: **end if**
14: **for** i **from** $l-1$ **to** 0 **do**
15: $C \leftarrow C^m \bmod n$
16: **if** $d_i = 1$ **then**
17: $C \leftarrow C \cdot M \bmod n$
18: **else**
19: **if** $d_i > 1$ **then**
20: $C \leftarrow C \cdot pc[d_i - 2] \bmod n$
21: **end if**
22: **end if**
23: **end for**
24: **return** C

TABLE II
AVERAGE NUMBER OF OPERATIONS

m	1024-bit	2048-bit	3072-bit	4096-bit
2	1023·S+512·M	2047·S+1024·M	3071·S+1536·M	4095·S+2048·M
4	1023·S+384·M	2047·S+768·M	3071·S+1152·M	4095·S+1536·M
8	1022·S+303·M	2046·S+604·M	3070·S+900·M	4094·S+1199·M
16	1021·S+252·M	2045·S+492·M	3069·S+732·M	4093·S+972·M
32	**1020·S+226·M**	2044·S+425·M	3068·S+588·M	4092·S+822·M
64	1019·S+228·M	**2043·S+396·M**	3067·S+564·M	4091·S+732·M
128	1018·S+269·M	2042·S+414·M	**3066·S+559·M**	**4090·S+705·M**
256	1017·S+380·M	2041·S+507·M	3065·S+635·M	4089·S+762·M

With a suitable choice of *m*, the *m-ary* method provides an improvement over the *repeated squaring and multiplication algorithm*. It should be noted that, additional memory is used to store the look-up table. Required memory sizes are given for different *m* values in TABLE III.

TABLE III
MEMORY USAGE-KB (1 KB = 1024-BYTE)

	4	8	16	32	64	128	256
1024-bit	0.25	0.75	1.75	3.75	7.75	15.75	31.75
2048-bit	0.5	1.5	3.5	7.5	15.5	31.5	63.5
3072-bit	0.75	2.25	5.25	11.25	23.25	47.25	95.25
4096-bit	1	3	7	15	31	63	127

C. The Modified m-ary Method

With the *m-ary* method, required number of operations decreases by considerable amounts for increasing values of

m. On the other hand, number of precomputation operations increases nearly twice when m is doubled. Basically, rapid increases in the required number of operations for large m values are caused by precomputation operations. The *modified m-ary* method decreases these multiplications by a factor of 2 with some modifications in the *m-ary* method. The idea stems from the fact that even powers of a number can be calculated with odd parts of the exponentiation. For instance, we can perform $(M^3)^2 \bmod n$ instead of $M^6 \bmod n$. The algorithm [**?**], [**?**] is given in Algorithm 3.

Algorithm 3 *modified m-ary* method

Input: $M,\ n,\ (d_l d_{l-1} \cdots d_0)_m$
Output: $M^d \bmod n$
1: $pc[0] \leftarrow M^3 \bmod n$
2: **for** i **from** 1 **to** $\frac{m-4}{2}$ **do**
3: $pc[i] \leftarrow pc[i-1] \cdot M^2 \bmod n$ {look-up table multiplications}
4: **end for**
5: **if** $d_l = 0$ **then**
6: $C \leftarrow 1$
7: **else**
8: $t \leftarrow 1$
9: **while** $(d_l \bmod 2) = 0$ **do**
10: $d_l \leftarrow d_l/2$
11: $t \leftarrow 2 \cdot t$
12: **end while**
13: **if** $d_l = 1$ **then**
14: $C \leftarrow M$
15: **else**
16: $C \leftarrow pc[\frac{d_l-3}{2}]$
17: **end if**
18: $C \leftarrow C^t \bmod n$
19: **end if**
20: **for** i **from** $l-1$ **to** 0 **do**
21: **if** $d_i = 0$ **then**
22: $C \leftarrow C^m \bmod n$
23: **else**
24: $t \leftarrow 1$
25: **while** $(d_i \bmod 2) = 0$ **do**
26: $d_i \leftarrow d_i/2$
27: $t \leftarrow 2 \cdot t$
28: **end while**
29: $C \leftarrow C^{\frac{m}{t}} \bmod n$
30: **if** $d_i = 1$ **then**
31: $C \leftarrow C \cdot M \bmod n$
32: **else**
33: $C \leftarrow C \cdot pc[\frac{d_i-3}{2}] \bmod n$
34: **end if**
35: $C \leftarrow C^t \bmod n$
36: **end if**
37: **end for**
38: **return** C

