Exponent Recoding and Regular Exponentiation Algorithms

Marc Joye[1] and Michael Tunstall[2]

[1] Thomson R&D France
Technology Group, Corporate Research, Security Laboratory
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
marc.joye@thomson.net – http://www.geocities.com/MarcJoye/
[2] 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 methods of recoding exponents to allow for regular implementations of m-ary exponentiation algorithms. Recoding algorithms previously proposed in the literature do not lend themselves to being implemented in a regular manner, which is required if the implementation needs to resist side-channel attacks based on simple power analysis. The advantage of the algorithms proposed in this paper over previous work is that the recoding can be readily implemented in a regular manner. Recoding algorithms are proposed for exponentiation algorithms that use both signed and unsigned exponent digits.

Keywords: Exponent recoding, exponentiation algorithms, side-channel analysis.

1 Introduction

Exponentiation algorithms have been shown to be vulnerable to side-channel analysis, where an attacker observes the power consumption [17] or electromagnetic emanations [9, 24]. These attacks are referred to as Simple Power Analysis (SPA) and Simple Electromagnetic Analysis (SEMA). A naïvely implemented exponentiation algorithm will reveal the exponent used, as the operations required are dependent on the (bitwise) representation of the exponent.

Recoding algorithms have been proposed to decrease the number of operations required to compute an exponentiation [4, 15]. However, these recoding algorithms are designed to produce efficient exponentiation algorithms, and not to produce side-channel resistant exponentiation algorithms.

There are several recoding algorithms that have been proposed in the literature [19–23, 26, 27] in order to thwart SPA or SEMA. However, as noted in [25], to achieve a regular exponentiation algorithm any recoding that is used also needs to be regular. We define "regular" to mean that an algorithm executes the same instructions in the same order for any input values. There is, therefore, no leakage through simply inspecting a side-channel. It could be argued that this recoding could be done when an exponent is generated. However, if an exponent is combined with a random value to prevent differential side-channel analysis, as detailed in [7, 16], the recoding will have to be conducted just prior to the exponentiation algorithm.

Algorithms are proposed for signed and unsigned exponentiation algorithms. They are described in general terms and can be readily adapted for use in $(\mathbb{Z}/N\mathbb{Z})^*$, $\mathbb{F}_p^*$, etc. It should be noted that differential side channel analysis and the required countermeasures are not considered in this paper (the interested reader is referred to [18] for a discussion of this topic). Only exponentiation algorithms that lend themselves to being implemented with a regular structure are considered in this paper. For a more generic survey of (fast) exponentiation algorithms, the interested reader is referred to [11].

The rest of this paper is organised as follows. In the next section, we review basic methods for evaluating an exponentiation. In Section 3, we explain how exponent recoding may prevent certain side-channel attacks and present new recoding methods. For further efficiency, we consider exponents recoded to both signed and unsigned digits. In Section 4, we describe (higher-radix) right-to-left exponentiation methods and point out why they are more suited for secure implementations. Finally, we conclude in Section 5. Specific applications of our recoding methods can be found in Appendix A.

2 Basic Methods

2.1 Square-and-multiply method

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. This algorithm is detailed in Algorithm 1, where we define the function $\mathrm{bit}(n, i)$ as a function returning the i^{th} bit of n.[3] 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 $\mathbb{G}$.

Algorithm 1: 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 ($\mathrm{bit}(n, i) \neq 0$) then  $A \leftarrow A \cdot R$
end

return $A$
```

[3] By convention, the first bit is bit number 0.

It has been shown that bit values of exponent n can be distinguished by observing a suitable side channel, such as the power consumption [17] or electromagnetic emanations [9, 24]. One countermeasure to prevent an attacker from being able to recover the bit values of an exponent is to execute the same code (i.e., without conditional branchings) whatever the value of exponent n, referred to as side-channel atomicity [5]. Specific implementations described in [5] assume that the multiplications and squaring operations behave similarly. However, it has been shown that squaring operations and multiplications may be distinguished by differences in the power consumption of hardware implementations [1], or the distribution of the Hamming weight of the result of the single precision operations required to compute multi-precision operations [2].

2.2 Square-and-multiply-always method

The first regular exponentiation algorithm was proposed in [7], where a multiplication is performed for each bit of an exponent. The value of the bit in question determines whether the result is used or discarded.

Algorithm 2: Square-and-Multiply-*Always* Algorithm

```
Input: $x \in \mathbb{G}$, $n \geq 1$, $\ell'$ the binary length of $n$
Output: $z = x^n$
$R[0] \leftarrow x$; $R[1] \leftarrow x$; $R[2] \leftarrow x$
for $i = \ell' - 2$ down to $0$ do
    $R[1] \leftarrow R[1]^2$
    $b \leftarrow \mathrm{bit}(n, i)$; $R[b] \leftarrow R[b] \cdot R[2]$
end
return $R[1]$
```

This algorithm is less efficient than many other exponentiation algorithms. More importantly, it has been shown that this algorithm can be attacked by a safe-error fault attack [29, 30]. If a fault is injected into a multiplication, it will change the output only if the result of that multiplication is used. An attacker could potentially use this to determine bits of the exponent by targeting chosen multiplications. A description of fault attacks that could potentially be applied to an exponentiation is beyond the scope of this paper (see [3, 10]). However, it would be prudent to avoid algorithms that are vulnerable to such attacks.

2.3 m-ary exponentiation

In order to speed up the evaluation of $z = x^n$ it is possible to precompute some values that are small multiples of x by breaking up the exponent into ℓ words in radix m [15]. 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 3, where function digit(n, i) returns the i^{th} digit of n (in radix m).

