Published in E. Prouff, Ed., *Smart Card Research and Advanced Applications (CARDIS 2011)*, vol. 7079 of *Lecture Notes in Computer Science*, pp. 84-101, Springer, 2011.

Memory-Efficient Fault Countermeasures

Marc Joye and Mohamed Karroumi

Technicolor, Security & Content Protection Labs
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
{marc.joye,mohamed.karroumi}@technicolor.com

Abstract. An efficient countermeasure against fault attacks for a right-to-left binary exponentiation algorithm was proposed by Boscher, Naciri and Prouff (WISTP, 2007). This countermeasure was later generalized by Baek (Int. J. Inf. Sec., 2010) to the 2^w-ary right-to-left algorithms for any $w \geqslant 1$ (the case $w = 1$ corresponding to the method of Boscher, Naciri and Prouff). In this paper, we modify theses algorithms, devise new coherence relations for error detection, and reduce the memory requirements without sacrificing the performance or the security. In particular, a full register (in working memory) can be gained compared to previous implementations. As a consequence, the implementations described in this paper are particularly well suited to applications for which memory is a premium. This includes smart-card implementations of exponentiation-based cryptosystems.

Keywords: Fault attacks, countermeasures, exponentiation, memory-constrained devices, smart cards.

1 Introduction

Implementation of exponentiation-based cryptosystems needs to be resistant against side-channel attacks. Simple Power Analysis (SPA) or Differential Power Analysis (DPA) target unprotected exponentiation algorithms like the classical square-and-multiply technique [20]. It has been shown that a private key used in RSA can be retrieved by observing the microprocessor's power consumption [21]. A possible countermeasure against SPA consists in always computing a squaring operation followed by a (sometimes dummy) multiplication, regardless the value of the exponent bit. The resulting algorithm is known as the "square-and-multiply always" algorithm [9]. Protection against DPA is usually achieved thanks to randomization techniques.

Fault attacks (FA) constitute another threat for public-key algorithms such as RSA [5]. Different methods have been proposed to protect RSA against fault analysis. There exist basically three main types of countermeasures.

The first type relies on a modification of the RSA modulus. This approach was initiated by Shamir [23] and gave rise to several follow-up papers, e.g. [1, 4, 18, 24]. When these methods are applied to RSA with Chinese remaindering, the main difficulty resides in the protection of the re-combination.

The second type of countermeasures uses the corresponding public operation to check the result before outputting it; for example, one can check the validity of an RSA signature using public exponent e. This however assumes the availability of e and is specific to a given cryptosystem; see [13] for a recent survey on these techniques.

The last type of countermeasures exploits coherence properties between internal variables during the exponentiation algorithm. Such algorithms are sometimes referred to as *self-secure* exponentiation methods. We focus our attention on this third type of countermeasures as they tend to be more generic.

Self-secure secure methods In 2005, Giraud proposed an exponentiation algorithm, which is secure against SPA and FA [11]. His key idea was to perform a coherence check at the end of the Montgomery powering ladder. Indeed, when evaluating x^d, both values $z := x^{d-1}$ and $y := x^d$ are available at the end of the computation. The check consists then in verifying that $z \cdot x = y$ before outputting the result. We note that this technique nicely combines with Chinese remaindering for RSA.

Boscher, Naciri and Prouff subsequently proposed another SPA-FA resistant exponentiation algorithm [7]. Their method built on an SPA-resistant version of the *right-to-left* binary algorithm[1] for evaluating x^d, that uses three registers. The authors observed that in each iteration of the main loop, the product of two registers is equal to the third one multiplied by the input value x. Their countermeasure consists in using this relation to derive a coherence check in the last iteration. Recently, Baek showed how the coherence check can be adapted to the Yao's right-to-left m-ary algorithm [2].

Yet another self-secure exponentiation method was proposed by Rivain [22]. The underlying idea is different. It relies on a double exponentiation, that is, an algorithm taking on input a pair of exponents (a, b) and returning (x^a, x^b). Applied to RSA, the pair of exponents is $(d, \phi(N) - d)$ and the coherence check verifies that $x^d \cdot x^{\phi(N)-d} = 1$ (modulo N). This method has the advantage of reducing the number of multiplications: on average, 1.65 modular multiplications per bit are required compared to the 2 modular multiplications for Giraud's or Boscher *et al.*'s binary methods. On the down side, the algorithm requires knowledge of $\phi(N)$ (or a multiple thereof like $ed - 1$), which is not necessarily available.

Contributions of the paper This paper deals with countermeasures against SPA and FA for right-to-left exponentiation algorithms. A drawback of Boscher *et al.*'s and Baek's proposals consists in requiring, on top of the internal registers for the computation of x^d, an additional register for storing the value of the input x, which is needed for the coherence check at the end of the algorithm. The main contribution of this paper is an optimized version of the protected right-to-left m-ary exponentiation algorithm. The optimizations consist of a rearrangement of Baek's algorithm by modifying the initialization steps and restructuring the

[1] This scan direction may be preferred as it usually eases the implementation (note that the Montgomery ladder processes the exponent bits from the left to the right).

inner operations so that a minimal number of registers in memory is used. Plugging $m = 2$ into our general algorithm yields a right-to-left binary method with *one register less* than Boscher *et al.*'s method. As a result, we obtain a *right-to-left* binary algorithm equally efficient (speed- *and* memory-wise) as Giraud's left-to-right algorithm. In the higher-radix case also (i.e., for $m > 2$), we also gain *one full memory register* over the best known right-to-left methods. Finally, as a side result, we offer a detailed memory analysis of Baek's algorithm and a slight variant thereof.

