On Double Exponentiation for Securing RSA against Fault Analysis

Duc-Phong Le[1], Matthieu Rivain[2], and Chik How Tan[1]

[1] Temasek Laboratories, National University of Singapore
{tslld,tsltch}@nus.edu.sg

[2] CryptoEpxerts, France
matthieu.rivain@cryptoexperts.com

Abstract. At CT-RSA 2009, a new principle to secure RSA (and modular/group exponentiation) against fault-analysis has been introduced by Rivain. The idea is to perform a so-called *double exponentiation* to compute a pair $(m^d, m^{\varphi(N)-d})$ and then check that the output pair satisfies the consistency relation: $m^d \cdot m^{\varphi(N)-d} \equiv 1 \bmod N$. The author then proposed an efficient heuristic to derive an addition chain for the pair $(d, \varphi(N)-d)$. In this paper, we revisit this idea and propose faster methods to perform a double exponentiation. On the one hand, we present new heuristics for generating shorter double addition chains. On the other hand, we present an efficient double exponentiation algorithm based on a right-to-left sliding window approach.

1 Introduction

Fault analysis is a cryptanalytic technique that takes advantage of errors occurring in cryptographic computations. Such errors can be induced in a device by physical means such as the variation of the power supply voltage, the increase in the clock frequency or an intensive lighting of the circuit. The erroneous results of the cryptographic computations can then be exploited in order to retrieve some information about the secret key. Fault attacks have first been introduced by Boneh et al. in [6] against RSA and other public key cryptosystems. In particular, they showed how to break RSA computed in CRT mode from a single faulty signature.

Many countermeasures have been proposed to protect embedded implementations of RSA against fault attacks. They can basically be classified in two different categories: countermeasures based on *a modulus extension* and *self-secure exponentiations*. The former countermeasures add redundancy in the computation by multiplicatively extending the RSA modulus. This approach was first introduced by Shamir in [26], and then further extended in [30, 1, 5, 10, 28]. The second approach consists in using an exponentiation algorithm that directly includes redundancy. It was first followed by Giraud in [14], who suggested to use the Montgomery ladder exponentiation algorithm. This approach was also followed by Bosher *et al.* in [8] with the square-and-multiply-always algorithm, and subsequently improved by Baek [2] and by Joye and Karroumi [17]. Eventually, Rivain proposed an alternative method in [22]. His approach is to compute a pair $(m^d, m^{\varphi(N)-d})$ in order to check the computation consistency by the relation $m^d \cdot m^{\varphi(N)-d} \equiv 1 \bmod N$. The author then presents an efficient heuristic to perform such a double exponentiation.

This paper revisits Rivain's idea and presents faster methods for double exponentiation. We first propose efficient improvements of the heuristic for double addition chains proposed in [22]. Namely, we present simple improvements that result in a speed up of 7% compared to the original method, and we investigate the use of sliding-window techniques to further improve its performances. On the other hand, we describe an efficient double exponentiation algorithm based on sliding-window technique and Yao's exponentiation [29]. Finally, we analyze the performances of our proposals and provide a comparison of the various self-secure exponentiation algorithms in the current literature.

2 Preliminaries

2.1 The RSA Cryptosystem

The RSA cryptosystem, introduced by Rivest, Shamir, and Adleman in 1978 [23], is currently the most widely used public key cryptosystem in smart devices. An RSA public key is composed of a public modulus N which is defined as the product of two large secret primes p and q, and of a public exponent e which is co-prime to $\varphi(N) = (p-1) \cdot (q-1)$ (the Euler's totient of N). The underlying RSA private key is composed of the public modulus N and the secret exponent $d = e^{-1} \bmod N$. A signature s (or deciphering) of a message m is computed by raising m to the power d modulo N, that is $s = m^d \bmod N$.

For the sake of efficiency, one often uses the Chinese Remainder Theorem (CRT). This theorem implies that $m^d \bmod N$ can be computed from $m^{d_p} \bmod p$ and $m^{d_q} \bmod q$ where $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$. The RSA in CRT mode (or RSA-CRT) then consists in computing these two *smaller* modular exponentiations and in combining the two results to recover the signature [13]. As pointed out in [21], this implementation reduces the size of the data stored in memory and is roughly four times faster than the standard implementation.

For both standard RSA and RSA-CRT, the core operation of the signature/deciphering is the modular exponentiation. The efficient implementation of RSA hence relies on an efficient exponentiation algorithm.

2.2 Addition Chains and Exponentiation

An addition chain for a positive integer a is a sequence of integers $\mathcal{C}(a) = \{a_i\}_{0 \leq i \leq n}$ beginning with $a_0 = 1$ and ending with $a_n = a$ such that each element is the sum of two previous elements in the sequence. Namely, for every $i \in \{1, 2, \ldots, n\}$ there exist $j, k \in \{0, 1, \ldots, i-1\}$ such that $a_i = a_j + a_k$. An addition chain for an integer a yields a way to evaluate the exponentiation $m \mapsto m^a$

by computing the sequence $m^{a_i} = m^{a_j} \cdot m^{a_k}$ for i from 1 to n. Conversely, any exponentiation process has an underlying addition chain. The problem of designing efficient exponentiation algorithms can hence be considered as the problem of finding short addition chains. A generalization of this problem whose instances are the tuples $(a_1, a_2, \ldots, a_k, n)$ is to find an addition chain of length n containing $a_1, \ldots, a_k$ (see for instance [12]). The later problem arises when one needs to compute simultaneously the monomials m^{a_1}, m^{a_2}, …, m^{a_k} given m and a_1, a_2, …, a_k. In this paper, we investigate the case $k = 2$; that is, given a pair of exponent (a, b) we aim at efficiently compute $m \mapsto (m^a, m^b)$. An addition chain for such a pair of exponent was called *double addition chain* in [22].