```
Algorithm 3: 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 $i = 2$ to $m - 1$ do  $R[i] \leftarrow R[i-1] \cdot x$
  $d \leftarrow \mathrm{digit}(n, \ell - 1)$; $A \leftarrow R[d]$
  for $i = \ell - 2$ down to $0$ do
      $A \leftarrow A^m$
      $d \leftarrow \mathrm{digit}(n, i)$
      if $(d \neq 0)$ then  $A \leftarrow A \cdot R[d]$
  end
  return $A$
```

This algorithm is more efficient than the square-and-multiply algorithm and has the advantage that it is more regular and will, therefore, leak less information. The algorithm is not entirely regular since no multiplication is conducted when a digit is equal to zero. It would, therefore, be expected that an attacker could determine the digits of n that are equal to zero by observing a suitable side channel.

3 Exponent Recoding

Several methods for recoding exponents have been proposed in the literature. The most commonly known example is Non-Adjacent Form (NAF) recoding [4] that recodes the bits of an exponent with values in $\{-1, 0, 1\}$. This reduces the number of multiplications that are required in the subsequent exponentiation algorithm, and can be generalised to a m-ary recoding [15]. Some methods of NAF recoding to produce a regular exponentiation algorithm have been proposed as a countermeasure to side-channel analysis [22, 23]. However, these recoding algorithms have been shown to be vulnerable to attack, since the recoding algorithm is not itself regular [25].

Other recoding algorithms have been proposed that make the subsequent exponentiation algorithm regular. 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 $\{1, \ldots, m-1\} \cup \{-m\}$. Combined with the m-ary exponentiation algorithm, this implies that x^{-m} should be precomputed. While this computation is "easy" on elliptic curves, this is not the case for the multiplicative group of integers modulo N. An unsigned version of Möller's algorithm is described in [27] where the digits are recoded in the set $\{1, \dots, m\}$: each zero digit is replaced with m and the next digit is decremented by one.

The problem with the recoding algorithms proposed in [19, 27] is that they cannot easily be implemented in a regular manner. In this section we present some recoding methods for regular exponentiation, where the exponent can be simply recoded in a regular fashion.

3.1 Unsigned-digit recoding

The goal is to compute $z = x^n$ for some integer n. Let $n = \sum_{i=0}^{\ell-1} d_i\, m^i$ denote the expansion of n in radix m (typically, as above, $m = 2^k$). Consider positive integer $s < n$ and define $n' := n - s$. If $n' = \sum_{i=0}^{\ell-1} d'_i\, m^i$ and $s = \sum_{i=0}^{\ell-1} s_i\, m^i$ respectively denote the m-expansion of n' and s, it follows that

$$\begin{aligned} x^n &= x^{n'+s} \\ &= x^{\sum_{i=0}^{\ell-1} \kappa_i\, m^i} \quad \text{with } \kappa_i := d'_i + s_i \ . \end{aligned}$$

We define the most significant digit of s in radix m to be zero. This means that the significant digit of n' in radix m (i.e., $\kappa_{\ell-1}$) will remain greater than, or equal to, zero. Otherwise, the recoding would no longer be unsigned and would not be suitable for groups where computing inversions is expensive.

There are several possible realisations of this idea. We detail below two such implementations.

First implementation Choose $s = \sum_{i=0}^{\ell-2} m^i$. This can be seen as setting all digits of s to 1, namely $s = (1, \dots, 1)_m$. Since $n'_i \in \{0, \dots, m-1\}$, it follows that $\kappa_i \in \{1, \dots, m\}$. We, therefore, obtain the following algorithm.

Algorithm 4: Unsigned-Digit Recoding Algorithm (I)

```
Input: $n \geq 1$, $m = 2^k$, $\ell$ the $m$-ary length of $n$
Output: $n = (\kappa_{\ell-1}, \dots, \kappa_0)_m$ with $\kappa_i \in \{1, \dots, m\}$, $0 \leq i \leq \ell - 2$

