More Flexible Exponentiation with Precomputation

Chae Hoon Lim and Pil Joong Lee

Department of Electrical Engineering, Pohang University of Science and Technology (POSTECH), Pohang, 790-784, KOREA

Abstract. A new precomputation method is presented for computing g^R for a fixed element g and a randomly chosen exponent R in a given group. Our method is more efficient and flexible than the previously proposed methods, especially in the case where the amount of storage available is very small or quite large. It is also very efficient in computing $g^R y^E$ for a small size E and variable number y, which occurs in the verification of Schnorr's identification scheme or its variants. Finally it is shown that our method is well-suited for parallel processing as well.

1 Introduction

The problem of exponentiating fast in a given group (usually Z_N, N a large prime or a product of two large primes) is very important for efficient implementations of most public key cryptosystems (hereafter it is assumed w.l.o.g. that the computation is performed over Z_N and thus multiplication denotes multiplication mod N). A typical method for exponentiation is to use the binary algorithm, known as the square-and-multiply method [1]. For 512 bit modulus and exponent, this method requires 766 multiplications on average and 1022 in the worst case. The signed binary algorithm [2-3] can reduce the required number of multiplications to around 682 on average and 768 in the worst case.

On the other hand, using a moderate amount of storage for intermediate values, the performance can be considerably improved again. Knuth's 5-window algorithm [1,4] can do exponentiation in about 609 multiplications on average, including the on-line precomputation of 16 multiplications. The fastest known algorithm for exponentiation is the windowing method based on addition chains, where we can use bigger windows such as 10 [4] and need more storage for intermediate values [5]. Though finding the shortest addition chain is an NP-complete problem [6], it is reported [4] that, by applying heuristics, an addition chain of length around 605 can be computed.

These general methods can be used for any cryptosystems requiring exponentiation such as RSA [7] and ElGamal [8]. However, in many cryptographic protocols based on the discrete logarithm problem, we need to compute g^R for a fixed base g but for a randomly chosen exponent R. Thanks to the fixed base element, a precomputation table can be used to reduce the number of multiplications required, of course at the expense of storage for precomputed values.

At Eurocrypt'92, Brickell et al. [9] proposed such a method for speeding up the computation of g^R (called the BGMW method, for the convenience of

Y.G. Desmedt (Ed.): Advances in Cryptology - CRYPTO '94, LNCS 839, pp. 95-107, 1994.
© Springer-Verlag Berlin Heidelberg 1994

reference). Their basic strategy is to represent an exponent R in base b, that is, $R = d_{t-1}b^{t-1} + \cdots + d_1 b + d_0$ where $0 \leq d_i < b$ $(0 \leq i < t)$, and precompute all powers $g_i = g^{b^i}$. Then g^R can be computed by $\prod_{i=0}^{t-1} g_i^{d_i} = \prod_{d=1}^{b-1}(\prod_{d_i=d} g_i)^d$. Using a basic digit set for base b, they extended the basic scheme so that the computation time can be further decreased while the storage required is increased accordingly.

In this paper, we propose another precomputation method for fast exponentiation. Our method is a generalization of the simple observation that if an n-bit exponent R is divided into two equal blocks (i.e., $R = R_1 \times 2^{n/2} + R_0$) and $g_1 = g^{2^{n/2}}$ is precomputed, then g^R can be evaluated in a half of the time required by the binary method in the worst case ($\frac{7}{12}$ on average) by computing $g_1^{R_1} g^{R_0}$. It will be seen that the proposed method is more efficient, especially when the storage available is very small or quite large, and also more flexible, giving a wide range of time-memory tradeoffs, than the BGMW method. The case of using a small amount of storage is of great importance for an application to smart cards having limited storage and computing power, but the BGMW method is not so efficient for this case.

