Random Order m-ary Exponentiation

Michael Tunstall

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol BS8 1UB, United Kingdom.
tunstall@cs.bris.ac.uk

Abstract. This paper describes a m-ary exponentiation algorithm where the radix-m digits of an exponent can be treated in a somewhat random order without using any more group operations than a standard right-to-left m-ary exponentiation. This paper demonstrates that the random order countermeasure, commonly used to protect implementations of secret key cryptographic algorithms, can be applied to public key cryptographic algorithms.

Keywords: Exponentiation algorithms, random order countermeasure, side channel analysis.

1 Introduction

Implementations of exponentiation algorithms in microprocessors need to be resistant to side channel analysis. For example, it has been demonstrated that a private key used in RSA [21] can be derived by observing a suitable side channel, such as power consumption [15] or electromagnetic emanations [11, 20]. These attacks targeted implementations of the square and multiply algorithm, where an exponent is read bit-by-bit and a zero in the exponent results in a squaring operation, whereas a one results in a squaring operation followed by a multiplication. Bits of a private key can be seen directly if these two operations can be distinguished by observing a suitable side channel while an exponentiation is being computed.

The first proposed countermeasure was to always compute a squaring operation followed by a (possibly fake) multiplication for each bit of an exponent, referred to as the square and multiply *always* algorithm [10]. This algorithm has a large impact on the efficiency of the computation of an exponentiation. An alternative was proposed in [9] that proposed efficient algorithms with a fixed structure, i.e. a structure independent to the value of the exponent. A further suggestion was made in [8], where it was proposed that a squaring operation and a multiplication be rendered indistinguishable via a side channel.

More complex attacks are proposed in [15], where numerous acquisitions are taken and an attacker attempts to observes a statistical relationship between an observed side channel and an intermediate state. In order to prevent this class of attack individual variables are combined with random values generated for each instantiation of an algorithm [6].

In block ciphers the order in which variables are treated can also be randomised to augment the side channel resistance of an implementation [17]. In this paper an algorithm is proposed that allows an exponentiation to be computed in a random order, although not all orders will occur with with the same probability. This algorithm is intended to allow an exponentiation to be computed without the currently used blinding algorithms. However, the proposed algorithm could also be used to augment the security of an exponentiation implementation, and will inhibit attacks that allow operations to be distinguished from one acquisition. The proposed algorithm requires a large amount of memory. However, some modern smart card chips are using 32-bit architectures and one can expect to have around 4k of RAM available [4, 18]. This would allow for the proposed algorithm to be implemented for elliptic curve cryptography, but it is unlikely to be possible for exponentiation in $(\mathbb{Z}/N\mathbb{Z})^*$.

The rest of this paper is organised as follows. In the next section, we review some methods for evaluating an exponentiation. In Section 3 we describe the methods of side channel analysis that could potentially be used to attack an exponentiation. The previously published uses of random values as a countermeasure to side channel analysis is described in Section 4. In Section 5 we detail an exponentiation algorithm where the digits can be treated in a random order. In Section 6 we describe how an attacker would try to derive information by side channel analysis. Finally, we conclude in Section 7.

2 Exponentiation Algorithms

2.1 The square and multiply algorithm

The simplest algorithm for computing an exponentiation is the square and multiply algorithm. This is where an exponent is read left-to-right bit-by-bit, a zero results in a squaring operation being performed, and a one results in a squaring operation followed by a multiplication with the original message. This algorithm is detailed in Algorithm 1, where we define the function `bit(`n, i`)` as a function returning the i-th bit of n. The input is an element x in a (multiplicatively written) group $\mathbb{G}$ and a positive ℓ-bit integer n; the output is the element $z = x^n$ in $\mathbb{G}$. This algorithm requires two group elements in memory and, on average, $1.5\,(\lfloor \log_2 n \rfloor - 1)$ group operations to compute x^n.

2.2 Left-to-Right m-ary Exponentiation

In order to compute $z = x^n$ more efficiently it is possible to use an algorithm that is described in [13]. In this algorithm one precomputes some values that are small powers of x and the exponent is broken up into ℓ words in base m (this is called the m-ary expansion and ℓ is called the m-ary length). Typically, m is chosen to be equal to 2^k, for some convenient value of k, to enable the relevant digits to simply be read from the exponent. The m-ary algorithm is shown in Algorithm 2, where we define the function `digit(`n, i`)` as a function returning

Algorithm 1: The Square and Multiply Algorithm

```
Input: $x \in \mathbb{G}$, $n \geq 1$, $\ell$ the binary length of $n$ (i.e. $2^{\ell-1} \leq n < 2^{\ell}$)
Output: $z = x^n$

$A \leftarrow x$
$R \leftarrow x$
for $i = \ell - 2$ down to 0 do
    $A \leftarrow A^2$
    if (bit($n, i$)$\neq 0$) then  $A \leftarrow A \cdot R$
end
return $A$
```

the i-th radix-m digit of n. This algorithm requires m group elements in memory and, on average, $\left(m + \frac{m-1}{m}\right)(\lfloor \log_m n \rfloor - 1) + m - 2$ group operations to compute x^n.