$s \leftarrow (1, \dots, 1)_m$; $n \leftarrow n - s$
for $i = 0$ to $\ell - 2$ do
    $d \leftarrow n \bmod m$; $n \leftarrow \lfloor n/m \rfloor$
    $\kappa_i \leftarrow d + 1$
end
$\kappa_{\ell-1} \leftarrow n$
```

It is interesting to note that recoding with digits in the set $\{1, \dots, m\}$ is unique. Indeed, suppose that $n = \sum_i \kappa_i\, m^i = \sum_i \kappa^*_i\, m^i \geq m + 1$. This implies $\kappa_0 \equiv \kappa^*_0 \pmod{m}$, which in turn yields $\kappa_0 = \kappa^*_0$ since, given their definition range, $|\kappa_0 - \kappa^*_0| \leq m - 1$. The same argument applies to $n \leftarrow (n - \kappa_0)/m$

which implies $\kappa_1 = \kappa_1^*$ and so on. Therefore, the recoded digits obtained by the algorithm in [27] correspond to those obtained by Algorithm 4. But, contrarily to [27], the proposed algorithm is *itself* regular.

Example 1. Consider, for example, an exponent $n = 31415$ whose binary representation is given by $(1,1,1,1,0,1,0,1,0,1,1,0,1,1,1)_2$. For $m = 2$, Algorithm 4 produces the equivalent representation $(2,2,2,1,2,1,2,1,2,2,2,1,1,1)_2$ in radix 2 with digits in the set $\{1,2\}$. In radix $m = 4$, it produces $(1,3,2,2,2,3,1,3)_4$ with digits in the set $\{1,2,3\}$.

A possible drawback of Algorithm 4 is that it requires the knowledge of the m-ary length of n (i.e., of ℓ) ahead of time. We describe hereafter an alternative implementation overcoming this limitation.

Second implementation Looking in more detail into the subtraction step, we have

$$d_i' = (d_i - s_i + \gamma_i) \bmod m \quad \text{and} \quad \gamma_{i+1} = \left\lfloor \frac{d_i - s_i + \gamma_i}{m} \right\rfloor \in \{-1, 0\}$$

where the "borrow" is initialised to 0 (i.e., $\gamma_0 = 0$). This is the classical schoolboy algorithm. Since $d_i, s_i \in \{0, \dots, m-1\}$, we get

$$\kappa_i = d_i' + s_i = \begin{cases} d_i + \gamma_i & \text{if } d_i + \gamma_i \geq s_i\,, \\ d_i + \gamma_i + m & \text{otherwise}\,. \end{cases}$$

Hence, we see that any choice for s_i with $s_i \neq 0$ when $d_i \in \{0,1\}$ leads to a non-zero value for κ_i. As in our first implementation, we choose $s = \sum_{i=0}^{\ell-2} m^i$. Further, to only resort to unsigned arithmetic, we define $\gamma_i' = \gamma_i + 1$ where $\gamma_i \in \{0,1\}$.

Algorithm 5: Unsigned-Digit Recoding Algorithm (II)

```
Input: $n \geq 1$, $m = 2^k$
Output: $n = (\kappa_{\ell-1}, \dots, \kappa_0)_m$ with $\kappa_i \in \{1, \dots, m\}$, $0 \leq i \leq \ell - 2$

$i \leftarrow 0$; $\gamma' \leftarrow 1$
while ($n \geq m + 1$) do
    $d \leftarrow n \bmod m$; $d' \leftarrow d + \gamma' + m - 2$
    $\kappa_i \leftarrow (d' \bmod m) + 1$; $\gamma' \leftarrow \lfloor d'/m \rfloor$
    $n \leftarrow \lfloor n/m \rfloor$
    $i \leftarrow i + 1$
end
$\kappa_i \leftarrow n + \gamma' - 1$
```

3.2 Signed-digit recoding

Again, we let $m = 2^k$. The goal is to rewrite the exponent into digits that take odd values in $\{-(m-1), \ldots, -1, 1, \ldots, m-1\}$,

$$n = \sum_{i=0}^{\ell-1} \kappa_i \, m^i \quad \text{with } \kappa_i \in \{\pm 1, \ldots, \pm(m-1)\} \text{ and } \kappa_i \text{ odd} \ .$$

An m-ary exponentiation algorithm using these signed digits would require the same amount of precomputed values as the unsigned version. When computing inverses is easy (for example on elliptic curves), it has been suggested that the exponent digits κ_i could be broken up into an unsigned integer value and the sign, i.e. $\kappa_i = s_i \tau_i$ where τ_i is unsigned and $s_i \in \{1, -1\}$ [12, 20, 21, 26]. In this case, only the values multiplied by the values in $\{1, \ldots, m-1\}$ need to be computed, and the inversion applied as required. The advantage of using this method is that the number of precomputed values is halved.

We rely on the observation that any odd integer in the range $[0, 2m)$ can be written as

$$\begin{aligned}
1 &= m + (-(m-1)) \\
3 &= m + (-(m-3)) \\
&\vdots \\
m-1 &= m + (-1) \\
m+1 &= m + (1) \\
&\vdots \\
2m-3 &= m + (m-3) \\
2m-1 &= m + (m-1)
\end{aligned}$$

which yields the following algorithm.

Algorithm 6: (Odd) Signed-Digit Recoding Algorithm