Another advantage of the proposed method is its efficiency in computing $g^R y^E$, where y is not fixed and the size of E is much less than that of R, which is needed for the verification of Schnorr's identification and signature scheme [10] or its variants, e.g., Brickell-McCurley's scheme [11] and Okamoto's scheme [12]. Note that, for this kind of computation, representing exponents in non-binary power base may considerably increase the on-line computational load (see section 4). Finally we show that the proposed method is also well-suited for parallel processing.

Throughout this paper, we will use g as a fixed element of Z_N and R as an n-bit random exponent over $[0, 2^n)$. We denote by $|S|$ the bit-length of S for an integer S or the cardinality of S for a set S. We also denote by $\lceil x \rceil$ the smallest integer not less than x and by $\lfloor x \rfloor$ the greatest integer not greater than x.

2 Review of Previous Work : BGMW Method

In this section, we briefly review the BGMW method, the precomputation method proposed by Brickell, Gordon, McCurley and Wilson at Eurocrypt'92. For more details, see the original paper [9].

A set of integers D is called a *basic digit set* for base b if any integer can be represented in base b using digits from the set D [13]. Suppose that we can choose a set M of multipliers and a parameter h for which

$$D(M, h) = \{mk | m \in M, 0 \leq k \leq h\} \tag{1}$$

is a basic digit set for base b. Then an n-bit exponent R can be represented as

$$R = \sum_{i=0}^{t-1} d_i b^i, \quad d_i = m_i k_i \in D(M, h). \tag{2}$$

With this representation of R, g^R can be computed by

$$g^R = \prod_{i=0}^{t-1} g^{m_i k_i b^i} = \prod_{k=1}^{h} \left(\prod_{k_i=k} g^{m_i b^i} \right)^k = \prod_{k=1}^{h} c_k^k. \tag{3}$$

Therefore, if we precompute and store powers g^{mb^i} for all $i < t$ and $m \in M$, then g^R can be computed in at most $t + h - 2$ multiplications using about $t|M|$ precomputed values by the following algorithm.

```
u := ∏_{k_i=h} g^{m_i b^i};
v := u;
for w := h − 1 to 1 step -1
    u := u * ∏_{k_i=w} g^{m_i b^i};
    v := v * u;
return(v);
```

It is easily seen that the number of multiplications performed by the above algorithm is $t + h - 2$ in the worst case ($t - h$ multiplications for computing products of the form $\prod_{k_i=w} g^{m_i b^i}$ for $w = 1, \cdots, h$ and $2h - 2$ multiplications for completing the for-loop) and $\frac{b-1}{b}t + h - 2$ on average (For a randomly chosen exponent, $\frac{t}{b}$ digits are expected to be zero.).

The most obvious example for D is the base b number system ($M = \{1\}$, $h = b - 1$, $t = \lceil \log_b (2^n - 1) \rceil$). For a 512 bit exponent, the choice of $b = 26$ minimizes the expected number of multiplications. This basic scheme requires 127.8 multiplications on average, 132 in the worst case, and storage for 109 precomputed values. More convenient choice of base will be $b = 32$, since then the digits for the exponent R can be computed without radix conversion by extracting 5 bits at a time. With this base, the required number of multiplications is increased only by one for the average case and remains unchanged for the worst case. Though the basic scheme is the obvious choice in the case where the storage available is small, its performance is considerably degraded as the number of storage is going down below 109. This means that the BGMW method does not provide an efficient way to perform the computation when the storage available is very small.

Brickell et al. also presented several schemes using other number systems to decrease the number of multiplications required, of course using more storage for precomputed values. One of the extreme examples is to choose the set M as $M_2 = \{m | 1 \le m < b, \omega_2(m) = 0 \bmod 2\}$, where $\omega_p(m)$ is the highest power of p dividing m. Then, for $1 \le d_i < b$, we have $d_i = m$ or $2m$ for some $m \in M_2$ (i.e., $h = 2$). Thus g^R can be computed in t multiplications on average and $\frac{b-1}{b}t$ multiplications in the worst case, with the storage to $|M_2| \lceil \log_b (2^n - 1) \rceil$ values. For example, taking $b = 256$ ($t = 64$, $|M_2| = 170$), we can achieve an average of 63.75 multiplications with 10880 precomputed values. Two tables that Brickell et al. presented in [9] are given in the appendix A for the purpose of comparison with our results.

