MIST:
An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis

Colin D. Walter

Comodo Research Laboratory
10 Hey Street, Bradford, BD7 1DQ, UK
www.comodo.net
and
Computation Department, UMIST
PO Box 88, Sackville Street, Manchester, M60 1QD, UK
www.co.umist.ac.uk

Abstract. The MIST algorithm generates randomly different addition chains for performing a particular exponentiation. This means that power attacks which require averaging over a number of exponentiation power traces becomes impossible. Moreover, attacks which are based on recognising repeated use of the same pre-computed multipliers during an individual exponentiation are also infeasible. The algorithm is particularly well suited to cryptographic functions which depend on exponentiation and which are implemented in embedded systems such as smart cards. It is more efficient than the normal square-and-multiply algorithm and uses less memory than 4-ary exponentiation.

Key words: Mist exponentiation algorithm, division chains, addition chains, power analysis, DPA, blinding, smart card.

1 Introduction

Recent progress in side channel attacks [4], [5] on embedded cryptographic systems has exposed the need for new algorithms which can be implemented in more secure ways than those currently in use. This is particularly true for exponentiation, which is a major process in many crypto-systems such as RSA and Diffie-Hellman. Timing attacks on modular multiplication can usually be avoided easily by removing data-dependent conditional statements [12]. But, with timing variations removed, power attacks on exponentiation become easier. Initial power attacks required averaging over a number of exponentiations [5]. Although the necessary alignment of power traces can be made more difficult by the insertion of obfuscating, random, non-data-dependent operations, the data transfers between operations usually reveal the commencement of every long integer operation very clearly. Fortunately, such attacks can be defeated by modifying the exponent e to $e{+}rg$ where r is a random number and g is the order of the (multiplicative) group in which the exponentiation is performed [4]. This results in a different exponentiation being performed every time.

However, the author showed recently [11] that there were strong theoretical grounds for believing that, given the right monitoring equipment [2], it would be possible to break the normal m-ary exponentiation method [3] and related sliding windows techniques using a single exponentiation. This method relies on being able to recognise the same multipliers being reused over and over, namely the pre-computed powers of the initial text, and requires no knowledge of the modulus, input text or output text. It renders useless the choice of $e{+}rg$ as a counter-measure, even for the case of $m = 2$, namely the standard square-and-multiply algorithm.

In an embedded system, the re-use of the same multipliers is useful because it reduces data movement. However, if such re-use is dangerous, other methods must be employed. Performing square-and-multiply in the opposite direction, namely consuming exponent bits from least to most significant, is the obvious starting point. With a random variation in the exponent as a counter-measure, this seems to defeat the attacks mentioned so far. Unfortunately, long integer squares and multiplications are among the easiest operations to distinguish in integer RSA [7]. Even the movement of data is different for the two operations. Thus an exponentiation algorithm is required which does not reveal its exponent through knowledge of the sequence of squares and multiplies.

A novel exponentiation algorithm is presented here which avoids all of the above-mentioned pitfalls. It can also be combined with most other counter-measures, such as using $e{+}rg$ instead of e. It relies on the generation of random addition chains [3] which determine the operations to be performed, and is based on previous work by the author [9] for finding efficient exponentiation schemes. This earlier work required extensive computation to establish near optimal addition chains, and so it is by no means obvious that the method can be used on-the-fly without an impractical overhead. Clearly, such computation can be performed in the factory prior to the production of an embedded crypto-system, and then each individual item can be issued with a different embedded addition chain which, although efficient, must be re-used on each exponentiation and is incompatible with the $e{+}rg$ counter-measure [1].

Here we develop that algorithm to the point where it is possible to efficiently generate fresh, efficient addition chains for any exponent and every exponentiation. The main aim here, besides presentation of the algorithm itself, is to establish that it has a time complexity better than square-and-multiply, and to note that, as far as space is concerned, only three long integer registers are required for executing the addition chain operations.

