Efficient Exponentiation using Precomputation and Vector Addition Chains

Peter de Rooij

PTT Research*

Abstract. An efficient algorithm for exponentiation with precomputation is proposed and analyzed. Due to the precomputations, it is mainly useful for (discrete log based) systems with a fixed base.
The algorithm is an instance of a more general method. This method is based on two ideas. Firstly, the idea from [3] of splitting the exponentiation into the product of a number of exponentiations with smaller exponents. Secondly, the use of the technique of vector addition chains to compute this product of powers.
Depending on the amount of precomputations and memory, the proposed algorithm is about two to six times as fast as binary square and multiply. It is slightly slower than the method from [3], but requires far less memory.

1 Introduction

In many cryptographic systems the users must perform one or more exponentiations in a group, such as $\mathbb{Z}_n$ or an elliptic curve group. This is a very time consuming operation, as it can be decomposed in a large number of multiplications in this group.

The time required for an exponentiation can be reduced by two orthogonal methods. On the one hand, one can reduce the time per multiplication by optimizing it. On the other hand, one can reduce the *number* of multiplications. This is the method discussed in this paper.

In the sequel, an exponentiation is indicated as g^x, irrespective of the group in which it occurs. Indeed the algorithms discussed below are independent of the group they are performed in; they are aimed at reducing the number of group operations for an exponentiation, irrespective of this group.

Assume we want to devise an algorithm to compute g^x with as few multiplications as possible. There a three approaches:

- The algorithm is independent of both g and x. An example is the binary square and multiply (BSM) algorithm, or its m-ary generalization [7]. For a random 512-bit exponent, binary square and multiply requires on average 767 multiplications. The 'sliding window' variant of the m-ary method reduces

* P.O. Box 421, 2260 AK Leidschendam, the Netherlands. E-mail: P.J.N.deRooij@research.ptt.nl

this to about 611 multiplications, using a temporary storage of 16 powers of g.

- The algorithm is independent of g but depends on x. This is useful for RSA [9], for example: there g is the message, while x is a key.
  An example is the precomputation of an optimal sequence of multiplications and squarings, a so-called *addition chain*, 'tailor-made' for x [7, 1, 5]. For a random 512-bit exponent, about 605 multiplications [1] are needed.
- The algorithm is independent of x but depends on g. This means that it is suitable for use in for example signature generation and identification as in DSS, Brickell-McCurley and Schnorr [8, 4, 10] and most other discrete log based protocols, as one has a fixed base g in that case.
  An example of this approach is the method with precomputation of some powers of g from [3]. Depending on the chosen parameters, a 512-bit exponent requires about 128 multiplications with a storage of 109 precomputed powers of g, down to as little as 64 multiplications with a storage of 10880 precomputed powers.

This paper takes this last approach: we consider exponentiations with a fixed base only. The algorithm from [3] is improved upon with respect to storage: with only 32 (resp. 8, 2) precomputed powers, the proposed algorithm takes about 133 (resp. 183, 411) multiplications for 512-bit exponents.

This paper is organized as follows. Section 2 describes the basic algorithm given in [3]. Furthermore, it is briefly discussed what the difference in approach of the new algorithm is. Section 3 is on vector addition chains, being the major tool that the proposed algorithm uses. Section 4 gives the algorithm itself. In Section 5, the complexity of this algorithm is assessed; Section 6 gives the conclusions.

2 Exponentiation with Precomputation

This section introduces and discusses the algorithm by Brickell, Gordon, McCurley and Wilson from [3]. Basically, this algorithm rewrites an exponentiation with a large exponent as a product of a number of exponentiations with smaller exponents, and cleverly computes this product.