3 The Proposed Method

We now present our method for fast evaluation of g^R using a precomputation table. Let R be an n-bit exponent for which we want to compute g^R. We first divide the exponent R into h blocks R_i, for $0 \le i \le h-1$, of size $a = \lceil \frac{n}{h} \rceil$ and then subdivide each R_i into v smaller blocks $R_{i,j}$, for $0 \le j \le v-1$, of size $b = \lceil \frac{a}{v} \rceil$ as follows (see figure 1):

$$R = R_{h-1} \cdots R_1 R_0 = \sum_{i=0}^{h-1} R_i 2^{ia},$$

$$R_i = R_{i,v-1} \cdots R_{i,1} R_{i,0} = \sum_{j=0}^{v-1} R_{i,j} 2^{jb}. \tag{4}$$

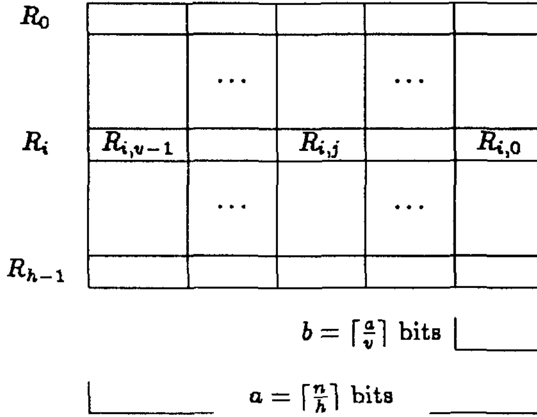

Figure 1 : Division and arrangement of an n-bit exponent R

Let $g_0 = g$ and define g_i as $g_i = g_{i-1}^{2^a} = g^{2^{ia}}$ for $0 < i < h$. Then, using the equations (4), we can express g^R as

$$g^R = \prod_{i=0}^{h-1} g_i^{R_i} = \prod_{j=0}^{v-1} \prod_{i=0}^{h-1} \left(g_i^{2^{jb}} \right)^{R_{i,j}}. \tag{5}$$

If we let $R_i = e_{i,a-1} \cdots e_{i,1} e_{i,0}$ be the binary representation of R_i $(0 \le i < h)$, then $R_{i,j}$ $(0 \le j < v)$ is represented in binary as

$$R_{i,j} = e_{i,jb+b-1} \cdots e_{i,jb+k} \cdots e_{i,jb+1} e_{i,jb}.$$

Therefore the expression (5) can be rewritten as follows :

$$g^R = \prod_{k=0}^{b-1} \left(\prod_{j=0}^{v-1} \prod_{i=0}^{h-1} g_i^{2^{jb} e_{i,jb+k}} \right)^{2^k} \tag{6}$$

Next suppose that the following values are precomputed and stored for all $1 \le i < 2^h$ and $0 \le j < v$.

$$G[0][i] = g_{h-1}^{e_{h-1}} g_{h-2}^{e_{h-2}} \cdots g_1^{e_1} g_0^{e_0},$$
$$G[j][i] = (G[j-1][i])^{2^b} = (G[0][i])^{2^{jb}}. \tag{7}$$

Here the index i is equal to the decimal value of $e_{h-1} \cdots e_1 e_0$. Then, using the precomputed values of (7), we can rewrite the expression (6) as

$$g^R = \prod_{k=0}^{b-1} \left(\prod_{j=0}^{v-1} G[j][I_{j,k}] \right)^{2^k}, \tag{8}$$

where $I_{j,k} = e_{h-1,bj+k} \cdots e_{1,bj+k} e_{0,bj+k}$ $(0 \le j < b)$, which corresponds to the k-th bit column of the j-th block column in the figure 1. Now it is straightforward to compute g^R using the expression (8) by the ordinary square-and-multiply method as follows :

```
Z := 1;
for k := b − 1 to 0 step -1
     Z := Z * Z;
     for j := v − 1 to 0 step -1
          Z := Z * G[j][I_{j,k}];
return(Z);
```