Algorithm 2: Left-to-Right m-ary Exponentiation Algorithm

```
Input: $x \in \mathbb{G}$, $n \geq 1$, $\ell$ the $m$-ary length of $n$
Output: $z = x^n$
Uses: $A$, $R[i]$ for $i \in \{1, 2, \ldots, m-1\}$

$R[1] \leftarrow x$
for $i \leftarrow 2$ to $m - 1$ do  $R[i] \leftarrow R[i-1] \cdot x$

$b \leftarrow$ digit($n, \ell - 1$)
$A \leftarrow R[b]$

for $i = \ell - 2$ down to 0 do
    $A \leftarrow A^m$
    $b \leftarrow$ digit($n, i$)
    if ($b \neq 0$) then  $A \leftarrow A \cdot R[b]$
end
return $A$
```

2.3 Right-to-Left m-ary Exponentiation

A right-to-left version of Algorithm 2 was described in [24]. This algorithm is detailed in Algorithm 3, and requires slightly more operations than the left-to-right algorithm. The structure of the algorithm is different as the effect of the value of each digit is not taken into account until the final stage of the algorithm. This algorithm requires $(m-1) + 1 = m$ group elements in memory and, on average, $\left(\frac{m-1}{m} + m\right)(\lfloor \log_m n \rfloor - 1) + 2m - 3$ group operations to compute x^n.

Algorithm 3: Right-to-Left m-ary Exponentiation

```
Input: $x \in \mathbb{G}$, $n \geq 1$
Output: $z = x^n$
Uses: $A$, $R[j]$ for $j \in \{1, \ldots, m-1\}$

for $j = 1$ to $m - 1$ do  $R[j] \leftarrow 1_{\mathbb{G}}$

$A \leftarrow x$
while $(n \geq m)$ do
    $d \leftarrow n \bmod m$
    if $(d \neq 0)$ then  $R[d] \leftarrow R[d] \cdot A$
    $A \leftarrow A^m$
    $n \leftarrow \lfloor n/m \rfloor$
end
$R[n] \leftarrow R[n] \cdot A$

$A \leftarrow R[m-1]$
for $j = m - 2$ down to $1$ do
    $R[j] \leftarrow R[j] \cdot R[j+1]$
    $A \leftarrow A \cdot R[j]$
end

return $A$
```

3 Side Channel Analysis

3.1 Simple Side Channel Analysis

The most basic form of side channel analysis is to simply inspect one acquisition of a suitable side channel, referred to as Simple Side Channel Analysis (SPA). This involves observing a suitable side channel whilst a computation is taking place. An attacker will then try and make deductions about what calculations are being performed based on these observations. If we consider the square and multiply algorithm, it has been shown that bit values of an exponent can be distinguished by observing a suitable side channel, such as the power consumption [15] or electromagnetic emanations [11, 20].

3.2 Differential Side Channel Analysis

It has been demonstrated that in microprocessors the instantaneous power consumption is typically proportional to the Hamming weight of data being manipulated at a given point in time [5], and the same relationship has been observed in electromagnetic emanations [11, 20]. This difference in Hamming weight was first exploited in [15] to attack block ciphers. This was extended in [5] to give a more complete analysis of the power consumption.

In this attack, an attacker acquires M power consumption traces (w_i for $i \in \{1, 2, \ldots, M\}$) during the computation of a cryptographic algorithm, and chooses one word, b, of an intermediate state generated while the acquisition is taking place. For a given hypothesis for a key value (or portion of the key) K

the Hamming weight of b is predicted and the correlation between this value and the instantaneous power consumption is calculated. A significant correlation will confirm the hypotheses, allowing an attacker to derive information on the key.

In order to attack an exponentiation, an attacker could use this to confirm hypotheses on an intermediate state of a multiplication at an arbitrary point in the computation to derive portions of the exponent. A significant correlation would indicate that the hypothesised exponent bits are correct.