While the main application we had in mind was RSA (in standard and CRT modes), our implementations readily extend to any (finite) abelian group $\mathbb{G}$. For RSA, one has $\mathbb{G} = (\mathbb{Z}/N\mathbb{Z})^\times$ or $(\mathbb{Z}/p\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times$. We give a fully *generic* treatment and consider the general problem of computing $y = x^d$ in $\mathbb{G}$. We assume that exponent d is given in a standard format. We do not make *a priori* assumptions on $\mathbb{G}$ so that the presented algorithms can be used in various settings, including elliptic curve cryptosystems — though we present certain shortcuts when for example inverses are easy to compute in $\mathbb{G}$. In particular, we do not assume that the order of $\mathbb{G}$ is known and available to the implementation. We note that memory issues are of paramount importance for devices with limited resources as the amount of working memory generally constitutes the limiting factor in the development of efficient implementations.

Outline The rest of this paper is organized as follows. In the next section, we review Baek's 2^w-ary exponentiation algorithm, generalizing a binary exponentiation algorithm due to Boscher, Naciri and Prouff. In Section 3, we present a slightly modified variant thereof. Section 4 is the core of the paper. We detail how a full register can be saved, reducing the memory requirements to their minimum: no additional memory is needed for fault detection. Finally, we conclude in Section 5.

2 Exponentiation and Fault Countermeasures

2.1 Yao's *m*-ary exponentiation

Consider the m-ary expansion of some positive integer d, $d = \sum_{i=0}^{\ell-1} d_i\, m^i$ where $0 \leqslant d_i \leqslant m-1$ and $d_{\ell-1} \neq 0$, and an element x in a (multiplicatively written) group $\mathbb{G}$. The goal is to efficiently compute $y = x^d$, that is, $x \cdot x \cdots x$ (d times), for some (secret) exponent d. When $m = 2$, the classical right-to-left binary exponentiation method proceeds from the relation $x^d = \prod_{\substack{0\leqslant i\leqslant \ell-1\\ d_i=1}} x^{2^i}$.

This was extended to a general radix $m \geqslant 2$ by Yao [25]. The evaluation of $y = x^d$ is then carried out from the relation

$$y = x^{\sum_{i=0}^{\ell-1} d_i\, m^i} = \prod_{j=1}^{m-1} \left(\prod_{\substack{0\leqslant i\leqslant \ell-1\\ d_i=j}} x^{m^i} \right)^j \quad . \tag{1}$$

In more detail, Yao's algorithm makes use of an accumulator A and of $(m-1)$ temporary variables $\mathsf{R}[j]$ $(1 \leqslant j \leqslant m-1)$. Accumulator A is used to contain the successive m^{th} powers of the input element x. Specifically, at the beginning of step i, A contains the value x^{m^i} and is then updated as $\mathsf{A} \leftarrow \mathsf{A}^m$ to contain the value $x^{m^{i+1}}$ for the next step. Temporary variables $\mathsf{R}[j]$ are initialized to $1_{\mathbb{G}}$ (the neutral element in $\mathbb{G}$). At step i, provided that $d_i \neq 0$, the temporary variable corresponding to digit d_i (i.e., $\mathsf{R}[d_i]$) is updated as $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$; the other temporary variables remaining unchanged. At the end of the computation, all the temporary variables $\mathsf{R}[d_i]$ are aggregated to get the final result as $y = \prod_{j=1}^{m-1} (\mathsf{R}[j])^j$.

The original version of Yao's algorithm, as previously described, is prone to SPA-type attacks since an attacker may distinguish zero digits from nonzero ones. Indeed, when at step i, digit d_i is zero, there is no update of a temporary variable, and thus no multiplication occurs. An easy way to make the algorithm regular is to insert a dummy multiplication when $d_i = 0$ so that the digit 0 is treated as the other digits; see e.g. [9]. This is achieved by using an additional temporary variable, $\mathsf{R}[0]$, which is updated as $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{A}$ when $d_i = 0$. The resulting implementation is depicted in Alg. 1.

Algorithm 1 Yao's algorithm (with dummy multiplication)

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_m \in \mathbb{N}$
Output: $y = x^d$

```
    /* Initialization */
 1: for $i = 0$ to $m - 1$ do $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
 2: $\mathsf{A} \leftarrow x$
    /* Main loop */
 3: for $i = 0$ to $\ell - 1$ do
 4:     $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$
 5:     $\mathsf{A} \leftarrow \mathsf{A}^m$
 6: end for
    /* Aggregation */
 7: $\mathsf{A} \leftarrow \mathsf{R}[m-1]$
 8: for $i = m - 2$ down to 1 do
 9:     $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
10:     $\mathsf{A} \leftarrow \mathsf{A} \cdot \mathsf{R}[i]$
11: end for
12: return A
```

Although protected against SPA-type attacks, the algorithm becomes now vulnerable to safe-error attacks [26]. By timely inducing a fault during the multiplication at step i, an attacker may guess whether the operation is dummy or not (and thus whether the corresponding digit is zero or not) from the output value: a correct output value indicates that digit d_i is 0. Again, countermeasures exist.

For example, exponent d can be recoded prior entering Yao's exponentiation algorithm in such a way that all digits are nonzero. This is the approach followed in [14] where a regular recoding algorithm with digits in the set $\{1, \ldots, m\}$ is presented.

2.2 Protecting against faults

This section addresses the more general question of protecting against fault attacks (which encompasses protecting against safe-error attacks).

Baek, generalizing an earlier method due to Boscher, Naciri and Prouff [7], showed in a recent paper [2] how Algorithm 1 can be adapted so as to resist fault attacks. Defining

$$L_j = \prod_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = j}} x^{m^i} \qquad (\text{for } 0 \leqslant j \leqslant m-1)\,,$$

Equation (1) can be rewritten as $y = \prod_{j=1}^{m-1} (L_j)^j$. Hence, defining

$$T = \prod_{j=0}^{m-2} (L_j)^{m-1-j}\,,$$

it follows that

$$\begin{aligned} y \cdot T &= \prod_{j=0}^{m-1} (L_j)^j \cdot \prod_{j=0}^{m-1} (L_j)^{m-1-j} = \left(\prod_{j=0}^{m-1} L_j \right)^{m-1} = \left(\prod_{0 \leqslant i \leqslant \ell-1} x^{m^i} \right)^{m-1} = \left(x^{m-1} \right)^{\sum_{0 \leqslant i \leqslant \ell-1} m^i} \\ &= x^{m^\ell - 1} \end{aligned}$$

and therefore

$$y \cdot T \cdot x = x^{m^\ell}\ . \tag{2}$$

This relation is the basic idea behind the protection against faults. It serves as a coherence check between the different values involved in the computation. If the content of one of the temporary variables or of the accumulator is corrupted during the computation, then the coherence check will very likely fail and therefore the faulty computation can be detected and notified.