2 The Mist Algorithm

For notation, let us assume that M^E has to be computed. D will always represent a "divisor" in the sense of [9], and R a residue modulo D. A key ingredient of this algorithm is that we can efficiently compute both A^R and A^D from A using a single addition chain. A set of divisors is chosen in advance, and an associated table of these addition chains is stored in memory. Several variables are used in the code below. There are three which contain powers of M, namely $StartM$, $TempM$ and $ResultM$. Rather than destroy the initial value of E, we also use a variable called $RemE$ which represents the power to which $StartM$ still has to be raised before the exponentiation is complete. When the divisor set consists of the single divisor 2, the algorithm simplifies to the following right-to-left binary exponentiation algorithm, that is, to the standard version of square-and-multiply in which the least significant exponent bit is processed first:

Right-to-Left Square-and-Multiply Exponentiation Algorithm

```
{ Pre-condition: E ≥ 0 }
RemE    := E  ;
StartM  := M  ;
ResultM := 1  ;
While RemE > 0 do
Begin
   If (RemE mod 2) = 1 then
      ResultM := StartM×ResultM ;
   StartM  := StartM^2 ;
   RemE    := RemE div 2 ;
   { Loop invariant: M^E = StartM^RemE × ResultM }
End ;
{ Post-condition: ResultM = M^E }
```

Thus $StartM$ contains the initial value M raised to a power of 2, and it is the starting point for each loop iteration. $ResultM$ is the partial product which contains M raised to the power of the processed suffix of E and ends up containing the required output.

Without the *essential* random feature made possible by a larger divisor set, this is an insecure special case of Mist. In fact, we are assuming that the attacker can distinguish between squares and multiplies. Hence he can read off the bits of the exponent in the above from the sequence of long integer instructions which he deduces. However, unlike the left-to-right version of square-and-multiply, both arguments in the conditional product are now changed for every multiplication. This, at least, makes an attack of the type [11] impractical.

Next is the generalisation of this, which is named "Mist" because of its use in obscuring the exponent from side channel attacks. For convenience and comparison, the processing of the exponent E is presented as being performed within the main loop. In practice, it *must* be scheduled differently. As will be pointed out below, the illustrated processing order is insecure from the point of differential power analysis because it can reveal the random choice of the divisor D, which *must* remain secret. In fact, the choices of divisors and computations with the exponent must be done initially in a secure way, or at least interleaved with the long integer operations in a more considered way. To be even more precise, the complete schedule of long integer operations, i.e. the addition chain which determines the exponentiation scheme, should be computed with great care in order to hide the choice of divisors.

THE MIST EXPONENTIATION ALGORITHM
{ before proper re-scheduling of addition chain choices }

```
{ Pre-condition: E ≥ 0 }
RemE    := E  ;
StartM  := M  ;
ResultM := 1  ;
While RemE > 0 do
Begin
   Choose a random "divisor" D ;
   R := RemE mod D ;
   If R ≠ 0 then
      ResultM := StartM^R × ResultM ;
   StartM  := StartM^D ;
   RemE    := RemE div D ;
   { Loop invariant: M^E = StartM^RemE × ResultM }
End ;
{ Post-condition: ResultM = M^E }
```

Observe that there are no powers of M which are repeatedly re-used during the exponentiation, so that the attack described in [11] on a single exponentiation is inapplicable in its current form. Also, the random choice of divisors achieves different exponentiation schemes on successive runs and so makes impossible the usual averaging process required for differential power analysis [5].

For the proof of correctness, it is immediate that the stated invariant

$$M^E \quad = \quad StartM^{RemE} \times ResultM$$

holds at the start of the first iteration of the loop. Because initial and final values of $RemE$ for one loop iteration satisfy

$$RemE_{Initial} \quad = \quad D \times RemE_{Final} + R$$

it is easy to check that if the invariant holds at the start of an iteration then it holds again at the end of the iteration. Consequently, the invariant holds at the end of every iteration. In particular, at the end of the last iteration, we have $RemE = 0$ and hence, by simplifying the invariant, $ResultM = M^E$. So $ResultM$ yields the required result on termination. Needless to say, termination is guaranteed because only divisors greater than 1 are allowed, and so RemE decreases on every iteration.