We next count the number of multiplications required by the above algorithm. Here we have to note that the $(v-1)$-th blocks in the figure 1 may not be full of b bits. In fact, they are of $bv - a$ bit size. Thus the number of terms to be multiplied together in the inner for-loop is $v + 1$ for the first $bv - a$ rounds and $v + 2$ for the remaining $b - bv + a$ rounds. Therefore, the total number of multiplications required is at most v(bv-a)+(v+1)(b-bv+a)-2 = a+b-2 in the worst case. Since we may assume that the probability of $I_{j,k}$ being zero is $\frac{1}{2^h}$ and there are a occurrences of $I_{j,k}$ in the above algorithm, the expected number of multiplications is given by $\frac{2^h-1}{2^h}a + b - 2$. Of course, this performance is achieved with storage for $(2^h - 1)v$ precomputed values.

In the above, we assumed that the exponent R is partitioned into hv blocks of almost equal size and that these hv blocks are arranged in a $h \times v$ rectangular shape. In most cases, such partitions and arrangements yield better performance than others for a given amount of storage, but sometimes this may not be the case. For example, consider two configurations shown in the figure 2 below, where a 512-bit exponent is partitioned and arranged in two different ways. The first configuration corresponds to the case we analyzed in the above and results in the performance of 118.78 multiplications on average (122 in the worst case) with storage for 155 values. On the other hand, with the second configuration, we can do the exponentiation in 117.13 multiplications on average (119 in the

worst case) using storage for 157 values. This shows that we had better choose the second configuration.

19	21	21	21	21
19				
18				
18				
18				

5 × 5 configuration

28	31	31

5 × 1|6 × 2 configuration

Figure 2 : Two different configurations with almost the same storage requirement

For the configuration of type $h_1 \times v_1 | h_2 \times v_2$ where $h_1 < h_2$, we can easily derive general formulae for the worst/average-case performance. Let b_1 and b_2 be the size of partitioned blocks in $h_1 \times v_1$ and $h_2 \times v_2$ respectively. For better performance, b_2 must be greater than or equal to b_1 and can be obtained in the same way as b in the $h \times v$ configuration. Thus we get $b_2 = \lceil \frac{n}{h_1 v_1 + h_2 v_2} \rceil$ and $b_1 = \lceil \frac{n - b_2 h_2 v_2}{h_1 v_1} \rceil$. Now the worst-case number of multiplications required for this configuration can be directly obtained from the formula for the $h \times v$ configuration. by replacing a by $h_1 v_1 + h_2 v_2$ and b by b_2 respectively, This results in $b_1 v_1 + b_2(v_2 + 1) - 2$ multiplications in the worst case. Similarly, the expected number of multiplications can be shown to be $\frac{2^{h_1}-1}{2^{h_1}} b_1 v_1 + \frac{2^{h_2}-1}{2^{h_2}} b_2 v_2 + b_2 - 2$. This performance can be achieved with the storage for $(2^{h_1} - 1)v_1 + (2^{h_2} - 1)v_2$ precomputed values. We can easily see that no configurations other than the two types, $h \times v$ and $h_1 \times v_1 | h_2 \times v_2 (h_2 = h_1 + 1)$, yield better performance for a given amount of storage.

The number of multiplications and storage requirements for a 512 bit modulus are summarized in tables B.1 and B.2 in the appendix B, for a 160-bit and 512-bit exponent respectively. Note that not only is the proposed method simpler, but it also achieves better performance than the BGMW method. In particular, due to its effectiveness over a wide range of storage, our method is flexibly applicable to various computing environments according to the amount of storage available. For example, to speed up the computation by smart cards, we may choose the configuration of 4×2. Then for 512 bit modulus and exponent the computation of g^R can be done in 182 multiplications on average with 1920 bytes of storage. On the other hand, when a relatively large amount of storage is available, we can choose, for example, the configuration of 7×4, achieving 90.42 multiplications on average with about 32 Kbytes of storage.