Binary case This case corresponds to the method of Boscher, Naciri and Prouff. When $m = 2$, the value of T simplifies to $T = L_0$. Further, noting that

$$d = \sum_{i=0}^{\ell-1} d_i \, 2^i = \sum_{\substack{0 \leqslant i \leqslant \ell-1 \\ d_i = 1}} 2^i$$

the binary case (i.e., $m = 2$) also implies $y = L_1$. As a result, the coherence test (Eq. (2)) becomes

$$L_0 \cdot L_1 \cdot x \overset{?}{=} x^{2^\ell} \quad . \tag{3}$$

Algorithmically, this translates into:

Algorithm 2 Binary SPA-FA resistant algorithm [7]

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_2 \in \mathbb{N}$
Output: $y = x^d$

```
   /* Initialization */
1: $\mathsf{R}[0] \leftarrow 1_{\mathbb{G}}$; $\mathsf{R}[1] \leftarrow 1_{\mathbb{G}}$
2: $\mathsf{A} \leftarrow x$
   /* Main loop */
3: for $i = 0$ to $\ell - 1$ do
4:     $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$
5:     $\mathsf{A} \leftarrow \mathsf{A}^2$
6: end for
   /* Error detection */
7: $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$
8: if ($\mathsf{R}[0] \cdot x \neq \mathsf{A}$) then return 'error'
9: return $\mathsf{R}[1]$
```

Remark 1. As presented, the previous implementation is not protected against second-order fault attacks. If an attacker induces a first fault and next a second fault during the error detection (at Line 8 in Alg. 2), then a faulty result may be returned and possibly exploited to infer information on secret value d. Such attacks were reported in [17]. An efficient countermeasure relying on the so-called lock-principle was later described in [10]. It is assumed that the error detection is done in this way or in an equivalent way so as to make it effective against second-order attacks.

Higher-radix case For the case $m = 2^w$, Baek suggests to evaluate, just after the main loop in Alg. 1, the quantities $y \leftarrow \prod_{j=1}^{m-1} \mathsf{R}[j]^j$ and $T \leftarrow \prod_{j=0}^{m-2} \mathsf{R}[j]^{m-1-j}$, and then to check whether $T \cdot y \cdot x = \mathsf{A}$ (note that A contains x^{m^ℓ} output of the main loop). The update of accumulator A, $\mathsf{A} \leftarrow \mathsf{A}^m$ with $m = 2^w$, is done via w repeated squarings. The evaluations of y and T are done as the aggregation in Alg. 1, the total cost of which amounts to $2 \times 2(m-2)$ multiplications. Therefore, as the coherence check requires two multiplications, if M and S respectively represent the cost of a multiplication and a squaring in $\mathbb{G}$, Baek's original algorithm requires altogether $\underline{(\ell + 4(2^w - 2) + 2)\mathsf{M} + \ell w \mathsf{S}}$, where $\ell = \lceil |d|_2 / w \rceil$ and $|d|_2$ is the bit-length of d.

3 A Variant of Baek's Algorithm

This section offers a detailed memory analysis for Baek's algorithm. We show how rearranging the operations allows a better management of the working memory. As a bonus, the total cost is also slightly reduced.

Let $m = 2^w$ for some $w > 1$. In a nutshell, using the notation of § 2.2, Baek's method proceeds in the following way:

1. Compute $y = x^d$ using Algorithm 1;
2. Compute T and next the product $S := y \cdot T$;
3. Check whether $S \cdot x \overset{?}{=} x^{m^\ell}$;
4. If so, return y.

The computation of y is evaluated in the aggregation step as $y = \prod_{i=1}^{m-1} \mathsf{R}[i]^i$; cf. Lines 7–11 in Alg. 1. First, we note the temporary variable $\mathsf{R}[m-1]$ can serve as the accumulator for the aggregation. This allows us to save A, which contains the value of x^{m^ℓ} output of the main loop — the value of x^{m^ℓ} being needed for the coherence check, $S \cdot x \overset{?}{=} x^{m^\ell}$. More explicitly, we rewrite the aggregation step as:

```
/* Aggregation */
for $i = m - 2$ down to 1 do
    $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
    $\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1] \cdot \mathsf{R}[i]$
end for
```

After the aggregation, the temporary variable $\mathsf{R}[m-1]$ contains the value of y. A close inspection shows also that $\mathsf{R}[1]$ contains $\prod_{i=1}^{m-1} L_i$. Further, since as shown in § 2.2

$$S = y \cdot T = \left(\prod_{i=0}^{m-1} L_i \right)^{m-1}$$

and since $\mathsf{R}[0]$ contains L_0, the value of S can be obtained as $(\mathsf{R}[0] \cdot \mathsf{R}[1])^{m-1}$. Therefore, instead of computing the quantity $T = \prod_{j=0}^{m-2} \mathsf{R}[j]^{m-1-j}$ as done in [2], we suggest to directly compute the product $y \cdot T$ by raising $\mathsf{R}[0] \cdot \mathsf{R}[1]$ to the power $m-1$. This avoid the use of any additional temporary variables. Furthermore, as in the binary representation of $m - 1 = 2^w - 1$, the bits are all equal to one, the powering to $m-1$ can be carried out through $w-1$ squarings and multiplications. This trades $2(2^w - 2)\mathsf{M}$ in Baek's original algorithm against $(w-1)\mathsf{S} + (w-1)\mathsf{M}$. The complete algorithm is depicted in Alg. 3.