4 Using Random Values as a Countermeasure

In this section we describe some of the countermeasures that can be used to prevent side channel analysis. We concentrate on countermeasures that are specific to exponentiation algorithms. The interested reader is referred to [16] for a discussion of countermeasures to side channel analysis in more general terms.

4.1 Blinding

The use of random values to modify the behaviour of an exponentiation has taken many different forms. One of the first proposals was to modify all the variables in a modular exponentiation [14]. If we consider the modular exponentiation $z = x^n \bmod m$, this could be implemented as

$$z = \left((x + r_1 \cdot m)^{n + r_2 \cdot \phi(m)} \bmod r_3 \cdot m \right) \bmod m \, ,$$

where r_1, r_2 and r_3 are random values and ϕ is Euler's Totient function. Typically, the bit lengths of r_1, r_2 and r_3 are chosen to increase x, d and m by one word of the processor computing the exponentiation. The above equation is specific to $(\mathbb{Z}/N\mathbb{Z})^*$, but equivalent algorithms exist for exponentiation algorithms in other groups (e.g. elliptic curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^q}$ [10]).

4.2 Randomised Algorithms

When implementing block ciphers one would combine masking with a random ordering. For example, if a given function were to be applied to a series of bytes, one would implement the function such that the bytes were treated in some random order. If an attacker were to try and predict a byte value occurring at a given point in time, the prediction would only be correct a small proportion of the time. This has a direct impact on the computed correlation coefficient, discussed in Section 3.2, since an attacker's prediction will often be incorrect.

An equivalent countermeasure for exponentiation algorithms has not previously been proposed in the literature, but other methods of randomising the computation of an exponentiation have been proposed. In this section we review some of these methods.

Random Digit Sizes

A right-to-left m-ary exponentiation algorithm is proposed in [23] where the radix of the digits of the exponent is varied during the computation of an exponentiation. The algorithm proceeds as described in Algorithm 3, but the radix of the digits of the exponent digits is chosen randomly between 2,3 and 5 whenever a new digit is required during the main loop of the algorithm.

Overlapping Exponent Digits

Another algorithm that modifies the exponent digits every time an exponentiation is computed is presented in [12]. When the exponent digits are read from the exponent the number if bits is extended such that the bits of one digit will overlap with the bits of the neighbouring digits. The digits are modified such that the sum of the effect of the overlapping digits is equivalent to the desired exponent. Given that there are numerous combinations of overlapping digits that are equivalent to the desired exponent, some bits of each digit can be randomly set each time the algorithm is executed.

5 Random Order Exponentiation

In this section we define a right-to-left m-ary exponentiation algorithm, where the radix-m digits of the exponent are treated in a random order. Before the algorithm is defined we demonstrate that, if enough memory is available, the digits of an exponent could be treated in an arbitrary order when using Algorithm 3.

Consider the right-to-left m-ary exponentiation algorithm to compute $z = x^n$. We can write the radix-m expansion of the exponent as $n = \sum_{i=0}^{\ell-1} d_i\, m^i$. Then

$$\begin{aligned} z &= \prod_{\substack{0\le i\le \ell-1\\ d_i=1}} x^{m^i} \cdot \prod_{\substack{0\le i\le \ell-1\\ d_i=2}} x^{2\cdot m^i} \cdots \prod_{\substack{0\le i\le \ell-1\\ d_i=j}} x^{j\cdot m^i} \cdots \prod_{\substack{0\le i\le \ell-1\\ d_i=m-1}} x^{(m-1)\cdot m^i} \\ &= \prod_{j=1}^{m-1} (R[j])^j \quad \text{where } R[j] = \prod_{\substack{0\le i\le \ell-1\\ d_i=j}} x^{m^i} \ . \end{aligned}$$

In Algorithm 3, each $R[j]$ for $j \in \{1, \dots m-1\}$ is computed and then combined in the final loop to raise each $R[j]$ to the power of j and compute the product of the results, i.e. $\prod_j (R[j])^j$.

We can note that each $R[j]$ is the product of x^{m^i} for all $i \in \{0, \dots, \ell-1\}$ where d_i is equal to j. This can be computed in any order if all the values x^{m^i} are known. Therefore, if it would be possible to precompute and store all the group elements x^{m^i}, for $i \in \{0, \dots, \ell-1\}$, they could be multiplied with the relevant $R[j]$ in an arbitrary order. A worked example of this is shown in Appendix A.

However, there would need to be enough memory available to store all ℓ group elements.

