# Chapter 17
# Randomized Exponentiation Algorithms

Colin D. Walter

## 17.1 Introduction

A *randomized algorithm* for function $f$ takes the usual inputs for $f$ together with a stream of random numbers and combines them in a way such that partial or complete knowledge of the atomic operations used to compute $f$ does not easily reveal the values of some or all inputs. The output of $f$ is still computed correctly, but the value is independent of the random input stream.

In this chapter we consider randomized algorithms for the exponentiation function and assume that side channel leakage reveals a certain level of partial knowledge about the arithmetic and read/write operations performed on the manipulated big numbers. Our object is to make it computationally infeasible for an attacker to use this information to deduce the secret exponent during its use over the lifetime of a cryptographic token.

For example, in the usual square-and-multiply algorithm (Figure 17.1, [6]), full knowledge of the sequence of squares and multiplies immediately determines the complete exponent uniquely. Specifically, there is an exponent bit for every square; and every time the square is followed by a multiplication the bit must be a 1, whereas it must be a 0 when the square is followed by another square.

As a rule, leaked information is rarely without error; a number of squares may be incorrectly recorded as multiplications and *vice versa*. Hence there is normally some error correction to be performed. If the number of errors is small enough, a search of nearby keys will discover the true value $D$ in a computationally feasible time. Its correctness can be confirmed easily by using the corresponding public key $E$ and the relation $P^{ED} = P$. Traversing the search space must often be done intelligently, selecting the most probable alternatives first in order to have any hope of finding the key.

In typical protocols using RSA [13], the same secret key $D$ is re-used a number of times, during which it may or may not be possible to blind it by, for example, adding

Comodo CA Ltd
e-mail: Colin.Walter@comodo.com

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_17,

**Inputs:** $M$, bit representation of $D = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$
**Output:** $C = M^D$

$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
    $C \leftarrow C^2$
    If $d_i \neq 0$ then $C \leftarrow C * M$
End

**Fig. 17.1** Square-and-Multiply Exponentiation.

a random multiple of $\phi(N)$ where $N$ is the public modulus. Data about the atomic operations can be accumulated over repeated use of the secret key and might be combined successfully to reveal the key. However, many protocols, such as ECDSA [2], generate a fresh random number on each occasion for use as the secret key. In this case there is only one chance to obtain the key and, moreover, it may be practically impossible to extract useful side channel information from the cryptographic token at the same time as using it to perform a real signature. Thus there is a variety of contexts for which protection against leakage is desirable. Different randomized algorithms are appropriate for these, and they have differing costs in terms of execution time and space, and code size, as well as demands on the supply of random numbers.

The body of this chapter describes the main randomized algorithms for exponentiation, provides an overview of possible attacks on them under likely leakage models, and considers how the inevitable errors affect results. One corollary of the definition of a randomized algorithm is that successive uses of the same inputs will result in different sequences of operations. Consequently, Kocher's averaging over many uses of the same key will prove useless [7]. However, we motivate the discussion further by showing how leaked data might be combined in Kocher-like fashion to extract operator and operand information from a *single* use of a secret key. This contrasts with attack details in earlier chapters where such information was only obtained after averaging over many applications of the same key. We also conclude with the important but counter-intuitive result that use of longer keys may actually *weaken* a cryptosystem rather than strengthen it when side channel leakage occurs.

## 17.2 The Big Mac Attack

This attack [16] applies to $m$-ary exponentiation (Figure 17.2, [6]) and to all similar algorithms which use a table of pre-computed digit powers of the input ciphertext $C$. Big Mac is so-called because the digits $d_i$ of the secret key are determined individually and independently, just like the flavours tomato, beef burger, lettuce, cheese, etc. of different layers in a certain well-known fast food product which is too large to be consumed in any other way.

**Inputs:** $M$, base $m$ representation of $D = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$
**Output:** $C = M^D$

Pre-computation of the table:
$M^{(1)} \leftarrow M$
For $i \leftarrow 2$ to $m-1$ do $M^{(i)} \leftarrow M * M^{(i-1)}$

Exponentiation of the message:
$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
    $C \leftarrow C^m$
    If $d_i \neq 0$ then $C \leftarrow C * M^{(d_i)}$
End

**Fig. 17.2** $m$-Ary exponentiation.

Using the notation of Figure 17.2 in which the exponent has a representation in base $m$, the attacker first has to distinguish the processes $C \leftarrow C^m$ of raising to the $m$th power and $C \leftarrow C * M^{(d_i)}$ of multiplication by the digit power of $M$. He must then partition the multiplications into disjoint sets for which the digits $d_i$ have the same values. The method for doing this is intimated below. Normally $m$ will be a power of 2 so that raising to the $m$th power is a sequence of $\log_2 m$ squarings. For convenience, we will assume that the $m$th power can be detected by recognizing squares from multiplies. Once the partitioning has been performed, there are $(m-1)!$ ways of associating specific different digit values with the $m-1$ sets of multiplications. One of these choices will yield the sought key. In fact, the pre-computations can be used to determine the map from sets of multiplications to digits.

The background to the attack is the fact that the power consumed by a hardware multiplier depends to some extent on the Hamming weight of the inputs [21]. This means that if we were to look at the averaged power trace for a large number of word-by-word multiplications $a \times b$ where $a$ varies randomly and $b$ is fixed then we would obtain a result which is, at least to some extent, characteristic of $b$. For standard classical multipliers, averaged traces for words $b$ of equal Hamming weight will be closer together than those for words of different weights. However, all we need for the attack to succeed is for the power trace to vary measurably between enough groups of word values. This depends mostly on the experimental technique and the trouble and expense to which attacker and designer are prepared to go. With a bit of experimentation, one can associate a probability of $b$ having a particular value for a given average trace.

The idea behind the attack is to apply the usual averaging process of Kocher's power analysis [7] to digit $\times$ big integer multiplication traces rather than to exponentiation traces. Kocher takes a number of exponentiation traces associated with the same secret exponent and arranges them so that parts corresponding to the same exponent digit are aligned. He then takes an average to improve the signal-to-noise ratio. Here we cut the trace of $C \leftarrow C \times M^{(d_i)}$ into sub-traces corresponding to constituent operations $C \leftarrow shift(C) + c_j \times M^{(d_i)}$ where $c_j$ is a word-level digit

of $C$, i.e., a group of consecutive bits of $C$ used as a single input to the hardware multiplier of the cryptographic token. These sub-traces (one for each $j$) are aligned and averaged to give a trace which should be characteristic of $M^{(d_i)}$. The dependency on the words $c_j$ of $C$ is averaged away. This is done for all $n$ digits of the key $D$. The $n$ averaged traces are compared using the Euclidean distance between them. It turns out that traces for which the digits have the same value are close together whereas those for different valued digits are noticeably further apart. This enables the partitioning to be performed.