The total cost of our variant drops to $\underline{(\ell + 2(2^w - 2) + w + 1)\mathsf{M} + (\ell w + w - 1)\mathsf{S}}$. It requires $2^w + 1$ registers (i.e., 2^w temporary variables $\mathsf{R}[j]$, $0 \leqslant j \leqslant 2^w - 1$, and accumulator A) as well as input value x for the coherence check.

Remark 2. When $m = 2^w$, the powering to $m-1$ for the computation of $S = R^{m-1}$ where $R := \prod_{0 \leqslant j \leqslant m-1} L_j$ could be optimized. In [8], Brauer explains how to obtain

Algorithm 3 Modified Baek's algorithm

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_{2^w} \in \mathbb{N}, w > 1$
Output: $y = x^d$

```
    /* Initialization */
 1: for $i = 0$ to $2^w - 1$ do R[$i$] $\leftarrow 1_{\mathbb{G}}$
 2: A $\leftarrow x$
    /* Main loop */
 3: for $i = 0$ to $\ell - 1$ do
 4:     R[$d_i$] $\leftarrow$ R[$d_i$] $\cdot$ A
 5:     A $\leftarrow$ A$^{2^w}$
 6: end for
    /* Aggregation */
 7: for $i = 2^w - 2$ down to 1 do
 8:     R[$i$] $\leftarrow$ R[$i$] $\cdot$ R[$i + 1$]
 9:     R[$2^w - 1$] $\leftarrow$ R[$2^w - 1$] $\cdot$ R[$i$]
10: end for
    /* Error detection */
11: R[0] $\leftarrow$ R[0] $\cdot$ R[1]; R[1] $\leftarrow$ R[0]
12: for $i = 1$ to $w - 1$ do
13:     R[1] $\leftarrow$ R[1]$^2$
14:     R[0] $\leftarrow$ R[0] $\cdot$ R[1]
15: end for
16: if (R[0] $\cdot x \neq$ A) then return 'error'
17: return R[$2^w - 1$]
```

a short addition chain for $2^w - 1$ from an addition chain for w. However, as the optimal value for w is rather small for typical cryptographic sizes (i.e., $w \leqslant 6$ as shown in Table 1), we do not discuss this issue further and stick to a simple binary algorithm for the computation of S. Alternatively, when the computation of an inverse in $\mathbb{G}$ is not expensive, R^{2^w-1} can be evaluated as $R^{2^w} \cdot R^{-1}$.

The next table compares the proposed variant for various sizes of d and w.

Table 1. Number of multiplications for various sizes of d and w.

		1024 bits		1536 bits		2048 bits	
		S/M=1	S/M=.8	S/M=1	S/M=.8	S/M=1	S/M=.8
Boscher *et al.* [7]	$w = 1$	**2050**	**1845**	**3074**	**2767**	**4098**	**3688**
Baek [2]	$w = 2$	1546	1341	2314	2007	3082	2672
	$w = 3$	1391	1187	2074	1767	2757	2347
	$w = 4$	**1338**	**1133**	1978	1671	2618	2208
	$w = 5$	1351	1146	**1965**	**1658**	**2580**	**2170**
	$w = 6$	1445	1230	2042	1735	2639	2230
Our variant	$w = 2$	1544	1339	2312	2005	3080	2670
	$w = 3$	1383	1178	2066	1758	2749	2339
	$w = 4$	1316	1111	1956	1648	2596	2186
	$w = 5$	**1299**	**1093**	**1913**	**1605**	2528	2117
	$w = 6$	1331	1125	1928	1620	**2525**	**2114**

4 Memory-Efficient Methods

We have seen in the previous section that a memory-optimized variant of Baek's algorithm requires $m+1$ registers *together with the input value x* for the coherence check:

$$S \cdot x \stackrel{?}{=} x^{m^{\ell}} \quad \text{where } S = y \cdot T$$

(see Eq. (2)). Likewise, in the binary case, Boscher *et al.*'s algorithm requires $2+1=3$ registers together with input value x.

When the computation of an inverse is not expensive in $\mathbb{G}$, a classical trick to avoid the storage of the complete value of x consists in computing a κ-bit digest thereof, say $h = H(x)$ for some function $H : \mathbb{G} \to \{0,1\}^{\kappa}$, at the beginning of the computation, and to replace the coherence check with

$$h \stackrel{?}{=} H\Big(S^{-1} \cdot x^{m^{\ell}}\Big) \ . \tag{4}$$

Such a method is mostly useful when $\kappa \ll |x|_2$. Note also that κ cannot be chosen too small, otherwise the coherence check (Eq. (4)) could be satisfied with a non-negligible probability, even in the presence of faults.

In this section, we consider *generic* countermeasures in the sense that they work equally well in any group $\mathbb{G}$ (even of unknown order). In particular, they do not require that computing inverses is fast. Moreover, they do not need further memory requirements: $m+1$ registers will suffice to get a protected implementation. Finally, they do not degrade the security level: the error detection probability remains unchanged compared to Baek's algorithm or to Boscher *et al.*'s algorithm in the binary case.

4.1 SPA-FA resistant right-to-left m-ary exponentiation

The main observation in Baek's algorithm (or in the binary version) for the computation of $y = x^d$ is that the product of the temporary variables is independent of exponent d.

