IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 22, 2020, accepted July 28, 2020, date of publication August 7, 2020, date of current version August 20, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3015099

Memory-Efficient Random Order Exponentiation Algorithm

DUC-PHONG LE AND ALI A. GHORBANI, (Senior Member, IEEE)
Canadian Institute for Cybersecurity, University of New Brunswick, Fredericton, NB E3B 9W4, Canada

Corresponding author: Duc-Phong Le (le.duc.phong@unb.ca)

This work was supported in part by the Canada Research Chair Program, and in part by the Canadian Institute for Cybersecurity.

ABSTRACT Randomizing the execution of the sequence of operations in an algorithm is one of the most frequently considered solutions to improve the security of cryptographic implementations against side-channel analysis. Such an algorithm for public-key cryptography was introduced by Tunstall at ACISP, 2009. In his right-to-left m-ary exponentiation algorithm, the radix-m digits of the exponent are treated in somewhat random order. This randomized solution will inhibit attacks that allow operations to be distinguished from *one acquisition*. In this article, we present a memory-efficient variant of Tunstall's random-order exponentiation algorithm, making it applicable to modular exponentiations in $(\mathbb{Z}/N\mathbb{Z})^*$ (for instance, the RSA cryptosystem). The proposed algorithm requires only $(m+1)$ memory registers instead of $(m+r)$, where $r > m$ as recommended in Tunstall's algorithm. Namely, the proposed algorithm saves about half the memory registers. Our analysis shows that our algorithm can be used as a supplement in order to defeat statistical side-channel analysis attacks, especially recent collision-correlation power analysis in the *horizontal setting*. Last but not least, we present a random order binary implementation, which is the first right-to-left binary implementation resisting attacks in the horizontal setting.

INDEX TERMS Right-to-left exponentiation, randomized algorithms, power analysis, horizontal collision-correlation attacks, Big Mac attacks.

I. INTRODUCTION

Side-channel analysis (SCA) attacks, formally introduced by Kocher *et al.* [16] and Kocher [17], are nowadays one of the most serious threats to the security of a given implementation of a cryptographic algorithm. This kind of attacks uses leaked side-channel information from cryptographic devices to determine the secret key. Kocher *et al.* described two main attacks: simple power analysis and differential power analysis. While the former uses the power consumption from one or several measurements directly to determine the secret information, the later (also called statistical side-channel analysis) requires a large number of consumption traces, and *statistical tools* to exploit the correlations between the leakage and processed data to recover the secret information. Regular algorithms, *e.g.,* square-multiply always [8], or Montgomery powering ladder [14] could be resistant to simple power analysis. However, to prevent differential power analysis, one would use blinding techniques [8], [16], or randomized techniques [19].

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang.

Shuffling that was first introduced to symmetric encryption algorithms in [12], provides additional resistance against statistical SCA attacks. Basically, shuffling randomizes the execution of the sequence of operations in an algorithm and can be applied to any set of independent operations. Nowadays, shuffling is one of the main approaches used to thwart different power analysis attacks for symmetric cryptographic implementations. Readers are referred to [10], [22] for comprehensive studies about this method.

For public-key cryptosystems, Tunstall introduced such a shuffling technique in [21]. In his randomized right-to-left m-ary exponentiation algorithm, operations are performed in somewhat random order. The author observed that in the right-to-left exponentiation algorithm, the multiplication operations can be performed independently, and in any order without influencing the final result. To our knowledge, this is so far the only random-order countermeasure for public-key cryptosystems.

One disadvantage of Tunstall's algorithm is to require $(m + r)$ memory registers to store group elements, where $r > m$ is large enough to provide a suitable level of random ordering. Compared to the usual right-to-left exponentiation,

his algorithm requires r extra memory registers. This may not matter in software implementation. However, in hardware implementation, *e.g.,* smart devices with constrained resources, this algorithm is unlikely to be possible for exponentiation in $(\mathbb{Z}/N\mathbb{Z})^*$.

In this article, we first propose a memory-efficient variant of Tunstall's algorithm. The proposed algorithm requires only $(m+1)$ group elements stored in memory instead of $(m+r)$. This memory requirement is comparable to the usual m-ary algorithm. We then analyze the security of the proposed algorithm, as well as existing right-to-left m-ary algorithms in the presence of the recent advanced differential power analysis in horizontal setting [6], [7], [11], [23]. Last but not least, we present an efficient implementation of the random order algorithm in the binary case. To the best of our knowledge, it is the first right-to-left binary exponentiation algorithm that resists to the horizontal collision-correlation attacks.

The rest of the paper is organized as follows. We briefly recall in Section II power analysis, and the random order m-ary exponentiation algorithm. Section III describes the proposed algorithm and analyzes its performance. Section IV describes our random order binary exponentiation and Section V analyzes the security of the proposed algorithm, as well as the security of the existing right-to-left m-ary exponentiation algorithms against the Big Mac attack and its extensions. We conclude in Section VI.

II. PHYSICAL ATTACKS AND COUNTERMEASURES

A. POWER ANALYSIS

Simple Power Analysis (SPA for short) attacks aim at recovering the secret key by just querying one message to the embedded devices. From the power consumption trace measured, an attacker makes use of a *distinguisher* to deduce a sequence of squaring and multiplication operations that is equivalent to the secret exponent in some implementations. Regular algorithms [8], [14] and atomic algorithms [5] can be used to thwart SPA attacks.

Differential Power Analysis (DPA for short) attacks, also known as *statistical side-channel analysis attacks*, exploit the correlations between the leakage and processed data to defeat countermeasures that are immune from SPA. In contrast to SPA attacks, DPA attacks require a large number of power consumption traces and then make use of *statistical tools* to deduce the secret information. Consequently, many improvements to DPA have been introduced. For example, Correlation Power Analysis [4] and Collision-Correlation Analysis [24] require far fewer power traces than the original DPA. Randomizing techniques [8], [16], [19], [21] can be used to inhibit DPA attacks.

Big MAC Attack and Its Extensions

All the statistical analysis attacks mentioned above require numerous traces to be taken to reduce noise to the point of time where an attack will perform. On the contrary, the Big Mac attack [23] requires only one power consumption trace to recover the secret key. This is a *horizontal statistical analysis attack*. The original Big Mac attack applies to m-ary exponentiation and to all similar algorithms which use a table of pre-computed values. Subsequent works further studied the Big Mac attack and demonstrated with experimental results. Instead of using a Euclidean distance, Clavier *et al.* [7] used the Pearson correlation to detect collision between two multiplications. Studies in [6], [11] presented the further refined attacks that use collision-correlation and applied not only to RSA but also to elliptic curve cryptosystems.