4 Speeding up Identification and Signature Verifications

Based on the discrete logarithm problem, a lot of identification and digital signature schemes have been developed (e.g., [10-12]). In all these schemes, along with a few modular multiplications the prover (or the signer) needs to compute g^R for a random R, which can be efficiently performed by the method described in section 3. On the other hand, to validate the prover's identity or the signature, the verifier needs to perform the computation of the form $g^R y^E$ where y corresponds to the public key of the prover (or the signer) and thus varies in each run of the protocol. The size of E typically lies between 20 and 40 in identification schemes and around 80 in the corresponding signature schemes. This section investigates the performance of the proposed method for computing $g^R y^E$.

Let t be the size of E. It is clear that if $t \leq b$, then $g^R y^E$ can be computed in $a + b + t - 2$ multiplications in the worst case and $\frac{2^h-1}{2^h}a + b + 0.5t - 2$ on average. In case of $t > b$, we can either proceed as above or do the computation after partitioning E into smaller blocks. The first case yields the performance of $a + 2t - 2$ multiplications in the worst case and $\frac{2^h-1}{2^h}a + 1.5t - 2$ on average. However, if t is much larger than b, the performance can be further improved by dividing E into smaller blocks.

Thus, for more general formulae, suppose that E is partitioned into u blocks of almost equal size (Consider the whole configuration for computing $g^R y^E$ as $u \times 1|h \times v$). Let c be the bit-length of the partitioned blocks (i.e., $c = \lceil \frac{t}{u} \rceil$). Then, we first have to compute $y^{2^{kc}}$ for $k = 1, 2, \cdots, u-1$, and each product of their possible combinations, which all together requires $(u-1)c + 2^u - u - 1$ multiplications. For the range of t we are interested in, i.e., up to $t = 80$, u takes on at most 3. Now, if $c \leq b$, then at most c additional multiplications are sufficient in the worst case ($\frac{(2^u-1}{2^u}c$ on average). Therefore the total number of multiplications required in this case is $a + b + uc + 2^u - u - 3$ in the worst case and $\frac{2^h-1}{2^h}a + b + \frac{u2^u-1}{2^u}c + 2^u - u - 3$ on average. Similarly, for the case of $c > b$ we can easily show that the number of multiplications is $a + (u+1)c + 2^u - u - 3$ in the worst case and $\frac{2^h-1}{2^h}a + \frac{(u+1)2^u-1}{2^u}c + 2^u - u - 3$ on average.

With the proposed method, the Schnorr-like identification and/or signature schemes can be made more practical for smart card implementations. For example, with a 512-bit modulus, 160-bit exponents and $t = 30$, the verification condition can be checked in 80.5 multiplications on average, if 1920 bytes of storage are available (4×2 configuration). Similarly, a signature with $t = 80$ can be verified in 144.13 multiplications on average using the same amount of storage. This is a considerable speedup only with a very small amount of storage, compared with the binary method requiring 246.5 multiplications for $t = 30$ and 259.0 multiplications for $t = 80$ on average. Moreover, identification or signature verifications are usually performed in much more powerful terminals capable of being equipped with a large amount of memory. In such an environment, we may adopt the 8×2 configuration and thus can perform, on average, identity verifications in 60.2 multiplications for $t = 30$ and signature verifications in 126.6 multiplications for $t = 80$, using about 32 Kbytes of storage.

Further improvement with additional communication : Small additional communication can considerably reduce the number of multiplications for computing $g^R y^E$ again. That is, the verifier can save the on-line computational load for preparing $y_k = y^{2^{kc}}$ for $k = 1, 2, \cdots, u-1$, if they are precomputed and stored by the signer (or the prover), since y is a fixed number to him, and then transmitted together with other data. For example, for the signature scheme with $t = 80$, if the signer sends 2 additional 512 bit blocks y_1, y_2 where $y_1 = y^{2^{27}}$ and $y_2 = y_1^{2^{27}}$, together with a signature for message, then the signature verification can be done in 90.13 multiplications on average with the 4×2 configuration. Therefore, 54 multiplications can be saved only with the increase of 128 bytes of communication. This corresponds to about a 3-fold speedup on average over the binary method which requires 259 multiplications on average.