Typically in RSA the inputs $C$ and $M$ will have around 1024 bits and an 8- or 16-bit hardware multiplier will be used. This means that $C$ will be broken into about 64 words, and this is the number of traces which are averaged. Unless $C$ has a really exceptional value (such as 0), this is enough to remove any dependency on $C$ in the averaged trace. Moreover, as $M^{(d_i)}$ also has about 64 words, it is very unlikely that a pair of them will share enough digits of similar characteristics to be confused when their averaged traces are compared.

The same technique is applied to distinguish the squarings (or multiplications) used in obtaining the $m$th power of $C$. In this case the averaged trace is that of a random operand $C'$ rather than that of $M^{(i)}$ for some digit value $i$. This trace is not close to that of any of the multiplications involving $M^{(d_i)}$ nor to any of the operands used in the other $m$th power computations. Hence, when the partitioning process is applied to all multiplications in the exponentiation, including those in the $m$th powers, we can identify the $m$th power computations and group together the sets of multiplications for which the same exponent digit has been used. So, unless the key lengths are small and the multipliers are large, one might expect the digits $d_i$, and hence the key $D$, to be recovered more or less accurately from a single exponentiation.

## 17.3 Digit Representation and Exponentiation Algorithms

All of the randomized exponentiation algorithms in this chapter depend on a randomized recoding of the binary representation of the secret key $D$. This is done by one of the change-of-base algorithms in Figures 17.3 and 17.4, the latter being a more complex version of the former in which the base $m$ can be randomly varied instead of being fixed. In both figures the function mod$'$ includes a random input which allows a limited number of alternative output digits $d_i$ subject to the property that the division $(D-d_i)/m$, resp. $(D-d_i)/m_i$, in the next line is exact. So the outputs satisfy

$$D=\sum_{i=0}^{n-1} d_i m^i = ((\dots(d_{n-1}m+d_{n-2})m+\cdots+d_2)m+d_1)m+d_0 \tag{17.1}$$

and

$$D=((\dots(d_{n-1}m_{n-2}+d_{n-2})m_{n-3}+\cdots+d_2)m_1+d_1)m_0+d_0 \tag{17.2}$$

respectively.

**Inputs:** $D \geq 0$, base $m > 1$
**Outputs:** $n$, base $m$ representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_m$
$i \leftarrow 0$
While $D > 0$ do
Begin
    $d_i \leftarrow D \bmod' m$
    $D \leftarrow (D - d_i)/m$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

**Fig. 17.3** Change-of-base algorithm for fixed base.

**Input:** $D \geq 0$
**Outputs:** $n$, base sequence $m_{n-1} \ldots m_2 m_1 m_0$, digit sequence $d_{n-1} \ldots d_2 d_1 d_0$
$i \leftarrow 0$
While $D > 0$ do
Begin
    Select base $m_i$
    $d_i \leftarrow D \bmod' m_i$
    $D \leftarrow (D - d_i)/m_i$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

**Fig. 17.4** Variable base representation algorithm.

Figure 17.3 just provides the usual, standard representation to some (fixed) base $m$ when mod$'$ is taken to be the usual mod function which returns the least non-negative remainder on division by $m$. So $m = 2$ will give the binary representation, and $m = 10$ the decimal version of $D$. With this fixed choice for mod$'$, successive executions of the algorithm will always give the same representation. It yields the normal $m$-ary exponentiation algorithm when the digits are fed into the exponentiation algorithm given in Figure 17.5.

A well-known example of the application of Figure 17.4 is in the sliding windows exponentiation algorithm. As before, take mod$'$ to be the mod function. The base $m_i$ is chosen to be $m = 2^r$ if the remaining, unrecoded part of $D$ is odd, and to be 2 otherwise. This gives windows of $r$ or 1 bits. The digits for the $r$-bit windows are all odd and those for the 1-bit windows are all 0. When these are fed into Figure 17.5, the sliding windows exponentiation algorithm is obtained.

**Inputs:** $M$, representation of $D = d_{n-1} \ldots d_2 d_1 d_0$
with respect to bases $m_{n-1} \ldots m_2 m_1 m_0$
**Output:** $C = M^D$

Pre-computation: a table containing $M^{(d)} = M^d$ for each digit value $d$
$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
$C \leftarrow C^{m_i}$
If $d_i \neq 0$ then $C \leftarrow C * M^{(d_i)}$
End

**Fig. 17.5** Left-to-right exponentiation.

The $m$-ary and sliding windows algorithms process the bits or digits of the exponent $D$ from left to right, i.e., from most to least significant. However, the square-and-multiply algorithm for exponentiation can also process the digits in the opposite order, as in Figure 17.6. The cost difference between the two directions for base 2 is almost entirely context specific, depending on, for example, how one moves data around in registers.

However, for larger $m$, the left-to-right and right-to-left versions of the exponentiation algorithm allow one to trade memory requirements for execution time. The left-to-right version requires space for the pre-computed powers of $M$ but the right-to-left version has to spend time computing the digit powers of $M$ (which has a new value) at each loop iteration.

For cryptographic purposes it is usually desirable to recode the exponent in the same order as the digits are consumed in the exponentiation. That means using the change-of-base algorithms here with the right-to-left exponentiation algorithm. The high point of this chapter is the MIST algorithm which does things in this order. If the opposite order is desired and $D$ is stored in binary, then the new

**Inputs:** $M$, representation $D = d_{n-1} \ldots d_2 d_1 d_0$
with respect to bases $m_{n-1} \ldots m_2 m_1 m_0$
**Output:** $C = M^D$

$C \leftarrow 1$
For $i \leftarrow 0$ to $n-1$ do
Begin
If $d_i \neq 0$ then $C \leftarrow C * M^{d_i}$
$M \leftarrow M^{m_i}$
End

**Fig. 17.6** Right-to-left exponentiation.

**Input:** Binary representation of $D = b_{z-1} \dots b_2 b_1 b_0 \geq 0$
**Outputs:** $n$, base sequence $m_0 m_1 m_2 \dots m_{n-1}$, digit sequence $d_0 d_1 d_2 \dots d_{n-1}$

$i \leftarrow 0$
carry $\leftarrow 0$
While $z > 0$ do
Begin
    borrow $\leftarrow$ carry
    Choose window size $r_i \leq z$
    if $r_i = z$ then carry $\leftarrow 0$ else choose carry
    $m_i = 2^{r_i}$
    $d_i \leftarrow$ borrow$\times m_i + (b_{z-1} \dots b_{z-r_i})_2 -$ carry
    $i \leftarrow i+1$
    $z \leftarrow z - r_i$
End
$n \leftarrow i$

**Fig. 17.7** Variable base recoding algorithm.

bases must be powers of 2, and the change of base is achieved by recoding groups of bits from left to right, as in Figure 17.7. Of course, this makes raising to the power $m$ particularly easy, but the subscripts are inevitably reversed from normal terminology, giving $D = ((\dots(d_0 m_1 + d_1)m_2 + \dots + d_{n-3})m_{n-2} + d_{n-2})m_{n-1} + d_{n-1}$. As we see shortly, this is used very imaginatively in Itoh's overlapping windows method [4].