In Algorithm 3, accumulator A is initialized to x and the temporary variables, $\mathsf{R}[j]$, to $1_{\mathbb{G}}$. In each step of the main loop, exactly *one* temporary variable is updated as $\mathsf{R}[d_i] \leftarrow \mathsf{R}[d_i] \cdot \mathsf{A}$ and the accumulator is updated as $\mathsf{A} \leftarrow \mathsf{A}^m$ (with $m = 2^w$). To avoid confusion, we let $\mathsf{R}[j]^{(i)}$ (resp. $\mathsf{A}^{(i)}$) denote the content of the temporary variable $\mathsf{R}[j]$ (resp. accumulator A) before entering step i. We have:

$$\begin{cases} \mathsf{R}[j]^{(i+1)} = \mathsf{R}[j]^{(i)} \cdot \mathsf{A}^{(i)} & \text{for } j = d_i \\ \mathsf{R}[j]^{(i+1)} = \mathsf{R}[j]^{(i)} & \text{for } j \neq d_i \\ \mathsf{A}^{(i+1)} = \big(\mathsf{A}^{(i)}\big)^m & \end{cases}$$

and consequently the product of all the temporary variables satisfies

$$\begin{aligned}
R^{(i+1)} := \prod_{j=0}^{m-1} \mathsf{R}[j]^{(i+1)} &= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(i)}\right) \cdot \mathsf{A}^{(i)} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(i-1)}\right) \cdot \mathsf{A}^{(i-1)} \cdot \mathsf{A}^{(i)} \\
&\ \vdots \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \mathsf{A}^{(0)} \cdots \mathsf{A}^{(i-1)} \cdot \mathsf{A}^{(i)} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \prod_{k=0}^{i} \mathsf{A}^{(k)} = \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \left(\mathsf{A}^{(0)}\right)^{1+m+m^2+\cdots+m^i} \\
&= \left(\prod_{j=0}^{m-1} \mathsf{R}[j]^{(0)}\right) \cdot \left(\mathsf{A}^{(0)}\right)^{\frac{m^{i+1}-1}{m-1}} \quad .
\end{aligned}$$

We observe that if the accumulator A is initialized to x^α and the temporary variables are initialized so that their product is equal to x^β (i.e., if $\mathsf{A}^{(0)} = x^\alpha$ and $\prod_{0\leqslant j\leqslant m-1} \mathsf{R}[j]^{(0)} = x^\beta$) then the previous relation becomes

$$R^{(i+1)} = x^{\beta+\frac{\alpha(m^{i+1}-1)}{m-1}} \quad . \tag{5}$$

Since Equation (5) is independent of d, our idea consists in using it to build a coherence check for appropriately chosen values for $\alpha \neq 0$ and β. There are several options.

Basic case: $\boldsymbol{\alpha = 1}$ and $\boldsymbol{\beta = 0}$ In this case, we get $R^{(i+1)} = x^{\frac{m^{i+1}-1}{m-1}}$ and in particular

$$R := R^{(\ell)} = x^{\frac{m^\ell-1}{m-1}} \quad .$$

This corresponds to the countermeasures proposed by Boscher *et al.* for the binary exponentiation and by Baek for higher radices. Indeed, setting $S = R^{m-1}$ leads to

$$S \cdot x = R^{m-1} \cdot x = x^{m^\ell-1} \cdot x = x^{m^\ell} \quad .$$

Fractional case: $\boldsymbol{\alpha = 1}$ and $\boldsymbol{\beta = \frac{1}{m-1}}$ Plugging these values in Eq. (5) yields $R^{(i+1)} = x^{\frac{m^{i+1}}{m-1}}$ and thus

$$R := R^{(\ell)} = x^{\frac{m^\ell}{m-1}} \quad .$$

In this case, it turns out that the numerator in the power of x is m^ℓ (and not $m^\ell - 1$ as in the basic case). Therefore, a simple coherence check is to compare $S := R^{m-1}$ with x^{m^ℓ}. The main advantage is that the value of x is no longer involved.

On the minus side, since $\beta = \frac{1}{m-1}$ is not an integer, the initialization of the temporary variables requires the computation of roots in $\mathbb{G}$. In general, this is a rather expensive operation but there exist cases where it is not. Examples include point halving (i.e., square roots in $\mathbb{G}$) in odd-order subgroups of binary elliptic curves [19] or cube roots in [the multiplicative group of] a finite field of characteristic three [3].

Generic case: $\alpha = \beta(m-1)$ This setting generalizes the fractional case. However, to avoid the computation of roots in $\mathbb{G}$, we restrict α and β to integer values. We have:

$$R := R^{(\ell)} = x^{\beta\, m^\ell} \implies R^{m-1} = x^{\alpha\, m^\ell} \ .$$

It is worth noting here that at the end of the main loop, A contains the value of $x^{\alpha\, m^\ell}$. The relation $R^{m-1} = x^{\alpha\, m^\ell}$ therefore provides a coherence check to protect against faults. The main advantage over the previous methods is that R^{m-1} appears as the exact value present in the accumulator and so there is no need to keep the value of x.

It remains now to show how to compute $y = x^d$ when A is initialized to $x^\alpha = x^{\beta(m-1)}$ and $\prod_{0\leqslant j\leqslant m-1} \mathsf{R}[j]$ is initialized to x^β. We write:

$$d = \alpha \cdot q + r \qquad \text{with } q = \left\lfloor \frac{d}{\alpha} \right\rfloor \text{and } r = d \bmod \alpha \ . \tag{6}$$

Hence, if $\sum_{i=0}^{\ell'-1} q_i\, m^i$ denotes the m-ary expansion of q then Yao's method yields

$$x^{d-r} = (x^\alpha)^{\sum_{i=0}^{\ell'-1} q_i\, m^i} = \prod_{j=1}^{m-1} (\boldsymbol{L}_j)^j \quad \text{where } \boldsymbol{L}_j = \prod_{\substack{0\leqslant i\leqslant \ell'-1\\ q_i=j}} X^{m^i} \text{ and } X = x^\alpha \ . \tag{7}$$

Remember from Section 3 that the temporary variable $\mathsf{R}[0]$ is not used in the computation of y. Now, assume in Algorithm 3 that temporary variables $\mathsf{R}[j]$ are initialized to x^{e_j} for some integer e_j, for $0 \leqslant j \leqslant m-1$, so that

$$\sum_{j=1}^{m-1} j \cdot e_j = r \quad \text{and} \quad \sum_{j=0}^{m-1} e_j = \beta \ . \tag{8}$$

In that case, it can be easily verified that if A is initialized to $x^{\beta(m-1)}$, then Algorithm 3 (with the error detection adapted as above explained) will return the value of $y = x^d$ (or an error notification). Indeed, from Eq. (7), we get

$$y = x^r \cdot \prod_{j=1}^{m-1} (\boldsymbol{L}_j)^j = \prod_{j=1}^{m-1} (x^{e_j} \cdot \boldsymbol{L}_j)^j$$

(and $R^{m-1} = X^{m^{\ell'}}$).