Randomizing intermediate computations can be used to thwart the horizontal (collision)-correlation analysis [7]. This approach requiring randomization of the intermediate long integer multiplication is generally costly from a performance viewpoint.

B. RANDOM ORDER EXPONENTIATION ALGORITHM

For an input key, the square-and-multiply algorithm [20] outputs a unique sequence of squares and multiplications. This allows an attacker to immediately determine the private key. On the other hand, given the same input, randomized algorithms output different sequences of operations. The idea is to randomize the number and the sequence of operations executed in the exponentiation algorithm itself.

Let $n = (d_{\ell-1}, \ldots, d_1, d_0)_m$ denote the radix-m representation of an exponent n, where $\ell = \lceil \log_2(n)/w \rceil$ and $w = \log_2(m)$, that is $n = \sum_i d_i m^i$ with $d_i \in \{0, 1, \ldots, m-1\}$ and $d_{\ell-1} \neq 0$. From this expansion, Yao [25] introduced a right-to-left m-ary exponentiation. Its principle is based on the following equality:

$$x^n = \prod_{0\le i\le \ell-1} \left(x^{m^i}\right)^{d_i} = \prod_{\substack{0\le i\le \ell-1\\ d_i=1}} x^{m^i} \times \prod_{\substack{0\le i\le \ell-1\\ d_i=2}} x^{2m^i} \times \cdots$$
$$\times \prod_{\substack{0\le i\le \ell-1\\ d_i=m-1}} x^{(m-1)m^i} = \prod_{j=1}^{m-1} \Big(\prod_{\substack{0\le i\le \ell-1\\ d_i=j}} x^{m^i} \Big)^j .$$

In Yao's algorithm (Algorithm 4 in the Appendix), one uses $(m-1)$ accumulators, $R[1], \ldots, R[m-1]$, each of them initialized the $1_{\mathbb{G}}$. A loop is processed which applies w successive squarings in every iteration to compute $A = x^{m^i}$ from $x^{m^{(i-1)}}$, and which multiplies the result to some accumulators $R[j]$, where $j = d_i$. Let $R[j]^{(i)}$ (resp. $A^{(i)}$) denote the value of the accumulator $R[j]$ (resp. A) before entering step i. We have:

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

At the end of the loop each accumulator $R[j]$ contains the product $\prod_{\substack{0\le i\le \ell-1\\ d_i=j}} x^{m^i}$. The different accumulators are finally aggregated as $\prod_{0\le j\le \ell-1} R[j]^j = x^n$. This algorithm requires more memory than the binary method but it is faster since the number of multiplications is roughly reduced to $\left(1 + \frac{m-1}{m \log_2 m}\right) \log_2 n$ (see [18]).

Now, suppose that all values of x^{m^i} (*i.e.,* a set of $S[i] = A^{(i)} = x^{m^i}$, for $i \in \{0, 1, \ldots, \ell - 1\}$) are available at the beginning of computation. The exponentiation can be computed by first choosing at random an $S[i]$, $0 \le i \le \ell - 1$, then updating $R[d_i] = R[d_i] \cdot S[i]$. This step is repeated until all of the values $S[i]$ have been considered. The random set having been chosen is a permutation of the set $\{0, 1, \ldots, \ell - 1\}$. This can be performed because the final results of $R[j]$ don't depend on the order of updated operations, and moreover the values of $R[j]$ are independent from each other.

This approach certainly costs extra memory to store precomputed values $S[i]$, for $0 \le i < \ell$. In order to reduce this required memory space, the idea can be repeatedly applied with only $r < \ell$ precomputed values. Tunstall [21] used this interesting observation to describe a random order m-ary exponentiation algorithm (Algorithm 1) as follows[1]:

Algorithm 1: Random Order Right-to-Left m-Ary Exponentiation Algorithm [21]

Input: $x \in \mathbb{G}, n = (d_{\ell-1}, \ldots, d_1, d_0)_m \in \mathbb{N}$
Output: x^n

```
   // Initialization
1  for i = 1 to m − 1 do
2      R[i] ← 1
3  S[0] ← x
4  for i = 1 to r − 1 do
5      S[i] ← S[i − 1]^m
6  for i = 0 to r − 1 do
7      D[i] ← n mod m ; n ← ⌊n/m⌋
8  γ ← r − 1
   // Main loop
9  while n > 0 do
10     τ ←R (0, r − 1)
11     if D[τ] ≠ 0 then
12         R[D[τ]] ← R[D[τ]] · S[τ]
13     S[τ] ← S[γ]^m ; D[τ] ← n mod m
14     n ← ⌊n/m⌋ ; γ ← τ
15 for i = r − 1 down to 0 do
16     R[D[i]] ← R[D[i]] · S[i]
   // Aggregation
17 A ← R[m − 1]
18 for i = m − 2 down to 1 do
19     R[i] ← R[i] · R[i + 1]
20     A ← A · R[i]
21 return A
```

Basically, Algorithm 1 makes use of a precomputed table S to store r values $A^{(i)} = x^{m^i}$, $0 \le i \le \ell$, and a list of r corresponding digits d_i of the exponent. For example:

$$S = \{A^{(3)}, A^{(6)}, A^{(2)}\} = \{x^{m^3}, x^{m^6}, x^{m^2}\},$$

corresponding with the list:

$$D = \{d_3, d_6, d_2\}.$$

Firstly, Algorithm 1 precomputes and stores x^{m^i}, for $i \in \{0, \ldots r - 1\}$. Then, at each step, the algorithm chooses at random a stored value $S[\tau]$, updates the corresponding accumulator $R[D[\tau]] = R[D[\tau]] \cdot S[\tau]$ (lines 10–12), computes the next precomputed value $S[\gamma]^m$, and finally overwrites this value to the register $S[\tau]$ that has been chosen to compute (line 13). By performing in such a random order, an attacker can't guess the value of digit being processed at a specific point in time for each acquisition in a set of acquisitions. The randomization of Algorithm 1 is performed within one exponentiation, and thus it is able to inhibit power analysis attacks that allow operations to be distinguished from one acquisition as discussed in Section II-A.

Although Algorithm 1 reduced the required memory compared to the original idea (i.e., r instead of ℓ group elements), it still requires a large amount of memory to store precomputed values to provide a suitable level of random ordering, and thus guarantee the security against Big Mac attacks and its extension. In [21], the author stated that one needs $r > m$ to add as much randomness as in the exponent n and that his algorithm is not suitable to implement exponentiations in $(\mathbb{Z}/N\mathbb{Z})^*$ (see an analysis in [21, Section 6] for more details).