## 17.4 Liardet–Smart

For elliptic curves, Liardet and Smart [8] suggested using the variable base recoding of Figure 17.4 where the base selection is a randomly chosen power $m_i = 2^{r_i}$ of 2 bounded above by $r_i \leq R$. This choice is detailed in Figure 17.8 where Random$(R)$ returns a randomly chosen integer in the interval [1,R]. The function mod$'$ of Figure 17.5 is deterministic, being the (signed) residue of least absolute value (taking 1 when $D$ is odd and $r_i = 2$).

The recoding is used in left-to-right exponentiation applied to perform point multiplication in an elliptic curve context. So the terminology becomes "additions" and "doublings" instead of "multiplications" and "squarings". Then the precomputed table of Figure 17.5 need only contain the odd multiples of the input point up to $2^{R-1}$: negative digits are dealt with by a point subtraction of the corresponding positive multiple of the input point. So the space efficiency is that of a sliding window with $(R-1)$-bit windows for which all digit multiples have to be computed.

If Random$(R)$ were to have a uniform distribution over $[1,R]$ then the average window size for the odd digits would be $(R+1)/2$. So the time efficiency of the algorithm would be close to that of the equivalent sliding windows algorithm whose window size is $(R+1)/2$. Of course, the distribution of bases could be biased to favour larger values in order to increase execution efficiency.

As the windows now occupy arbitrary positions in the addition/doubling sequence, there will be both adds and doubles in any given position of the side channel traces if the same key is re-used. This should make it virtually impossible to deduce meaningful information from averaging a number of traces. Moreover, if the pattern of adds and doubles can be determined for a single use of the key, there is still the problem of identifying which digit occurs. Even more difficult is to distinguish the sign of the occurring digit because the same operands are used for both signs.

This is an excellent algorithm for protocols such as ECDSA [2] where the secret key is used just once, provided there is not too much side channel leakage. There is an exercise at the end of the chapter to determine the computational cost of key recovery under an expected leakage model. If the classical algorithms for point addition and point doubling are used then the different numbers and types of the constituent field operations could lead to a very accurate determination of the sequence of adds and doubles. So balanced code [1] may be advisable in combination with this algorithm.

**Inputs:** $D, R$
**Output:** $m_i$

If $(D \bmod 2) = 0$ then $m_i \leftarrow 2$
else
Begin
    $r \leftarrow$ Random$(R)$
    $m_i \leftarrow 2^r$
End

**Fig. 17.8** Liardet–Smart base selection.

**Table 17.1** Some recodings of $13 = 1101_2$ with $R = 3$ and their add/double traces.

| 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 0 | 1 | | DA | DA | D | DA |
| 1 | – | $\bar{1}$ | 0 | 1 | DA | D | DA | D | DA |
| – | – | 3 | 0 | 1 | D | D | DA | D | DA |
| | 1 | 1 | – | 1 | | DA | DA | D | DA |
| 1 | – | $\bar{1}$ | – | 1 | DA | D | DA | D | DA |
| – | – | 3 | – | 1 | D | D | DA | D | DA |
| 1 | 0 | – | – | $\bar{3}$ | DA | D | D | D | DA |

### 17.4.1 Attacking the Algorithm

If the secret key is re-used unblinded, and the pattern of adds ($A$) and doubles ($D$) is leaked with few errors, then the situation is less happy. An example is given in Table 17.1 where possible digit sequences on the left are spaced out to indicate the intervening doubling operations, and the corresponding operation sequences, referred to as "traces", are given on the right. So $- \; - \; \bar{3}$ indicates base $2^3$ with digit $-3$ and the corresponding sequence of adds and doubles is $D\,D\,DA$. By pairing each $A$ with a "D", corresponding $D$s are aligned with their associated bit position given at the head of each column. For uniformity, there is an initial $\ldots DA$ for the leading non-zero digit, although efficient code would omit it.

In order to determine the value of bits at position $i$ or just above, we will ignore the parts of the traces to the right of position $i$. For simplicity, assume this part is deleted, i.e., the $i$ rightmost occurrences of $D$ are removed, and any $A$s therein. Next, partition the trace segments into two sets, $Tr_i^A$ and $Tr_i^D$, according to whether their rightmost operation (in position $i$) is $A$ or $D$. Assume there are enough traces to show all the possible patterns of adds and doubles around this position. If only $D$ occurs (i.e., $Tr_i^A$ is empty), then the $i$th bit must be 0 since the representation using only base 2 would generate $A$ if the bit were 1. The same argument applies if only $A$ occurs. So the bits of index 0 and 1 must be 1 and 0, respectively, in the example of Table 17.1.

Write $D_i$ for the value of the input binary key $D$ from the most significant bit down to, and including, bit $i$. Then the traces in $Tr_i^A$ represent the value $D_i$ or $D_i + 1$ according to whether the next (i.e., less-significant) digit is non-negative or not. Clearly, as digit $d_i$ is odd for these traces, it must be the odd one of the values $D_i$ or $D_i + 1$ which is represented. So the bit pattern in the number represented by the traces $Tr_i^A$ is identical to that in $D_i$ with the sole possible exception of position $i$. Assuming there are enough traces for base 2 to have been chosen at position $i$, we will have $A$ at position $i+1$ if, and only if, the bit is 1 at that position. This can be seen in Table 17.1 where we can use this to deduce the values of bits in positions 1 and 3. In fact, we can have no more $A$s until the next bit which is 1. So we can deduce that bit 2 is 1 from the trace set $Tr_0^A$. As the first trace only goes up to position 3, we know the input has at most 4 bits, all of which have now been determined.

An attacker may only be able to make, say, 10 measurements with enough accuracy to deduce the patterns of adds and doubles. Consequently, a few bits may be undetermined [19]. (If the arguments are performed carefully, none should actually be incorrect.) However, it then requires surprisingly little computational power to deduce the key.

On the other hand, if adds can scarcely be distinguished from doubles, life is much more difficult for the attacker. His main problem in performing this attack is to align the doubles. Without the ability to do this, the sub-traces corresponding to given key bits cannot be aligned. Their random movement within side channel traces seems to average away useful information except at the key ends. He could select the very longest traces to guarantee only base 2 was used and average them

to deduce their common pattern, but, of course, there will be no such traces because the probability of generating them is too small for cryptographic-sized keys. So, overall, any uncertainty over interpreting the side channel leakage seems to increase dramatically the difficulty of extracting the key.

In conclusion, this algorithm should only be used after careful regard to its context. In particular, it should not be employed where the same key is used repeatedly without blinding unless side channel leakage is low.