The choices of divisor set and associated addition chains for each residue R are made with security and efficiency in mind. In particular, for efficiency the choice of addition chain for raising to the power D always includes R so that the computation of $StartM^D$ provides $StartM^R$ *en route* at little or no extra cost. Thus, these two power computations are not performed independently, as might be implied by the code. They are to be implemented so that $StartM^D$ uses all the work done already to compute $StartM^R$. So, in the case of RSA, the main cost of a loop iteration is only the cost of computing $StartM^D$ plus the conditional extra multiplication involving $ResultM$. In terms of space, an extra long integer variable $TempM$ is required to enable the addition chain operations to be carried out. Its value is always a power of the value of $StartM$ at the beginning of the main loop above, but its value does not need to persist between successive iterations of the loop. Of course, on the final iteration, once $StartM^R$ has been computed there is no need to continue to compute the rest of $StartM^D$. Likewise, the initial multiplication of $ResultM$ by 1 can be easily omitted.

For security reasons, some care is necessary in the initial choice of divisor set and addition chains, and in how and when the divisors are chosen for each exponentiation. The selection of the divisor and associated addition chain instructions can be performed by the CPU on-the-fly while the exponentiation is performed in parallel by the co-processor, or it can be done in advance when there is no co-processor. At any rate, these computations must be scheduled so as not to reveal the end points of each iteration of the main loop. Otherwise, the number and type of long integer operations during the loop iteration may leak the values of D and R, enabling E to be reconstructed.

A typical safe set of divisors is $\{2,3,5\}$. If the exponent is represented using a radix which is a multiple of every divisor (say 240 in this case if an 8-bit processor is being used) then the divisions of the exponent by D become trivial. So, over and above the operations for executing the addition chain, the cost of the algorithm is little more than that for calling the random number generator to select each D.

3 The Addition Sub-Chains and Space Requirements

When divisor choices are made during pre-processing, the sequence of operations to perform the exponentiation is stored as an *addition chain* [3]. For a safe implementation, the pattern of squares and multiplies in this chain must not reveal too much about the divisors and residues. However, for the complexity considerations of this and the following section, we only need to look at the subchain associated with a single divisor. Such subchains can be concatenated to yield an addition chain for the whole exponentiation, if desired.

Let us choose the divisor set to be $\{2,3,5\}$. The full list of minimal addition subchains can be represented as follows:

1+1=2	for divisor 2 with any residue R
1+1=2, 1+2=3	for divisor 3 with any residue R
1+1=2, 1+2=3, 2+3=5	for divisor 5 with any residue except 4
1+1=2, 2+2=4, 1+4=5	for divisor 5 with any residue except 3

The third case corresponds to using an initial M^1 to compute M^2 with one multiplication, then M^3 with another multiplication, and finally M^5 with a third multiplication. The first three addition chains provide M^R when R is 0, 1, 2 or 3: for $0 < R < D$ the chain already contains the value of R, while the case $R = 0$ requires no multiplication and so 0 does not need to appear. The last addition chain can be used when $R = 4$. *Minimal* here means that any other addition chains which give a power equal to the divisor are longer. The subchains above are minimal. To achieve the fastest exponentiation, we will not include longer chains. However, there may be extra cryptographic strength in extending the choice in such a way.

There is no instruction which updates the value of $ResultM$ in the above addition subchains, but it can be represented explicitly using the following notation. Suppose we number the registers 1 for $StartM$, 2 for $TempM$ and 3 for $ResultM$. Then the subchains can be stored as sequences of triples $ijk \in \{1, 2, 3\}^3$, where ijk means read the contents from registers i and j, multiply them together, and write the product into register k. In particular, $ResultM$ will always be updated using a triple of the form $i33$ and 3 will not appear in triples otherwise. Now, adding in the instruction for updating $ResultM$ yields the following as a possible list of subchains, with one representative for each divisor/residue pair $[D, R]$. Such a table requires only a few bytes of storage.

$[2,0]$	(111)
$[2,1]$	(112, 133)
$[3,0]$	(112, 121)
$[3,1]$	(112, 133, 121)
$[3,2]$	(112, 233, 121)
$[5,0]$	(112, 121, 121)
$[5,1]$	(112, 133, 121, 121)
$[5,2]$	(112, 233, 121, 121)
$[5,3]$	(112, 121, 133, 121)
$[5,4]$	(112, 222, 233, 121)

Table 3.1. A Choice for the Divisor Sub-Chains.