Several strategies are possible in order to fulfill Eq. (8). The simplest one is to select $\beta = 1$ (and thus $\alpha = m-1$). Since $0 \leqslant r < \alpha$ $(= m-1)$, a solution to Eq. (8) is then given by

$$\begin{cases} e_r = 1 \\ e_j = 0 \quad \text{for } 0 \leqslant j \leqslant m-1 \text{ and } j \neq r \end{cases} .$$

We detail below the corresponding algorithm for $m > 2$. The case $m = 2$ is presented in § 4.3.

Algorithm 4 Memory-efficient SPA-FA resistant algorithm

Input: $x \in \mathbb{G}$ and $d \in \mathbb{N}$
Output: $y = x^d$

```
    /* Initialization */
 1: $\mathsf{A} \leftarrow x^{m-1}$
 2: for $i = 0$ to $m-1$ do $\mathsf{R}[i] \leftarrow 1_{\mathbb{G}}$
 3: $\mathsf{R}[d \bmod (m-1)] \leftarrow x$
 4: $d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
    /* Main loop */
 5: for $i = 0$ to $\ell' - 1$ do
 6:     $\mathsf{R}[d'_i] \leftarrow \mathsf{R}[d'_i] \cdot \mathsf{A}$
 7:     $\mathsf{A} \leftarrow \mathsf{A}^m$
 8: end for
    /* Aggregation */
 9: for $i = m-2$ down to 1 do
10:     $\mathsf{R}[i] \leftarrow \mathsf{R}[i] \cdot \mathsf{R}[i+1]$
11:     $\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1] \cdot \mathsf{R}[i]$
12: end for
    /* Error detection */
13: $\mathsf{R}[0] \leftarrow \mathsf{R}[0] \cdot \mathsf{R}[1]$;
14: $\mathsf{R}[0] \leftarrow \mathsf{R}[0]^{m-1}$
15: if ($\mathsf{R}[0] \neq \mathsf{A}$) then return 'error'
16: return $\mathsf{R}[m-1]$
```

Remark 3. For the sake of clarity and in order not to focus to a specific implementation, the error detection in Alg. 4 is written with an if-branching. This may be subject to second-order fault attacks. In practice, if second-order attacks are a concern, precautions need to be taken and the if-branching should be rewritten with appropriate measures; cf. Remark 1.

4.2 Dealing with the neutral element $1_{\mathbb{G}}$

In certain cases, multiplication by neutral element $1_{\mathbb{G}}$ may be distinguished, which is turn, may leak information on the secret exponent d.

Small order elements To avoid this, the temporary variables $\mathsf{R}[j]$ can be multiplied by an element of small order in $\mathbb{G}$ in the initialization step. As an illustration, suppose that they are all multiplied by some element h of order 2. More specifically, suppose that the initialization in Alg. 4 (where $\alpha = m-1$ and $\beta = 1$) is

```
/* Initialization */
$\mathsf{A} \leftarrow x^{m-1}$
for $i = 0$ to $m-1$ do $\mathsf{R}[i] \leftarrow h$
$\mathsf{R}[d \bmod (m-1)] \leftarrow h \cdot x$
$d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
```

for some $h \in \mathbb{G}$ such that $h^2 = 1_{\mathbb{G}}$. Then in each iteration, it is easily seen that the product of all $\mathsf{R}[j]$'s will contain a surplus factor h^m, or from Eq. (5) that

$$R^{(i+1)} := \prod_{j=0}^{m-1} \mathsf{R}[j]^{(i+1)} = h^m \cdot x^{\beta + \frac{\alpha(m^{i+1}-1)}{m-1}} = h^m \cdot x^{m^{i+1}} \implies R := R^{(\ell)} = h^m \cdot x^{m^\ell} \ .$$

When m is even (which is always the case for $m = 2^w$) then $h^m = 1$ and so the coherence check is unchanged: $R^{m-1} \overset{?}{=} x^{\alpha\, m^\ell}$ with $\alpha = (m-1)$. Furthermore, the computation of y is also unchanged when $m = 2^w$ and $w > 1$. Indeed, letting $r = d \bmod (m-1)$, we have from Eq. (7):

$$\prod_{j=1}^{m-1} \left(\mathsf{R}[j]^{(\ell)}\right)^j = \left(\prod_{\substack{1 \leqslant j \leqslant m-1 \\ j \neq r}} h^j \right) \cdot (h \cdot x)^r \cdot x^{d-r} = h^{\sum_{j=1}^{m-1} j} \cdot x^d = h^{\frac{m(m-1)}{2}} \cdot y = y$$

since $m(m-1)/2$ is even when $m = 2^w$ and $w > 1$. In the binary case (i.e., when $w = 1$), we have $m(m-1)/2 = 1$ and consequently the above product needs to be multiplied by h to get the correct output; see also § 4.3 for alternative implementations. Note that for the RSA cryptosystem with a modulus N, we can take $h = N - 1$ which is of order 2 since $(N-1)^2 = 1 \pmod N$. The technique can also be adapted to other elements of small order.

Invertible elements Prime-order elliptic curves obviously do not possess elements of small order. We present below a solution for groups $\mathbb{G}$ wherein the computation of inverses is fast (as it is the case for elliptic curves).