## 17.5 Oswald–Aigner Exponentiation

Another randomized algorithm was proposed by Oswald and Aigner [11]. For ease of presentation, the description here is slightly modified. The base is fixed at $m = 2$ and the digit set is $\{-1,0,1,2\}$. In the digit recoding phase (Figure 17.3), the randomization occurs in the digit selection function mod′ which chooses –1 or 1 when $D$ is odd and 0 or 2 when $D$ is even. However, choice is only possible in certain cases: 2 is only allowed when the previous non-zero digit was $-1$, namely when a "Carry" has been propagated to obtain the next value of $D$ in the recoding; and $-1$ is only allowed when there is no such Carry being propagated. Termination is forced by selecting 1 if $D = 1$ and 2 if $D = 2$. This is described in detail in Figure 17.9, and some recodings of 29 are given on the left side of Table 17.2.

**Input:** $D \geq 0$
**Outputs:** $n$, and representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_2$
$i \leftarrow 0$
Carry $\leftarrow$ False
While $D > 0$ do
Begin
    If $D = 1$ then $d_i \leftarrow 1$ else
    If $D = 2$ then $d_i \leftarrow 2$ else
    If Carry then
    Begin
        If $(D \bmod 2) = 1$ then $d_i \leftarrow 1$ else $d_i \leftarrow 0$ or 2
        If $d_i \neq 0$ then Carry $\leftarrow$ False
    End
    else
    Begin
        If $(D \bmod 2) = 0$ then $d_i \leftarrow 0$ else $d_i \leftarrow 1$ or $-1$
        If $d_i = -1$ then Carry $\leftarrow$ True
    End
    $D \leftarrow (D - d_i)/2$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

**Fig. 17.9** Oswald–Aigner digit generation.

For exponentiation, the right-to-left method of Figure 17.6 is preferred because the digits are then consumed in the same order as they are generated. In an elliptic curve context, the digit $-1$ causes no problems as point subtraction is as easy as point addition. The digit 2 is processed by re-ordering the loop iteration to perform the point addition (with digit $d_i = 1$) after the point doubling instead of before it. Hence the space efficiency matches that of the equivalent square-and-multiply algorithm. On average half the recoded digits are odd (+1 or –1) and half are even (0 or 2). More precisely, the digits $\{-1, 0, 1, 2\}$ occur in the ratio $\frac{1}{8} : \frac{3}{8} : \frac{3}{8} : \frac{1}{8}$. So the average time efficiency is a little poorer than square-and-multiply because occurrences of digit 2 are more expensive than those of digit 0.

### *17.5.1 Attacking the Algorithm*

As in the attack on the Liardet–Smart algorithm, suppose that adds ($A$) can be distinguished from doubles ($D$) reliably in each execution of the exponentiation procedure and that the same key is used many times unblinded as the exponent.

Again, the behaviour of the algorithm at any point depends on the local bit pattern in the binary representation of the key. This bit pattern is reflected in a restricted set of patterns over $\{A, D\}$. From these, the bit pattern can be deduced. For example, the pattern $DAAD$ only arises from the recoding 12 or $\bar{1}2$ (more significant digit on the left). This means a corresponding bit pattern 111 must occur in the binary representation: the middle 1 is needed to generate the digit 2 using a Carry which can only come from the bit on its right being 1, and a 1 is needed on its left to give the recoded digit 1 or $\bar{1}$. This occurs for some traces representing the top three bits in Table 17.2.

Now suppose there are enough traces to generate every possible pattern of operations near a given bit position. From the previous paragraph, we will know every occurrence of 111. Also, the bit pair 00 always causes two doublings with no intervening addition, but for every other bit pair an intervening add is possible. So we can identify every occurrence of 00. Thus 00 cannot occur in the example of Table 17.2. Furthermore, the bit pattern 10 always has one $A$ between the $D$s of its two bit positions whereas every other bit pattern allows the $D$s to be adjacent for

**Table 17.2** Recodings of $29 = 11101_2$ and their traces, both generated right to left.

| 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | DA | DA | DA | D | DA |
| 2 | $\bar{1}$ | 1 | 0 | 1 | AD | DA | DA | D | DA |
| 1 | 2 | $\bar{1}$ | 0 | 1 | DA | AD | DA | D | DA |
| 2 | 0 | $\bar{1}$ | 0 | 1 | AD | D | DA | D | DA |
| 1 | 1 | 1 | 1 | $\bar{1}$ | DA | DA | DA | DA | DA |
| 2 | $\bar{1}$ | 1 | 1 | $\bar{1}$ | AD | DA | DA | DA | DA |
| 1 | 2 | $\bar{1}$ | 1 | $\bar{1}$ | DA | AD | DA | DA | DA |
| 2 | 0 | $\bar{1}$ | 1 | $\bar{1}$ | AD | D | DA | DA | DA |

some recoding. Hence all occurrences of 10 will be determined. This shows that 10 must occur over positions 2,1 in the example of Table 17.2. These two cases enable every bit 0 to be determined as well as every bit immediately to the left of a 0. Of course, the remaining undetermined bits must all be 1s, otherwise they would have been determined as belonging to a pattern 00 or 10. In fact, if some traces contain one or more $A$s and others contain no $A$s between two neighbouring $D$s, then the corresponding bit pair must be $*1$ for some bit $*$. So every bit 1 can be determined that way. This is the case for the example of Table 17.2 and it reveals the whole key.

However, the attacker may have too few traces to be sure of his deductions about the bits. In this case he looks at the ratios of the number of traces with zero, one or two $A$s between the $D$s of a bit pair. Most occurrences of 1 will be determined unequivocally as above, including the majority of occurrences of 111. Otherwise, it is possible to use the operation pattern to assign a probability to the value of each bit pair. For example, no intervening $A$s will make 00 the most likely bit pair, and with a probability that increases with the number of traces available for inspection. As each bit belongs to two pairs (except at the ends), almost all bits are determined with high or complete accuracy. Indeed, with as few as 10 correct traces and a standard key length for elliptic curve cryptography, it is computationally possible to determine any unknown bits and reveal the secret key ([20], Thm. 1).

Greater accuracy is obtained from looking at patterns corresponding to sequences of three or more bits instead of just two and this might overcome problems arising from errors in the traces.

However, the above analysis depends critically on precise alignment of all occurrences of doubles in the traces. With balanced code for adds and doubles [1], this may be difficult because the adds and doubles cannot be distinguished so easily. In fact, it is not clear how to align the traces satisfactorily even if the bits of the key are known as far as the point of interest in the traces. Inexact alignment seems to average away any useful data about the bits except at the ends.

As with the Liardet–Smart algorithm, the security of the Oswald–Aigner method relies on the key being different on each use, or for it to be very difficult to use side channel leakage to distinguish adds from doubles reliably.

## 17.6 Ha–Moon

There are two randomizing algorithms due to Ha and Moon et al. [3, 25], both presented as left-to-right exponentiation methods.

The first [3] has fixed base $m = 2$ and simply employs the most general binary signed digit (BSD) coding in the change-of-base algorithm (Figure 17.3): it selects digit $d = 0$ when $D \equiv 0 \bmod 2$ and randomly chooses between $d = \pm 1$ when $D \equiv 1 \bmod 2$. When $D$ is odd, the random choice makes the next value of $D$ odd or even with equal probability, and so the occurrence or otherwise of a multiplication does not indicate the value of the next bit in the original input value of $D$. However, in exactly the same way as with the Oswald–Aigner method, the pattern of additions

and multiplications reveals the exponent with feasible computation when it is re-used about 10 times [10].