Assume we want to compute powers g^x for random x smaller than some upper bound N. A typical size is $N \sim 2^{512}$. Now suppose we have precomputed some powers $g^{b_0}, g^{b_1}, \ldots, g^{b_{m-1}}$. If we can write

$$x = \sum_{i=0}^{m-1} x_i b_i, \tag{1}$$

then obviously

$$g^x = \prod_{i=0}^{m-1} (g^{b_i})^{x_i}. \tag{2}$$

Clearly, the set of exponents has to be chosen cleverly to make the computation of Equation 2 more efficient than the usual algorithms. An obvious choice is $\{1, b, b^2, \ldots, b^{m-1}\}$, with m the smallest integer satisfying $b^m > N$ (viz. $b = \lceil N^{1/m} \rceil$). That is, Equation 1 represents the b-ary representation of x for a suitably chosen base b. As in [3], we limit the discussion to this case.

In the rest of this paper, the notation $g_i = g^{b^i}$ will be employed. In [3] the following algorithm is given to compute the product

$$g^x = \prod_{i=0}^{m-1} g_i^{x_i}, \tag{3}$$

with h denoting the maximal possible value for the x_i's:

Algorithm B
$A, B \leftarrow 1$
for $d = h$ down to 1 do
    for each i with $x_i = d$ do
        $B \leftarrow B \cdot g_i$
    endfor
    $A \leftarrow A \cdot B$
endfor
return A

For any x_i the following holds: as soon as $d = x_i$, g_i is multiplied into B. In that step and the following $d - 1$ steps, A is multiplied by B; so in the end A is multiplied by $g_i^{x_i}$.

Note that this method is essentially *unary*: to multiply by $g_i^{x_i}$ we just multiply x_i times by g_i. The clever part is that all m of those unary exponentiations are done simultaneously. The complete algorithm requires at most $m + h - 2$ multiplications [3].

Note that we have $m = \lceil \log_b N \rceil$, and $h = b - 1$. That is, the number of multiplications is at most $b + \lceil \log_b N \rceil - 3$; on average it is at most $b + \frac{b-1}{b} \lceil \log_b N \rceil - 3$, as an x_i is 0 with probability $\frac{1}{b}$. Obviously, we must have relatively small values of b to keep this number low. For example, for $N = 2^{512}$, the optimal choice is $b = 26$, yielding $m = 109$ and $h = 25$.

If we want to decrease the number of factors m in Equation 3 significantly, the 'height' h of the unary exponentiation becomes prohibitively large: m is inversely proportional to $\log_2 b$ (viz. $m \approx \log_2 N / \log_2 b$), while h is proportional to b itself. For example, halving m requires a squaring of b, and consequently of h.

By the same argument, a significant decrease of storage is infeasible for Algorithm B or variants: we obviously need at least m stored powers. That is, this algorithm is not especially suited to minimize storage. (Indeed this is not attempted in [3]; instead, extensions are proposed that further reduce the number of multiplications at the cost of more storage.)

Informally, this can be seen as follows. The algorithm deals optimally with the number of factors in Equation 3, as one cannot do better than $m-1$ multiplications for a product of m independent factors. Each of those factors is an exponentiation as well, and those are not at all dealt with optimally: the used algorithm is unary. One might say that this favors large m, and consequently large storage, and small size h of the exponents in each factor.

3 Vector Addition Chains

3.1 Introduction

In this paper, a different way of computation of Equation 3 is proposed. It is based on the observation that any vector addition chain algorithm can be used to perform the computation of a product of a number of exponentiations. This will become clear below. The proposed algorithm has a complexity that is roughly logarithmic in h. This allows much larger bases and therefore less storage is needed. Informally: the factors each are dealt with more efficiently, and though the complexity of the algorithm does not split nicely in an 'combination-part' (m) and an 'exponentiation-part' (h), one might say that this is paid for by a less efficient combination of the factors.

3.2 Addition Chains

Before turning to vector addition chains, we briefly introduce the concept of *addition chains*. The main reference for this subject is [7], for newer developments, see [2, 1, 5].