III. THE PROPOSED ALGORITHM

A. OBSERVATIONS

Our improvements are from the following observations. In the precomputed table of Algorithm 1, there may exist two values $(S[\tau], S[\tau']) = (A^{(i)}, A^{(i')})$ such that $d_i = d_{i'}$. That is, table S may contain some values $A^{(i)}$ having the same digit values d_i. For example, if we have:

$$S = \{S[0], S[1], S[2]\} = \{A^{(3)}, A^{(6)}, A^{(2)}\} = \{x^{m^3}, x^{m^6}, x^{m^2}\},$$

and corresponding digits d_i may be,

$$D = \{d_3, d_6, d_2\} = \{2, 3, 3\}.$$

The elements $S[1]$, $S[2]$ stores values x^{m^6}, x^{m^2} corresponding digits d_6, d_2, and these two digits have the same value of 3.

As mentioned in Section II-A, the horizontal collision-correlation power analysis attacks assume that a collision-correlation can be detected when the output of an operation is the input to another operation, *i.e.,* these two operation are processing the same value of d_i. In the above example, if $S[1]$ and $S[2]$ are consecutively processed, an attacker will be able to detect a collision.

In order to avoid such a potential collision attack, we won't allow any repetition in the precomputed table. We store in $S[j]$ the value of $A^{(i)}$, where $d_i = j$. Namely, if $d_i = 0$, we store

[1]Note that, updating of $R[D[i]]$ at *line* 16 of Algorithm 1 is only able to be performed if $D[i] \neq 0$. This was not mentioned in [21].

$A^{(i)}$ in $S[0]$, if $d_{i'} = 1$, we store $A^{(i')}$ in $S[1]$ and so on till $S[m-1]$. For example, if we have:

$$S = \{A^{(3)}, A^{(6)}, A^{(2)}\} = \{x^{m^3}, x^{m^6}, x^{m^2}\},$$

then the corresponding digits d_i must be:

$$D = \{d_3, d_6, d_2\} = \{0, 1, 2\}.$$

The size of the precomputed table thus could be fixed to m instead of r as in Algorithm 1.

Even, we can do better by using the sliding window technique to reduce the number of accumulators required. Since the digits d_i are only odd numbers, *i.e.,* $d_i \in \{1, 3, \ldots, m-1\}$, our algorithm thus requires only $\frac{m}{2}$ memory registers for the precomputed table. For example, $m = 8$:

$$S = \{A^{(3)}, A^{(6)}, A^{(2)}, A^{(5)}\} = \{x^{m^3}, x^{m^6}, x^{m^2}, x^{m^5}\},$$

then,

$$D = \{d_3, d_6, d_2, d_5\} = \{1, 3, 5, 7\}.$$

B. ALGORITHM

This section describes our random-order sliding window algorithm, which offers the following features:

1) it requires less memory registers than Tunstall's algorithm, that is, $(m + 1)$ instead of $(m + r)$ registers. It is thus more likely to implement exponentiations in $(\mathbb{Z}/N\mathbb{Z})^*$;
2) it doesn't use a fixed base (i.e., m), but varies the base. Hence, more potential values of digits d_i are generated for a fixed exponent n (line 7–10 in Algorithm 2). This increases the level of randomness compared to Algorithm 1 and thus minimizes the collision-correlation between operations.

Likewise, we denote $R[j]^{(i)}$ (resp. $A^{(i)}$) for the value of the accumulator $R[j]$ (resp. A) before entering iteration i. Like the left-to-right sliding window method, Algorithm 2 is treating w binary digits $d = (n_{i+w-1}, \ldots, n_i)_2$ in the case $n_i = 1$ as follows:

(i) Algorithm 2 randomly chooses $e \in \{1, 3, \ldots, m-1\}$, then update the accumulator $R[e] = R[e] \times S[e]$, and $S[e] = 1$.
(ii) If the register $S[d_i]$ is *available* (that is, $S[d_i] = 1$), Algorithm 2 stores $S[d_i]$ in the precomputed table, $S[d_i] = A$, and updates n (*lines* 16–18).
(iii) Otherwise, if the register $S[d_i]$ is not available (*i.e.,* $S[d_i] \neq 1$), we reduce the size of the current window to find another digit that has not yet been delayed (*lines* 14–15). If there is no available register found, the algorithm repeats Step (i), perfoms one multiplication and releases one register $S[e]$.

The explicit description of the proposed algorithm is given in Algorithm 2. Instead of decomposing the exponent in k fixed windows of $w(= \log m)$ bits, the proposed algorithm treats bit-by-bit from the least significant bit to the most significant bit. The algorithm performs a square of A (*line* 7) if $n_i = 0$, however, this may not be the actual bit value as n is further processed at the *line* 18. To inhibit the simple power analysis attack, the proposed algorithm requires squaring and multiplication operations to be performed in the same routine, *i.e.,* using atomic principle (see [5]).

Algorithm 2: Random Order Sliding Window Exponentiation

Input: $x \in \mathbb{G}$, $w = \log_2 m$, and an k-bit integer $n = (n_{k-1}, \ldots, n_1, n_0)_2 \in \mathbb{N}$
Output: x^n

```
1  for j = 1 to m/2 do
2      R[2j − 1] ← 1
3      S[2j − 1] ← 1
4  i ← 0 ;      A ← x
   // Main loop
5  while i < k do
6      if (n_i = 0) then
7          A ← A × A
8          i = i + 1
9      else
10         e ←R {1, 3, ... m − 1}
11         R[e] ← R[e] × S[e]
12         S[e] ← 1
13         d ← (n_{i+w−1}, ..., n_{i+1}, n_i)_2
14         while (S[d] ≠ 1) and (d > 0) do
15             lshift(d, 1) // left shift d by one bit
16         if d > 0 then
17             S[d] ← A
18             n ← n − 2^i d
19 for j = 1 to m/2 do
20     R[2j − 1] ← R[2j − 1] × S[2j − 1]
21 A ← ∏_{d∈{1,3,5,...,m−1}} R[d]^d
22 return A
```

C. EXAMPLE

We demonstrate the correctness of the proposed algorithm by the following example. Let us compute x^n, where $n = 7871 = (1\,1110\,1011\,1111)_2$. The digits will be read three bits per time, that is, $w = 3$ and $m = 8$. Algorithm 2 hence needs 9 (*i.e.,* $m + 1$) memory registers to store group elements, that include 4 accumulators $R[j]$, and 4 registers $S[j]$ for $j \in \{1, 3, 5, 7\}$. Initially, all these registers are set to 1 (*lines* 2–3).