The second, improved version by S.-M. Yen et al. [25] uses any fixed 2-power radix and employs digit recoding from most to least significant, so that conversion can be done on-the-fly during a left-to-right exponentiation. An example with base 4 is given in Figure 17.10. It is readily verified that the digit $d_i$ is always in the range 1–14.

**Input:** Base 4 representation of $D = (b_{n-1} \ldots b_2 b_1 b_0)_4$, $b_{n-1} > 0$
**Output:** Base 4 representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_4 + \delta$

Carry $\leftarrow b_{n-1}$
$i \leftarrow n-1$
While $i > 0$ do
Begin
    Borrow $\leftarrow 4*$Carry
    Carry $\leftarrow$ Random from $1, 2, 3$
    $d_{i-1} \leftarrow$ Borrow $-$ Carry $+ b_{i-1}$
    $i \leftarrow i-1$
End
$\delta \leftarrow$ Carry

**Fig. 17.10** A Yen–Chen–Ha–Moon digit recoding with base 4.

Regarding time efficiency, the method is similar to $m$-ary exponentiation. (Here with $m = 4$.) It has the same number of squarings. However, it also has the same number of multiplications as squarings because all the digits are now non-zero, whereas $m$-ary exponentiation has only $\frac{m}{m-1} = \frac{3}{4}$ of this number. The pre-computations also add marginally to the time, as does the extra digit $\delta$. The space requirement is close to that of $m^2$-ary exponentiation since the pre-computed table contains $m^2 - 2 = 14$ values. As the digits are non-negative, the technique can be used for modular exponentiation as well as for point multiplication on elliptic curves.

### *17.6.1 Attacking the Algorithm*

The non-zero property of the digits ensures that the pattern of squares and multiplications is always the same and there are no dummy operations to which to apply the safe error attack [24]. The attacks mounted on Liardet–Smart and Oswald–Aigner are therefore impossible here.

However, there are features of the recoded digit values which might be used to extract the bits of the key. Park and Lee [14] observed that the average value of the digit $d_{i-1}$ has 2 for *Carry* and 8 for *Borrow*, making an average of $6 + b_{i-1}$. Hence, minimal leakage of the value of the recoded digit from enough traces will be sufficient to determine $b_{i-1}$ correctly with high probability. For example, the leakage can be turned into probable digit values using the Big Mac attack [16] which was

described in Section 17.2. Now the identical pattern of operations for every exponentiation is in the attacker's favour: he can easily align operations at position $i-1$ and so pool any weak leakage in order to find the average $6+b_{i-1}$. This enables him to recover the secret key $D$ with very few errors. He just needs to collect more trace data to add into his averages if the signal-to-noise ratio is not giving enough correct digits. Consequently, any re-use of a key with such recoding should be combined with random blinding of it. Then the used value of $b_{i-1}$ varies randomly and its average value contains no information.

Thus the second Ha–Moon algorithm exhibits different strengths and weaknesses from those of the Liardet–Smart and Oswald–Aigner algorithms. This may make it more suitable than the others in some contexts. As usual, message whitening and key blinding appear necessary if the same key is to be re-used a number of times.

## 17.7 Itoh's Overlapping Windows

The algorithm of Itoh et al. [4] is a sliding window technique which, in its general form, essentially includes all the preceding algorithms except that digits are non-negative. It allows any representation given by the variable base representation algorithm (Figure 17.4) subject to the base being a 2-power. As in the (second) Ha–Moon algorithm, the authors describe the conversion from binary as a recoding from left to right, enabling a table-driven exponentiation to consume the digits in the order they are generated.

The method is illustrated by several examples, the first being the overlapping windows method (O-WM) in Figure 17.11. There are two main parameters, $k$ and $h$, with $k>h$ and a recommended relationship $h \geq k/2$. The base for both input and output is fixed at $m=2^{k-h}$ for this example. In the figure, the $k$-bit variable $Left$ consists of two parts. Its lowest $k-h$ bits are the next set of bits of $D$ to be processed, namely the base $m$ digit $b_{i-1}$. Its top $h$ bits, the value of $Top$, is the remainder left from processing the more significant bits of $D$. The output digit $d_{i-1}$ is no larger than $Left$ and so the digit range is from 0 to $2^k-1$. Consequently, the process can be viewed as a $k$-bit sliding windows method with an overlap of $h$ bits.

Figure 17.11 is just a fixed base version of the variable base recoding in Figure 17.7 with appropriate simplifications and details about "choose". It yields a left-to-right exponentiation method whose time efficiency is similar to that of $m$-ary exponentiation and whose space efficiency is that of $2^k$-ary exponentiation. For smart card applications, $k$ needs to be kept very small, which limits the amount of randomness which can be introduced. We need $h \geq k/2$ to add as much randomness as is in the key $D$.

The full O-WM method still keeps $k$ fixed but allows $h$ to vary, so that the base $m$ also varies. This uses the recoding method of Figure 17.7 where Carry is chosen to keep output digits in the range 0 to $2^h-1$. Interested readers should consult the original paper of Itoh [4].

### 17.7.1 Attacking the Algorithm

The O-WM method is very similar to the second Ha–Moon algorithm for a fixed base $m = 2^{k-h}$. The main difference is in the range for Carry in Figure 17.10. The similarity means that the same attacks are likely to work for both algorithms, although there are more complications here. The presence of zero digits and/or variable bases means that matters are easier when squares can be recognized. Then the multiplications can be correctly aligned in the same way as in Table 17.1 for Liardet–Smart. Leakage of Hamming weight enables this to be done, and so that is assumed in the leakage model here.

In particular, the attack described in Section 17.6.1 works here, using Park and Lee's averaging technique [14]. Usually a good first approximation to $b_{i-1}$ is obtained by ignoring the effect of $Left$ on the range of randoms assigned to $Top$: when the previous value of $Top$ is non-zero, $Left$ has a value of at least $m = 2^{k-h}$, which is at least $2^h$ if $h \le k/2$. There is a large cost in selecting $h > k/2$, but even if this were to occur, very few previous values of $Top$ are small enough to reduce the range of the following value of $Top$. Hence $Top$ has an average value only slightly less than $\frac{1}{2}(2^h-1)$, making the average value of $d_{i-1}$ a little less than $b_{i-1} + \frac{1}{2}(m-1)(2^h-1)$. This enables an approximate value for $b_{i-1}$ to be deduced once squaring operations in the traces have been aligned. Of course, at least 1 in $2^h$ of the random values will be 0, so the average for $Top$ should be reduced by $O(2^{-h})$ and that for $d_{i-1}$ reduced by $O((m-1)2^{-h}) = O(2^{k-2h})$. So in most cases this rough calculation should yield $b_{i-1}$ simply by rounding down. With a bit more effort the accuracy of the digit prediction would be improved.