In the computation of an exponentiation, every intermediate result is the product of two (not necessarily distinct) preceding intermediate results. The computation is fully described by the sequence of intermediate exponents. In this sequence $s_0 = 1, s_1, s_2, \ldots, s_L = x$, each s_i $(1 \leq i \leq L)$ is the *sum* of two preceding terms. The number of multiplications for the corresponding exponentiation equals the *length* L of the sequence; x is called the *target*.

The sequence $(s_i)_i$ is called an *addition chain for* x. The problem of finding the algorithm that computes g^x with the minimal number of multiplications now reduces to finding the shortest addition chain for x.

3.3 Vector Addition Chains

In the computation of a product of powers of g_i's, every intermediate result is the product of such powers. The idea of addition chains can be generalized to such products by representing them as vectors. That is, a product $\prod_{i=0}^{m-1} g_i^{e_i}$ is represented by the vector $\boldsymbol{e} = (e_0, \ldots, e_{m-1})$. The property that each intermediate result is a product of powers of the g_i's now translates to the property that each term in the corresponding sequence of vectors is the sum of two previous terms. It is easily verified that a sequence of vectors $(\boldsymbol{s}_i)_i$ describing the computation of Equation 3 must satisfy the requirements:

- the first m terms s_{1-m} up to s_0 are the unit vectors (representing the g_i's);
- every term, except those unit vectors, is the sum of two preceding terms;
- the last term s_L is the target $\boldsymbol{x} = (x_0, x_1, \ldots, x_{m-1})$.

A sequence of vectors satisfying those requirements is called a *vector addition chain* of length L. Obviously, any algorithm to compute Equation 3 can be described in terms of vector addition chains, and conversely, any algorithm that produces a vector addition chain with the appropriate target can be used to compute this equation.

3.4 An Example

Let $m = 3$, and $\boldsymbol{x} = (30, 10, 24)$. With base 32, for example, this would correspond to exponent $30 + 10 \cdot 32 + 24 \cdot 1024 = 24926$. The following vector addition chain represents a computation of $g_0^{30} g_1^{10} g_2^{24}$.

(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 0, 1), (2, 0, 2), **(2, 1, 2), (3, 1, 2), (5, 2, 4), (6, 2, 5),** (12, 4, 10), **(15, 5, 12), (30, 10, 24).**

That is, with 9 multiplications, one can compute $g_0^{30} g_1^{10} g_2^{24}$ from g_0, g_1 and g_2. The computation of g_0^{30}, g_1^{10}, g_2^{24} by BSM and subsequently their product would take $7 + 4 + 5 + 2 = 18$ multiplications. The computations for an exponent 24926 by BSM would even require 21 multiplications.

4 Exponentiation using Vector Addition Chains

4.1 Introduction

For our purpose, one can choose any vector addition chain algorithm. See, for example, [11, 5, 2] or "Shamir's trick" [6] or further developments of the latter such as [13] (see also [12]) or indeed the method by Brickell et al. from [3]. In the given context, the best choice seems to be the one made below; this is discussed in the final paper. The vector addition chain algorithm given below is derived from [2, appendix to Chapter 4] and is discussed in [5] as well.

A vector addition chain algorithm can be described by the rule that selects the two terms of the vector addition chain used to form the next term. Since we aim at an algorithm with minimal memory requirements, we require that one of the terms used may be discarded from memory. That is, it is not needed later on in the sequence. This implies that the new term may *replace* one of the old terms in memory.

Such an algorithm repeatedly updates a fixed-size collection of powers of the g_i, until one of the elements of the collection equals the target $\prod_i g_i^{x_i}$. The proposed algorithm not only updates this collection held in memory, but also restates the target in terms of the currently stored elements. That is, given a collection $\{b_i\}$, the target is rewritten as a product $\prod_i b_i^{t_i}$. The elements b_i of the collection are called the (current) *bases*; the exponents t_i are called the (current) *target exponents*. The initialization is trivially performed as $b_i \leftarrow g_i$, and $t_i \leftarrow x_i$ for all i.