Many other choices are possible. In particular, we might prefer to preserve the location of $StartM$ to be in register 1 from one divisor to the next. This could be achieved by choosing (133, 111) instead of (112, 133) for [2,1]. Then subchains of triples ijk could be concatenated without modification to provide the complete exponentiation scheme. However, for the given subchain, the new value for $StartM$ is in register 2 instead of register 1. Rather than waste time copying, the computation should continue with 2 as the address for $StartM$, and the next subchain then has to be updated with the register addresses 1 and 2 interchanged. However, as noted in the penultimate section, this apparently less obvious choice for [2,1] makes the implementation more secure against differential power analysis.

Following this last remark, it is clear that we could provide an additional source of randomness by writing the product into any register whose current contents are no longer required [6]. Thus the purpose of each register can be changed. If we include every possible minimal addition subchain with such variations, then we obtain 2, 6, 2, 6, 4, 4, 16, 8, 4 and 4 possibilities respectively for the 10 divisor/residue cases. The storage is still only a few bytes, but it provides an extra source of randomness which may provide added security.

Thus the space order required for MIST is low. In the next section we turn to the time complexity, which we measure in terms of the number of multiplications in the exponentiation scheme.

4 The Time Complexity

The usual square-and-multiply algorithm uses $1.5\times\lfloor\log_2 E\rfloor$ multiplicative operations (including squarings) on average and a maximum of at most $2\times\lfloor\log_2 E\rfloor$. In this section almost the same upper bound will be established for MIST, and an improved average. In practice the variance is very small. Consequently, if a pre-computed scheme involved more than $1.5\times\lfloor\log_2 E\rfloor$ multiplications, say, it could be abandoned and an alternative scheme computed.

> **Theorem 4.1** Assume minimal subchains are provided as above for the divisor set $\{2,3,5\}$ and $E > 0$, and unnecessary initial and final multiplications have been omitted. Then the maximum number of operations in an addition chain for E is at most $2\log_2 E$ with equality possible only for $E = 1$.

Proof. We use a proof by descent: assuming E is a smallest counter-example, we will construct a smaller counter-example E' for which the theorem fails. Such a contradiction will prove the theorem.

First assume that at least two divisors are required to reduce the minimal counter-example E to 0. We will consider the three choices for the first divisor of E in turn.

Suppose the first divisor is 3. Then $E' = E$ div 3 is the next value of $RemE$ after E and $3E' \le E$. Let m be the number of multiplications for E, and k the number for this first divisor 3. Then $k = 2$ or 3, so that $k < 2(\log_2 3) \approx 2\times 1.585$. By assumption, $m \ge 2\log_2 E$. So $m \ge 2\log_2(3E') = 2\log_2(E') + 2(\log_2 3)$ gives $m-k > m-2(\log_2 3) \ge 2\log_2(E')$. But $m-k$ is the number of multiplications for E'. Hence we obtain a smaller E for which the theorem does not hold. This is a contradiction unless $E' = 0$. However, that is impossible as there is another divisor in the exponentiation scheme.

Similarly, suppose the first divisor is 5 and E' is the next value for $RemE$. Then the associated subchain requires $k = 3$ or 4 multiplications, so that $k < 2(\log_2 5) \approx 2\times 2.322$. Thus, a similar argument yields a contradiction again.

Lastly, suppose the first divisor is 2 and E' is defined again as the next value for $RemE$. If E is odd, then $2E' < E$ and the corresponding division requires $k = 2 = 2(\log_2 2)$ multiplications. By assumption, $m \geq 2\log_2 E > 2{+}2\log_2 E'$. So the number of multiplications used by E' is $m{-}2 > 2\log_2(E')$ and E' is also a failing instance unless $E' = 0$, which is a contradiction as there is still another divisor to come. On the other hand, if E is even, then the corresponding division requires $k = 1$ multiplication. By assumption, $m \geq 2\log_2 E = 2{+}2\log_2 E'$. So the number of multiplications used by E' is $m{-}1 > 2\log_2(E')$ and E' is also a failing instance unless $E' \leq 0$, which, as before, is a contradiction.