In comparison with earlier algorithms, it is clear that this one is more difficult to break if suitable parameters can be chosen (such as large $k$ and small $k-h$), especially if the base is made variable. So security can be improved, but it is at the cost of run-time efficiency.

**Input:** $h, k$ with $0 < h < k$, $n > 0$, $D = (b_{n-1} \dots b_2 b_1 b_0)_m$ where $m = 2^{k-h}$
**Output:** Random base $m$ representation of $D = (d_{n-1} \dots d_2 d_1 d_0)_m$

$m \leftarrow 2^{k-h}$
Top $\leftarrow 0$
$i \leftarrow n$
While $i > 0$ do
Begin
    Left $\leftarrow m *$ Top $+ b_{i-1}$
    If $i = 1$ then Top $\leftarrow 0$
    else Top $\leftarrow$ Random from $\{0, 1, \dots, \min\{\text{Left}, 2^h - 1\}\}$
    $d_{i-1} \leftarrow$ Left $-$ Top
    $i \leftarrow i - 1$
End

**Fig. 17.11** O-WM recoding.

## 17.8 Randomized Table Method

Itoh et al. [4] enhance O-WM with a "randomized table" technique (RT-WM) which modifies the digit range $\{d_{min},\dots,d_{max}\}$ to $\{r+d_{min}2^c,\dots,r+d_{max}2^c\}$ where $r$ is a random $c$-bit number fixed for each exponentiation. The required pre-computed table then contains the powers of the input text under the new digit range. As the method can be applied as an additional counter-measure to any recoding scheme, the translation is described as a separate process here. However, it is a fully integrated part of the recoding in [4].

For the desired sequence of bases $m_{n-1},m_{n-2},\dots,m_0$, let

$$D_0 = r(((\dots(m_{n-2}+1)m_{n-3}+\cdots+1)m_1+1)m_0+1)$$

Compute $D' = (D-D_0-\delta)/2^c$ where $D-D_0=\delta \bmod 2^c$, and apply the chosen recoding method with the chosen bases to $D'$ to obtain

$$D' = ((\dots(d'_{n-1}m_{n-2}+d'_{n-2})m_{n-3}+\cdots+d'_2)m_1+d'_1)m_0+d'_0$$

where the digits $d'_i$ are in the range $\{d_{min},\dots,d_{max}\}$. Then

$$D = ((\dots(d_{n-1}m_{n-2}+d_{n-2})m_{n-3}+\cdots+d_2)m_1+d_1)m_0+d_0+\delta$$

for digits $d_i = r+d'_i2^c$ in the required range $\{r+d_{min}2^c,\dots,r+d_{max}2^c\}$.

### *17.8.1 Attacking the Algorithm*

The motivation behind RT-WM is clearly the disruption of the digit averaging attacks described in Sections 17.6.1 and 17.7.1. Currently there are no published attacks on the method.

If the leakage were strong enough, an attack which yields any information from individual traces might be applied to the table construction phase first in order to reveal $r$ and then applied to the exponentiation phase. This is unlikely to work without averaging over many traces: devices which employ the algorithm are likely to use hardware counter-measures which are sufficient to defeat attacks on a single trace.

The average of $r$ is $\frac{1}{2}(2^c-1)$, which leads to an average value for $D_0$. In the case of Ha–Moon 2 or O-WM this would lead to average values for each digit $d'_i$. Ostensibly, this leads to recovery of the average for $D'$ and hence to $D$. However, the borrows in computing $D' = (D-D_0-\delta)/2^c$ mean that every $d'_i$ has the same average, namely that of a random digit. Hence the average $d'_i$ contains no information, and $D$ cannot be recovered in this way.

There are new methods being published which enable weak leakage to be combined successfully in the presence of randomizing counter-measures, e.g., [23]. If the same key is used sufficiently many times, the information theoretic content of the

```
m_i ← 0
If Rand(8) < 7 then
        If D ≡ 0 mod 2 then m_i ← 2 else
        If D ≡ 0 mod 5 then m_i ← 5 else
        If D ≡ 0 mod 3 then m_i ← 3
If m_i = 0 then
Begin
        p ← Rand(8)
        If p < 6 then m_i ← 2 else
        If p < 7 then m_i ← 5 else
        m_i ← 3
End
```

**Fig. 17.12** One choice for digit recoding in MIST.

side channels is enough to determine the key uniquely. The only question is whether or not the information can be combined into a computationally feasible attack.

## 17.9 The MIST Algorithm

In the preceding algorithms, the change-of-base and variable base representation algorithms in Figures. 17.3 and 17.4 only made use of bases which are powers of 2. This means that these algorithms can be expressed as left-to-right recodings of binary, and digits can be consumed as they are generated by the usual left-to-right exponentiation algorithm. The MIST algorithm [17] deliberately selects bases which are not all powers of the same prime, but this forces digit generation to be from right to left. However, by separating digit generation from exponentiation, the exponentiation can be performed in either direction.

The original description suggests choosing the recoding base $m_i$ randomly from the set $S = \{2,3,5\}$. An example algorithm for this is given in Figure 17.12 where $Rand(n)$ returns a random non-negative integer less than $n$. Because raising to the power 3 or 5 is less efficient than raising to a power of 2, the choice is biased towards base 2. However, a multiplication is saved in the exponentiation when the digit is zero, so there also a bias towards selecting bases for which the digit is 0. The digit choice mod$'$ could be the least non-negative value, but alternatives are possible, such as the residue of least absolute value. $D$ would be stored efficiently in base 240 if the machine word were 8-bits long, so that recoding digit and base selection could be done by looking only at the lowest digit. A typical recoding example is $235_{10} = (((((0\times2+1)\times3+0)\times2+1)\times5+4)\times2+0)\times3+1$. This can be abbreviated to $235 = 1_2 0_3 1_2 4_5 0_2 1_3$ using the obvious notation to indicate the base of each digit.

Space efficiency is similar to that of binary exponentiation except for an extra register required to store one more intermediate product and space for the recoding.

Time efficiency is between that of binary and quaternary exponentiation. The details to check this require modelling the recoding as a Markov process and computing its eigenvectors [17]. The left-to-right exponentiation method of Figure 17.5 uses table entries for the multiplications. However, to achieve the best efficiency in the right-to-left method of Figure 17.6, the computations of $M^{d_i}$ and $M^{m_i}$ must be combined to minimize the total number of long integer multiplications. Specifically, $M^{d_i}$ should be computed *en route* to $M^{m_i}$. The details can be expressed using an addition chain in which $a+b=c$ stands for the computation of $M^a \times M^b$ to obtain $M^c$. For example, the addition chain $1+1=2$, $2+1=3$, $2+3=5$ enables $M^2$, $M^3$ and $M^5$ to be computed with three multiplications, and so is suitable for base 5 with any digit except 4.