Given integers a and n, the decision problem of whether there exists an addition chain of length n for a is NP-complete. As a result, finding the shortest addition chain for an exponent a is difficult on average. That is why one relies on heuristics to perform exponentiations in practice. Some heuristics require to perform a preprocessing of the exponent and store the indices (j, k) such that $m^{a_i} = m^{a_j} \cdot m^{a_k}$ for every i. Other heuristics decide on the multiplication to perform at each step by processing the exponent on the fly.

A well-known such heuristic is the binary method also known as *square-and-multiply* method. Let $(a_{\ell-1}, \ldots, a_1, a_0)_2$ denote the binary expansion of a, namely $a = \sum_i 2^i a_i$ where $a_i \in \{0, 1\}$. The equality

$$m^a = \prod_i \left(m^{2^i}\right)^{a_i} = \prod_{i|a_i=1} m^{2^i}$$

gives rise to a simple exponentiation algorithm. At each step one computes m^{2^i} by squaring $m^{2^{i-1}}$ and then multiply it to some accumulator if $a_i = 1$. After ℓ such steps, the accumulator contains the value m^a. This process is summarized in Algorithm 1. This algorithm processes the exponent bits, from the less significant one to the most significant one and is hence often referred to as the *right-to-left* (R2L) binary algorithm. Note that a common *left-to-right* variant also exists that processes the bits in the inverse order (see for instance [19]).

Algorithm 1 R2L binary algorithm

```
Input: $m$, $a = (a_{\ell-1}, \ldots, a_1, a_0)_2 \in \mathbb{N}$
Output: $m^a$
1. $M \leftarrow m$
2. $A_1 \leftarrow 1$
3. for $i = 0$ to $\ell - 1$ do
4.     if $a_i = 1$ then $A_1 \leftarrow A_1 \cdot M$
5.     $M \leftarrow M^2$
6. end for
7. return $A_1$
```

Algorithm 2 R2L window algorithm

```
Input: $m$, $a = (u_{n-1}, \ldots, u_1, u_0)_{2^w} \in \mathbb{N}$
Output: $m^a$
1. $M \leftarrow m$
2. for $u \in \{1, 2, \ldots, 2^w - 1\}$ do $A_u \leftarrow 1$
3. for $i = 0$ to $\ell - 1$ do
4.     if $u_i \neq 0$ then $A_{u_i} \leftarrow A_{u_i} \cdot M$
5.     $M \leftarrow M^{2^w}$
6. end for
7. return $\prod_u A_u^u$
```

A generalization of the binary method consists in processing the exponent by window of w bits. Let $(u_{n-1}, \ldots, u_1, u_0)_{2^w}$ denote the expansion of some exponent a in radix 2^w where $n = \lceil \log_2(a)/w \rceil$, that is $a = \sum_i u_i 2^{iw}$ with $u_i \in \{0, 1, \ldots 2^w - 1\}$ and $u_{n-1} \neq 0$. The principle of the window method is analogous to that of the binary method and is based on the equality

$$m^a = \prod_i \left(m^{2^{iw}}\right)^{u_i} = \prod_{u=1}^{2^w-1} \Big(\prod_{i|u_i=u} 2^{iw} \Big)^u .$$

A loop is processed which applies w successive squarings in every iteration to compute $m^{2^{iw}}$ from $m^{2^{(i-1)w}}$, and which multiplies the result to some accumulator A_{u_i}. At the end of the loop each accumulator A_u contains the product $\prod_{i|u_i=u} 2^{iw}$. The different accumulators are finally aggregated as $\prod_u A_u^u = m^a$. The resulting algorithm is summarized in Algorithm 2. This algorithm was first put forward by Yao in [29] and is often referred as Yao's algorithm. It requires more memory than the binary method (specifically 2^w memory registers) but it is faster since the number of multiplications is roughly reduced to $\big(1 + \frac{1-2^{-w}}{w}\big)\ell$.

3 RSA and Fault Analysis

The first fault attacks against RSA were published in the pioneering work of Boneh *et al.* [6]. In particular, this paper describes a very efficient attack against RSA in CRT mode. The principle of the so-called *Bellcore attack* is to corrupt one of the two CRT exponentiations, and to exploit the difference between the correct and faulty signatures to recover the secret prime factors of the modulus N. For example, suppose an attacker injects a fault during the computation of $s_p = m^{d_p} \bmod p$ so that the RSA computation results in a faulty signature $\tilde{s}$ which is correct modulo q and faulty modulo p (*i.e.* $\tilde{s} \equiv s \bmod q$ and $\tilde{s} \not\equiv s \bmod p$). The difference $\tilde{s} - s$ is hence a multiple of q but is not a multiple of p, and the prime factor q can be recovered by computing $q = \gcd(\tilde{s} - s, N)$.

Implementations of RSA in standard mode (*i.e.* without CRT) are also vulnerable to fault analysis. Some attacks have been described which target the exponent [3], the public modulus [4, 9, 25] or an intermediate power of the exponentiation [6, 7, 24]. Although these attacks require several faulty signatures for a full recovery of the secret key, they constitute a practical threat that must be considered by implementors.

3.1 Securing RSA against Fault Analysis

The simplest method to thwart fault analysis is to compute the signature s twice and compare the two results. This method implies a doubling of the computation time and it cannot detect permanent errors. A more efficient way is to verify the signature s with the public exponent e. That is, the cryptographic device checks whether $m \equiv s^e \bmod N$ before returning the signature s. This method provides a perfect security since a faulty signature is systematically detected. On the other hand, this method is efficient as long as e is small (which is widely common), but in the presence of a random e, it is as inefficient as the computation doubling. Besides, in some applications (*e.g.* the Javacard API for RSA signature [27]), the public exponent e is not available to the implementor. That is why, many works have focused on finding alternative solutions.