4.2 An Algorithm

The selection rule used for the update is based on the simple observation that $x^a y^b = (xy^{b \operatorname{div} a})^a y^{b \bmod a}$. That is, if we replace the target exponents (a, b) by $(a, b \bmod a)$, we may simultaneously replace the base x by $xy^{b \operatorname{div} a}$.

The idea of the algorithm is the repeated application of the above observation to the two largest target exponents. That is, the observation is applied with $b = t_{max}$ and $a = t_{next}$. Thus, the largest target value is decreased to a value smaller than the next largest value.

This is now described more formally. Let t_i be the current target exponents, let b_i be the corresponding current bases $(0 \leq i < m)$. Furthermore, denote by '*max*' and '*next*' the indices of the two largest t_i. That is, for all i, $t_i \leq t_{max}$, and for all $i \neq max$, $t_i \leq t_{next}$.

In each step, the algorithm performs the following. First, raise b_{max} to the power $q = t_{max} \operatorname{div} t_{next}$. This takes $\lfloor \log q \rfloor$ squarings and $\mathrm{wt}(q) - 1$ multiplications. If $q = 1$ this is a void step; if $q > 1$ this is done by for example BSM. (The non-bold terms in Section 3.4 correspond to the computation of such a qth power with $q > 1$. This computation requires an extra temporary variable.) Next, replace t_{max} by $t_{max} \bmod t_{next}$ and simultaneously replace b_{next} by $b_{next} \cdot b_{max}^q$.

The whole step requires $\lfloor \log_2 q \rfloor + \mathrm{wt}(q)$ multiplications. For the next step, the indices *max* and *next* are re-evaluated, so that they are the indices of the largest two current t_i again. (Note that the new value of *next* will be the old value of *max*.)

The algorithm ends as soon as $t_{next} = 0$. The final result then is $b_{max}^{t_{max}}$; if $t_{max} > 1$, we compute it by BSM, else we are done.

Algorithm V
for $i = 0$ to $m - 1$ do
    $b_i \leftarrow g_i$
    $t_i \leftarrow x_i$
endfor
while $(t_{next} > 0)$ do
    $q \leftarrow t_{max} \operatorname{div} t_{next}$
    $t_{max} \leftarrow t_{max} \bmod t_{next}$
    $b_{next} \leftarrow b_{max}^q \cdot b_{next}$
endwhile
return $b_{max}^{t_{max}}$

Fig. 1. The basic Vector Addition Chain Algorithm. The powers b_{max}^q and $b_{max}^{t_{max}}$ can be computed by any algorithm.

The pseudocode in Figure 1 gives the basic vector addition chain algorithm to compute $g^x = \prod_{i=0}^{m-1} g_i^{x_i}$. The power b_{max}^q and the return value $b_{max}^{t_{max}}$ can be

computed by any algorithm, for example BSM. This is the case considered in the sequel.

Note that for $m = 2$, this algorithm essentially performs Euclid's algorithm for the computation of the greatest common divisor of t_0 and t_1.

4.3 Variants

In the final paper several variants and small improvements will be described. Almost all of those use the intermediate results in the computation of b_{max}^q to reduce the new value t_{max} even further. Therefore, these variants are most useful for small m. On the other hand, especially for $m = 2$ this method increases the number of quotients larger than 1. For some results, see Table 1.

5 Complexity Analysis

5.1 Memory Requirements

The required permanent storage is m powers of g, compared to 1 (namely g itself) for BSM. During the computation we need space for the targets t_i (about $\log N$ bits), for the bases b_i (m powers of g) and one temporary variable used in BSM used to compute b_{max}^q. Since the exponent must be stored in any exponentiation algorithm, it is not recorded in the tables below.

So, for exponentiation in $\mathbb{Z}_p$ the storage for intermediate results amounts to $m + 1$ times $\log p$ bits. For example, for 512-bits p, and depending on the choice of m (between 2 and 32) this ranges from 192 to 2112 bytes, see Table 1.