### *17.9.1 Attacking the Algorithm*

The algorithm is designed to make it much more difficult to apply any of the previous attack methods to deduce the exponent $D$. Specifically, the variable base choice means there is no alignment between operations and bits of $D$ which could be exploited. So attacks similar to those against the Liardet–Smart and Oswald–Aigner algorithms are not possible. In general, the patterns of squarings and multiplications do not seem to narrow the search space sufficiently to allow key recovery [18].

In any exponentiation, detection of operand re-use may be possible by observing Hamming weights on the bus or repeated access of the same memory locations. This makes every table-based left-to-right exponentiation potentially vulnerable. Indeed, all the previous algorithms are fatally compromised unless there is enough noise to ensure a number of mistakes in determining the operand sharing. However, in the original right-to-left MIST, the operand sharing pattern still leaves an ambiguity between the digit/base pairs $1_2$ and $0_3$. These occur sufficiently frequently to make it computationally infeasible to traverse the search space for the correct key $D$.

Nevertheless, Oswald [12] has reported analysing patterns of squarings and multiplications from a single trace by using Viterbi's algorithm [15] to select the most likely sequence of digits. This chooses about 83% of digits correctly, but apparently does not identify which are the correct ones. It is an improvement on the 74% predicted by independently selecting the most likely digits when the pattern for each digit has been identified [18]. The latter choice ignores a strong dependence between consecutive digit choices resulting from the efficiency-driven bias in Figure 17.12. With short elliptic curve keys, Oswald's technique leaves some 30 bits to modify, which is infeasible without good information about their positions. Further work on the attack may reveal this so that a prioritized search can be performed. But the result also assumes perfect information from the trace. In practice, noise leads to some degradation in the deduced pattern, and this is likely to render the attack infeasible.

## 17.10 Conclusions

The first attacks on exponentiation by Kocher et al. [7] showed that key recovery is possible from weak side channel information when keys are re-used with the same unprotected pattern of long integer operations. Similar trace averaging methods can reveal repeated patterns of operand re-use and of data movements in algorithms. This can still create problems for algorithms that appear to have key-independent computation patterns at the highest level [5]. Randomization techniques are required to prevent this. Key blinding provides one solution, but it may be insufficient [22]. Further randomization to confuse the attacker can be provided through the randomary exponentiation in this chapter. Most of the methods that have been developed thus far have been seen to be weak on their own and require key blinding as well if the key is to be re-used. However, for once-off key use the randomization provides the algorithms with considerably increased security and an efficiency which is often better than that of algorithms with uniformly balanced code – such as square-and-always-multiply.

## 17.11 Exercises

These exercises are aimed at developing an appreciation of just how difficult it is to discover the correct key from imperfect side channel leakage even when the degree of error is very low.

1. In the Liardet–Smart algorithm choose a key size of 160 bits, an upper bound $R = 4$ on the window size, and assume that there is side channel leakage from a point multiplication which provides the sequence of adds and doubles without error.
   a. What is the average number of windows which occur?
   b. Calculate the average number of different keys for which the same pattern of adds and doubles will occur.
   c. Does the algorithm become more or less secure if the value of $R$ is altered?
   d. Is there a most secure value for $R$ with this level of leakage?
   e. Is it computationally feasible to attack an implementation of this algorithm with so much leakage?
   f. What are the answers to these questions if the key size is doubled to 320 bits?
   g. Repeat the previous parts under the assumption that adds and doubles are only determined correctly with a probability of $p = 0.95$. (So about 10 errors may need to be corrected before guessing the digits corresponding to each addition.) Make any reasonable simplifications you wish. For example, ignore the fact that some patterns of adds and doubles are impossible.
2. This exercise involves some programming, but it can be done entirely by hand. If so, take a smaller key size such as 20 bits, adjust the probabilities,

e.g., $p_S = 1-\frac{1}{20} = 0.95$, and use the parity of word lengths in this question as the random number generator.

a. Use a random number generator to obtain a random 160-bit key $D$. For convenience, pick a key for which exactly 80 bits are 0 and 80 bits are 1 (including the first). Convert $D$ into the string of squarings (S) and multiplications (M) which occur when it is used as the exponent in the usual square-and-multiply algorithm. Make sure your implementation omits unnecessary operations, such as an initial squaring of 1.
b. Suppose this key and algorithm are used in a cryptographic token which suffers from side channel leakage. Assume that $p_S = 1-\frac{1}{80}$ and $p_M = 1-\frac{1}{40}$ are the probabilities that each S and M is determined correctly before taking account of the fact that two multiplications cannot be adjacent. Use these probabilities to generate a string $\lambda$ over the alphabet $\{$S,M$\}$ which might have been deduced from the trace information.
c. Write down a formula for the expected number of errors in $\lambda$. In how many cases is it expected to be possible to correct the error when "MM" occurs? Are there any end conditions which $\lambda$ must also satisfy before it can correspond to the true sequence of operations? Compare your figures with those from the key and string which you generated.
d. What are the probabilities of having (a) no errors, (b) exactly one error, (c) exactly two errors, (d) exactly five errors to correct? (You may assume, for simplicity, that observable errors, such as "MM", have not occurred.)
e. Take your string $\lambda$ and correct the obvious errors such as occurrences of "MM". Mark all characters in $\lambda$ which might represent isolated errors. (So assume there are no adjacent errors.) How many keys need to be tested for correctness if $\lambda$ contains exactly (a) one error, (b) two errors, or (c) five errors?
f. Are there any substrings which must be correct if they contain at most one error?
g. Suppose the search for the correct key is done using a machine with a 32-bit processor. Using information in earlier chapters about the cost of elliptic curve operations, estimate how many 32-bit operations are needed to check the correctness of the key if it has exactly five errors and has been used in an elliptic curve point multiplication over a 160-bit field? (Assume the test requires a 160-bit point multiplication.)

## 17.12 Projects

1. In this project we assume the same key is re-used without blinding, so that there are a number of traces corresponding to the same exponent. We try to reconstruct the key from these traces.

a. Write a program to generate the sequences of squarings (S) and multiplications (M) given by (a) the Liardet–Smart and (b) the Oswald–Aigner algorithms. Collect 50 such sequences, initially for 20-bit keys.
b. As in Table 17.1 align the squarings of a number of such sequences and determine whether or not the ratio of squares to multiplications is the same for each column in the averaged sequence. If it is not, can anything be deduced about the corresponding bit in the exponent?
c. Look at the patterns of S and M which are possible for an adjacent pair of bits. Do the possibilities determine either or both of the bits? Mark the bits which are determined. How many bits are doubly determined as a result of pairing them with the bits on either side? How many bits remain undetermined? If those bits had different values, would they have been determined? If so, does this mean all bits can be determined?
d. Repeat the previous part of the question, this time looking at the patterns of S and M for three adjacent bits.
e. Mechanize the bit determination process worked out in the previous parts. Apply the process to a number of sets of $2^n$ sequences, $n = 1, 2, \ldots$ and a 160-bit key. Calculate the average number of incorrectly determined bits for each size of trace set. If there are cases where all the bits are recovered correctly, how often is the correct key recovered for each set size?