Shamir [26] first suggested a non-trivial countermeasure that computes exponentiations with some redundancy. The principle is to perform the two CRT exponentiations with extended moduli $p \cdot t$ and $q \cdot t$ where t is a small (random) integer. One can then efficiently check the consistency of the computation modulo t. This method has been extended and improved in many subsequent works [30, 1, 5, 10, 28]. In the present paper, we focus on a different approach in which the redundancy is not included in the modular operations but at the exponentiation level. Namely, we focus on *self-secure exponentiations* that provide a direct way to check the consistency of the computation.

3.2 Self-Secure Exponentiation Algorithms

The first exponentiation algorithm with built-in security against fault analysis was proposed by Giraud in [14] and is based on the Montgomery ladder [20]. It uses the fact that this exponentiation algorithm works with a pair of registers (R_0, R_1) storing values of the form $(m^\alpha, m^{\alpha+1})$. At the end of the exponentiation loop, the registers contain the pair of values (m^{d-1}, m^d), which enables to verify the computation consistency by checking whether $R_0 \cdot m$ well equals R_1. If a fault is injected during the computation, the coherence between R_0 and R_1 is lost and the fault is detected by the final check.

Another self-secure exponentiation algorithm was proposed by Boscher *et al.* [8] which is based on the right-to-left *square-and-multiply-always* algorithm (originally devoted to thwart *simple power analysis* [11]). Their algorithm works as Algorithm 1 but it involves an additional register A_0 initialized to 1, and when $d_i = 0$ the multiplication $A_0 \leftarrow A_0 \cdot M$ is processed. It is observed in [8] that the triplet (A_0, A_1, M) stores $(m^{2^\ell - d - 1}, m^d, m^{2^\ell})$ at the end of the exponentiation. The computation consistency can be verified by checking whether $A_0 \cdot A_1 \cdot m = M$. In case of fault injection, the relation between the three register values is broken and the fault is detected by the final check. This approach was then generalized by Baek in [2] to the use of the right-to-left window square-and-multiply-always algorithm. It works as Algorithm 2 with a register A_0 so that the multiplication $A_{u_i} \leftarrow A_{u_i} \cdot M$ is also performed whenever $u_i = 0$. At the end of the exponentiation, computing

$$R \leftarrow \prod_{b=1}^{2^w-1} A_u^u \quad \text{and} \quad L \leftarrow \prod_{b=0}^{2^w-1} A_u^{2^w-1-u} \ ,$$

one then gets a triplet (R, L, M) storing $(m^d, m^{2^\ell - d - 1}, m^{2^\ell})$ as in the binary case. If the computation was performed correctly then the equation $R \cdot L \cdot m =$

M must hold. The obtained self-secure exponentiation achieves better timing performances than the previous ones: it roughly involves $(1+\frac{1}{w})\ell$ multiplications on average against 2ℓ multiplications for the Giraud and Bosher *et al.* schemes. In [17] Joye and Karroumi further improved this approach with a variant for the aggregation and consistency check achieving a better time-memory trade-off than the original Baek proposal.

Remark 1. The self-secure exponentiations presented above have a drawback: they do not provide detection of errors induced in the exponent. For instance, if the exponent is corrupted before or during the Montgomery ladder (*e.g.* by flipping the current bit in any iteration), it shall output a pair $(m^{d'-1}, m^{d'})$ where d' denotes the corrupted exponent value. One hence clearly see that such a fault is not detected by the consistency check since we still have $m^{d'-1} \cdot m = m^{d'}$. The same applies for the Bosher *et al.* scheme and its variants. The final triplet in presence of a corrupted exponent equals $(m^{d'}, m^{2^\ell - d' - 1}, m^{2^\ell})$ and the fault is undetected by the consistency check. In his paper, Giraud suggests to include integrity checks for the exponent and the loop counter in every iteration of the exponentiation loop. In their paper, Bosher *et al.* suggest to recompute the read exponent on the fly in order to check that it well matches the correct exponent value at the end of the exponentiation (see also [16]). Although such methods may circumvent the problem in theory, their practical implementation is not straightforward and it might leave some unexpected flaws.

3.3 Securing Exponentiation with Double Addition Chains

In [22], Rivain introduced another principle for self-secure exponentiation. It consists in performing a *double exponentiation* to compute m^d and $m^{\varphi(N)-d}$ at the same time and then checking the following consistency relation:

$$m^d \cdot m^{\varphi(N)-d} \equiv 1 \pmod N .$$

If there is no fault injected during the computation, then the above equation well holds. Otherwise, if the computation is corrupted, it doesn't hold with high probability (see analysis in [22]).

In order to get an efficient self-secure exponentiation from the above principle, one must then find a way to raise m to both powers d and $\varphi(N)-d$ with the least multiplications possible. In other words, one must find a short addition chain containing both exponents. For such a purpose, Rivain introduces a heuristic to compute a *double addition chain* for any pair of integers (a, b). To construct such a chain, he defines a sequence $\{(\alpha_i, \beta_i)\}_i$ starting from the pair $(\alpha_0, \beta_0) = (a, b)$ down to the pair $(\alpha_n, \beta_n) = (0, 1)$ for some $n \in \mathbb{N}$, such that the inverse sequence is an addition chain for (a, b). Formally, he defines

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i/2) & \text{if } \alpha_i \le \beta_i/2 \text{ and } \beta_i \text{ is even} \\ (\alpha_i, (\beta_i - 1)/2) & \text{if } \alpha_i \le \beta_i/2 \text{ and } \beta_i \text{ is odd} \\ (\beta_i - \alpha_i, \alpha_i) & \text{if } \alpha_i > \beta_i/2. \end{cases}$$