Note that Algorithm B [3] requires only two variables for intermediate results (A and B), but one also needs quick read access to all precomputed g_i's. For Algorithm V one needs a variable (with *write* access) for each base, not just read access. Depending on the hardware platform, this may make a difference. This distinction is shown in the tables by prefixing the temporary memory by an asterisk. This signifies that the permanent memory is read only. For example, the entry *10882 in Table 2 means that only 2 variables require write access; the other 10880 (as found in the column permanent memory) require only read access.

5.2 Computational Complexity

The complexity is determined by the computation performed in the while-loop in Fig. 1. The computation of the indices *next* and *max* are negligible. So only the division of t_{max} by t_{next}, yielding both quotient and remainder, and the multiplications in $b_{next} \leftarrow b_{max}^q \cdot b_{next}$ have a significant influence.

We assume that the division is performed as in [7]; there is no reason to use a slower algorithm. The complexity of this (long) division is $|q| \times |t_{next}|$: it takes $O(|q|)$ steps of complexity $O(|t_{next}|)$ each. The corresponding multiplications have a far higher complexity: namely $O(|q| \times (\log N)^2)$.

length	algorithm		time		memory	
$\log N$	type	m	$\bar{L}$	$\tilde{L}$	perm	temp
512	V	64	128	145	64	66
	V	32	133	153	32	34
	V	16	150	177	16	18
	V	8	183	216	8	10
	V	4	253	277	4	6
	V'	2	402		2	5
	V	2	411	369	2	4

length	algorithm		time		memory	
$\log N$	type	m	$\bar{L}$	$\tilde{L}$	perm	temp
160	V	32	50	61	32	34
	V	16	52	60	16	18
	V	8	59	70	8	10
	V	4	80	87	4	6
	V'	2	126		2	5
	V	2	128	115	2	4

Table 1. The performance of the Vector Addition Chain Algorithm. Summary of the memory requirements, empirical average number of multiplications $\bar{L}$ and the estimate $\tilde{L}$ from Section 5.3 for some relevant values of N (size of the exponent) and m (memory). V' denotes a variant of algorithm V, see Section 4.3.

Furthermore, note that for larger values of m, we will have target exponents that fit into one or two machine words. This means that we can use a single or double-precision division, rather than a multi-precision division as considered above.

In any case, the division has negligible complexity compared to the multiplications. This is confirmed by empirical results: these are only a few percent worse than would be expected on the basis of the number of multiplications only. Of course this is highly dependent on the used hardware platform. Note that the *availability* of a multi-precision division may be a problem on some architectures.

Finally, note that relatively few multiplications in Algorithm V are squarings. Many squaring algorithms are faster than an ordinary multiplication, by making use of the fact that both muliplicands are equal (see, e.g., [7]). This too may result in performance slightly worse than expected on the basis of the number of multiplications only.

We conclude that the computational complexity is determined by the number of multiplications. That is, by the length of the vector addition chain. This is worked out below.

5.3 Length of a Vector Addition Chain

Denote the maximal length of a vector addition chain as constructed by Algorithm V for exponents $x \leq N$ and for storage m by $L(N, m)$. The average value will be denoted $\bar{L}(N, m)$.

Below, we give heuristics for estimates for $L(N, m)$ and $\bar{L}(N, m)$. It turns out to be very hard to find a closed formula for either of these, especially for smaller values of m, as in that case it is not possible to make the simplifying assumption that $q = 1$. Therefore we give only an estimate for the complexity for larger values of m, and the worst case for $m = 2$. Both results can be found

length	algorithm	memory		time
log N	reference	perm	temp	mult.'s
512	B [3]	10880	*10882	64
	B [3]	109	*111	128
	V	64	65	128
	V	16	17	150
	Shamir [6]	15	*16	239
	[13]	11	*12	366
	V'	2	4	411
	Shamir [6]	3	*4	447
	m-ary [7]	1	*17	611
	BSM	1	*2	767

length	algorithm	memory		time
log N	reference	perm	temp	mult.'s
160	B [3]	2751	*2753	21
	B [3]	45	*47	50
	V	32	33	50
	V	8	15	59
	Shamir [6]	15	*16	74
	[13]	11	*12	114
	V'	2	4	128
	Shamir [6]	3	*4	139
	m-ary [7]	1	*9	197
	BSM	1	*2	239