```
**Input**: $n$ odd, $m = 2^k$
**Output**: $n = (\kappa_{\ell-1}, \ldots, \kappa_0)_{\pm m}$ with $\kappa_i \in \{\pm 1, \ldots, \pm(m-1)\}$ and $\kappa_i$ odd

$i \leftarrow 0$
**while** $(n > m)$ **do**
    $\kappa_i \leftarrow (n \bmod 2m) - m$
    $n \leftarrow (n - \kappa_i)/m$
    $i \leftarrow i + 1$
**end**
$\kappa_i \leftarrow n$
```

The correctness of the recoding follows by inspecting that for an odd integer $n > m$, we have:

1. $|\kappa_i| \in [1, m-1]$: $1 - m \leq (n \bmod 2m) - m \leq m - 1$;
2. $(n \bmod 2m) - m$ is odd: $(n \bmod 2m) - m \equiv n \equiv 1 \pmod 2$;
3. $(n - \kappa_i)/m < n$: for $n > m > 1$, $n + m \leq nm \implies (n + m - 1)/m < n$.

Moreover, it is easy to see that the updating step, $n \leftarrow (n - \kappa_i)/m$, does not change the parity:

$$(n - \kappa_i)/m \bmod 2 = \frac{(n - \kappa_i) \bmod 2m}{m} = \frac{m \bmod 2m}{m} = 1 \ .$$

Example 2. Again, with the example of exponent $n = 31415$, Algorithm 6 produces the equivalent representation $(1, 1, 1, 1, 1, \overline{1}, 1, \overline{1}, 1, \overline{1}, 1, 1, \overline{1}, 1, 1)_{\pm 2}$ for $m = 2$. For $m = 4$, it produces $(1, 3, 3, \overline{1}, \overline{1}, \overline{1}, 1, 3)_{\pm 4}$.

Algorithm 6 requires that n is odd. It is noted in [13] that when n is even it can be replaced with $n' = n+1$ and that the result is multiplied by the inverse of the input, i.e. x^{-1}. To make this regular for any exponent an equivalent treatment can be conducted when n is odd. In this case, n is replaced with $n' = n + 2$ and the result is multiplied by x^{-2}.

It is also possible to modify Algorithm 6 so that the length of the recoded integer is fixed. This allows one to hide the Hamming weight, by setting consecutive digits more significant than the most significant word in base m to $-m$ and one, as described in [19]. This, assuming an even number of digits are available, will not change the output of the exponentiation algorithm.

4 Right-to-left Exponentiation Algorithms

4.1 Right-to-left m-ary exponentiation

While it is well-known that the square-and-multiply algorithm can be improved by using a higher radix, it is sometimes believed that such improvements only apply to left-to-right versions (e.g., in [6, p. 10]). In [15, §4.6.3], Knuth suggests as an exercise to design an exponentiation algorithm that is analogous to the right-to-left binary method, but based on a general radix m.

For $m = 2$, the right-to-left binary method works as follows. Let $n = \sum_{i=0}^{\ell'-1} b_i \, 2^i$ with $b_i \in \{0, 1\}$. The algorithm evaluates $z = x^n$ in $\mathbb{G}$ as

$$z := x^{\sum_{i=0}^{\ell'-1} b_i \, 2^i} = \prod_{\substack{0 \leq i \leq \ell'-1 \\ b_i \neq 0}} x^{2^i} = \prod_{\substack{0 \leq i \leq \ell'-1 \\ b_i \neq 0}} Y_i \quad \text{with} \begin{cases} Y_0 = x \\ Y_i = Y_{i-1}^2 \,, \quad i \geq 1 \end{cases} .$$

Algorithm 7: Right-to-left m-ary Exponentiation

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

// Step 1: Evaluation of $T_j$ for $1 \leq j \leq m-1$
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
$R[n] \leftarrow R[n] \cdot A$

// Step 2: Evaluation of $z = \prod_{j=1}^{m-1} \left(T_j\right)^j$
$A \leftarrow R[m-1]$
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 $A$
```

Likewise, for a general radix m, we can consider the m-ary expansion of n, $n = \sum_{i=0}^{\ell-1} d_i \, m^i$ where $0 \leq d_i < m$. We can then write $z = x^n$ as

$$\begin{aligned} z &= \prod_{\substack{0 \leq i \leq \ell-1 \\ d_i = 1}} x^{m^i} \cdot \prod_{\substack{0 \leq i \leq \ell-1 \\ d_i = 2}} x^{2 \cdot m^i} \cdots \prod_{\substack{0 \leq i \leq \ell-1 \\ d_i = j}} x^{j \cdot m^i} \cdots \prod_{\substack{0 \leq i \leq \ell-1 \\ d_i = m-1}} x^{(m-1) \cdot m^i} \\ &= \prod_{j=1}^{m-1} \left(T_j\right)^j \quad \text{where } T_j = \prod_{\substack{0 \leq i \leq \ell-1 \\ d_i = j}} x^{m^i} \quad . \end{aligned}$$

Therefore, the evaluation of $z = x^n$ in $\mathbb{G}$ amounts to evaluating the T_j's followed by $\prod_j \left(T_j\right)^j$. It is worth noting that for a given index i, only one T_j is affected (i.e. T_j is only affected by $j = d_i$).