- **Step 1**: Algorithm 2 chooses e at random from the set $\{1, 3, 5, 7\}$ (line 12). Without loss of generality, we assume $e = 3$, Algorithm 2 updates $R[3] \leftarrow R[3] \times S[3]$ $(= 1 \times 1)$ (*line* 11). It then takes first 3 bits from right, *i.e.,* $d = (111)_2$ (*line* 13). Since the stored table is empty, the proposed algorithm assigns $S[7] = x$

(*line* 17), and computes $n = n - 7$ (*line* 18), *i.e.,* $n = 7864 = (1\,1110\,1011\,1000)_2$.
Algorithm 2 then performs 3 consecutive multiplications $A \leftarrow A \times A$. After these operations, $i = 2, A = x^8$ and the array $(S[1], S[3], S[5], S[7]) = (1,\ 1,\ 1,\ x)$.

- **Step 2**: Algorithm 2 chooses a random e. Assuming $e = 1$, it then updates $R[1] \leftarrow R[1] \times S[1] = 1 \times 1$. Since the next 3 bits have the same value with the digit that has just been stored, *i.e.,* $d = (111)_2$, Algorithm 2 seeks another digit (*lines* 14–15) and finds $d = (11)_2$. It assigns $S[3] = A = x^8$ (*line* 17), and computes $n = n - 2^3 \times 3$, *i.e.,* $n = 7840 = (1\,1110\,1010\,0000)_2$. Then, it performs two consecutive multiplications $A \leftarrow A \times A$ (*line* 7). At the end of this step, $i = 4, A = x^{32}$ and the array $(S[1], S[3], S[5], S[7]) = (1,\ x^8,\ 1,\ x)$.
- **Step 3**: Algorithm 2 chooses a random e from the set $\{1, 3, 5, 7\}$, *e.g.,* $e = 7$, computes $R[7] \leftarrow R[7] \times S[7]$ $(= 1 \times x)$, and updates $S[7] = 1$ (*lines* 11–12). It takes next 3 bits, *i.e.,* $d = (101)_2$, updates $S[5] = x^{32}$ and $n = n - 2^5 \cdot 5$, *i.e.,* $n = 7680 = (1\,1110\,0000\,0000)_2$ (*lines* 17-18). Algorithm 2 then performs 4 consecutive multiplications $A \leftarrow A \times A$ (*line* 7). After that, $i = 8$, $A = x^{512}$ and the array $(S[1], S[3], S[5], S[7]) = (1,\ x^8,\ x^{32},\ 1)$.
- **Step 4**: Algorithm 2 randomly chooses e from the set $\{1, 3, 5, 7\}$. Assuming $e = 1$, it updates $R[1] = R[1] \times S[1]$ $(= 1 \times 1)$, $S[1] = 1$ (*lines* 11–12). It then takes next 3 bits, *i.e.,* $d = (111)_2$. Since $S[7] = 1$, it assigns $S[7] = x^{512}$, and compute $n = n - 2^9 \times 7$, *i.e.,* $n = 4096 = (1\,0000\,0000\,0000)_2$. Then, it performs three consecutive multiplications $A \leftarrow A \times A$ (*line* 7). At the end of this step, $i = 11, A = x^{4096}$ and the array $(S[1], S[3], S[5], S[7]) = (1,\ x^8,\ x^{32},\ x^{512})$.
- **Step 5**: Algorithm 2 chooses a random e, for example $e = 5$; then updates $R[5] \leftarrow R[5] \times S[5]$ $(= 1 \times x^{32})$, and $S[5] = 1$. It takes the last bit, *i.e.,* $d = 1$, updates $S[1] = x^{4096}$, and $n = n - 2^{12} \cdot 1 = 0$. It also performs one multiplication $A \leftarrow A \times A$. After that, $i = 12$, $A = x^{8192}$ and the array $(S[1], S[3], S[5], S[7]) = (x^{4096},\ x^8,\ 1,\ x^{512})$.
- **Step 6**: Algorithm 2 does a final update for accumulators $R[j]$ (*lines* 19–20). That is, $R[1] = R[1] \times S[1] = 1 \times x^{4096}$, $R[3] = R[3] \times S[3] = 1 \times x^8$, $R[5] = R[5] \times S[5] = x^{32} \times 1$ and $R[7] = R[7] \times S[7] = x \times x^{512} = x^{513}$.
- **Step 7**: Finally, *line* 21 of Algorithm 2 computes the result of the product $\prod_{j\in\{1,3,5,7\}} R[j]^j = x^{4096} \times (x^8)^3 \times (x^{32})^5 \times (x^{513})^7 = x^{7871}$.

D. PERFORMANCE CONSIDERATION

From the memory point-of-view, the main advantage of Algorithm 2 is the number of memory registers required to be only $m + 1$ instead of $m + r$ as in the original algorithm. It is also worth to note that, unlike Tunstall's algorithm, the proposed algorithm doesn't require an array D of r elements (see Algorithm 1) to store the values of the digits.

TABLE 1. Performance and security comparison between right-to-left *m*-ary algorithms.

Algorithm	# registers	Secure against Big Mac-like attacks
Algorithm 4	m	No
Tunstall's algorithm [21]	$m + r$	Yes
Proposed algorithm	$m + 1$	Yes

As suggested in [21], r should be greater than m. Thus, the proposed algorithm approximately requires a half of the number of registers required in Tunstall's algorithm. It also is worth to note that the number of registers required in the proposed algorithm is competitive with that in Yao's m-ary algorithm (Algorithm 4) that is insecure against the Big Mac attack (as analyzed in Section V). Since the size of window will reduce in some cases (*lines* 11–12), Algorithm 2 may require slightly more multiplications than the m-ary window method. A summary of comparison is given in Table 1.

IV. A BINARY IMPLEMENTATION

A. ALGORITHM

This section presents a variant of the proposed algorithm in the binary case, *i.e.,* $m = 2$. The explicit description of the proposed solution is given in Algorithm 3.

In this description, $A[j]$ (resp. $D[j]$), for $j \in \{0, 1\}$, denote the delayed value of $A^{(i)}$ at a previous round i (resp. the value of bit that was delayed, *i.e.,* $D[j] = n_i \in \{0, 1\}$). When the value of $D[j]$ is set to $\emptyset$, it means that there is no delayed operation associated to the register $A[j]$. We also define a function $lookup(e, D)$, looking for the first element in the array D whose value is equal to e. If found, it returns the index j of that element, that is, $e = D[j]$. If not, it returns $\emptyset$. Finally, as the right-to-left square-and-multiply always, the *accumulator* $R[1]$ (resp., $R[0]$) accumulates and outputs the values x^n (resp., x^{2^k-n-1}), where k is the bit-length of the exponent n.