Without loss of generality, b is assumed to be greater than a and the above sequence keep $\alpha_i \leq \beta_i$ as invariant. Moreover it is shown in [22] that there exists $n \in \mathbb{N}$ such that $(\alpha_n, \beta_n) = (0, 1)$. Then by defining

$$\tau_j = \begin{cases} 0 & \text{if } \beta_i \geq 2\alpha_i, \\ 1 & \text{if } \beta_i < 2\alpha_i, \end{cases} \quad \text{and} \quad \nu_j = \begin{cases} \beta_i \pmod 2 & \text{if } \tau_j = 0, \\ \perp & \text{if } \tau_j = 1, \end{cases}$$

where $j = n - i$, the inverse sequence $(a_j, b_j) = (\alpha_i, \beta_i)$ can be computed by initializing (a_0, b_0) to $(0, 1)$ and iterating

$$(a_{j+1}, b_{j+1}) = \begin{cases} (a_j, 2b_j) & \text{if } \tau_{j+1} = 0 \text{ and } \nu_{j+1} = 0 \\ (a_j, 2b_j + 1) & \text{if } \tau_{j+1} = 0 \text{ and } \nu_{j+1} = 1 \\ (b_j, a_j + b_j) & \text{if } \tau_{j+1} = 1 \end{cases} \tag{1}$$

to finally get $(a_n, b_n) = (a, b)$. The sequence (τ_j, ν_j) is hence a sound encoding to process the above double addition chain. The method of [22] consists in computing the (α_i, β_i) sequence in order to derive and store the (τ_j, ν_j) sequence. The double exponentiation $m \mapsto (m^a, m^b)$ is then efficiently computed by evaluating the sequence (m^{a_j}, m^{b_j}) with respect to (1). According to [22], the resulting encoding has a bit-length lower than 2.2ℓ with overwhelming probability and the underlying exponentiation involves 1.65ℓ multiplications on average, where $\ell = \log_2 b$.

4 New Heuristics for Double Addition Chains

As shown above, the problem of finding a double addition chain for a pair (a, b) can be thought as the problem of finding a way to go from the pair (a, b) to the pair $(0, 1)$ using only (and the least possible) subtractions, decrementations and divisions by two. From this starting point, Rivain's method works with two intermediate variables α_i and β_i according to the following principle. If α_i is close to β_i then $\beta_i - \alpha_i$ is small so a *subtraction step* $\beta_{i+1} = \beta_i - \alpha_i$ should be used. Otherwise, if α_i is significantly lower than β_i, then $\beta_i - \alpha_i$ is not significantly lower than β_i, so such a subtraction step should be avoided. One should rather lower β_i so that it get closer to α_i by using a *binary step* $\beta_{i+1} = \beta_i/2$ or $\beta_{i+1} = (\beta_i - 1)/2$ depending on the parity of β_i. In this section, we show that this principle can be improved in several ways.

4.1 First Improvements

Our first improvement starts from the observation that when β_i is odd and lies in $[2\alpha_i; 3\alpha_i]$, it is more efficient to perform a subtraction step $\beta_{i+1} = \beta_i - \alpha_i$ and get $\beta_{i+1} \leq \frac{2}{3}\beta_i$ at the cost of one subtraction (inducing one multiplication at the exponentiation level), rather than performing a binary step $\beta_{i+1} = (\beta_i - 1)/2$ and get $\beta_{i+1} \leq \frac{1}{2}\beta_i$ at the cost of one decrementation and one division by two (inducing two multiplications at the exponentiation level).

Our second improvement focuses on the situation where $\beta_i \geq 2^k \alpha_i$ for some $k \geq 2$. In such a situation, the original method applies k binary steps involving k divisions by two and $H(r)$ decrementations where $r = \beta_i \pmod{2^k}$ and H is the Hamming weight function. We observe that if we have $\alpha_i \pmod{2^k} = \beta_i \pmod{2^k} \neq 0$, then it is more efficient to perform a subtraction step $\beta_{i+1} = \beta_i - \alpha_i$ so that β_{i+1} is a multiple of 2^k. Doing so, the k next steps are divisions by 2 and β_i is lowered by a factor 2^k in $k+1$ operations instead of $k + H(r)$.

From these observations, we suggest to modify the above method by defining the (α_i, β_i) sequence such that $(\alpha_0, \beta_0) = (a, b)$ and for every $i \geq 0$:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\beta_i - \alpha_i, \alpha_i) & \text{if } \beta_i < 2\alpha_i \\ (\alpha_i, \beta_i - \alpha_i) & \text{if } (\beta_i \in [2\alpha_i; 3\alpha_i] \text{ and } \beta_i \text{ is odd}) \text{ or } (\beta_i > 2\alpha_i \\ & \ \text{and } \exists\ k \text{ s.t. } \alpha_i \ (\mathrm{mod} 2^k) = \beta_i \ (\mathrm{mod} 2^k) \neq 0) \\ (\alpha_i, (\beta_i - \gamma)/2) & \text{otherwise,} \end{cases}$$

where $\gamma = \beta_i \ (\mathrm{mod} 2)$. Our simulations revealed that the double exponentiation obtained from our heuristic involves an average of 1.55ℓ multiplications, which represents a gain of 7% compared to the original method.

Example 1. We illustrate the effectiveness of the above variant for the pair $(7, 35)$. For this pair, the original method gives:

$$(0,1) \xrightarrow{+} (1,1) \xrightarrow{\times 2\ +1} (1,3) \xrightarrow{\times 2\ +1} (1,7) \xrightarrow{+} (7,8) \xrightarrow{\times 2\ +1} (7,17) \xrightarrow{\times 2\ +1} (7,35)$$

which requires 10 multiplications at the exponentiation level. Using our improvement, we obtain the chain:

$$(0,1) \xrightarrow{\times 2\ +1} (0,3) \xrightarrow{\times 2\ +1} (0,7) \xrightarrow{+} (7,7) \xrightarrow{\times 2} (7,14) \xrightarrow{\times 2} (7,28) \xrightarrow{+} (7,35)$$

which only needs 8 multiplications at the exponentiation level.