In Algorithm 7, we use $(m-1)$ temporary variables, $R[1], \ldots, R[m-1]$, each of them initialised to $1_{\mathbb{G}}$, the identity element of $\mathbb{G}$. At the end of the loop, $R[j]$ will contain the value of T_j. We also make use of an accumulator A that keeps track of the successive values of x^{m^i}. For $i = 0, \ldots, \ell-1$, if at iteration i, digit d_i of n is equal to d then, provided that $d \neq 0$, $R[d]$ is updated as $R[d] \leftarrow R[d] \cdot A$. Accumulator A is initialised to x and, at iteration i, it is updated by computing $A \leftarrow A^m$ so that it contains the value of $x^{m^{i+1}}$ for the next iteration. To avoid the useless computation of x^{m^ℓ} (i.e., when $i = \ell-1$), we stop the loop at iteration $\ell - 2$, and only update $R[d_{\ell-1}]$ as $R[d_{\ell-1}] \leftarrow R[d_{\ell-1}] \cdot A$ for the last iteration.

It now remains to compute $\prod_{j=1}^{m-1}\bigl(R[j]\bigr)^j$ to get the value of $z = x^n$, as expected. This can be done with only $(2m-4)$ multiplications in $\mathbb{G}$. Indeed, letting $U_{\bar{j}} := \prod_{j=\bar{j}}^{m-1} T_j$ and $V_{\bar{j}} := U_{\bar{j}} \cdot \prod_{j=\bar{j}}^{m-1}\bigl(T_j\bigr)^{j-\bar{j}}$, we observe that $V_1 = \prod_{j=1}^{m-1}\bigl(T_j\bigr)^j = z$ and it is easy to check that

$$U_{\bar{j}} = T_{\bar{j}} \cdot U_{\bar{j}+1} \quad \text{and} \quad V_{\bar{j}} = V_{\bar{j}+1} \cdot U_{\bar{j}} \ .$$

Consequently, $\prod_{j=1}^{m-1}\bigl(R[j]\bigr)^j$ can be evaluated by using accumulator A initialised to $R[m-1]$ and then containing the successive values of V_j for $j = m-2, \dots, 1$. Further, as the content of $R[j]$ (i.e., T_j) is only needed in iteration j for the evaluation of U_j, $R[j]$ can be used to store the value of U_j. Accumulator A is so updated as $A \leftarrow A \cdot R[j]$.

4.2 Right-to-left vs. left-to-right exponentiation

As with the left-to-right m-ary algorithm, it would be expected that an attacker would be able to determine the digits of n that are equal to zero in Algorithm 7. An easy way to prevent this is to treat the digit 0 as the other digits. Namely, we can remove the "**if** $(d \neq 0)$ **then**" by using an additional temporary variable $R[0]$. However, as for the left-to-right version, the resulting implementation would become vulnerable to a safe-error attack. Alternatively, the recent attack of [14] can be extended to recover the zero digits of n. All these attacks can be prevented by making use of the recoding methods described in Section 3. This will be exemplified by the algorithms detailed in Appendix A.

Overall, right-to-left exponentiation methods are superior than their left-to-right counterparts, from a security viewpoint. Known left-to-right exponentiations repeatedly multiply the accumulator by the *same* precomputed value for a non-zero exponent digit. While this can be advantageous from an efficiency viewpoint (for example when the precomputed values feature some properties allowing for a faster multiplication), this can have dramatic consequences from a security viewpoint. In [28], Walter exploits the fact the same values are used to mount what he calls "Big Mac" attacks. Surprisingly, Big Mac attacks are more powerful when there are more precomputed values (i.e. when the algorithms are faster). Right-to-left exponentiation methods are not subject to these attacks.

Another class of attacks exploiting specific properties of left-to-right methods relies on collisions for carefully chosen inputs. These attacks were introduced by Fouque and Valette [8] and subsequently extended by Yen et al. [31]. The most general presentation of these attacks with several extensions applying to all known left-to-right methods (including the Montgomery ladder) is given by Homma et al. [14]. Although not described in [14], as aforementioned, it is an easy exercise to show that a similar attack can be extended to recover the *zero* digits of the exponent — but not all the digits as in the left-to-right algorithms — in the *basic* right-to-left m-ary exponentiation algorithm (Algorithm 7).

5 Conclusion

In this paper, we describe regular recoding algorithms to aid in the implementation of regular m-ary exponentiation algorithms. If a recoding algorithm is not regular then it may itself become subject to a side channel attack, as noted in [25]. In this paper we detail secure implementations for recoding exponents into signed and unsigned digits. The side-channel resistance of the left-to-right m-ary exponentiation algorithm is compared to that of the right-to-left m-ary exponentiation algorithm. This provides a base for a side channel resistant implementation of an exponentiation algorithm, but further countermeasures will need to be included to prevent differential side channel analysis. (Some examples of how the recoded exponents can be used to provide regular exponentiation algorithms are presented in the appendix.)