A method used in [13] consists in initializing all the registers to x. This initialization method does not work as is and has to be adapted here. In order to verify Eq. (5), $\mathsf{R}[j]$ should be initialized to x^{1+e_j} so that

$$\sum_{j=1}^{m-1} j \cdot e_j = r' \quad \text{and} \quad \sum_{j=0}^{m-1} (1 + e_j) = 1 \ .$$

Again there are several possible solutions to the previous relation. We may for example define:

– if $r' = 0$

$$\begin{cases}(e_0,\ e_{m-1}) = (2-m,\ -2) & \\ e_j = 0 & \text{for } 1 \leqslant j \leqslant m-2\end{cases};$$

– if $r' \neq 0$

$$\begin{cases}(e_0,\ e_{m-1}) = (1-m,\ -2) & \\ e_j = 0 & \text{for } 1 \leqslant j \leqslant m-2 \text{ and } j \neq r' \\ e_{r'} = 1 & \end{cases}.$$

$\mathsf{R}[j]$ (for $1 \leqslant j \leqslant m-2$) are then initialized to x and $\mathsf{R}[m-1]$ is set to $x^{1+e_{m-1}} = x^{-1}$. Since $\mathsf{R}[j]$ is raised to the power j within the aggregation step, we subtract $\sum_{j=1}^{m-2} j - (m-1) = \frac{(m-1)\cdot(m-4)}{2}$ from d prior to the exponentiation, i.e., we compute $d' = d - \frac{(m-1)\cdot(m-4)}{2}$. Next, we write $d' = (m-1)\cdot q' + r'$ with $0 \leqslant r' < m-1$. Finally, $\mathsf{R}[0]$ is initialized to $x^{1+e_0} = x^{-(m-2)}$ and $\mathsf{R}[r']$ is initialized to

$$x^{1+e'_r} = \begin{cases} x^{-(m-3)} & \text{if } r' = 0 \\ x^{1+1} = x^2 & \text{otherwise}\end{cases}.$$

Noting that $x^{-(m-3)} = x^{-(m-2)} \cdot x$ and $x^2 = x \cdot x$, and since $r' \leqslant m-2$ the above procedure can be implemented in a regular fashion by replacing the initialization of Alg. 4 with

```
/* Initialization */
$\mathsf{A} \leftarrow x^m \cdot x^{-1}$
$\mathsf{R}[0] \leftarrow \mathsf{A}^{-1} \cdot x$
for $i = 1$ to $m-1$ do $\mathsf{R}[i] \leftarrow x$
$d \leftarrow d - \frac{(m-1)\cdot(m-4)}{2}$
$r' \leftarrow d \bmod (m-1)$
$\mathsf{R}[r'] \leftarrow \mathsf{R}[r'] \cdot \mathsf{R}[r'+1]$;
$\mathsf{R}[m-1] \leftarrow \mathsf{R}[m-1]^{-1}$
$d \leftarrow \lfloor d/(m-1) \rfloor = (d'_{\ell'-1}, \ldots, d'_0)_m$
```

This initialization works for all $w > 1$. The next section proposes an efficient alternative in the binary case for the RSA exponentiation (i.e. d is odd)

4.3 Binary case

In the binary case, we have $m-1 = 1$ and thus $q = d$ and $r = 0$. We can use Algorithm 4 as is, where m is set to 2.

The resulting algorithm is very simple. It is slightly faster than Boscher *et al.*'s algorithm (Alg. 2) as it saves one multiplication. When d is odd (as is the case for RSA), A can be initialized to x^2, $\mathsf{R}[1]$ to x, and the for-loop index at $i = 1$; this allows to save one more multiplication — and not to involve $1_{\mathbb{G}}$. More importantly, Algorithm 5 saves one register compared to Algorithm 2 as the value of x is no longer needed for the error detection. We so obtain a right-to-left algorithm as efficient as Giraud's algorithm. Being based on Montgomery ladder, Giraud's method scans however the exponent in the opposite direction which may be less convenient.

Algorithm 5 Binary right-to-left SPA-FA resistant algorithm

Input: $x \in \mathbb{G}$ and $d = (d_{\ell-1}, \ldots, d_0)_2 \in \mathbb{N}$
Output: $y = x^d$

```
   /* Initialization */
1: A ← $x$
2: R[0] ← $x$; R[1] ← $1_{\mathbb{G}}$
   /* Main loop */
3: for $i = 0$ to $\ell - 1$ do
4:     R[$d_i$] ← R[$d_i$] · A
5:     A ← A$^2$
6: end for
   /* Error detection */
7: R[0] ← R[0] · R[1]
8: if (R[0] ≠ A) then return 'error'
9: return R[1]
```

4.4 Efficiency

The initialization phase in Alg. 4 for the higher-radix case involves seemingly cumbersome operations. We detail below efficient implementations.

Algorithm 4 starts with the evaluation of x^{m-1}. We suggest to rely on a binary algorithm similarly to the implementation of the error detection in our variant of Baek's algorithm (i.e., Lines 12–15 in Alg. 3). For $m = 2^w$, this costs at most $(w - 1)$ multiplications and squarings in $\mathbb{G}$ (see Remark 2).

The initialization in Alg. 4 also requires the integer division (with remainder) of exponent d by $m - 1$. One may argue that these two values could be pre-computed and d represented by the pair (q, r) with $q = \lfloor d/(m - 1) \rfloor$ and $r = d \bmod (m - 1)$. We rule out this possibility as it supposes that d is fixed (which is for example not the case for ECDSA). Further, even for RSA with a *a priori* fixed exponent, such a non-standard format for d, $d = (q, r)$, is not always possible as it may be incompatible with the personalization process or with certain randomization techniques used to protect against DPA-type attacks.