The algorithm works as follows. At each step, depending on the randomly chosen value b (*line* 5), Algorithm 3 will perform a multiplication related to $R[n_i]$ (*line* 7 or *line* 18) or to $R[\neg n_i]$ if there is previously a delayed multiplication related to $R[\neg n_i]$(*line* 11). In the case there is no delayed multiplication related to $R[\neg n_i]$ in the queue, Algorithm 3 searches if there is an available register (either $A[0]$ or $A[1]$) to store the current A (*i.e.,* try to delay the current multiplication involved to $R[n_i]$) (*line* 14–16). If there is no such a register (*i.e.,* both registers $A[0]$ and $A[1]$ are occupied by delayed multiplications involved to $R[n_i]$), Algorithm 3 updates $R[n_i]$ with one of the previous stored values, $A[n_i]$ (*line* 18), then saves the current A to $A[n_i]$ (*line* 19).

B. EXAMPLE

We demonstrate the correctness of the above algorithm by the following example. Let us compute x^n, where $n = 135 = (1000\,0111)_2$.

- **Step 1**: Algorithm 3 processes the first bit, $n_0 = 1$. At first, it randomly chooses $b \leftarrow \{0, 1\}$ (*line* 5).

Algorithm 3: Random Order Binary Exponentiation

Input: $x \in \mathbb{G}, n = (n_{k-1}, \ldots, n_1, n_0)_2 \in \mathbb{N}$
Output: x^n

1 **for** $i = 0$ **to** 1 **do**
2 $R[i] \leftarrow 1; A[i] \leftarrow 1; D[i] \leftarrow \emptyset$;
3 $A \leftarrow x$
4 **for** $i = 0$ **to** $k-1$ **do**
5 $b \xleftarrow{R} \{0, 1\}$;
6 **if** $b = 1$ **then**
7 $R[n_i] \leftarrow R[n_i] \times A$;
8 **else**
9 $j \leftarrow lookup(\neg n_i, D)$;
10 **if** $(j \neq \emptyset)$ **then** // Exist a delayed multiplication related to $R[\neg n_i]$; update $R[\neg n_i]$ and delay the multiplication for $R[n_i]$
11 $R[\neg n_i] \leftarrow R[\neg n_i] \times A[j]$;
12 $A[j] \leftarrow A$; $D[j] = n_i$;
13 **else**
14 $j \leftarrow lookup(\emptyset, D)$;
15 **if** $j \neq \emptyset$ **then** // have a free space
16 $A[j] \leftarrow A$; $D[j] = n_i$;
17 **else** // $A[0], A[1]$ occupied by delayed multiplications involved $R[n_i]$
18 $R[n_i] \leftarrow R[n_i] \times A[n_i]$;
19 $A[n_i] \leftarrow A$;
20 $A \leftarrow A \times A$;
21 **for** $i = 0$ **to** 1 **do**
22 $R[D[j]] \leftarrow R[D[j]] \times A[j]$;
23 **return** $R[1]$

Assuming $b = 0$, Algorithm 3 executes *line* 9, looking for a delayed operation. As there is no such an operation, it goes to *line* 13 to look for an available accumulator to delay the operation (*line* 14). As found ($D[0] = D[1] = \emptyset$), Algorithm 3 executes *line* 16, assigns $A[0] = x$, $D[0] = 1$. *Line* 20 updates $A = x^2$. After this, $R[0] = R[1] = 1, A[0] = x, A[1] = 1, D[0] = 1, D[1] = \emptyset$.

- **Step 2**: Algorithm 3 processes the 2^{nd} bit, $n_1 = 1$ by first choosing a random b. Assuming $b = 1$, it then updates $R[1] \leftarrow R[1] \times A = x^2$ (*line* 7) and $A = A \times A = x^4$ (*line* 20).
- **Step 3**: Algorithm 3 processes the 3^{rd} bit, $n_2 = 1$. It randomly chooses b. Assuming $b = 0$, Algorithm 3 executes *line* 9, looking for a delayed multiplication involved to the bit value 0. As there is not such an operation, it goes to *line* 13 to look for an available accumulator to delay the operation (*line* 14). Since $A[1]$ is free ($D[1] = \emptyset$), Algorithm 3 executes *line* 16, assigns $A[1] = x^4$, $D[1] = 1$. Line 20 updates $A = x^8$. At the end of this step, $R[0] = 1, R[1] = x^2, A[0] = x, A[1] = x^4, D[0] = 1, D[1] = 1$.
- **Step 4**: Algorithm 2 processes the 4^{th} bit, $n_3 = 0$ by randomly choosing b. Assuming $b = 1$, it then updates $R[0] \leftarrow R[0] \times A = x^8$ (*line* 7) and $A = A \times A = x^{16}$ (*line* 20). At the end of this step, $R[0] = x^8, R[1] = x^2, A[0] = x, A[1] = x^4, D[0] = 1, D[1] = 1$.
- **Step 5**: Algorithm 3 processes the 5^{th} bit, $n_4 = 0$. Assuming a random $b = 0$ was chosen, Algorithm 3 executes line 9, looking for a delayed multiplication involved to the bit value 1. Because $D[0] = D[1] = 1$, it executes *lines* 11–12, assigns $R[1] = R[1] \times A[0] (= x^2 \times x = x^3)$, $A[0] = x^{16}$, $D[0] = 0$. *Line* 20 updates $A = x^{32}$. At the end of this step, $R[0] = x^8, R[1] = x^3, A[0] = x^{16}, A[1] = x^4, D[0] = 0, D[1] = 1$.
- **Step 6**: Algorithm 3 processes the 6^{th} bit, $n_5 = 0$. Assuming a random $b = 0$ was chosen, Algorithm 3 executes *line* 9, looking for a delayed operation involved to the bit value 1. Because $D[0] = 0$, and $D[1] = 1$, Algorithm 3 executes *lines* 11–12, assigns $R[1] = R[1] \times A[1] (= x^3 \times x^4 = x^7)$, $A[1] = x^{32}$, $D[1] = 0$. *Line* 20 updates $A = x^{64}$. At the end of this step, $R[0] = x^8, R[1] = x^7, A[0] = x^{16}, A[1] = x^{32}, D[0] = 0, D[1] = 0$.
- **Step 7**: Algorithm 3 processes the 7^{th} bit, $n_6 = 0$. It randomly chooses b. Again, assuming $b = 0$, Algorithm 3 executes *line* 9, looking for a delayed operation involved to the bit value 1. Since both $D[0]$ and $D[1]$ are equal to 0, Algorithm 3 goes to *line* 13, searches if there is any available memory, but it does not happen, it goes to *line* 17, then *lines* 18–19 update $R[0] = R[0] \times A[0] (= x^8 \times x^{16} = x^{24})$, and $A[0] = x^{64}$. *Line* 20 updates $A = x^{128}$. At the end of this step, $R[0] = x^{24}, R[1] = x^7, A[0] = x^{64}, A[1] = x^{32}, D[0] = 0, D[1] = 0$.
- **Step 8**: Algorithm 3 processes the 8^{th} bit, $n_7 = 1$. It randomly chooses b. Assuming $b = 0$, Algorithm 3 executes *line* 9, looking for a delayed multiplcation involved to the bit value 0. As there exist such an operation ($D[0] = D[1] = 0$), it executes *lines* 11–12, assigns $R[0] = R[0] \times A[0] (= x^{24} \times x^{64} = x^{88})$, $A[0] = x^{128}$, and $D[0] = 1$. *Line* 20 updates $A = x^{256}$. At the end of this step, $R[0] = x^{88}, R[1] = x^7, A[0] = x^{128}, A[1] = x^{32}, D[0] = 1, D[1] = 0$.
- **Step 9**: *Lines* 21–22 of Algorithm 3 compute the final results $R[1] = R[1] \times A[0] = (x^7 \times x^{128} = x^{135})$, and $R[0] = R[0] \times A[1] = (x^{88} \times x^{32} = x^{120})$. Finally, Algorithm 3 returns $R[1]$ as its output, that is x^{135}.