Encoding. We have to slightly modify the (τ_j, ν_j) encoding defined in [22] in order to include the proposed improvements. In the original proposal recalled in Section 3.3, a step $(\alpha_{i+1}, \beta_{i+1}) = (\beta_i - \alpha_i, \alpha_i)$ is encoded by a bit $\tau_j = 1$. On the other hand, a step $(\alpha_{i+1}, \beta_{i+1}) = (\alpha_i, (\beta_i - \gamma)/2)$, where $\gamma = \beta_i \pmod 2$, is encoded by a bit $\tau_j = 0$ followed by a bit $\nu_j = \gamma$. In order to include our improvements, we must define an encoding for a step $(\alpha_{i+1}, \beta_{i+1}) = (\alpha_i, \beta_i - \alpha_i)$, namely a subtraction step without swapping of the elements. We suggest to encode every subtraction step by a bit $\tau_j = 1$ followed by a bit ν_j that equals 1 if there is no swap (*i.e.* $\beta_i \geq 2\alpha_i$) and that equals 0 if there is a swap (*i.e.* $\beta_i < 2\alpha_i$). Specifically, we define:

$$\tau_j = \begin{cases} 0 & \text{if } (\alpha_{i+1}, \beta_{i+1}) = (\alpha_i, (\beta_i - \gamma/2)), \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\nu_j = \begin{cases} \beta_i \pmod 2 & \text{if } \tau_j = 0, \\ 0 & \text{if } \tau_j = 1 \text{ and } \beta_i < 2\alpha_i, \\ 1 & \text{if } \tau_j = 1 \text{ and } \beta_i \geq 2\alpha_i, \end{cases}$$

where $j = n - i$. The addition chain $(a_j, b_j) = (\alpha_i, \beta_i)$ can be computed by initializing (a_0, b_0) to $(0, 1)$ and iterating

$$(a_{j+1}, b_{j+1}) = \begin{cases} (a_j, 2b_j + \nu_{j+1}) & \text{if } \tau_{j+1} = 0, \\ (b_j, a_j + b_j) & \text{if } \tau_{j+1} = 1 \text{ and } \nu_{j+1} = 0, \\ (a_j + b_j, b_j) & \text{if } \tau_{j+1} = 1 \text{ and } \nu_{j+1} = 1. \end{cases}$$

Example 2. We construct the encoding $\Gamma(7, 35)$ for the double addition chain given in Example 1. First, we obtain

$$(\tau_0, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5) = (0, 0, 1, 0, 0, 1) \text{ and } (\nu_0, \nu_1, \nu_2, \nu_3, \nu_4, \nu_5) = (1, 1, 0, 0, 0, 1)$$

giving the following encoding:

$$\Gamma(7, 35) = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1$$

Each pattern of two bits correspond to a single step: 00 indicates a $(\times 2)$-step, 01 indicates a $(\times 2 + 1)$-step, 10 and 11 indicate a $(+)$-step (with and without swapping respectively).

4.2 Improved Method based on Sliding Window

The original and improved methods presented above require only 3 registers to compute a double exponentiation $m \mapsto (m^a, m^b)$ (one for m, one for m^{a_i} and one for m^{b_j}). In this section, we look at the context where more memory is available. For a single exponentiation, window-based methods are natural extensions of the binary method for reducing the number of multiplications. We show hereafter how to improve the performances of double addition chains by using a *sliding window* (see *e.g.* [18]).

In the original method, if we have $\beta_i \geq 2^k \alpha_i$, then the heuristic performs k binary steps to lower β_i by a factor 2^k. At the exponentiation level, this translates by a binary exponentiation involving k squarings and an average of $\frac{k}{2}$ multiplications. It is then natural to replace such binary exponentiation by a more efficient sliding-window exponentiation. The principle is to precompute and store odd values $3, 5, \ldots, 2^w - 1$, for some widow parameter w, so that when $\beta_i \geq 2^k \alpha_i$ for some $k \leq w$, we first subtract $r_i = \beta_i \pmod{2^k} \in \{1, 3, \ldots, 2^w - 1\}$ to β_i and then we perform k successive divisions by two (note that we assume β_i to be odd, otherwise it is simply divided by two). This translates by a multiplication and k squarings at the exponentiation level.

We then modify the original method by defining the sequence (α_i, β_i) such that $(\alpha_0, \beta_0) = (a, b)$, and for every $i \geq 0$:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i/2) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \text{ is even}, \\ (\alpha_i, (\beta_i - r_i)/2^{k_i}) & \text{if } \alpha_i \leq \beta_i/2,\ \beta_i \text{ is odd}, \\ (\beta_i - \alpha_i, \alpha_i) & \text{if } \alpha_i > \beta_i/2, \end{cases}$$

where $r_i = \beta_i \pmod{2^{k_i}}$ and k_i is the greatest integer in $\{1, 2, \ldots, w\}$ such that $\beta_i \geq 2^{k_i} \alpha_i$ and $2^{k_i - 1} \leq r_i < 2^{k_i}$. The latter condition means that the most significant bit of r_i (viewed as a k_i-bit string) is at 1. It is equivalent to the equality $k_i = \lfloor \log_2(r_i) \rfloor + 1$, which is required for our encoding (see below).

Note that the double addition chain is not strictly the inverse of the above sequence. Since it must start with a sequence:

$$1 \xrightarrow{\times 2} 2 \xrightarrow{+1} 3 \xrightarrow{+2} 5 \xrightarrow{+2} 7 \xrightarrow{+2} \cdots \xrightarrow{+2} 2^w - 1$$

in order to precompute the odd values 3, 5, ..., $2^w - 1$. This chain translates to one square and $2^{w-1} - 1$ multiplications to compute the powers m^3, m^5, ..., m^{2^w-1} at the exponentiation level. Moreover the resulting implementation has a greater memory consumption since these precomputed powers must be stored during the exponentiation.

Our simulations revealed that the double exponentiation obtained from our sliding-window-based heuristic involves an average of multiplications ranging from 1.59ℓ to 1.53ℓ depending on the window size. This represents a gain between 4% and 8% compared to the original method.