2. a. Choose a random 160-bit key. Write a program implementing the Ha–Moon recoding algorithm of Figure 17.10 and use it to generate a number of randomized digit sequences for the key. Extend the program so that it will average the digit values at each base 4 position in the key. Use this information to predict the base 4 digit of the key as described in Section 17.6.1.
b. Apply the digit prediction process to a number of sets of $2^n$ sequences, $n = 1, 2, \ldots$. Calculate the average number of incorrectly determined digits for each size of trace set. If there are cases where all the digits are recovered correctly, how often is the correct key recovered for each set size?
c. Extend the digit prediction process so that it returns a probability with each digit, namely the 1 minus half the distance of $\bar{b}_i$ from its nearest integer, where $\bar{b}_i$ is the real number average used to predict the digit $b_i$. Repeat the previous part, this time treating each incorrect digit prediction as correct if it is among the 10% with the lowest probabilities and the next nearest integer is the correct value.
d. Estimate the cost of checking a key prediction in terms of the number of 32-bit operations required to perform a 160-bit elliptic curve point multiplication. For a case where digit errors were predictable in the sense of the previous part of the question, is it computationally feasible to correct all the digit errors identified as correctable? Decide your answer by estimating the number of hours a PC of your choice would require to test half the key possibilities. Is it computationally feasible to recover a key in this way, given that you must now divide the cost of correcting digit errors by the probability of having a key prediction that is correctable?

## References


1. E. Brier and M. Joye. *Weierstraß Elliptic Curves and Side-Channel Attacks*, Public Key Cryptography (Proc. PKC 2002), D. Naccache and P. Paillier (editors), LNCS 2274, pp. 335–345, Springer-Verlag, 2002.
2. Federal Information Processing Standard 186-2. *Digital Signature Standard (DSS)*, National Institute of Standards and Technology, Maryland, USA, 2001.
3. J. C. Ha and S. J. Moon. *Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 551–563, Springer-Verlag, 2002.
4. K. Itoh, J. Yajima, M. Takenaka, and N. Torii. *DPA Countermeasures by Improving the Window Method*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 303–317, Springer-Verlag, 2002.
5. M. Joye and S.-M. Yen. *The Montgomery Powering Ladder*, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 291–302, Springer-Verlag, 2002.
6. D. E. Knuth. *The Art of Computer Programming*, vol. 2, "Semi-numerical Algorithms", 2nd Edition, pp. 441–466, Addison-Wesley, 1981.
7. P. Kocher, J. Jaffe, and B. Jun. *Differential Power Analysis*, Advances in Cryptology – CRYPTO '99, M. Wiener (editor), Lecture Notes in Computer Science, 1666, pp. 388–397, Springer-Verlag, 1999.
8. P.-Y. Liardet and N. P. Smart. *Preventing SPA/DPA in ECC Systems Using the Jacobi Form*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science 2162, pp. 391–401, Springer-Verlag, 2001.
9. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. *Power Analysis Attacks of Modular Exponentiation in Smartcards*, Cryptographic Hardware and Embedded Systems (Proc CHES 99), Ç. K. Koç and C. Paar (editors), Lecture Notes in Computer Science, 1717, pp. 144–157, Springer-Verlag, 1999.
10. K. Okeya and K. Sakurai. *On Insecurity of the Side Channel Attack Countermeasure Using Addition-Subtraction Chains under Distinguishability between Addition and Doubling*, Information Security and Privacy - ACISP '02, Lecture Notes in Computer Science, 2384, pp. 420–435, Springer-Verlag, 2002.
11. E. Oswald and M. Aigner. *Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science, 2162, pp. 39–50, Springer-Verlag, 2001.
12. E. Oswald. *Markov Model Side-Channel Analysis* SCA-Lab Technical Report IAIK - TR 2004/03/01, Institute for Applied Information Processing and Communication, 2004. http://www.iaik.tu-graz.ac.at/research/sca-lab/index.php

13. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems, Comm. ACM 21:120–126, 1978.
14. D. J. Park and P. J. Lee. *DPA Attack on the Improved Ha-Moon Algorithm* Information Security Applications, WISA 2005, J. Song, T. Kwon, M. Yung (editors), Lecture Notes in Computer Science, 3786, pp. 283–291, Springer-Verlag, 2006.
15. A. J. Viterbi. *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Trans. Information Theory, **13**(2): 260–269, 1967.
16. C. D. Walter. *Sliding Windows succumbs to Big Mac Attack*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science, 2162, pp. 286–299, Springer-Verlag, 2001.
17. C. D. Walter. *MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis*, Topics in Cryptology – CT-RSA 2002, B. Preneel (editor), Lecture Notes in Computer Science, 2271, pp. 53–66, Springer-Verlag, 2002.
18. C. D. Walter. *Some Security Aspects of the MIST Randomized Exponentiation Algorithm*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 276–290, Springer-Verlag, 2002.
19. C. D. Walter. *Breaking the Liardet-Smart Randomized Exponentiation Algorithm*, Proc. Cardis 2002, Usenix Assoc, Berkeley, CA, pp. 59–68 2002.
20. C. D. Walter. *Issues of Security with the Oswald-Aigner Exponentiation Algorithm*, Topics in Cryptology – CT-RSA 2004, T. Okamoto (editor), Lecture Notes in Computer Science, 2964, pp. 208–221, Springer-Verlag, 2004.
21. C. D. Walter and D. Samyde. *Data Dependent Power Use in Multipliers*, 17th IEEE Symposium on Computer Arithmetic – ARITH-17, IEEE Computer Society, pp. 4–12, 2005.
22. C. D. Walter. *Longer Randomly Blinded RSA Keys may be Weaker than Shorter Ones*, Information Security Applications, 8th International Workshop – WISA 2007, S. Kim, M. Yung and H.-W. Lee (editors), Lecture Notes in Computer Science, 4867, pp. 303–316, Springer-Verlag, 2008.
23. C. D. Walter. *Recovering Secret Keys from Weak Side Channel Traces of Differing Lengths*, Cryptographic Hardware and Embedded Systems — CHES 2008, E. Oswald and P. Rohatgi (editors), Lecture Notes in Computer Science, 5154, pp. 214–227, Springer-Verlag, 2008.
24. S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. *A countermeasure against one physical cryptanalysis may benefit another attack*, Information Security and Cryptology - ICISC 2001, K. Kim (editor), Lecture Notes in Computer Science, 2288, pp. 414-427, Springer-Verlag, 2002.
25. S.-M. Yen, C.-N. Chen, S. J. Moon, and J. C. Ha. *Improvement on Ha-Moon Randomized Exponentiation Algorithm*, Information Security and Cryptology – ICICS 2004, C. Park and S. Chee (editors), Lecture Notes in Computer Science, 3506, pp. 154–167, Springer-Verlag, 2005.