For comparison, it is worth mentioning that the BGMW method is less efficient for the computation of the form $g^R y^E$ in either case we considered above. In case of no additional communication, if the exponents are represented in non-binary power base, more computations are needed in performing the on-line precomputation required for y^E. When addtional communication is allowed, more precomputed values must be transmitted due to the use of small base.

The above method of combining precomputation and additional communication can be used to speed up the verification of the digital signature standard (DSS) [14] as well. In DSS, we have to perform the computation of the type $g^R y^E$ with $|R| = |E| = 160$ and thus without additional communication we can gain no advantage with precomputation. However, if the signer sends 3 additional blocks $\{y_1, y_2, y_3\}$ where $y_i = y_{i-1}^{2^{40}}$ for $i = 1, 2, 3$ and if the verifier adopts the 4×2 configuration, then the signature can be verified in 124 multiplications on average. This is more than a 2-fold improvement over the binary method which requires 279 multiplications on average, only with 1920 bytes of storage and 192 bytes of additional communication (for a 512 bit modulus).

signature generation				
schemes	n / t	binary	4 × 2	8 × 2
Schnorr	160 / 80	318 / 238	58 / 55.5	28 / 27.9
DSS	160 / 160	318 / 238	58 / 55.5	28 / 27.9
BM	512 / 80	1022 / 766	190 / 182.0	94 / 93.8

signature verification				
schemes	n / t	binary	4 × 2	8 × 2
Schnorr	160 / 80	319 / 259	89 / 85.3	69 / 67.7
DSS	160 / 160	319 / 279	129 / 124.0	109 / 106.4
BM	512 / 80	1023 / 787	221 / 211.8	125 / 123.5

Table 1 : Worst/average performance w/ 3 block additional commun.

The table 1 shows the number of multiplications required for signature generation and verification in three signature schemes (Schnorr [10], DSS [14] and

Brickell-McCurley [11]), under the assumption that the signer sends additionally 3 precomputed values for his public key together with a signature, as mentioned above. Here we only take into account the number of multiplications for exponentiation operations, neglecting some other necessary operations such as reduction mod q and multiplicative inverse mod q where q is a prime of about 160 bit size. Two configurations of 4×2 and 8×2 are taken as examples, since the former is suitable for smart card applications and the latter for more general applications with a relatively large amount of storage available. For comparison, the performance of the binary method is also presented.

5 Parallel Processing

With multiple processors, the proposed method can be parallelized, much more efficiently than the BGMW method, by assigning the j-th processor to the j-th column of the $h \times v$ configuration (see figure 1). That is, if v processors are available, then the j-th processor can be assigned to compute

$$\prod_{k=0}^{b-1} (G[j][I_{j,k}])^{2^k} \tag{9}$$

in the expression of

$$g^R = \prod_{j=0}^{v-1} \prod_{k=0}^{b-1} (G[j][I_{j,k}])^{2^k}, \tag{10}$$

where we assume that each processor stores in its local memory $2^h - 1$ precomputed values. The computation of each processor can be completed in at most $2(b-1)$ multiplications. After then, we need $\lceil log_2 v \rceil$ multiplications in addition to produce the final result. Therefore, the total number of multiplications is $2(b-1) + \lceil log_2 v \rceil$.

Table B.3 in the appendix B shows the required number of multiplications for 160/512 bit exponents, according to the number of processors and the storage needed per processor. Note that only with a small number of processors the performance can be greatly improved. For example, for 512-bit modulus and exponent, we can compute g^R in 32 multiplications , when 4 processors are available and each processor has a local storage for 255 precomputed values (about 16 Kbytes). With more processors, say 16, the exponentiation can be done in 10 multiplications with the same storage requirement.