Acknowledgements

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. Toru Akishita and Tsuyoshi 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.
2. Frédéric Amiel, Benoît Feix, Michael Tunstall, Claire Whelan, and William P. Marnane. Distinguishing multiplications from squaring operations. In *Selected Areas in Cryptography — SAC 2008*, Lecture Notes in Computer Science. Springer-Verlag, To appear.
3. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer’s apprentice guide to fault attacks. *Proceedings the IEEE*, 94(2):370–382, 2006. Earlier version in Proc. of FDTC 2004.
4. Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1999.
5. Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
6. Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1993.
7. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç.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.

8. Pierre-Alain Fouque and Frédéric Valette. The doubling attack — Why upwards is better than downwards. In C.D. Walter, Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 269–280. Springer-Verlag, 2003.
9. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. 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 251–261. Springer-Verlag, 2001.
10. Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In J.-J. Quisquater et al., editors, *Smart Card Research and Advanced Applications VI (CARDIS 2004)*, pages 159–176. Kluwer, 2004.
11. Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.
12. Mustapha Hédabou, Pierre Pinel, and Lucien Bénéteau. A comb method to render ECC resistant against side channel attacks. Report 2004/342, Cryptology ePrint Archive, 2004. `http://eprint.iacr.org/`.
13. Mustapha Hédabou, Pierre Pinel, and Lucien Bénéteau. Countermeasures for preventing comb method against SCA attacks. In R.H. Deng et al., editors, *Information Security Practice and Experience — ISPEC 2005*, volume 3439 of *Lecture Notes in Computer Science*, pages 85–96. Springer-Verlag, 2005.
14. Naofumi Homma, Atsushi Miyamoto, Takafumi Aoki, Akashi Satoh, and Adi Shamir. Collision-based power analysis of modular exponentiation using chosen-message pairs. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2008.
15. Donald E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, 2nd edition, 1981.
16. Paul 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.
17. Paul Kocher, Joshua Jaffe, and Benjamin 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.
18. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
19. Bodo 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, 2001.
20. Bodo Möller. Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In A.H. Chan and V. Gligor, editors, *Information Security (ISC 2002)*, volume 2433 of *Lecture Notes in Computer Science*, pages 402–413. Springer-Verlag, 2002.
21. Bodo Möller. Fractional windows revisited: Improved signed-digit representation for effcient exponentiation. In C. Park and S. Chee, editors, *Information Security and Cryptology — ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 137–153. Springer-Verlag, 2004.
22. Katsuyuki Okeya and Tsuyoshi Takagi. A more flexible countermeasure against side channel attacks using window method. In C.D. Walter, Ç.K. Koç, and C. Paar,

editors, *Cryptographic Hardware and Embedded Systems — CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 397–410. Springer-Verlag, 2003.
23. Katsuyuki Okeya and Tsuyoshi Takagi. The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In M. Joye, editor, *Topics in Cryptology — CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 328–342. Springer-Verlag, 2003.
24. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In I. Attali and T.P. Jensen, editors, *Smart Card Programming and Security (E-Smart 2001)*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2001.
25. Yasuyuki Sakai and Kouichi Sakurai. A new attack with side channel leakage during exponent recoding computations. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 298–311. Springer-Verlag, 2004.
26. Nicolas Thériault. SPA resistant left-to-right integer recodings. In B. Preneel and S.E. Tavares, editors, *Selected Areas in Cryptograhy (SAC 2005)*, volume 3156 of *Lecture Notes in Computer Science*, pages 345–358. Springer-Verlag, 2006.
27. Camille Vuillaume and Katsuyuki 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.
28. Colin D. Walter. Sliding windows succumbs to Big Mac attack. 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 286–299. Springer-Verlag, 2001.
29. Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.
30. Sung-Ming Yen, Seung-Joo Kim, Seon-Gan Lim, and Sang-Jae Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In K. Kim, editor, *Information Security and Cryptology — ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 417–427. Springer-Verlag, 2002.
31. Sung-Ming Yen, Wei-Chih Lien, Sang-Jae Moon, and JaeCheol Ha. Power analysis by exploiting chosen message and internal collisions — Vulnerability of checking mechanism for RSA-decryption. In E. Dawson and S. Vaudenay, editors, *Progress in Cryptology — Mycrypt 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 183–195. Springer-Verlag, 2005.

A Regular Exponentiation Algorithms

A.1 Unsigned-digit recoding

A regular equivalent of Algorithm 7 can designed using the exponent recoding as described in Section 3.1.

The resulting algorithm requires m temporary variables for storing the respective accumulated values for recoded digits $d' + 1$ with $d' \in \{0, 1, \ldots, m-1\}$. In Algorithm 8 these values are stored in locations $R[i]$ for $i \in \{0, 1, \ldots, m-1\}$ which avoids having to add 1 to the index in the main loop.