Table 2. Comparison of some exponentiation algorithms. Summary of the memory requirements, (empirical) average number of multiplications for some relevant values of N (size of the exponent). m-ary is its m-ary generalization using a "sliding window". [13] is a combination of this sliding window idea and Shamir's trick.

partly in [5] as well. More details will be given in the final paper. A summary of empirical results is given in Table 1.

In practice, it turns out that $q = 1$ almost always, unless $\log b \gg m$. This is not surprising, as for Euclid's Algorithm this already holds about 41% of the time [7, pp. 352–353]. By choosing larger values of m, the t_i will be 'closer together' than for $m = 2$ as in Euclid's Algorithm. This implies that $q = 1$ will occur significantly more often than in Euclid's Algorithm.

For the reasoning below, the possibility that $q > 1$ is ignored. This seems reasonable for m not too small; empirical results show that for $N = 2^{512}$ and $m = 32$ values of $q > 1$ are indeed rare. It can be expected that calculations based on this assumption will provide a reasonable (and conservative) estimate $\tilde{L}(N, m)$ for the actual value of the expected length $\bar{L}(N, m)$. This is confirmed by empirical results; even for smaller values of m.

If $q = 1$ always, we have that each term $\boldsymbol{s}_i$ in the vector addition chain is the sum of the previous term $\boldsymbol{s}_{i-1}$ and some other term in the set $\{\boldsymbol{s}_{i-m}, \ldots, \boldsymbol{s}_{i-2}\}$. It is not hard to see that $\boldsymbol{s}_i$ and all subsequent terms are minimal (using the usual norm) if $\boldsymbol{s}_i = \boldsymbol{s}_{i-1} + \boldsymbol{s}_{i-m}$. This observation yields $\tilde{L}(N, m)$.

So the minimal sequence satisfies $\boldsymbol{s}_i = \boldsymbol{s}_{i-1} + \boldsymbol{s}_{i-m}$ for all $i > 0$.[2] Moreover, all entries s_{ij} of the vectors $\boldsymbol{s}_i$ satisfy the same recurrence, all with a different initial segment: $s_{ij} = s_{i-1,j} + s_{i-m,j}$ for all $i > 0$ and for $1 \leq j \leq m$, and the initial m terms of the sequences $(s_{ij})_i$ represent the m unit vectors of length m.

It is readily verified that these m sequences of entries of the vectors $\boldsymbol{s}_i$ just are shifts of each other, namely $s_{im} = s_{i+1,1} = s_{i+2,2} = \cdots = s_{i+m-1,m-1}$. Therefore, we have $\tilde{L}(N, m) = max_i\{s_{im} < N^{1/m}\}$. Note that s_{im} can be approximated

[2] W.l.o.g. we reorder the m initial entries to satisfy this requirement also for $1 \leq i < m$.

by ρ_m^i where ρ_m is the largest real solution of $x^m = x^{m-1} + 1$. It follows that $\tilde{L}(N,m) < \log b / \log \rho_m$. (An approximation for ρ_m is $\rho_m = 1 + \ln m/m$.) For more details, see the final paper.

For some values of $\tilde{L}(N,m)$, see Table 1. Indeed for $m \geq 4$, $\tilde{L}(N,m)$ proves to be a reasonable estimate for $\bar{L}(N,m)$.

Note that the bound $\tilde{L}(N,m)$ is constructive: there is a vector addition chain of length $\tilde{L}(N,m)$. That is, $L(N,m) \geq \tilde{L}(N,m)$.