C. DISCUSSION

As the right-to-left square-and-multiply always algorithm, the proposed binary algorithm performs two group operations, one multiplication and one squaring, per bit. From the security viewpoint, at the round i, an attacker couldn't determine the value of bit n_i. That is because at that round the proposed algorithm would perform a multiplication related to $R[n_i]$ or to $R[\neg n_i]$ with the probability $1/2$ if the value of bits has a uniform distribution. The proposed algorithm is thus resistant to the Big Mac attack and its extensions.

TABLE 2. Performance and security Comparison between right-to-left binary algorithms.

	R2L multiply always [3]	Algorithm 3 in [15]	Algorithm 3
No of registers	3	3	5
Fault attacks	Yes	Yes	Yes
Safe-fault attacks	No	Yes	Yes
Big Mac attack	No	No	Yes
Combined attack	No	Yes	Yes

To the best of our knowledge, it is the first right-to-left binary exponentiation algorithm that resists to such attacks, and remains the same performance, *i.e.,* it requires 2 group operations per bit. On the other hand, Algorithm 3 requires 5 instead of 3 registers in comparison to the right-to-left square-and-multiply always algorithm.

Secure Against Fault Analysis

The outputs of $R[0]$ and $R[1]$ of Algorithm 3 can be used to prevent the fault attacks as suggested by Boscher *et al.* in [3] due to this relation: $x \times R[0] \times R[1] = A$. As it can be seen in the above example, we have $x \times x^{120} \times x^{135} = x^{256}$.

Both Algorithm 2 and Algorithm 3 are resistant to combined attack [2] and safe-error attacks [26] because at the i-th loop, the digit processed may not be n_i (counting from right to left) with high probability, and hence the attacker learns nothing about the value of n_i. In addition, Algorithms 2–3 are both secure against safe-error attacks because they don't have dummy operations. A security comparison between right-to-left binary algorithms is shown in Table 2.

V. SECURITY ANALYSIS

A. SIMPLE POWER ANALYSIS

Using the atomic principle [5], Algorithm 2 requires that the multiplication and squaring operations are implemented by using identical code and, therefore, cannot be distinguished easily. In the case the attacker can distinguish a multiplication from a squaring by using statistical methods (*e.g.,* [1]), she/he may learn about the number of bits '0' in the secret exponent but it is unclear whether she/he would determine the real value of the current bit because at each iteration the proposed algorithm performs a squaring without considering the current bit is '0' or '1'.

B. DIFFERENTIAL POWER ANALYSIS

To thwart DPA attacks, classical blinding techniques (*e.g.,* message, exponent blinding) with a big enough randomness (*e.g.,* 48-bits) can be applied. However, as discussed in Section II-A, the Big Mac attack may defeat all these blinding techniques.

In the following section, we focus on analyzing the security of the proposed algorithm in the presence of the Big Mac attack and its extensions, that is statistical side-channel analysis attacks in the horizontal setting. We start with an extension of the Big Mac collision-correlation attack to analyze the security of the right-to-left m-ary exponentiation and the security of the random order right-to-left m-ary exponentiation algorithm. Then, we analyze the security of the proposed algorithm.

C. HORIZONTAL COLLISION-CORRELATION ANALYSIS

1) ON RIGHT-TO-LEFT m-ARY EXPONENTIATION

For implementations of the right-to-left binary algorithms, Hanley *et al.* [11] presented horizontal collision correlation analysis on Joye's add-only exponentiation algorithm [13], and then Feix *et al.* [9] presented a similar attack in the right-to-left square-and-multiply always [13]. While, the former uses the fact that the register R_0 (resp. R_1) remains the same when the value of bit being processed $n_j = 0$ (resp. $n_j = 1$), the later uses the fact that if the two consecutive bits have the same value then the output of the multiplication in the previous loop will be the input of the multiplication in the next loop.

In the case of the right-to-left m-ary exponentiation with $m > 2$, we assume that m is a power of 2 so that raising to the m-th power is a sequence of $\log_2 m$ squarings. For convenience, we also assume that the mth power can be detected by recognizing squares from multiplications. Similar to [9], the attack uses the fact that the adversary can detect a collision-correlation when the output of an operation is the input to another operation (*i.e.,* operations processing the same value of d_i). As Big Mac attack, the attacker must then partition the multiplications (*line* 7 in Algorithm 4) into disjoint sets for which the digits d_i have the same values. Once the partitioning has been performed, there are $(m-1)!$ ways of associating specific different digit values with the $m-1$ sets of multiplications. One of these choices will yield the sought key. Because the value of m shouldn't be too big, this attack computationally can be performed.

2) ON THE RANDOM ORDER m-ARY EXPONENTIATION

We revise the security of Tunstall's algorithm against the Big Mac collision-correlation analysis attack and show that under this attack, we don't need to set $r > m$ to get more randomness than $r = m$. Likewise, we use the fact that the attacker would detect a collision-correlation when the output of a multiplication is the input to another multiplication (*i.e.,* the multiplication involving in the accumulators $R[j]$). If an attacker attempts a collision-correlation analysis attack, it would be assumed that the digit treated at the t-th loop has the same value with the $(t + r - 1)$-th digit of the exponent, d_{t+r-1} (counting from right) that has just been included the set of digit from which the algorithm will randomly choose. As long as such a digit chosen, the attacker can detect a collision-correlation because the accumulator $R[d_{t+r-1}]$ will be the first operand of the multiplication in line 13, Algorithm 1. This is different from the security analysis in [21, Section 6.2], where the partial correlation is detected due to the second operand of the multiplication, that is there is correlation when the digit treated must be one that has been included. Let us assume that the values of digits have a uniform distribution. So, it doesn't matter what size r is

of (for $r > m$), the probability a digit chosen having the same value with d_{k+r-1} is $1/m$. In this setting of attacks, to balance the memory performance and the security, the best value of r should be m. On the other hand, since the digit treated is randomly chosen at the t-th loop, this digit wouldn't have the same value of d_{t+r-1} with the probability $\frac{(m-1)}{m}$. This randomization is performed within one execution of the exponentiation, Tunstall's algorithm is hence secure against the horizontal collision-correlation power attacks.