Let $m = 2^w$. We first remark that given the m-ary expansion of d, $d = \sum_{i=0}^{\ell-1} d_i\, m^i$, the value of $r = d \bmod (m - 1)$ can simply be obtained as $r = \sum_{i=0}^{\ell-1} d_i \pmod{m - 1}$. For the computation of the quotient, $q = \lfloor d/(m - 1) \rfloor$, the schoolboy method can be significantly sped up by noting that the divisor (i.e., $m-1 = 2^w-1$)

has all its bits set to 1 [20, p. 271] (see also [28]). It is also possible to evaluate q without resorting on a division operation. A common approach is the division-free Newton-Raphson method [16]. Being quadratically convergent, the value of q is rapidly obtained after a few iterations, i.e., roughly $\log_2(\log_2 d)$ iterations. The cost is typically upper-bounded to 2 multiplications in $\mathbb{G}$. The algorithm is presented in Appendix A. Of course, when available, the pair (q, r) can be computed with the co-processor present on the smart card; some of them come with an integer division operation.

To sum up, noting that Algorithm 4 saves a few multiplications, we see that all in all its expected performance is globally the same — when not faster — as our variant of Baek's algorithm. But the main advantage of Algorithm 4 resides in its better usage of memory.

5 Conclusion

This paper presented several memory-efficient implementations for preventing fault attacks in exponentiation-based cryptosystems. Furthermore, they are by nature protected against SPA-type attacks and can be combined with other existing countermeasures to cover other classes of implementation attacks. Remarkably, the developed methodology is fully generic (i.e., applies to any abelian group) and allows one to save one memory register (of size a group element) over previous implementations. This last feature is particularly attractive for memory-constrained devices and makes the proposed implementations well suited for smart-card applications.

Acknowledgments We are grateful to an anonymous reviewer for useful comments.

References

1. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2002.
2. Y.-J. Baek. Regular 2^w-ary right-to-left exponentiation algorithm with very efficient DPA and FA countermeasures. *International Journal of Information Security*, 9(5):363–370, 2010.
3. P. S. L. M. Barreto. A note on efficient computation of cube roots in characteristic 3. Cryptology ePrint Archive, Report 2004/305, 2004. `http://eprint.iacr.org/`.
4. J. Blömer, M. Otto, and J.-P. Seifert. A new CRT-RSA algorithm secure against Bellcore attacks. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 311–320. ACM Press, 2003.

5. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001. Earlier version published in EUROCRYPT '97.
6. A. Boscher, H. Handschuh, and E. Trichina. Blinded exponentiation revisited. In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2009*, pages 3–9. IEEE Computer Society, 2009.
7. A. Boscher, R. Naciri, and E. Prouff. CRT RSA algorithm protected against fault attacks. In D. Sauveron et al., editors, *Information Security Theory and Practices (WISTP 2007)*, volume 4462 of *Lecture Notes in Computer Science*, pages 229–243. Springer-Verlag, 2007.
8. A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.
9. J.-S. 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.
10. E. Dottax, C. Giraud, M. Rivain, and Y. Sierra. On second-order fault analysis resistance for CRT-RSA implementations. In O. Markowitch et al., editors, *Information Security Theory and Practices (WISTP 2009)*, volume 5746 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2007.
11. C. Giraud. An RSA implementation resistant to fault attacks and to simple power analysis. *IEEE Transactions on Computers*, 55(9):1116–1120, 2006.
12. M. Joye. Highly regular m-ary powering ladders. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography (SAC 2009)*, volume 5867 of *Lecture Notes in Computer Science*, pages 350–363. Springer-Verlag, 2009.
13. M. Joye. Protecting RSA against fault attacks: The embedding method. In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2009*, pages 41–45. IEEE Computer Society, 2009.
14. M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. In B. Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 334–349. Springer-Verlag, 2009.
15. M. Joye and S.-M. Yen. The Montgomery powering ladder. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, 2002.
16. A. H. Karp and P. W. Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997.
17. C. H. Kim and J.-J. Quisquater. Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In D. Sauveron et al., editors, *Information Security Theory and Practices (WISTP 2007)*, volume 4462 of *Lecture Notes in Computer Science*, pages 215–228. Springer-Verlag, 2007.
18. C. H. Kim and J.-J. Quisquater. How can we overcome both side channel analysis and fault attacks on RSA-CRT? In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2007*, pages 21–29. IEEE Computer Society, 2007.
19. E. W. Knudsen. Elliptic scalar multiplication using point halving. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptology – ASIACRYPT '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.
20. D. E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, 2nd edition, 1981.

21. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
22. M. Rivain. Securing RSA against fault analysis by double addition chain exponentiation. In M. Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 459–480. Springer-Verlag, 2009.
23. A. Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. US Patent #5,991,415, Nov. 1999. Presented at the rump session of EUROCRYPT '97.
24. D. Vigilant. RSA with CRT: A new cost-effective solution to thwart fault attacks. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 130–145. Springer-Verlag, 2008.
25. A. C.-C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.
26. S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.
27. S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. 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.
28. C. Yungui, Y. Xiaodong, and W. Bingshan. A fast division technique for constant divisors $2^m(2^n \pm 1)$. *Scientia Sinica (Series A)*, XXVII(9):984–989, 1984.

A Newton-Raphson Iterated Division Algorithm

Algorithm 6 Integer division by $m-1$ (initialization step)

```
Input: $d = \sum_{i=0}^{\ell-1} d_i m^i$ where $m = 2^w$
Output: $q = \lfloor d/(m-1) \rfloor$ and $r = d \bmod (m-1)$
   /* Remainder */
1: $r \leftarrow d_0$
2: for $i = 1$ to $\ell - 1$ do $r \leftarrow (r + d_i) \bmod (2^w - 1)$
   /* Quotient */
3: $d \leftarrow d - r$; $B \leftarrow |\ell(w-1)|_2$
4: $q \leftarrow 1$; $s \leftarrow 2^w - 1$
5: for $i = 1$ to $B$ do $q \leftarrow q \cdot (2 - s \cdot q) \bmod 2^{2^i}$
6: $q \leftarrow q \cdot d \bmod 2^{2^B}$
7: return $(q, r)$
```