Average number of required operations is:

- For precomputation: S + $\frac{m-2}{2}$·M
- For powering: $(\frac{k}{r} - 1) \cdot r$·S
- For multiplication: $\frac{m-1}{m} \cdot (\frac{k}{r} - 1)$·M
- For division by 2: $\frac{k}{m \cdot r} \cdot (m - r - 1)$(Division by 2)

For the *modified m-ary method*, the required memory sizes are in TABLE IV . As we see from the table, the required memory sizes decrease to exactly half. Therefore, running time of preparation of the look-up table decreases to half.

TABLE IV
MEMORY USAGE-KB (1 KB=1024-BYTE)

	4	8	16	32	64	128	256
1024-bit	0.125	0.375	0.875	1.875	3.875	7.875	15.875
2048-bit	0.25	0.75	1.75	3.75	7.75	15.75	31.75
3072-bit	0.375	1.125	2.625	5.625	11.625	23.625	47.625
4096-bit	0.5	1.5	3.5	7.5	15.5	31.5	63.5

D. Reducing Precomputation Multiplications

Especially with small exponents, all base *m* numerals are not expected to be seen in the base *m* expansion of the exponent. In these situations, calculating all look-up table values becomes unnecessary. With the *reducing precomputation multiplications*, we ignore these unnecessary precomputations and reduce the number of required multiplications.

The *reducing precomputation multiplications* cannot be applied for large exponents since all base *m* numerals are expected to be seen in the expansion. Therefore, this method cannot be used in decryption effectively. But, it allows considerable improvements in encryption when the public key exponent *e* = 65537 [**?**] is used.

III. IMPLEMENTATION RESULTS OF STUDIED METHODS

In this section, implementation results are presented. These results are obtained using the *MPIR* library with *Microsoft Visual Studio* on Intel Core i7 2.00 GHz. Running times of these methods are compared with the *repeated squaring and multiplication algorithm* and the most efficient cases are noted. Encryption and decryption steps were discussed separately.

Aim of this study is to find more efficient methods than the *repeated squaring and multiplication algorithm* for modular exponentiation. Thus, the RSA cryptosystem parameters were chosen as in [**?**]. In the following tables, there are modulus size, m value, running time, comparison with *m*=2 and saving columns. With saving column, we intend to express obtained improvement over percent. If the obtained result is worse than *m*=2 case, then we express them with minus sign.

A. Encryption

The public key *e* was taken as 65537 in all experiments.

The *m-ary* method's running time results for different plaintext sizes are tabulated in TABLE V. According to the information in the table, the *m-ary* method is not suitable for encryption with *e* = 65537. Running time increases rapidly when *m* grows. The reason behind this increase is due to the computation of a larger look-up table. But, only one multiplication is needed or there is no need for any multiplication during the procedure. This means that all of the look-up table multiplications are never used. For example, if we take *m*