Example 3. We illustrate the effectiveness of our sliding-window-based method for the pair $(6, 27)$. For this pair, the original method gives:

$$(0,1) \xrightarrow{\times 2\ +1} (0,3) \xrightarrow{\times 2} (0,6) \xrightarrow{+} (6,6) \xrightarrow{\times 2\ +1} (6,13) \xrightarrow{\times 2\ +1} (6,27)$$

Using our sliding-window-based method, we obtain the chain:

$$(0,1) \xrightarrow{\times 2\ +1} (0,3) \xrightarrow{\times 2} (0,6) \xrightarrow{+} (6,6) \xrightarrow{\times 2\ \times 2\ +3} (6,27)$$

saving one multiplication at the exponentiation level.

Encoding. In order to define a sound encoding for our window-based double addition chains, we define the three following sequences:

$$\tau_j = \begin{cases} 0 & \text{if } \beta_i \geq 2\alpha_i, \\ 1 & \text{if } \beta_i < 2\alpha_i, \end{cases} \qquad \nu_j = \begin{cases} \beta_i \pmod 2 & \text{if } \tau_j = 0, \\ \perp & \text{if } \tau_j = 1, \end{cases}$$

and

$$\gamma_j = \begin{cases} (r_i - 1)/2 & \text{if } \tau_j = 0 \text{ and } \nu_j = 1, \\ \perp & \text{if } \tau_j = 1 \text{ or } \nu_j = 0, \end{cases}$$

where $j = n - i$. Note that when r_i is odd (*i.e.* when $\nu_j = 1$), γ_j is the value obtained by shifting $r_i = \beta_i \pmod{2^{k_i}}$ by one bit to the left, and we have $r_i = 2\gamma_j + 1$.

The double addition chain $(a_j, b_j) = (\alpha_i, \beta_i)$ can then be computed from the $(\tau_j, \nu_j, \gamma_j)$ sequence by initializing (a_0, b_0) to $(0, 1)$ and iterating

$$(a_{j+1}, b_{j+1}) = \begin{cases} (a_j, 2b_j) & \text{if } \tau_{j+1} = 0 \text{ and } \nu_{j+1} = 0 \\ (a_j, 2^{k_i} b_j + r_i) & \text{if } \tau_{j+1} = 0 \text{ and } \nu_{j+1} = 1 \\ (b_j, a_j + b_j) & \text{if } \tau_{j+1} = 1, \end{cases}$$

where $r_i = 2\gamma_j + 1$ and $k_i = \lfloor \log_2(r_i) \rfloor + 1$.

Each step is hence encoded by a bit τ_j, followed by a bit ν_j if and only if $\tau_j = 0$, followed by $w - 1$ bits encoding γ_j if and only if $\tau_j = 0$ and $\nu_j = 1$.

Example 4. We construct the encoding $\Gamma(6, 27)$ for the double addition chain given in Example 3 (with window size $w = 2$). First, we obtain

$$(\tau_0, \tau_1, \tau_2, \tau_3) = (0, 0, 1, 0) \ , \quad (\nu_0, \nu_1, \nu_2, \nu_3) = (1, 0, \perp, 1) \ ,$$

and

$$(\gamma_0, \gamma_1, \gamma_2, \gamma_3) = (0, \perp, \perp, 1)$$

giving the following encoding:

$$\Gamma(6, 27) = 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$

The first three bits 010 indicate a $(\times 2 + 1)$-step. The next two bits 00 indicate a $(\times 2)$-step. The next bit 1 indicates a $(+)$-step. And the final three bits 011 indicate a $(\times 2 \times 2 + 3)$-step.

4.3 Combined Improvements

In the previous section we have introduced two different kinds of improvements to Rivain's heuristic for double addition chains. The purpose of our first improvements is to perform subtraction steps (without swapping) instead of binary steps in some cases where it is more advantageous to do so. The purpose of the sliding-window method is to speed up a succession of several binary steps. These two kinds of improvements are hence fully compatible and we can combine them by defining the (α_i, β_i) sequence as:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\beta_i - \alpha_i, \alpha_i) & \text{if } \beta_i < 2\alpha_i \\ (\alpha_i, \beta_i - \alpha_i) & \text{if } (\beta_i \in [2\alpha_i; 3\alpha_i] \text{ and } \beta_i \text{ is odd) or } (\beta_i > 2\alpha_i \\ & \quad \text{and } \exists\, k \text{ s.t. } \alpha_i \pmod{2^k} = \beta_i \pmod{2^k} \neq 0) \\ (\alpha_i, (\beta_i - r_i)/2^{k_i}) & \text{otherwise,} \end{cases}$$

where $r_i = \beta_i \pmod{2^{k_i}}$ and k_i is the greatest integer in $\{1, 2, \ldots, w\}$ such that $\beta_i \geq 2^{k_i}\alpha_i$ and $2^{k_i - 1} \leq r_i < 2^{k_i}$. The encoding of the obtained double addition chain $(a_j, b_j) = (\alpha_i, \beta_i)$ with $j = n - i$ is easily deduced from the encodings of both previous heuristics. Specifically, it is based on the three following sequences:

$$\tau_j = \begin{cases} 0 & \text{if } \beta_i \geq 2\alpha_i, \\ 1 & \text{if } \beta_i < 2\alpha_i, \end{cases} \qquad \nu_j = \begin{cases} \beta_i \pmod 2 & \text{if } \tau_j = 0, \\ 0 & \text{if } \tau_j = 1 \text{ and } \beta_i < 2\alpha_i, \\ 1 & \text{if } \tau_j = 1 \text{ and } \beta_i \geq 2\alpha_i, \end{cases}$$

and

$$\gamma_j = \begin{cases} (r_i - 1)/2 & \text{if } \tau_j = 0 \text{ and } \nu_j = 1, \\ \perp & \text{if } \tau_j = 1 \text{ or } \nu_j = 0, \end{cases}$$