If we consider the bit lengths of variables in exponentiations that would be of use in cryptography, it is unlikely that all the values of x^{m^i}, for $i \in \{0, \ldots, \ell-1\}$, could be stored in the memory of a microprocessor.

However, if we assume that there is enough memory available to store r group elements, we can precompute and store x^{m^i}, for $i \in \{0, \ldots, r-1\}$. Suppose these are stored in an array

$$S = \{x, x^m, x^{m^2}, \ldots, x^{m^{r-1}}\},$$

and list the first r digits of the exponent as

$$D = \{d_0, d_1, d_2, \ldots, d_{r-1}\} \ .$$

If we initialise a set of registers $R[i]$, for $i \in \{1, \ldots, m-1\}$, to $1_{\mathbb{G}}$. We can treat the r values held in S and D in some arbitrary order, where, for each $j \in \{0, \ldots, r-1\}$, we compute $R[D[j]] = R[D[j]] \cdot S[j]$. The contents of $R[i]$, for $i \in \{1, \ldots, m-1\}$, will then contain the same group elements one would expect if the standard right-to-left m-ary exponentiation had treated the first r digits of the exponent (see Algorithm 3). One could then assign the next r values that would be required to continue computing the exponentiation to S and D, which then become

$$S = \{x^{m^r}, x^{m^{r+1}}, x^{m^{r+2}}, \ldots, x^{m^{2r-1}}\},$$

and

$$D = \{d_r, d_{r+1}, d_{r+2}, \ldots, d_{2r-1}\} \ .$$

These values could also be treated in some arbitrary order, as for for each $j \in \{0, \ldots, r-1\}$ we compute $R[D[j]] = R[D[j]] \cdot S[j]$. After which the values held in $R[i]$, for $i \in \{1, \ldots, m-1\}$, will be the same as if the standard right-to-left m-ary exponentiation has treated the first $2r$ digits of the exponent. This could be continued until all the digits of the exponent have been treated.

However, we can do better by noting that once $R[D[j]] = R[D[j]] \cdot S[j]$ has been computed, for a given $j \in \{0, \ldots, r-1\}$, then $D[j]$ and $S[j]$ are no longer required by the exponentiation algorithm. These values in memory can be safely overwritten. Consider again the initial state of S and D

$$S = \{x, x^m, x^{m^2}, \ldots, x^{m^{r-1}}\}, D = \{d_0, d_1, d_2, \ldots, d_{r-1}\} \ .$$

If we compute $R[D[j]] = R[D[j]] \cdot S[j]$, for some arbitrary $j \in \{0, \ldots, r-1\}$, we can replace $D[j]$ with d_r and $S[j]$ with x^{m^r}. If, for example, we take $j = 1$ then, after computing $R[D[1]] = R[D[1]] \cdot S[1]$, we can replace $D[1]$ and $S[1]$. That is,

$$S = \{x, x^{m^r}, x^{m^2}, \ldots, x^{m^{r-1}}\},$$

and

$$D = \{d_0, d_r, d_2, \ldots, d_{r-1}\} \ .$$

This could be repeated for another $D[j]$ and $S[j]$, for some arbitrary $j \in \{0, \ldots, r-1\}$, and the j-th values of S and D replaced with $x^{m^{r+1}}$ and d_{r+1} respectively. For an ℓ-bit exponent this could be repeated $\ell - r$ times, at which point one would be unable to include any new digits in D. One could then compute $R[D[j]] = R[D[j]] \cdot S[j]$ for each $j \in \{0, \ldots, r-1\}$ without replacing any of the values in D or S. After which, all of the digits of the exponent, i.e. all d_i for $i \in \{0, \ldots, \ell - 1\}$, will have been treated. A worked example of this process is given in Appendix A.

An example of how this could be implemented to produce an exponentiation algorithm where the digits are treated in some random order is shown in Algorithm 4, where we define the function `RandomInteger(`x, y`)` as returning a random integer in the interval $[x, y]$.

In order to minimise the number of operations required it is necessary to keep track of which element of S contains the largest power of x, so that x^{m^i} can be computed from $x^{m^{i-1}}$ with a minimum, and constant, number of operations. In Algorithm 4 we use the variable γ as a pointer to the largest power of x present in S.

It is interesting to note that this algorithm does not require any more group operations than Algorithm 3 so the performance should remain unaffected. However, this does assume that random values can be generated instantly. The difference in performance between Algorithm 3 and Algorithm 4 will be the time required to generate $\ell - r$ random values.

Algorithm 4 requires significantly more memory than Algorithm 3, as $(m-1) + r + 1 = m + r$ group elements need to be stored in memory. A further r radix-m digits will also need to be held in memory.