It remains to consider the cases of a single divisor reducing E to 0. For all of these, the chosen divisor satisfies $D > E$ since E div $D = 0$ and so $R = E$. The only multiplications are those which construct M^E. For $E = 1$, 2, 3, 4 the number of multiplications are 0, 1, 2 and 2 respectively, all of which satisfy the theorem. Hence, by the method of descent, there are no failing instances, and the theorem holds. $\square$

It may be worth noting that removal of a divisor 3 or 5 at any point from the addition chain for E (other than the last) yields a chain for which the bound in the theorem is tighter. The same is true when the divisor is 2 and the residue 0. Thus, the tightest bounds (i.e. the most multiplications for given E) will occur when only divisor 2 is chosen and the residue is always 1. Then $E = 2^n{-}1$ for some n and E requires $n{-}1$ subchains of 2 operations to reduce $RemE$ to $2^1{-}1$, leading to $2n{-}2$ operations in all. So $2{\times}\lfloor\log_2 E\rfloor$ would normally hold as an upper bound. However, it requires more care to establish the exceptions for this slightly better bound.

5 A Weighting for the Choice of Divisor

Suppose k is the number of multiplications required for the divisor/residue pair (D, R). The result of picking divisor D is that $RemE$ is reduced by a factor which is very close to D, especially when $RemE$ is large. So, we would expect the total number of multiplications to be close to $k\log_D E$ if (D, R) occurred for every divisor. Consequently, the cost of using (D, R) is proportional to $k/\log D$. We might use this to bias the choice of divisor in an attempt to decrease the number of multiplications. However, this is not the best measure because the effect of larger divisors is longer lasting than that of smaller divisors. A small divisor with many multiplications, which is expected to be followed by divisors with an average number of multiplications, may be a better choice than picking a larger divisor with a better, but poorer than average, ratio $k/\log D$ (*see* [9]).

Suppose α is the average number of multiplications required to reduce E by a factor of 2. Then reducing E by a factor F will require $\alpha\log_2 F$ multiplications, on average. For the usual square-and-multiply algorithm $\alpha = 1.5$, for 4-ary exponentiation $\alpha = 1.375$, and for the sliding window version of 4-ary exponentiation $\alpha = 1.333$... Here we can manage just a little better than $\alpha = 1.4$

with appropriate choices for the divisors[1]. Comparing (3,0) with (5,0) we find 3 multiplications will reduce E by a factor of 5 when (5,0) is chosen, whereas the same factor of 5 takes $2+\alpha\log_2(5/3)$ multiplications on average if (3,0) is chosen instead. Equating these costs, $3 = 2+\alpha\log_2(5/3)$, provides the cross-over point $\alpha' = 1/\log_2(5/3) = 1.357$ at which one becomes a better choice than the other. Since $\alpha' < \alpha$, for speed we should choose (5,0) in preference to (3,0) although the ratios $k/\log D$ suggested the opposite preference. In a similar way, for the expected range of α, (3,0) is preferred to $(5,R)$ for $R \neq 0$; $(5,R)$ to $(3,S)$ for R, $S \neq 0$; $(3,R)$ to (2,1) for $R \neq 0$; and $(5,R)$ to (2,1) for $R \neq 0$. This yields the following order of desirability for the pairs (D,R):

$$(2,0) < (5,0) < (3,0) < (5,1) = (5,2) = (5,3) = (5,4) < (3,1) = (3,2) < (2,1)$$

So (2,0) will lead to the shortest chains, and (2,1) to the longest.

Once more we caution that this is only a better approximation than the previous one. Successive divisors are not independent, so that the above argument is not quite accurate. This is clear from an example. We consider the extreme case where divisibility by a divisor always leads to the choice of that divisor. If 5 is chosen as the divisor and its residue is 0 then its residue mod 30 was 5 or 25. Residue 5 mod 30 leads to the next residue mod 30 being 1, 7, 13, 19 or 25, and residue 25 mod 30 leads to it being 5, 11, 17, 23 or 29. Hence 5 becomes the most likely choice for the next divisor as divisibility by 2 or 3 cannot to occur. Indeed, the dependence is inherited by the next residue beyond this as well, since these residues favour 5 as the next divisor and so 5 or 25 as the following residue.