(This shows that Conjecture 6 from [5] is not completely correct. The bound conjectured there translates to $L(N,m) < (1 + m/\ln m) \log b$;[3] for $N = 2^{512}$ and $m = 64$ this yields $L(2^{512}, 64) < 139$, while $\tilde{L}(2^{512}, 16) = 145$.)

Finally, we remark that Theorem 7 in [5] translates to $L(N,2) = \log N$. Indeed this bound is reached for the all-one exponent.

6 Conclusions

An method for faster exponentiation using a limited amount of precomputed powers is proposed. This method is based on two ideas. Firstly, the idea from [3] of splitting the exponentiation into the product of a number of exponentiations with smaller exponents. Secondly, the use of the technique of vector addition chains to compute this product of powers.

Furthermore, a specific vector addition chain algorithm [5, 2] has been proposed and analyzed. Depending on the amount of precomputations and memory, this provides an algorithm that is about two to six times as fast as binary square and multiply. It is only slightly slower than the method from [3] using far less memory.

More specifically, a 512-bit exponentiation can be performed in 128 up to 402 multiplications, using 32 down to 2 precomputed powers. The faster algorithms from [3] take 128 down to 64 multiplications, but require 109 up to 10880 precomputed powers. Binary square and multiply and a variant of the m-ary method take 767 respectively about 611 multiplications without precomputations.

The fact that precomputations have to be done limits the applicability to those cryptographic systems where the same base is used very often. This holds for most discrete log based systems.

There are several interesting open problems, such as other vector addition chain algorithms requiring less temporary variables, with different time/memory trade-off, etc.

Acknowledgements

I would like to thank Jean-Paul Boly, Arjen Lenstra, Berry Schoenmakers and Hans van Tilburg for their useful comments.

[3] It is derived along the same lines as above; in the conjecture itself $\ln m$ is misprinted as $\log m$.

References

1. J. Bos and M. Coster, "Addition chain heuristics", *Advances in Cryptology – Proceedings of Crypto'89* (G. Brassard, ed.), Lecture Notes in Computer Science, vol. 435, Springer-Verlag, 1990, pp. 400–407.
2. J. N. E. Bos, *Practical Privacy*, Ph.D. thesis, Technical University of Eindhoven, March 1992.
3. E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson, "Fast exponentiation with precomputation (extended abstract)", *Advances in Cryptology – Proceedings of Eurocrypt'92* (R. A. Rueppel, ed.), Lecture Notes in Computer Science, vol. 658, Springer-Verlag, 1993, pp. 200–207.
4. E. F. Brickell and K. S. McCurley, "An interactive identification scheme based on discrete logarithms and factoring", *Journal of Cryptology* **5** (1992), no. 1, pp. 29–39.
5. M. Coster, *Some Algorithms on Addition Chains and their Complexity*, Tech. Report CS-R9024, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.
6. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory* **IT-31** (1985), no. 4, pp. 469–472.
7. D. E. Knuth, *Seminumerical Algorithms*, second ed., The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, Massachusetts, 1981.
8. National Institute of Technology and Standards, *Specifications for the Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication XX, US. Department of Commerce, February 1 1993.
9. R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM* **21** (1978), no. 2, pp. 120–126.
10. C. P. Schnorr, "Efficient signature generation by smart cards", *Journal of Cryptology* **4** (1991), no. 3, pp. 161–174.
11. A. Yao, "On the evaluation of powers", *SIAM Journal on Computing* **5** (1976), no. 1, pp. 100–103.
12. S.-M. Yen and C.-S. Laih, "The fast cascade exponentiation algorithm and its application to cryptography", *Abstracts of Auscrypt 92*, 1992, pp. 10–20 – 10–25.
13. S.-M. Yen, C.-S. Laih, and A. K. Lenstra, "A note on multi-exponentiation", *IEE Proceedings, Computers and Digital Techniques* **141** (1994), no. 5, to appear.