TABLE V
RUNNING TIME RESULTS FOR THE M-ARY METHOD ON ENCRYPTION

M	*m*	running time (ms)	comparison with *m*=2	saving (%)
1024-bit	2	0.22	1	0
	4	0.26	1.179	-17.9
	8	0.34	1.513	-51.3
	16	0.52	2.344	-134.4
	32	0.86	3.856	-285.6
	64	1.53	6.827	-582.7
	128	3.57	15.978	-1497.8
	256	9.49	42.447	-4144.7
2048-bit	2	0.98	1	0
	4	1.09	1.11	-11
	8	1.28	1.298	-29.8
	16	1.82	1.846	-84.6
	32	2.74	2.781	-178.1
	64	4.73	4.809	-380.9
	128	10	10.166	-916.6
	256	23.53	23.913	-2291.3
3072-bit	2	2.08	1	0
	4	2.34	1.123	-12.3
	8	2.73	1.309	-30.9
	16	3.88	1.858	-85.8
	32	5.85	2.803	-180.3
	64	9.98	4.785	-378.5
	128	20.19	9.679	-867.9
	256	44.85	21.498	-2049.8
4096-bit	2	3.51	1	0
	4	3.98	1.134	-13.4
	8	4.52	1.286	-28.6
	16	6.28	1.787	-78.7
	32	9.34	2.656	-165.6
	64	15.9	4.508	-350.8
	128	31.6	8.998	-799.8
	256	68.6	19.519	-1851.9

TABLE VI
RUNNING TIME RESULTS FOR THE REDUCING PRECOMPUTATION MULTIPLICATION METHOD ON ENCRYPTION

M	*m*	running time (ms)	comparison with *m*=2	saving (%)
1024-bit	2	0.05	1	0
	4	0.05	0.981	1.9
	8	0.06	1.086	-8.6
	16	0.05	0.979	2.1
	32	0.06	1.078	-7.8
	64	0.06	1.086	-8.6
	128	0.06	1.079	-7.9
	256	0.05	0.98	0.2
2048-bit	2	0.18	1	0
	4	0.18	0.976	2.4
	8	0.18	0.997	0.3
	16	0.17	0.965	0.5
	32	0.18	0.991	0.9
	64	0.18	0.993	0.7
	128	0.18	0.983	1.7
	256	0.17	0.959	4.1
3072-bit	2	0.37	1	0
	4	3.66	0.988	1.2
	8	3.69	0.995	0.5
	16	3.64	0.982	1.8
	32	3.63	0.980	2
	64	3.67	0.989	1.1
	128	3.68	0.997	0.3
	256	3.62	0.976	2.4
4096-bit	2	5.78	1	0
	4	5.76	0.997	0.3
	8	5.82	1.007	-0.7
	16	5.77	0.998	0.2
	32	5.83	1.008	-0.8
	64	5.81	1.005	-0.5
	128	5.81	1.005	-0.5
	256	5.72	0.995	0.5

= 256, there is no need for a look-up table. However, 254 precomputation multiplications are computed because of the *m-ary* method execution. To overcome this, we should consider the *reducing precomputation multiplications*. In TABLE VI, running time results of the method for different plaintext sizes are shown. By the table, small savings are obtained with the use of small amounts of memory. Obtained ratios are very close to 1. This means that the method does not provide a significant efficiency over the *repeated squaring and multiplication algorithm*. But, the *reducing precomputation multiplications* offers an alternative way for encryption.

B. Decryption

Decryption is the most time consuming part of the RSA algorithm because of the size of *d*. Hence, making improvements on this part has great importance. In TABLE VII running times of the *m-ary* method, comparision of the results with *m*=2 case and also amount of savings are listed.

By the table, for 1024-bits modulus size, choosing *m*=32 gives 28% improvement by using 3.75 KB memory. If we choose 2048-bits modulus size, *m*=64 provides 23% improvement with a cost of 15.5 KB memory. For 3072-bits modulus size, the best option is *m*=64 with 23% improvement and the required memory size is 23.25 KB. If the modulus size is chosen to be 4096-bits, *m*=128 provides 23% acceleration by using 64 KB of memory.

With the *m-ary* method, efficient cases are obtained for each modules size. But, we know that when the *modified m-ary* method is used, efficiency can be increased by using less memory. To observe running time relations between the *modified m-ary* and the *m-ary* method, these methods were compared according to the running times for different modulus sizes. Obtained running times and ratios are tabulated in the TABLE VIII. By the table, the *modified m-ary* method becomes more efficient than the *m-ary* method after *m*=16. If we look at the recommended cases for the *m-ary* method, these *m* values are larger than 16. Therefore, we should consider the *modified m-ary method* to get more efficient results.