3) SECURITY OF THE PROPOSED ALGORITHM

The proposed exponentiation algorithm (Algorithm 2) randomly performs multiplications from the set of delayed multiplications. Similar with Tunstall's algorithm, assume that a side-channel attacker learns the value of digit e (line 13, Algorithm 2) being processed at the loop t, however she/he may not learn about the real position of this e. Thus, the proposed is secure against the Big Mac attack and its extensions that deduce the secret information from a single power trace.

Unlike Tunstall's algorithm, Algorithm 2 doesn't use a fixed base m, but varies it by using the sliding-window technique. In each iteration, the proposed algorithm tries to find a good digit to execute, it therefore minimizes the possibility that the digit treated has the same value of the digit that has been included. Moreover, this allows the proposed algorithm to generate plural unpredictable values of digits $\{d_0, d_1, \ldots, d_{\ell-1}\}$ for a fixed exponent n. If the attacker collects different power traces, she/he would deduce different combinations of digits d_i.

As analyzed in [21], the random-order exponentiation algorithms can be used as a supplement, rather than as a replacement, to the blinding countermeasures. By combining the random order algorithm with a blinding countermeasure, exponentiation implementations should be resistant to statistical side-channel analysis in both vertical and horizontal setting.

VI. CONCLUSION

In this article, we revisited Tunstall's random order m-ary exponentiation algorithm. We considered its security against the Big Mac collision-correlation analysis attack and then present a memory-efficient variant. The proposed algorithm requires only $(m + 1)$ group elements in memory instead of $(m + r)$. Finally, we presented an efficient implementation in the binary case. To the best of our knowledge, it is the first right-to-left binary exponentiation algorithm that resists to side-channel attacks only using a single consumption trace such as the combined attacks or the horizontal collision-correlation attacks.

APPENDIX. EXPONENTIATION ALGORITHMS

A right-to-left m-ary version of Algorithm 4 was described by Yao in [25] and hence it is often referred as Yao's algorithm.

For example, one needs to compute x^n, where $n = 871$. The binary representation of n is $(1101100111)_2$. If one use Algorithm 4 with $m = 4$ (i.e., $n = (31213)_4$), the register A will be initialized to x, and then raise to the power of 4 at each iteration, *i.e.*, $x \to x^4 \to x^{16} \to x^{64} \to x^{256}$. The order updating accumulators will be $R[3] \to R[1] \to R[2] \to R[1] \to R[3]$. That is, $R[3]^{(1)} \xrightarrow{R[3]^{(0)} \times x} x$, $R[1]^{(2)} \xrightarrow{R[1]^{(1)} \times x^4} x^4$, $R[2]^{(3)} \xrightarrow{R[2]^{(2)} \times x^{16}} x^{16}$, $R[1]^{(4)} \xrightarrow{R[1]^{(3)} \times x^{64}} x^4 \cdot x^{64} = x^{68}$, $R[1]^{(5)} \xrightarrow{R[1]^{(4)} \times x^{256}} x \cdot x^{256} = x^{257}$.

Algorithm 4: Right-to-Left m-Ary Exponentiation Algorithm

Input: $x \in \mathbb{G}$, $n = (n_{k-1}, \ldots, n_1, n_0)_m \in \mathbb{N}$
Output: x^n

1 **for** $j = 1$ **to** $m - 1$ **do**
2 $R[j] \leftarrow 1_{\mathbb{G}}$
3 **for** $i = 0$ **to** $k - 1$ **do**
4 **if** $n_i \neq 0$ **then**
5 $R[n_i] \leftarrow R[n_i] \cdot A$
6 $A \leftarrow A^m$
7 $A \leftarrow R[m - 1]$
8 **for** $i = m - 2$ **down to** 1 **do**
9 $R[i] \leftarrow R[i] \cdot R[i + 1]$
10 $A \leftarrow A \cdot R[i]$
11 **return** A

$$R[1] = 1 \cdot x^4 \cdot x^{64}$$
$$R[2] = 1 \cdot x^{16}$$
$$R[3] = 1 \cdot x \cdot x^{256}$$

Finally, $x^n = R[1] \cdot R[2]^2 \cdot R[3]^3 = x^{68}(x^{16})^2(x^{257})^3 = x^{871}$.

REFERENCES

[1] F. Amiel, B. Feix, M. Tunstall, C. Whelan, and P. W. Marnane, "Distinguishing multiplications from squaring operations," in *Selected Areas Cryptography*, R. M. Avanzi, L. Keliher, and F. Sica, Eds. Berlin, Germany: Springer-Verlag, 2009, pp. 346–360.
[2] F. Amiel, K. Villegas, B. Feix, and L. Marcel, "Passive and active combined attacks: Combining fault attacks and side channel analysis," in *Proc. Workshop Fault Diagnosis Tolerance Cryptography (FDTC)*, Washington, DC, USA, Sep. 2007, pp. 92–102.
[3] A. Boscher, R. Naciri, and E. Prouff, "CRT RSA algorithm protected against fault attacks," in *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems* (Lecture Notes in Computer Science), vol. 4462, D. Sauveron, K. Markantonakis, A. Bilas, and J.-J. Quisquater, Eds. Berlin, Germany: Springer, 2007, pp. 229–243.
[4] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems—CHES 2004* (Lecture Notes in Computer Science), vol. 3156, M. Joye and J.-J. Quisquater, Eds. Berlin, Germany: Springer, 2004, pp. 16–29.
[5] B. Chevallier-Mames, M. Ciet, and M. Joye, "Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity," *IEEE Trans. Comput.*, vol. 53, no. 6, pp. 760–768, Jun. 2004.
[6] C. Clavier, B. Feix, G. Gagnerot, C. Giraud, M. Roussellet, and V. Verneuil, "Rosetta for single trace analysis," in *Progress in Cryptology—INDOCRYPT 2012* (Lecture Notes in Computer Science), vol. 7668, S. Galbraith and M. Nandi, Eds. Berlin, Germany: Springer, 2012, pp. 140–155.
[7] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil, "Horizontal correlation analysis on exponentiation," in *Proc. 12th Int. Conf. Inf. Commun. Secur.* Berlin, Germany: Springer-Verlag, 2010, pp. 46–61.