6 Security Analysis

In this section we describe how an attacker would attempt to derive digits of the exponent used in an implementation of Algorithm 4 by side channel analysis.

6.1 Simple Side Channel Analysis

Algorithm 4 is, to a certain extent, vulnerable to Simple Side Channel Analysis. It would be expected that an attacker would be able to detect when zero digits are treated. In order to counter this, the multiplication and squaring operations can be implemented such that they use identical code and, therefore, cannot be distinguished [8]. There are methods of statistically distinguishing a multiplication and a squaring operation based on the design [2] or the distribution of the bits of single-precision operations [3]. It is expected that similar attacks may be possible on a single acquisition using template attacks [7].

If an attacker is able to distinguish multiplications from squaring operations the amount of information available is limited. An attacker would be able to determine the total number of zero digits in an exponent, but only approximately

Algorithm 4: Random Order Right-to-Left m-ary Exponentiation

```
Input: $x \in \mathbb{G}$, $n \geq 1$, $r$ number of values to store in memory.
Output: $z = x^n$
Uses: $A$, $R[j]$ for $j \in \{1, \ldots, m-1\}$, $D[i]$ for $i \in \{0, \ldots, r-1\}$, $S[i]$ for
        $i \in \{0, \ldots, r-1\}$
for $j = 1$ to $m - 1$ do  $R[j] \leftarrow 1_{\mathbb{G}}$
$S[0] \leftarrow x$
for $i = 1$ to $r - 1$ do  $S[i] \leftarrow S[i-1]^m$
for $i = 0$ to $r - 1$ do
    $D[i] \leftarrow n \bmod m$
    $n \leftarrow \lfloor n/m \rfloor$
end
$\gamma \leftarrow r - 1$
while $(n > 0)$ do
    $\tau = \texttt{RandomInteger}(0, r-1)$
    if $D[\tau] \neq 0$ then
        $R[D[\tau]] \leftarrow R[D[\tau]] \cdot S[\tau]$
    end
    $S[\tau] \leftarrow S[\gamma]^m$
    $D[\tau] \leftarrow n \bmod m$
    $n \leftarrow \lfloor n/m \rfloor$
    $\gamma \leftarrow \tau$
end
for $i = r - 1$ down to $0$ do
    $R[D[i]] \leftarrow R[D[i]] \cdot S[i]$
end
$A \leftarrow R[m-1]$
for $i = m - 2$ down to $1$ do
    $R[i] \leftarrow R[i] \cdot R[i+1]$
    $A \leftarrow A \cdot R[i]$
end
return $A$
```

where they are in an exponent. The position of the first zero digit in the exponent could be determined by taking enough acquisitions that it is possible to determine the earliest in an exponentiation that a zero digit is visible in a side channel. However, it is unclear how an attacker would derive further zero digits unless they occur infrequently in an exponent, i.e. an attacker is able to locate the earliest point each zero digit is used during the computation of an exponentiation. This is unlikely to be the case, especially if the number of elements in D and S is of a similar size to the radix of the digits being taken from the exponent, in which case we would expect, statistically, there to be one zero digit in D most of the time.

In [19], Möller describes a recoding algorithm for m-ary exponentiation where each digit that is equal to zero is replaced with $-m$, and the next most significant digit is incremented by one. This leads to an exponent recoded with digits in the set $\{1,\ldots,m-1\}\cup\{-m\}$. An unsigned version of Möller's algorithm is described in [22] where the digits are recoded with digits in the set $\{1,\ldots,m\}$. Where each zero digit is replaced with m and the next digit is decremented by one, which removes the need to compute a potentially costly inversion. Both of these algorithms render a subsequent m-ary exponentiation regular, and, therefore, resistant to simple side channel analysis.

6.2 Differential Side Channel Analysis

In order to conduct a side channel attack using differential side channel analysis, an attacker needs to construct hypotheses on data being manipulated at a specific point in time for each acquisition in a set of acquisitions. An attacker can then attempt to confirm these hypotheses by computing the correlation between them and each instant in time of during the acquisitions.

In this section we describe how differential side channel analysis could be applied to Algorithm 4. We assume that an attacker has acquired sufficient traces of a side channel to conduct a differential side channel attack, as described in Section 3.2, on Algorithm 4. We also assume that the exponent has been recoded such that the digits are in the set $\{1,\ldots,m\}$, as described above.