Choosing one of the three divisors with equal probability leads to an inefficient process. So we make a choice which is biased towards the better pairs (D,R). For non-trivial reasons outlined in the penultimate section, it is less safe cryptographically to make a deterministic choice of divisor even if the exponent is modified randomly on every occasion. So 2 is not chosen whenever its residue is 0. Instead, we might use code such as the following for choosing the divisor non-deterministically. (*Random* returns a fresh random real in the range [0,1].)

```
D := 0 ;
If Random(x) < 7/8 then
    If 0 = RemE mod 2 then D := 2 else
    If 0 = RemE mod 5 then D := 5 else
    If 0 = RemE mod 3 then D := 3 ;
If D = 0 then
Begin
    p := Random(x) ;
    If p < 6/8 then D := 2 else
    If p < 7/8 then D := 3 else
                    D := 5 ;
End ;
```

[1] Choose $D = 2$ if the residue is 0, else $D = 3$ if the residue is 0, else $D = 5$ if the residue is 0, else $D = 2$. However, the probabilities of these choices need reducing to less than 1 in order to yield randomly different exponentiation schemes.

The parameters, such as 6/8 and 7/8, are free for the implementor to choose, and might even be adjusted dynamically. This is the code which will be used for calculating the various parameters of interest in the rest of this article, such as α, which turns out to be 1.4205 here. So a good implementation of MIST can be expected to have a time efficiency midway between the square-and-multiply and 4-ary exponentiation methods.

6 A Markov Process

As the successive residues of $RemE$ mod 30 are not independent, we must study them as forming a Markov process. By forming the probability matrix of output residues against input residues and iterating a number of times, it is possible to obtain the limiting relative frequencies of the residues. These are given in Table 6.1, from which it is apparent that the residues do not occur with equal frequency, and so, as one would expect, the probabilities of divisors and divisor/residue pairs are also not what we might initially expect from the code above.

0	0.02914032	1	0.03448691	2	0.01660770
3	0.05345146	4	0.01884630	5	0.03655590
6	0.02920902	7	0.04100675	8	0.02436897
9	0.04985984	10	0.02011853	11	0.04919923
12	0.02014301	13	0.04214864	14	0.03407472
15	0.03681055	16	0.01526795	17	0.03368564
18	0.03160194	19	0.03600811	20	0.03187408
21	0.04915191	22	0.02166433	23	0.04323007
24	0.03197409	25	0.02706484	26	0.03224476
27	0.04020936	28	0.02102305	29	0.04897205

Table 6.1. The limit probabilities of residues mod 30.

From this table the probability $p_{D,R}$ of each divisor/residue pair (D, R) can be obtained as well as the probability $p_{\sim 0}$ of selecting a divisor with a non-zero residue and the probability p_D of each divisor D:

(D, R)	0	1	2	3	4
2	0.35012341	0.27101192	-	-	-
3	0.18867464	0.02086851	0.02419582	-	-
5	0.09792592	0.01202820	0.01060216	0.01227849	0.01229092

Table 6.2. The limit probabilities $p_{D,R}$ of the divisor/residue pairs (D, R)

$$p_2 = 0.62113534$$
$$p_3 = 0.23373897$$
$$p_5 = 0.14512570$$

Table 6.3. The limit probability p_D for each divisor D.

7 Average Properties of the Addition Chain

The probabilities in the above tables are fairly accurate after only a small number of divisors have been applied. So the following results hold very closely for any exponents related to integer RSA decryption.

> **Theorem 7.1** The average subchain length is just under 1.89 operations per divisor as $E \to \infty$.

Proof. With probability $p_{2,0} = 0.35012341$ the subchain has length 1, with probability $p_{3,0}+p_{2,1} = 0.45968656$ the subchain has length 2, with probability $p_{5,0}+p_{3,1}+p_{3,2} = 0.14299025$ the subchain has length 3, and with probability $p_{5,1}+p_{5,2}+p_{5,3}+p_{5,4} = 0.04719977$ the subchain has length 4. The average length of a subchain is therefore $1(p_{2,0}) + 2(p_{3,0}+p_{2,1}) + 3(p_{5,0}+p_{3,1}+p_{3,2}) + 4(p_{5,1}+p_{5,2}+p_{5,3}+p_{5,4}) = 1.88726638$. □

> **Theorem 7.2** The average number of subchains in the addition chain for E is approximately $0.75\times \log_2 E$ as $E \to \infty$.