The final digit is in $\{0, 1, \ldots, m\}$ and can, therefore, require special treatment. If the final digit is in $\{1, \ldots, m\}$ the digit can be treated like the rest of the digits

Algorithm 8: Regular Right-to-left m-ary Exponentiation

```
Input: $x \in \mathbb{G}$, $n \geq 1$, $k \geq 1$, $m = 2^k$
Output: $z = x^n$
Uses: $A$, $R[i]$ for $i \in \{0, 1, \ldots, m-1\}$
for $j = 0$ to $m-1$ do  $R[j] \leftarrow 1_{\mathbb{G}}$
$A \leftarrow x$; $\gamma' \leftarrow 1$; $n' \leftarrow \lfloor n/m \rfloor$
while $(n' \geq m+1)$ do
    $d \leftarrow n \bmod m$; $d' \leftarrow d + \gamma' + m - 2$; $\gamma' \leftarrow \lfloor d'/m \rfloor$
    $d' \leftarrow d' \bmod m$; $n \leftarrow n'$; $n' \leftarrow \lfloor n/m \rfloor$
    $R[d'] \leftarrow R[d'] \cdot A$
    $A \leftarrow A^m$
end
$d \leftarrow n \bmod m$; $d' \leftarrow d + \gamma' + m - 2$
$\gamma' \leftarrow \lfloor d'/m \rfloor$; $d' \leftarrow d' \bmod m$
$R[d'] \leftarrow R[d'] \cdot A$
$d' \leftarrow n' + \gamma' - 1$
if $(d' \neq 0)$ then
    while $(d' < 1)$ do
        $d' \leftarrow 2 \cdot d'$; $k \leftarrow k - 1$
    end
    $A \leftarrow A^{2^k}$; $R[d'-1] \leftarrow R[d'-1] \cdot A$
end
$A \leftarrow R[m-1]$
for $j = m-2$ down to $0$ do
    $R[j] \leftarrow R[j] \cdot R[j+1]$; $A \leftarrow A \cdot R[j]$
end
$A \leftarrow A \cdot R[0]$
return $A$
```

of the exponent. If the final digit is equal to zero the final multiplication can be avoided or replaced with a multiplication with $1_{\mathbb{G}}$ (where such a multiplication would not be apparent by observing a side channel).

Another potential problem is the first multiplication for each $R[i]$, for $i \in \{0, 1, \ldots, m-1\}$, which will be with $1_{\mathbb{G}}$. Depending on the arithmetic involved with this multiplication, it may be visible in a side channel. In a side channel resistant implementation one would expect the operands to be blinded such that there are numerous values that represent $1_{\mathbb{G}}$ [7, 16]. In this case, multiplications with $1_{\mathbb{G}}$ would no longer be visible.

A.2 Signed-digit recoding

The same exponent recoding presented in Algorithm 6 in Section 3.2 can be used in a right-to-left algorithm. The algorithm given in Algorithm 9 describes how the recoded exponent can be used to compute an exponentiation.

In Algorithm 9 the variable $R[i]$ for $i \in \{0, \ldots, m/2-1\}$ is used to store the product of the negative digits of the recoded exponent without inverting them. The product of the positive digits is stored in $R[i]$ for $i \in \{m/2, \ldots, m-1\}$. When the main loop is terminated the result of combining $R[i]$ for $i \in \{0, \ldots, m/2-1\}$ can be inverted to produce the value of x raised to the power of the negative digits. This can then be combined with the values corresponding to the positive digits to produce the result of the exponentiation.

Algorithm 9: Right-to-left Signed Recoding m-ary Exponentiation (I)

```
**Input**: $x \in \mathbb{G}$, $n \geq 1$ and odd, $m = 2^k$
**Output**: $z = x^n$
**Uses**: $A$, $R[i]$ for $i \in \{0, 1, \ldots, m-1\}$
**for** $j = 0$ **to** $m - 1$ **do**  $R[j] \leftarrow 1_{\mathbb{G}}$
$A \leftarrow x$
**while** $(n > m)$ **do**
    $\kappa \leftarrow (n \bmod 2m) - m$; $d' \leftarrow \lfloor (\kappa + m)/2 \rfloor$
    $R[d'] \leftarrow R[d'] \cdot A$
    $A \leftarrow A^m$
    $n \leftarrow (n - \kappa)/m$
**end**
$d' \leftarrow \lfloor (n + m)/2 \rfloor$; $R[d'] \leftarrow R[d'] \cdot A$
$A \leftarrow R[0]$
**for** $j = 1$ **to** $m/2 - 1$ **do**
    $A \leftarrow A \cdot R[j-1]$; $R[j] \leftarrow R[j] \cdot R[j-1]$; $A \leftarrow A \cdot R[j]$
**end**
$R[0] \leftarrow A$; $A \leftarrow R[m-1]$
**for** $i = m - 2$ **down to** $m/2$ **do**
    $A \leftarrow A \cdot R[j+1]$; $R[j] \leftarrow R[j] \cdot R[j+1]$; $A \leftarrow A \cdot R[j]$
**end**
$A \leftarrow A \cdot R[0]^{-1}$
**return** $A$
```

As with Algorithm 8, the initial multiplications with $1_{\mathbb{G}}$ may be visible in a side channel in Algorithm 9. This problem is easier to deal with in this algorithm since an inversion is computed at the end of the algorithm. For example, the line "$R[j] \leftarrow 1_{\mathbb{G}}$" can be replaced with "$R[j] \leftarrow x$". This will, effectively, blind the exponentiation and be removed when "$A \leftarrow A \cdot R[0]^{-1}$" is computed at the end of the algorithm.