If an attacker were to try and conduct an differential side channel attack it would be assumed that the digit treated at the ℓ-th round is the $(\ell+r-1)$-th digit of the exponent (counting from the right). This is because at this at this point the $(\ell+r-1)$-th digit will just have been included into the digits from which the algorithm will randomly select a digit to treat and will occur at this point with the highest frequency.

This can be seen if we consider that once a digit has been included, it has a probability of $1/r$ of being used in the next round. It has the same probability of being used in the following round but this can only occur if the digit was not treated in the previous round, so the probability of the digit not being used in the first round but being used in the second round is $\left(1-\frac{1}{r}\right)\frac{1}{r}$. In general, we can say that the probability of a digit being treated θ rounds after being included is $\frac{1}{r}\left(1-\frac{1}{r}\right)^{\theta-1}$. The highest probability occurs one round after a digit is included when $\theta=1$.

In [5] it is pointed out that if the correlation coefficient of η independent bits amongst δ is calculated, a partial correlation still exists and its size can be predicted as a function of the coefficient that would be generated if all the bits of δ were included. This is given as:

$$\rho_{\eta/\delta}=\rho\sqrt{\frac{\eta}{\delta}}$$

where ρ is the correlation if everything is known and $\rho_{\eta/\delta}$ is the predicted partial correlation.

This also applies to η intermediate states being correctly predicted out of a total of δ. Therefore, in conducting a side channel attack against a random order exponentiation the correlation coefficient seen would be reduced to $\sqrt{1/r}$ of the correlation coefficient that would be seen if a non-randomised algorithm were under attack if R is known. This would indicate to an attacker which previously treated digits of the exponent are equal to the $(\ell + r - 1)$-th digit.

However, the state of R will not be known to an attacker. To conduct a differential side channel attack an attacker would be obliged to form hypotheses on the number and positions of the digits with the same value as the $(\ell+r-1)$-th digit and therefore predict the state of R. An attacker would then be obliged to generate a correlation trace from the acquired traces for all of the possible values of R.

The state of R will be different for each execution since the digits that have not been treated at the ℓ-th round will vary from one execution to another. An attacker would, therefore, be obliged to predict the most likely state of R and assume this is the state for every acquisition. This will further reduce the correlation coefficient visible via a differential side channel attack.

An attacker can, therefore, expect to be able to derive a correlation whose size is reduced to $\sqrt{1/r}$ of what one would be able to produce when attacking a naïve implementation in the first round of the algorithm. In subsequent rounds the largest correlation an attacker could hope to produce will diminish rapidly. It is not clear exactly how the correlation will diminish, and it is left as open problem, as to exactly how an attacker would attempt to derive the exponent via differential side channel analysis.

7 Conclusion

In this paper we present a method of computing an exponentiation where radix-m digits can be treated in a random order. The algorithm is intended to provide resistance to side channel analysis, and some informal arguments are given as to the side channel analysis resistance of this algorithm. However, we can note that not all the possible orderings of exponent digits will be equally likely.

Algorithm 4 will most likely find use as a supplement, rather than as a replacement, to the blinding countermeasure described in Section 4.1. This is because it may be possible to derive the digits used in an m-ary exponentiation from one trace using template attacks [7]. In the example given, $d+\phi(m)$ gives a value equivalent to the private key and could be used to start to make attempts at factoring m. If Algorithm 4 were also used, an attacker would have to test all the possible orderings of the exponent to find $d+\phi(m)$. This is an advantage over the previously proposed randomised exponentiation algorithms [12, 23], described in Section 4.2, that would provide a value equivalent to the exponent if one were able to derive the digits of an exponent from one acquisition.

Another advantage over previously proposed countermeasures [12, 23], is that the digit size can be chosen such that it evenly divides a computer word. It is, therefore, not necessary to read digits that will have bits on two computer words, which has implications for both security and efficiency.

As stated in the introduction, Algorithm 4 requires a large amount of memory, as Algorithm 4 requires $m + r$ group elements to be stored in memory. There are 32-bit secure microprocessors that have enough memory to allow exponentiation in $\mathbb{F}_p$ or $\mathbb{F}_{2^q}$ to be implemented [4, 18]. More recently, processors with larger memories are being studied with regard to their side channel resistance [1], on which one would be able to implement Algorithm 4 in $(\mathbb{Z}/N\mathbb{Z})^*$. This is unlikely to be possible on a smart card microprocessor unless a cryptographic coprocessor with a large amount of registers is included.

The side channel resistance of the algorithm proposed in this paper is only briefly analysed. There are some open problems that arise from this work.