In TABLE IX, running times of the *modified m-ary* method, comparisons with *m*=2 case and obtained savings are shown. In accordance with the information in the table, for 1024-bits modulus size choosing *m*=32 provides 28% acceleration with using 1.875 KB memory. For 2048-bits modulus size, *m*=64 provides 26% efficiency with using 7.75 KB memory. When the key size is 3072-bits, we should choose *m*=128 to get the best result. In this case, performance improvement is nearly 28% and required memory size is 23.625 KB. For 4096-bit key size, *m*=128 provides 25% acceleration with using 31.5 KB memory. The improvement does not seem to be significant compared to the *m-ary* method but the required memory size decreases to exactly the half in for each case.

IV. CONCLUSION

In this paper, we present some fast modular exponentiation methods. The aim is to find particular parameters for this oper-

TABLE VII
RUNNING TIME RESULTS FOR THE M-ARY METHOD WITH POWER OF 2

modulus size	*m*	running time (ms)	comparison with *m*=2	saving (%)
1024-bit	2	23.6	1	0
	4	20.24	0.857	14.3
	8	18.69	0.791	20.9
	16	17.6	0.745	25.5
	32	**17.09**	**0.723**	**27.7**
	64	17.39	0.736	26.4
	128	19.44	0.823	17.7
	256	25.49	1.078	-7.8
	512	51.31	2.171	-117.1
	1024	148.04	6.264	-526.4
2048-bit	2	173.77	1	0
	4	156.14	0.899	10.1
	8	145.95	0.84	16
	16	139.44	0.802	19.8
	32	**135.34**	**0.779**	**22.1**
	64	**134.5**	**0.774**	**22.6**
	128	138.91	0.799	22.1
	256	156.01	0.898	10.2
	512	177.77	1.023	-2.3
	1024	301.24	1.734	-73.4
3072-bit	2	98.01	1	0
	4	88.84	0.907	9.3
	8	82.59	0.843	15.7
	16	78.92	0.805	19.5
	32	76.64	0.782	21.8
	64	**75.6**	**0.771**	**22.9**
	128	77.51	0.791	20.9
	256	86.9	0.887	11.3
	512	123.39	1.259	-25.9
	1024	263.14	2.685	-168.5
4096-bit	2	1219.61	1	0
	4	1108.288	0.909	9.1
	8	1037.402	0.85	15
	16	990.694	0.812	18.8
	32	962.834	0.789	21.1
	64	**947.358**	**0.777**	**22.3**
	128	**944.52**	**0.774**	**22.6**
	256	977.716	0.802	19.8
	512	1043.298	0.855	14.5
	1024	1256.364	1.03	-3

TABLE VIII
RUNNING TIME RESULTS FOR COMPARISON OVER MODIFIED M-ARY METHOD & M-ARY METHOD

modulus size	*m*	*modified m-ary* (ms)	*m-ary* (ms)	ratio
1024-bit	4	4.14	3.99	1.038
	8	3.72	3.65	1.02
	16	3.44	3.43	1.005
	32	3.35	3.39	0.988
	64	3.32	3.54	0.938
	128	3.56	4.52	0.787
	256	4.48	8	0.56
2048-bit	4	29.94	29.49	1.015
	8	27.43	26.44	1.037
	16	25.28	25.13	1.005
	32	24.51	24.589	0.997
	64	23.92	24.44	0.979
	128	24.14	26.07	0.926
	256	25.9	32.53	0.796
3072-bit	4	88.27	87.9	1.004
	8	82.84	82.84	1
	16	79.24	79.24	1
	32	76.72	76.85	0.998
	64	74.96	76.09	0.985
	128	74.71	78.01	0.957
	256	77.01	87.39	0.881
4096-bit	4	193.04	192.08	1.005
	8	179.93	178.66	1.007
	16	171.32	170.72	1.004
	32	165.16	165.16	1
	64	161.17	162.4	0.992
	128	159.77	164.13	0.973
	256	162.23	176.66	0.918

ation which have better performance compared to the *repeated squaring and multiplication algorithm*. For encryption, we implemented the *m-ary* method and the *reducing precomputation multiplications*. According to the results, the *m-ary* method is not suitable for encryption. When the *reducing precomputation multiplications* is implemented, some alternatives are observed which use small memory sizes. However, we do not recommend an option that is better than the *repeated squaring and multiplication algorithm* for encryption.