where $j = n - i$. The double addition chain $(a_j, b_j) = (\alpha_i, \beta_i)$ can finally be computed from the $(\tau_j, \nu_j, \gamma_j)$ sequence by initializing (a_0, b_0) to $(0, 1)$ and iterating

$$(a_{j+1}, b_{j+1}) = \begin{cases} (a_j, 2b_j) & \text{if } (\tau_{j+1}, \nu_{j+1}) = (0, 0), \\ (a_j, 2^{k_i}b_j + r_i) & \text{if } (\tau_{j+1}, \nu_{j+1}) = (0, 1), \\ (b_j, a_j + b_j) & \text{if } (\tau_{j+1}, \nu_{j+1}) = (1, 0), \\ (a_j + b_j, b_j) & \text{if } (\tau_{j+1}, \nu_{j+1}) = (1, 1), \end{cases}$$

where $r_i = 2\gamma_j + 1$ and $k_i = \lfloor \log_2(r_i) \rfloor + 1$.

5 Sliding-Window Double Exponentiation

In the previous section, we have presented several heuristics for double addition chain improving the original method from [22]. These heuristics give rise to efficient double exponentiation algorithms with different time-memory trade-offs (see Section 6 for a detailed comparison). However, these algorithms have a drawback in practice: they require the precomputation of the chain encoding (involving the evaluation of the (α_i, β_i) sequence). Although this precomputation only involves simple operations compared to the modular multiplications used in the exponentiation, it might not be negligible in practice, especially for implementations using a hardware accelerator for modular arithmetic (which is common in smart cards and other embedded systems).

In this section, we propose an alternative by describing an efficient double exponentiation algorithm that does not rely on any form of precomputation. Our proposed algorithm is a generalization of Yao algorithm for the double exponentiation scenario and it is based on a sliding window approach.

Algorithm 3 Sliding-Window Double Exponentiation

```
Input: $m$, $a$, $b$
Output: $(m^a, m^b)$
 1. $M \leftarrow m$
 2. for $d \in \{1, 3, \ldots, 2^w - 1\}$ do
 3.     $A_d \leftarrow 1$ ; $B_d \leftarrow 1$
 4. end for
 5. for $i = 0$ to $\ell - 1$ do
 6.     if $(a_i = 1)$ then
 7.         $d \leftarrow (a_{i+w-1}, \ldots, a_{i+1}, a_i)_2$
 8.         $A_d \leftarrow A_d \cdot M$
 9.         $a \leftarrow a - 2^i d$
10.     endif
11.     if $(b_i = 1)$ then
12.         $d \leftarrow (b_{i+w-1}, \ldots, b_{i+1}, b_i)_2$
13.         $B_d \leftarrow B_d \cdot M$
14.         $b \leftarrow b - 2^i d$
15.     endif
16.     $M \leftarrow M^2$
17. end for
18. $A_1 \leftarrow \prod_d A_d^d$
19. $B_1 \leftarrow \prod_d B_d^d$
20. return $(A_1, B_1)$
```

In a nutshell, the proposed algorithm works as two parallel executions of Yao's algorithm (see Algorithm 2), by using two sets of 2^{w-1} accumulators: A_1, A_3, $\ldots$, A_{2^w-1} for exponent a, and B_1, B_3, $\ldots$, B_{2^w-1} for exponent b. On the other hand, a single register M is used and the squarings involved to derive the successive powers m, m^2, m^4, ..., m^{2^ℓ} are computed only once, which results in a saving of ℓ squarings compared to two independent applications of Algorithm 2. Moreover, for the sake efficiency, we use a sliding window rather than a fixed window. Namely, instead of cutting the exponent in n fixed windows of w bits, each bit is treated from the less significant one to the most significant one. If the current bit a_i equals 0, the algorithm just squares M and continue with the next bit. Otherwise if a_i equals 1, then the algorithm processes the current w-bit digit $d = \sum_{j=0}^{j=w-1} a_{i+j} 2^j$ by multiplying M to the corresponding accumulator A_d, and by setting the next w bits of a to 0 with $a \leftarrow a - 2^i d$. The same process is performed simultaneously for the exponent b and corresponding accumulators.

The explicit description of the obtained double exponentiation is given in Algorithm 3. This algorithm involves ℓ squarings as a regular sliding window exponentiation. On the other hand, it involves twice more multiplications, that is $2 \times \frac{\ell}{w+1}$ multiplications on average for the exponentiation loop. For the aggregation, we adapt the method proposed in [15] for fixed window algorithms to the case of sliding-window algorithms. Specifically, the aggregation is computed as follows:

```
1. $M \leftarrow A_{2^w-1}$
2. for $d = 2^w - 3$ to 3 step $i \leftarrow i - 2$ do
3.     $A_d \leftarrow A_d \cdot A_{d+2}$
4.     $M \leftarrow M \cdot A_d$
5. end for
6. $A_1 \leftarrow M^2 \cdot A_3 \cdot A_1$
```

The above process takes $2^w - 1$ multiplications, and it must be performed twice (in steps 18 and 19 of Algorithm 3). This makes a total of $\big(1 + \frac{2}{w+1}\big)\ell + 2^{w+1} - 2$ multiplications on average.

Example 5. We illustrate hereafter the processing of Algorithm 3 by detailing the successive values of the pair (a, b) and the different registers between each loop iteration for the input pair of exponents $(14, 25)$:

$$\begin{pmatrix} (a,b) \\ M \\ (A_1, A_3) \\ (B_1, B_3) \end{pmatrix} : \begin{pmatrix} (29,50) \\ m \\ (1,1) \\ (1,1) \end{pmatrix} \rightarrow \begin{pmatrix} (28,50) \\ m^2 \\ (m,1) \\ (1,1) \end{pmatrix} \rightarrow \begin{pmatrix} (28,48) \\ m^4 \\ (m,1) \\ (m^2,1) \end{pmatrix} \rightarrow \begin{pmatrix} (16,48) \\ m^8 \\ (m,m^4) \\ (m^2,1) \end{pmatrix} \rightarrow \begin{pmatrix} (16,48) \\ m^{16} \\ (m,m^4) \\ (m^2,1) \end{pmatrix} \rightarrow \begin{pmatrix} (0,0) \\ m^{32} \\ (m^{17},m^4) \\ (m^2,m^{16}) \end{pmatrix}$$

At the end of the exponentiation loop we well have $m^{17} \cdot \big(m^4\big)^3 = m^{29}$ on the one hand and $m^2 \cdot \big(m^{16}\big)^3 = m^{50}$ on the other hand.