The described parallel computation can be more efficiently implemented by a special-purpose hardware. For example, with 8 pairs of multiply and squaring circuits together with read-only memory for 120 precomputed values (4×8 configuration), we can compute g^R with $|R| = 512$ in 18 multiplications and 15 squarings. If we use storage for 2040 values (8×8 configuration) with the same circuits, then the computation can be done in 10 multiplications and 7 squarings.

6 Conclusion

We have proposed a new method for fast exponentiation with precomputation. The proposed method is very simple but achieves better performance than the BGMW method [9]. Our method is also preferable since it is flexibly applicable to various computing environments due to its wide range of time-storage tradeoffs. In particular, using the proposed method, we can substantially speed up the computation by smart cards with only a very small amount of storage. We also showed that the proposed method can also speed up the computation of the form $g^R y^E$ with y variable. This can make much more practical the Schnorr-type identification and signature scheme, since the verifier as well as the prover (signer) can gain great computational advantage with a moderate amount of storage. Finally we presented how the proposed algorithm can be parallelized. Such parallel processing may be useful in high performance server machines with multiple processors.

References

1. D.E.Knuth, *The art of computer programming, Vol.2 : Seminumerical algorithms*, second Edition, Addison-Wesley (1981).
2. J.Jedwab and C.J.Mitchell, "Minimum weight modified signed-digit representations and fast exponentiation," *Elect. Let.* 25 (17), 1171-1172 (1989).
3. C.N.Zhang, "An improved binary algorithm for RSA," *Computers Math. Applic.* 25 (6), 15-24 (1993).
4. J.Bos and M.Coster, "Addition chain heuristics," In *Advances in Cryptoloy-Crypto'89, Lecture Notes in Computer Science 435*, (edited by G.Brassard), pp.400-407, Springer-Verlag (1990).
5. J.Sauerbrey and A.Dietel, "Resource requirements for the application of addition chains modulo exponentiation," In Proc. Eurocrypt'92, Balatonfured, Hungary (1992).
6. P.Downey, B.Leony and R.Sethi, "Computing sequences with addition chains," Siam J. Comput. 3, 638-696 (1981).
7. R.L.Rivest, A.Shamir and L.Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, 21 (2), 120-126 (1978).
8. T.ElGmal, "A public key cryptosystem and a signature scheme based on the discrete logarithm," *IEEE Trans. Inform. Theory* 31 (4), 469-472 (1985).
9. E.F.Brickell, D.M.Gordon, K.S.McCurley and D.Wilson, "Fast exponentiation with precomputation," In *Proc. Eurocrypt'92*, Balatonfured, Hungary (1992).
10. C.P.Schnorr, "Efficient signature generation by smart cards," *J. Cryptology* 4 (3), 161-174 (1991).
11. E.F.Brickell and K.S.McCurley, "An interactive identification scheme based on discrete logarithms and factoring," *J. Cryptology* 5 (1) 29-39, (1992).
12. T.Okamoto, "Provably secure and practical identification schemes and corresponding signature schemes," In *Proc. Crypto'92*, Santa Barbara, CA (1992).
13. D.W.Matula, "Basic digit sets for radix representation," *J.ACM* 29, 1131-1143 (1982).
14. A proposed Federal information processing standard for digital signature standard (DSS), *Federal Register* 56 (169), 42980-42982 (1991).