The most obvious question is exactly how one would mount a side channel attack against Algorithm 4. Some brief details are given, but the magnitude of the correlation coefficient one would expect to be able to observe for a given r is not defined. Indeed, it remains to be shown what values of r would provide a suitable level of random ordering.

Another question is how one would attack Algorithm 4 if it were combined with other countermeasures. For example, one could easily combine Algorithm 4 with Walter's MIST algorithm [23] where the radix of the digits read from an exponent is also randomised.

Acknowledgements

The author would like to thank Elisabeth Oswald and Dan Page for their helpful comments related to the ideas presented in this paper. The author would also like to thank Steven Galbraith for his help in preparing the final version of this paper. The work described in this paper has been supported in part by the European Commission IST Programme under Contract IST-2002-507932 ECRYPT and EPSRC grant EP/F039638/1 "Investigation of Power Analysis Attacks".

References

1. Side-channel attack standard evaluation board (SASEBO). Available at: `http://www.rcis.aist.go.jp/special/SASEBO`.
2. T. Akishita and T. Takagi. Power analysis to ECC using differential power between multiplication and squaring. In J. Domingo-Ferrer, J. Posegga, and D. Schreckling, editors, *Smart Card Research and Advanced Applications — CARDIS 2006*, volume 3928 of *Lecture Notes in Computer Science*, pages 151–164. Springer-Verlag, 2006.
3. F. Amiel, B. Feix, M. Tunstall, C. Whelan, and W. P. Marnane. Distinguishing multiplications from squaring operations. In *Selected Areas in Cryptography — SAC 2008*, Lecture Notes in Computer Science. Springer-Verlag, To appear.
4. ARM. SecurCore family. `http://www.arm.com/products/CPUs/families/SecurCoreFamily.html`.
5. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer-Verlag, 2004.

6. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards approaches to counteract power-analysis attacks. In M. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag, 1999.
7. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. Kaliski Jr., C. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer-Verlag, 2002.
8. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
9. C. Clavier and M. Joye. Universal exponentiation algorithm. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 300–308. Springer-Verlag, 2001.
10. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In C. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 99*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer-Verlag, 1999.
11. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In C. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, 2001.
12. K. Itoh, J. Yajima, M. Takenaka, and N. Torii. DPA countermeasures by improving the window method. In B. Kaliski, C. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 303–317. Springer-Verlag, 2002.
13. D. E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, 2nd edition, 1981.
14. P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.
15. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
16. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks — Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.
17. T. S. Messerges. *Power Analysis Attacks and Countermeasures for Cryptographic Algorithms*. PhD thesis, University of Illinois, Chicago, 2000.
18. MIPS-Technologies. SmartMIPS ASE. `http://www.mips.com/content/Products/`.
19. B. Möller. Securing elliptic curve point multiplication against side-channel attacks. In G. Davida and Y. Frankel, editors, *Information Security — ISC 2001*, volume 2200 of *Lecture Notes in Computer Science*, pages 324–334. Springer-Verlag, 201.
20. J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards — E-smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2001.

21. R. Rivest, A. Shamir, and L. M. Adleman. Method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
22. C. Vuillaume and K. Okeya. Flexible exponentiation with resistance to side channel attacks. In J. Zhou, M. Yung, and F. Bao, editors, *Applied Cryptography and Network Security — ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2006.
23. C. D. Walter. MIST: An efficient, randomized exponentiation algorithm for resisting power analysis. In B. Preneel, editor, *Topics in Cryptology — CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 53–66. Springer-Verlag, 2002.
24. A. C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

A Worked Example

We wish to compute $z = x^n$, where n is set to 738530 (`B44E2` in hexadecimal) and the digits will be read from this exponent two bits at a time, i.e. $m = 4$. The digits and powers of x (variable A) computed in the main loop of Algorithm 3 are shown below. If enough memory were available then x^{m^i} for $i \in \{0, \ldots, 9\}$ could all be precomputed. Then we can set

$$S = \{x, x^4, x^{16}, x^{64}, x^{256}, x^{1024}, x^{4096}, x^{16384}, x^{65536}, x^{262144}\},$$

and

$$D = \{2, 0, 2, 3, 0, 1, 0, 1, 3, 2\} \ .$$

The exponentiation can be computed by treating the elements of S and D in an arbitrary order. We initially set $R[j]$ for $j \in \{1, 2, 3\}$ to $1_{\mathbb{G}}$. An arbitrary $j \in \{0, \ldots, 9\}$ is chosen, and we compute $R[D[j]] = R[D[j]] \cdot S[j]$ except when $D[j]$ is equal to zero when no operation is performed. This is repeated once for each possible value of j, which will result in:

$$\begin{aligned} R[1] &= S[5] \cdot S[7] = x^{1024}\, x^{16384} = x^{17408} \\ R[2] &= S[0] \cdot S[2] \cdot S[9] = x\, x^{16}\, x^{262144} = x^{262161} \\ R[3] &= S[3] \cdot S[8] = x^{64}\, x^{65536} = x^{65600} \end{aligned}$$

where $z = R[1] \cdot R[2]^2 \cdot R[3]^3 = x^{17408}\, x^{2 \cdot 262161}\, x^{3 \cdot 65600} = x^{738530}$, which is computed by the final loop in Algorithm 3.

If the above computation of $z = x^n$ were conducted using Algorithm 4, where we set $r = 6$ so that there is only enough memory to store six group elements in S. The initial values in memory would be x^{m^i} for $i \in \{0, \ldots, 5\}$, i.e.

$$S = \{x, x^4, x^{16}, x^{64}, x^{256}, x^{1024}\},$$

and the corresponding digits of the exponent would be

$$D = \{2, 0, 2, 3, 0, 1\} \ .$$

Again, initially set $R[j]$ for $j \in \{1,2,3\}$ to $1_{\mathbb{G}}$. An arbitrary $j \in \{0,\ldots,5\}$ is chosen, and we compute $R[D[j]] = R[D[j]] \cdot S[j]$. As above, all $R[j]$ for $j \in \{1,2,3\}$ are initialised to $1_{\mathbb{G}}$. We will take $j = 3$ which give $S[3] = x^{64}$ and $D[3] = 3$, and we compute $R[3] = R[3] \cdot x^{64}$. The values in $S[3]$ and $D[3]$ are no longer required and can be replaced. We, therefore set $S[3] = x^{m^7} = x^{4096}$ and $D[3] = 0$. The values contained in memory would then be

$$S = \{x, x^4, x^{16}, x^{4096}, x^{256}, x^{1024}\}\,,$$

and

$$D = \{2, 0, 2, 0, 0, 1\}\ .$$

Another arbitrary $j \in \{0,\ldots,5\}$ can then be selected. We will take $j = 0$, which will mean we will compute $R[2] = R[2] \cdot x$. After which $S[0]$ and $D[0]$ can be replaced with x^{m^8} and 1 respectively, giving

$$S = \{x^{16384}, x^4, x^{16}, x^{2048}, x^{256}, x^{1024}\}\,,$$

and

$$D = \{1, 0, 2, 0, 0, 1\}\ .$$

Next, we take $j = 4$. $D[4]$ is equal to zero, so no operation is conducted with $S[4]$, and these elements can be replaced with 3 and x^{m^9}, giving

$$S = \{x^{16384}, x^4, x^{16}, x^{4096}, x^{65536}, x^{1024}\}\,,$$

and

$$D = \{1, 0, 2, 0, 3, 1\}\ .$$

We now take $j = 1$. Again, the chosen digit, $D[1]$, is equal to zero and no operation takes place. $S[1]$ and $D[1]$ can be replaced with $x^{m^{10}}$ and the last digit of the exponent, giving

$$S = \{x^{16384}, x^{262144}, x^{16}, x^{4096}, x^{65536}, x^{1024}\}\,,$$

and

$$D = \{1, 2, 2, 0, 3, 1\}\ .$$

There are now no remaining digits that could be D, so there is no further need to select digits to be replaced. The remaining digits can be treated, where for each $j \in \{0,\ldots,5\}$ we compute $R[D[j]] = R[D[j]] \cdot S[j]$ except when $D[j]$ is equal to zero when no operation is performed. This can be performed in an arbitrary order and will result in:

$$\begin{aligned}
R[1] &= S[7] \cdot S[5] = x^{16384}\, x^{1024} = x^{17408} \\
R[2] &= S[0] \cdot S[9] \cdot S[2] = x\, x^{262144}\, x^{16} = x^{262161} \\
R[3] &= S[3] \cdot S[8] = x^{64}\, x^{65536} = x^{65600}
\end{aligned}$$

This is exactly the same result as given where all the x^{m^i} for $i \in \{0,\ldots,9\}$ are precomputed. The only difference being the order in which the x^{m^i}, for $i \in \{1,\ldots,10\}$, are multiplied together. The final stage is the same as described above, since $z = R[1] \cdot R[2]^2 \cdot R[3]^3 = x^{17408}\, x^{2\cdot 262161}\, x^{3\cdot 65600} = x^{738530}$.