Proof. Using the divisor probabilities listed in Table 6.3, the average decrease in size of $RemE$ due to a single subchain is by the factor $2^{p_2}3^{p_3}5^{p_5} = 2\uparrow\{0.6211353\times \log_2 2 + 0.233739\times \log_2 3 + 0.1451257\times \log_2 5\} \approx 2^{1.328574}$. Hence the average number of subchains is about $\log_{2\uparrow(1.328574)} E = 0.75268656 \times \log_2 E$. □

> **Theorem 7.3** The average number of operations in the addition chain for E is approximately $1.42\times \log_2 E$ as $E \to \infty$. This is about 3% above the $1.375 \times \log_2 E$ of the 4-ary method, and noticeably less than the $1.5\times \log_2 E$ of the square-and-multiply method.

Proof. Using the results of the last two theorems, the average number of operations for the whole addition chain is approximately

$$1.88726638 \times 0.75268656 \times \log_2 E \quad \approx \quad 1.42052005\times \log_2 E$$

For small exponents E the approximations are slightly more inaccurate because modular division by the divisor produces a result which differs more from rational division. However, each subchain reduces the exponent by *at least* the divisor. Hence exact calculations here should yield an *upper* bound on the average.

Lastly, we note that the number of multiplications in the exponentiation schemes for a given exponent E does vary between different executions. The variance is usually small but depends on the method of picking divisors. Typically, the choice of divisors is restricted, as here, in order to improve efficiency, and this reduces the variance. Indeed, in the limit, deterministic choices lead to zero variance for fixed E.

8 Data Leakage

Because of the difficulty of successfully hiding all the differences between squaring and non-squaring multiplications, the MIST algorithm has been created in order to make it impossible to deduce the secret exponent E from leaked knowledge of the sequence of square or multiply instructions which perform an exponentiation. A detailed exposition of how easy it is to reconstruct the exponent from this and other related data is beyond the scope of this article. However, we will outline some of the issues. Fuller details will be published elsewhere.

Standard power analysis attacks [5] average traces over a number of exponentiations in order to determine information such as which operations are multiplications and which are squares. This requires the same exponentiation scheme to be used every time. However, assuming the choice of divisors does vary from one exponentiation to the next (or at least changes with sufficient frequency), such an averaging process cannot be carried out here.

Differences between squaring and multiplying are so great that one should assume that they can be correctly distinguished for a single exponentiation. In the case of the standard square-and-multiply methods, such knowledge can be translated immediately into the bit sequence for E. But here the equivalent process requires first parsing the sequence of squares and multiplies into divisor subchains in order to deduce each choice of divisor/residue pair. Then E is reconstructed by working backwards from the final value of 0. However, the subsequences are identical for various pairs, such as for $[2,1]$ and $[3,0]$, and for $[3,1]$, $[3,2]$ and $[5,0]$. It is easy enough to verify that these ambiguities put the number of possibilities for E far outside the limits of feasible computation, making MIST secure against such an attack. This was one reason for choosing the less obvious subchain for $[2,1]$ given in Table 3.1.

The m-ary and sliding windows methods of exponentiation can be defeated for a single exponentiation if the power attack described in [11] can be applied. That attack depends on identifying the re-use of multiplicands. In particular, whenever a digit i is encountered in the exponent, a multiplication is performed using the pre-computed value M^i. Then careful averaging of the power trace subsections enables multiplications which involve the same power M^i to be identified, and this leads to recovery of the exponent E.

The same attack has an analogue here. Every multiplication here involves a new power of M. More precisely, $StartM$ and $ResultM$ represent higher powers of M every time they are updated. So the possibilities of two multiplications sharing the same multiplicand are mostly limited to local considerations. If operations which share a common argument can be identified, it usually becomes possible to determine uniquely the subchains which correspond to each divisor. Of course, a careless implementation of MIST might do this anyway: for example, if the exponentiation is halted in every iteration of the main loop while determination of the next divisor is made. However, although knowledge of argument sharing now distinguishes $[3,1]$, $[3,2]$ and $[5,0]$, it does not distinguish $[2,1]$ from $[3,0]$. Thus it does not necessarily determine the divisor which was chosen. A more careful estimate of the number of exponents E which satisfy all the operand sharing requirements shows that such an attack is still infeasible.