For decryption, the *m-ary* and the *modified m-ary* methods were implemented. According to the results, the *m-ary* method provides a considerable improvement in running time. This improvement is greater than 23% for each modulus size. For the *modified m-ary* method, more efficient results are obtained compared to the *m-ary* method. With this method at least 25% improvement is achieved by using less memory than the *m-ary* method.

Consequently, acceleration of the RSA algorithm especially for decryption is very important to get efficient implementations. With the analyzed methods in this paper, we observed considerable improvements in the running time for some of the methods compared to the *repeated squaring and multiplication algorithm*.

V. ACKNOWLEDGEMENTS

The second author is partially supported by TÜBİTAK under Grant No. BIDEB114C052.

REFERENCES

[1] E. Akyldz, Ç. Çalık, M. Özarar, Z. Tok, O. Yayla, RSA Kriptosistemi Parametreleri için Güvenlik Testi Yazlm, ISCTURKEY 2013, Proceedings of 6th International Security & Cryptology Conference, pp.124-127
[2] D. Boneh, Twenty Years of Attacks on the RSA Cryptosystem, Notices of the AMS, pp. 203-213, February 1999
[3] J. Chung, M. A. Hasan, Asymmetric Squaring Formulae, Computer Arithmetic.ARITH'07. 18th IEEE Symposium on, pp.113-122, 2007
[4] D. M. Gordon, A survey of Fast Exponentiation Methods, Journal of Algorithms, Vol.27, Issue 1, pp.129-146, April 1998
[5] B. Kaliski, The Mathematics of the RSA Public-Key Cryptosystem, RSA Laboratories, 2006
[6] D. E. Knuth, The Art of Computer Programming, Vol.2/Seminumerical Algorithms, Second Edition, Addision-Wesley Publishing Company, 1981
[7] Ç. K. Koç, High-Speed RSA Implementation, RSA Laboratories, 1994
[8] A. Kumar, S. Jakhar, S. Makkar, Comparative Analysis Between DES and RSA Algorithm's, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 7, 2012
[9] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996
[10] R. L. Rivest, A. Shamir, L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, 1978
[11] M. Welschenbach, Cryptography in C and C++, Second Edition, Apress, 2005

TABLE IX
RUNNING TIME RESULTS FOR THE MODIFIED M-ARY METHOD WITH POWER OF 2

modulus size	m	running time (ms)	comparison with m=2	saving (%)
1024-bit	2	4.7	1	0
	4	4.07	0.867	13.3
	8	3.67	0.781	21.9
	16	3.5	0.745	25.5
	32	**3.38**	**0.719**	**28.1**
	64	3.39	0.721	27.9
	128	3.59	0.766	23.4
	256	4.54	0.967	3.3
2048-bit	2	32.32	1	0
	4	29.03	0.898	10.1
	8	26.20	0.811	18.9
	16	24.94	0.772	22.8
	32	24.14	0.747	25.3
	64	**23.74**	**0.735**	**26.5**
	128	23.9	0.74	26
	256	25.58	0.791	20.9
3072-bit	2	103.96	1	0
	4	93.48	0.899	10.1
	8	87.08	0.838	16.2
	16	81.79	0.787	21.3
	32	77.33	0.744	25.6
	64	76.02	0.731	26.9
	128	**75.32**	**0.724**	**27.6**
	256	77.63	0.747	25.3
4096-bit	2	213.79	1	0
	4	194.59	0.91	9
	8	182.15	0.852	14.8
	16	173.11	0.81	19
	32	168.24	0.787	21.3
	64	163.15	0.763	23.7
	128	**161.25**	**0.754**	**24.6**
	256	164.07	0.767	23.3