Another option, when a random element in $\mathbb{G}$ can be computed efficiently, is to replace the initialisation loop, "**for** $j = 0$ **to** $m - 1$ **do** $R[j] \leftarrow 1_{\mathbb{G}}$", with

```
**for** $j = 0$ **to** $m/2 - 1$ **do**
    $R[j] \leftarrow$ RandomElement()
    $R[m - j - 1] \leftarrow R[j]$
**end**
```

where the function `RandomElement()` returns a random element in $\mathbb{G}$. Given that the random elements are balanced, the computation of a group inversion in the last line will remove the effect of these group elements. This provides more security than if x is used, since an attacker may attempt to manipulate x to produce specific effects in a side channel.

Algorithm 9 can be rewritten so that it uses $m/2$ values stored in memory. This means that inversions will need to be computed as required, rather than deferred to the end of the algorithm.

Algorithm 10: Right-to-left Signed Recoding m-ary Exponentiation (II)

```
Input: $x \in \mathbb{G}$, $n \geq 1$ and odd, $m = 2^k$ with $k > 1$
Output: $z = x^n$
Uses: $R[i]$ for $i \in \{0, 1, \ldots, m/2\}$
for $j = 1$ to $m/2$ do  $R[j] \leftarrow 1_{\mathbb{G}}$
$R[0] \leftarrow x$
while $(n > m)$ do
    $\kappa \leftarrow (n \bmod 2m) - m$; $d' \leftarrow \lceil |\kappa|/2 \rceil$
    $s \leftarrow \mathrm{sign}(\kappa)$; $d \leftarrow [(1 + s)((d' \bmod (m/2)) + 1)]/2$
    $R[d] \leftarrow R[d]^{-1}$; $R[d'] \leftarrow R[d'] \cdot R[0]$; $R[d] \leftarrow R[d]^{-1}$
    $R[0] \leftarrow R[0]^m$
    $n \leftarrow (n - \kappa)/m$
end
$d' \leftarrow \lceil n/2 \rceil$; $R[d'] \leftarrow R[d'] \cdot R[0]$
$R[0] \leftarrow R[m/2]$
for $j = m/2 - 1$ down to $1$ do
    $R[0] \leftarrow R[0] \cdot R[j + 1]$; $R[j] \leftarrow R[j] \cdot R[j + 1]$; $R[0] \leftarrow R[0] \cdot R[j]$
end
return $R[0]$
```

Depending on the group $\mathbb{G}$ and the target device, this may be problematic to compute in a regular manner, as discussed in Section 3.2. Some methods of achieving a regular solution when using elliptic curve arithmetic can be envisaged where κ_i is broken down into s_i and τ_i, e.g.:

- In $\mathbb{F}_{2^m}$ the unsigned representation of s_i can be used as a look up to either zero or the binary representation of the irreducible polynomial used to represent $\mathbb{F}_q$ over $\mathbb{F}_2$ which will be XOR-ed with the value of τ_i to produce the required value for the κ_i.
- An inversion over $\mathbb{F}_p$ can be achieved if we add p to the y coordinate and subtract zero or two times the value of the y coordinate, depending on the value of s_i. This will produce the required value corresponding to the value of κ_i in a regular manner. However, this will be vulnerable to a safe-error attack on the function that multiplies the y coordinate by two.

Another approach for elliptic curve arithmetic would be to compute the triplet $\{x, y, -y\}$ for each point produced by multiplying the initial point by $\{1, 3, \ldots,$

$m - 1\}$. The unsigned representation of s_i can then be used to select y or $-y$ as required. The memory requirements are reduced when compared to a standard m-ary algorithm, but not as much as would be possible if the computation of an inversion were completely free.

Yet another approach, provided that inversion can be computed for free, with fewer memory requirements, is to apply the previous trick but "on-the-fly". This requires an additional temporary variable, say B. In the **while**-loop, at each step, the inverse of A is computed and stored in B. The sign of digit d' is then used to compute $R[d'] \leftarrow R[d'] \cdot A$ or $[d'] \leftarrow R[d'] \cdot B$ as required. Provided that $k > 1$, the same effect can be obtained but without additional temporary variable with two inversions. This is detailed in Algorithm 10, where, to make the presentation easier, we rename accumulator A to $R[0]$ and reorganise the other variables: $R[i]$ for $i \in \{1, 2, \ldots, m/2\}$ is used to accumulated the values for recoded digits $\{1, 3, \ldots, m - 1\}$ (in absolute value).