A The Performance of the BGMW Method

Table A.1 : Selected parameters for a 160-bit exponent

b	M	h	storage	worst / average
13	$\{1\}$	12	45	54 / 50.25
19	$\{\pm1\}$	9	76	45 / 43.00
29	$\{\pm1, \pm2\}$	9	134	41 / 39.83
36	$\{\pm1, 9, \pm14, \pm17\}$	7	219	37 / 36.11
36	M_3	3	620	32 /31.14
64	M_2	2	1134	27 / 26.58
128	M_3	3	1748	24 / 23.82
256	M_2	2	2751	21 / 20.92

Table A.2 : Selected parameters for a 512-bit exponent

b	M	h	storage	worst / average
26	$\{1\}$	25	109	132 / 127.81
45	$\{\pm1\}$	22	188	114 / 111.91
53	$\{\pm1, \pm2\}$	17	362	106 / 104.28
67	$\{\pm1, \pm2, \pm23\}$	16	512	100 / 98.72
64	M_3	3	3096	87 / 85.66
122	M_3	3	5402	75 / 74.39
256	M_2	2	10880	64 / 63.75

B The Performance of the Proposed Method

Table B.1 : Selected parameters for a 160 bit exponent

configuration	storage	worst / average
2×2	6	118 / 98.00
$2 \times 1 \vert 3 \times 1$	10	94 / 82.00
3×2	14	79 / 72.25
$3 \times 1 \vert 4 \times 1$	22	67 / 62.69
4×2	30	58 / 55.50
4×3	45	52 / 49.50
5×2	62	46 / 45.00
$4 \times 1 \vert 5 \times 2$	77	44 / 42.63
5×3	93	41 / 40.00
5×4	124	38 / 37.00
$5 \times 1 \vert 6 \times 2$	157	36 / 35.44
6×3	189	34 / 33.58
6×4	252	32 / 31.58
$6 \times 1 \vert 7 \times 2$	317	30 / 29.75
7×3	381	29 / 28.82
7×4	508	27 / 26.82
7×6	762	25 / 24.82
8×4	1020	23 / 22.92
8×7	1785	21 / 20.92
$8 \times 1 \vert 9 \times 4$	2299	20 / 19.96

Table B.3 : Performance on parallel processing ($|R| = 160/512$)

np \ sp	15	31	63	127	255	511	1023
2	39 / 127	31 /103	27 / 85	23 / 73	19 / 63	17 / 57	15 / 51
4	20 / 64	16 / 53	14 / 44	12 / 38	10 / 32	- / 30	8 / 26
6	15 / 45	13 / 37	11 / 31	9 / 27	- / 23	7 / 21	- / 19
8	11 / 33	9 / 27	- / 23	7 / 21	- / 17	- / -	5 / 15
16	8 / 18	6 / 16	- / 14	- / 12	- / 10	- / -	4 / -
32	7 / 11	5 / -	- / 9	- / -	- / 7	- / -	- / -

* np = no. of processors (v), sp = storage / processor ($2^h - 1$)

Table B.2 : Selected parameters for a 512 bit exponent

configuration	storage	worst / average
2×2	6	382 / 318.00
$2 \times 1 \| 3 \times 1$	10	306 / 267.63
3×2	14	255 / 233.63
$3 \times 1 \| 4 \times 1$	22	218 / 204.38
4×2	30	190 / 182.00
4×3	45	169 / 161.00
5×2	62	153 / 149.78
5×3	93	136 / 132.78
$5 \times 2 \| 6 \times 1$	125	126 / 123.50
$5 \times 1 \| 6 \times 2$	157	106 / 104.66
6×3	189	113 / 111.66
6×4	252	106 / 104.66
$6 \times 1 \| 7 \times 2$	317	101 / 100.20
7×3	381	97 / 96.42
7×4	508	91 / 90.42
7×5	635	87 / 86.42
$7 \times 1 \| 8 \times 3$	892	81 / 80.68
8×4	1020	78 / 77.75
8×5	1275	75 / 74.75
8×6	1530	73 / 72.75
8×8	2040	70 / 69.75
9×5	2555	67 / 66.89
9×6	3066	65 / 64.89
$9 \times 6 \| 10 \times 1$	4089	62 / 61.90
$9 \times 1 \| 10 \times 5$	5626	59 / 58.94
$9 \times 1 \| 10 \times 7$	7672	57 / 56.95
$10 \times 2 \| 11 \times 4$	10234	54 / 53.97
$10 \times 2 \| 11 \times 6$	13305	52 / 51.98