Finally, it would be nice to make deterministic choices of the divisor. In particular, one would like to pick $[2,0]$ when possible because of its higher efficiency. However, this is generally a bad idea, and was deliberately avoided in the code suggested in Section 5. This is because when such choices are used to prune the possible values of E which are deduced from knowledge of operand sharing, it may become feasible to recover E.

The choices of the divisor set $\{2,3,5\}$ and the addition subchains in Table 3.1 have been made with some care to ensure such attacks as the above leave an infeasible number of values which E might be. Small divisors lead to short addition subchains and hence more ambiguity over which divisor occurred, but large divisors may lead to characteristic patterns which identify conditions under which the search tree for E can be pruned to a computationally feasible size.

9 Conclusion

An exponentiation algorithm "MIST" has been presented which has a variety of features which make it much more resilient to attack by differential power analysis than the normal m-ary or sliding window methods. MIST uses randomly different multiplication schemes on every run in order to avoid the averaging which is normally required for power analysis attacks to succeed. It also avoids re-using multiplicands within a single exponentiation, thereby defeating some of the more recent power analysis attacks.

There are many different ways in which to program the random selection of the so-called divisors of the algorithm. For the code presented here, about $1.42\log_2 E$ multiplications are required for the exponent E, thereby making it more efficient than square-and-multiply. Three read/write registers are required for storing intermediate powers, together with somewhat less memory for storing a pre-computed addition chain which determines the exponentiation scheme. Otherwise there is little extra overhead in terms of either space or time. In a processor/co-processor set-up, the additional minor computing associated with the exponent can be carried out on the processor in parallel with, and without holding up, the exponentiation on the co-processor.

As with all algorithms, poor implementation can lead to data leakage. The main sources of such weakness have been identified. In particular, divisor sets and addition subchains must be chosen carefully and a predictable choice of any divisor must be avoided.

MIST is compatible with most other blinding techniques, and independent of the methods used for other arithmetic operators. So, in conjunction with existing methods, it offers a sound basis for high specification, tamper resistant crypto-systems.

References

1. C. Clavier & M. Joye, *Universal Exponentiation Algorithm*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 300–308.
2. K. Gandolfi, C. Mourtel & F. Olivier, *Electromagnetic Analysis: Concrete Results*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 251–261.
3. D. E. Knuth, *The Art of Computer Programming*, vol. **2**, "Seminumerical Algorithms", 2nd Edition, Addison-Wesley, 1981, 441–466.
4. P. Kocher, *Timing Attack on Implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology – Crypto '96, N. Koblitz (editor), Lecture Notes in Computer Science, **1109**, Springer-Verlag, 1996, 104–113.
5. P. Kocher, J. Jaffe & B. Jun, *Differential Power Analysis*, Advances in Cryptology – Crypto '99, M. Wiener (editor), Lecture Notes in Computer Science, **1666**, Springer-Verlag, 1999, 388–397.
6. D. May, H.L. Muller & N.P. Smart, *Random Register Renaming to Foil DPA*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 28–38.
7. T. S. Messerges, E. A. Dabbish & R. H. Sloan, *Power Analysis Attacks of Modular Exponentiation in Smartcards*, Cryptographic Hardware and Embedded Systems (Proc CHES 99), C. Paar & Ç. Koç (editors), Lecture Notes in Computer Science, **1717**, Springer-Verlag, 1999, 144–157.
8. E. Oswald & M. Aigner, *Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 39–50.
9. C. D. Walter, *Exponentiation using Division Chains*, IEEE Transactions on Computers, **47**, No. 7, July 1998, 757–765.
10. C. D. Walter & S. Thompson, *Distinguishing Exponent Digits by Observing Modular Subtractions*, Topics in Cryptology – CT-RSA 2001, D. Naccache (editor), Lecture Notes in Computer Science, **2020**, Springer-Verlag, 2001, 192–207.
11. C. D. Walter, *Sliding Windows succumbs to Big Mac Attack*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. Koç, D. Naccache & C. Paar (editors), Lecture Notes in Computer Science, **2162**, Springer-Verlag, 2001, 286–299.
12. C. D. Walter, *Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli*, Proceedings of CT-RSA 2002, Lecture Notes in Computer Science, Springer-Verlag, 2001, *this volume.*