[8] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* (Lecture Notes in Computer Science), vol. 1717, C. K. Koç and C. Paar, Eds. Berlin, Germany: Springer, 1999, pp. 292–302.
[9] B. Feix, M. Roussellet, and A. Venelli, "Side-channel analysis on blinded regular scalar multiplications," *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 191, Dec. 2014.
[10] L. Gaspar, "Combining leakage-resilient PRFs and shuffling," in *Proc. 13th Int. Conf. Smart Card Res. Adv. Appl. (CARDIS)*, vol. 8968. Paris, France: Springer, 2015, p. 122.
[11] N. Hanley, H. Kim, and M. Tunstall, "Exploiting collisions in addition chain-based exponentiation algorithms using a single trace," in *Topics in Cryptology—CT-RSA 2015* (Lecture Notes in Computer Science), vol. 9048, K. Nyberg, Ed. Berlin, Germany: Springer, 2015, pp. 431–448.
[12] C. Herbst, E. Oswald, and S. Mangard, "An AES smart card implementation resistant to power analysis attacks," in *Applied Cryptography and Network Security* (Lecture Notes in Computer Science), vol. 3989. Berlin, Germany: Springer, 2006, pp. 239–252.
[13] M. Joye, "Highly regular right-to-left algorithms for scalar multiplication," in *Cryptographic Hardware and Embedded Systems—CHES 2007* (Lecture Notes in Computer Science), vol. 4727. Berlin, Germany: Springer, 2007, pp. 135–147.
[14] M. Joye and S.-M. Yen, "The Montgomery powering ladder," in *Cryptographic Hardware and Embedded Systems—CHES 2002*, vol. 2523, B. S. Kaliski, Ç. K. Koç, and C. Paar, Eds. Berlin, Germany: Springer. 2003, pp. 291–302.
[15] H. Kim, Y. Choi, D. Choi, and J. Ha, "A secure exponentiation algorithm resistant to a combined attack on RSA implementation," *Int. J. Comput. Math.*, vol. 93, no. 2, pp. 258–272, 2016.
[16] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO' 99* (Lecture Notes in Computer Science), vol. 1666, M. J. Wiener, Ed. Berlin, Germany: Springer, 1999, pp. 388–397.
[17] C. P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO' 96* (Lecture Notes in Computer Science), vol. 1109, N. Koblitz, Ed. Berlin, Germany: Springer, 1996, pp. 104–113.
[18] D.-P. Le, M. Rivain, and C. H. Tan, "On double exponentiation for securing RSA against fault analysis," in *Topics in Cryptology—CT-RSA 2014* (Lecture Notes in Computer Science), vol. 8366, J. Benaloh, Ed. San Francisco, CA, USA: Springer, 2014, pp. 152–168.
[19] D.-P. Le, C. H. Tan, and M. Tunstall, "Randomizing the montgomery powering ladder," in *Information Security Theory and Practice* (Lecture Notes in Computer Science), vol. 9311. Heraklion, Greece: Springer, 2015, pp. 169–184.
[20] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, 1997.
[21] M. Tunstall, "Random order *m*-ary exponentiation," in *Information Security and Privacy* (Lecture Notes in Computer Science), vol. 5594, C. Boyd and J. M. G. Nieto, Eds. Berlin, Germany: Springer, 2009, pp. 437–451.
[22] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *Proc. 18th Int. Conf. Theory Appl. Cryptol. Inf. Secur.* Berlin, Germany: Springer-Verlag, 2012, pp. 740–757.
[23] C. D. Walter, "Sliding windows succumbs to big Mac attack," in *Cryptographic Hardware and Embedded Systems—CHES 2001* (Lecture Notes in Computer Science), vol. 2162, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Berlin, Germany: Springer, 2001, pp. 286–299.
[24] M. F. Witteman, J. G. V. Woudenberg, and F. Menarini, "Defeating RSA multiply-always and message blinding countermeasures," in *Topics in Cryptology—CT-RSA 2011* (Lecture Notes in Computer Science), vol. 6558, A. Kiayias, Ed. Berlin, Germany: Springer, 2011, pp. 77–88.
[25] A. C.-C. Yao, "On the evaluation of powers," *SIAM J. Comput.*, vol. 5, no. 1, pp. 100–103, 1976.
[26] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Trans. Comput.*, vol. 49, no. 9, pp. 967–970, Sep. 2000.

DUC-PHONG LE received the Ph.D. degree in computer science from the University of Pau et des Pays de l'Adour, in August 2009. He was a Postdoctoral Fellow with the Algorithms Research Group, University of Caen, Base Normandie, from 2009 to 2010, a Research Scientist with the National University of Singapore (NUS), Singapore, from October 2010 to May 2016, a Senior Security Analyst with Underwriters Laboratories, from May 2016 to June 2017, and the Scientist II of the Institute for Infocomm Research (I2R), Agency of Science, Technology and Research (A*STAR), Singapore, from July 2017 to March 2019. He is currently working as the Research Team Lead of the Canadian Institute for Cybersecurity (CIC), University of New Brunswick, Canada. His research interests include applied cryptography, elliptic curve cryptography, secure and efficient implementations, applied machine learning to cybersecurity issues, and blockchain.

ALI A. GHORBANI (Senior Member, IEEE) has held a variety of academic positions for the past 39 years. He is currently a Professor of computer science, the Tier 1 Canada Research Chair in cybersecurity, and the Director of the Canadian Institute for Cybersecurity, which he established, in 2016. He has served as the Dean of the Faculty of Computer Science, University of New Brunswick, from 2008 to 2017. He is also the Founding Director of the Laboratory for Intelligence and Adaptive Systems Research. He has spent over 29 years of his 39-year academic career, carrying out fundamental and applied research in machine learning, cybersecurity, and critical infrastructure protection. He is the Co-Inventor on three awarded and one filed patent in the fields of cybersecurity and web intelligence. He has published over 280 peer-reviewed articles during his career. He has supervised over 190 research associates, postdoctoral fellows, and students during his career. His book *Intrusion Detection and Prevention Systems: Concepts and Techniques* (Springer, October 2010). He developed several technologies adopted by high-tech companies and co-founded three startups, Sentrant Security, EyesOver Technologies, and Cydarien Security, in 2013, 2015, and 2019, respectively. He was a recipient of the 2017 Startup Canada Senior Entrepreneur Award, and the Canadian Immigrant Magazine's RBC top 25 Canadian immigrants of 2019. He is the Co-Founder of the Privacy, Security, Trust (PST) Network in Canada and its annual international conference. He has served as the Co-Editor-in-Chief for the *International Journal of Computational Intelligence*, from 2007 to 2017.