6 Performances and Comparison

In this section, we provide performance figures for our proposals and we compare them with other self-secure exponentiations in the literature. Basically, we consider:

– the binary self-secure exponentiations by Giraud [14] (Montgomery ladder) and by Bosher *et al.* [8] (square and multiply always),
– the w-ary square-and-multiply-always (w-ary SMA) method by Baek [2], with the Joye-Karroumi improvement [17],
– the double addition chain method by Rivain [22],
– the improved heuristics for double addition chains described in Section 4 of this paper,
– the sliding-window double exponentiation described in Section 5 of this paper.

The performances of the different methods are summarized in Table 1, for exponent bit-length $\ell \in \{512, 1024, 2048\}$, and for various window sizes. Specifically, we give the average number of multiplications per bit of the exponent as well as the number of ℓ-bit memory registers require by each self-secure exponentiation. We also give the memory overhead required to store the exponent, the chain encoding (for Rivain's method and our improvements) or the pair of exponents (for the double exponentiation described in Section 5). This overhead is given in number of required ℓ-bit registers.

Table 1. Performances of various self-secure exponentiations.

	Window size	Multiplications/bit $\ell = 512$	$\ell = 1024$	$\ell = 2048$	Reg.	Memory overhead
Binary exp. [14, 8]	-	2	2	2	3	1
w-ary SMA [2, 17]	$w = 2$	1.52	1.51	1.50	5	1
	$w = 3$	1.37	1.35	1.34	9	1
	$w = 4$	**1.32**	1.26	1.27	17	1
	$w = 5$	1.34	**1.28**	**1.23**	33	1
	$w = 6$	1.43	1.29	1.23	65	1
Double addition chain [22]	-	1.66	1.66	1.66	3	2.2
First improvements (§4.1)	-	1.55	1.55	1.55	3	3.19
Sliding-window improvement (§4.2)	$w = 2$	1.59	1.59	1.59	4	1.74
	$w = 3$	1.56	1.56	1.56	6	1.89
	$w = 4$	**1.54**	**1.54**	1.54	10	2.09
	$w = 5$	1.54	1.54	**1.53**	18	2.31
	$w = 6$	1.55	1.54	1.53	34	2.55
Combined improvements (§4.3)	$w = 2$	1.55	1.55	1.55	4	2.57
	$w = 3$	1.54	1.54	1.54	6	2.56
	$w = 4$	**1.53**	1.53	1.53	10	2.66
	$w = 5$	1.53	**1.52**	**1.52**	18	2.78
	$w = 6$	1.54	1.53	1.52	34	2.92
Sliding-window double exponentiation (§5)	$w = 2$	1.68	1.67	1.67	5	2
	$w = 3$	1.53	1.51	1.51	9	2
	$w = 4$	**1.46**	1.43	1.42	17	2
	$w = 5$	1.46	**1.39**	1.36	33	2
	$w = 6$	1.53	1.40	**1.35**	65	2

We see that our techniques provide significant improvements of the original heuristic for double addition chain proposed in [22]. The first improved heuristic (Section 4.1) is roughly 7% faster than the original method, whereas the window-based method is 4% to 8% faster depending on the window size. When combined, the two kind of improvements provide a performance gain between 7% and 9%. On the other hand the double exponentiation algorithm described in Section 5 achieves a speed up factor up to 19% depending on the available memory and the exponent length. In comparison, Baek w-ary self-secure exponentiation is roughly 10% faster for a similar memory consumption. However our algorithms (as Rivain's initial proposal) have the advantage of inherently protecting the implementation against corruption of the exponent whereas all other proposals require additional countermeasures for this purpose (see Remark 1).

7 Conclusion

In this paper we have revisited double exponentiation algorithms for fault-analysis resistant RSA. We have introduced new variants of Rivain's heuristic for double addition chains that achieve speed up factors up to 9%. We have also presented a generalization of Yao's right-to-left exponentiation to efficiently perform a double exponentiation. This algorithm achieves a performance gain up to 19% compared to the original double addition chain exponentiation, while requiring no precomputation. These improvements are of interest as self-secure exponentiations based on double exponentiation are currently the only ones that protect the exponent from fault attacks (whereas other self-secure exponentiations need additional countermeasures to this aim).

Interesting open issues include the design of more efficient double exponentiation algorithms (either based on addition chains or not), as well as the investigation of alternative approaches for designing self-secure exponentiation algorithms.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In Kaliski Jr., B., Koç, Ç., Paar, C., eds.: Cryptographic Hardware and Embedded Systems – CHES 2002. Volume 2523 of Lecture Notes in Computer Science. (2002) 260–275
2. Baek, Y.J.: Regular 2^w-ary right-to-left exponentiation algorithm with very efficient dpa and fa countermeasures. International Journal of Information Security **9**(5) (2010) 363–370
3. Bao, F., Deng, R., Han, Y., Jeng, A., Narasimhalu, A.D., Ngair, T.H.: Breaking Public Key Cryptosystems an Tamper Resistance Devices in the Presence of Transient Fault. In: 5th Security Protocols Workshop. Volume 1361 of Lecture Notes in Computer Science. (1997) 115–124
4. Berzati, A., Canovas, C., Goubin, L.: Perturbating RSA Public Keys: An Improved Attack. In Oswald, E., Rohatgi, P., eds.: CHES. Volume 5154 of Lecture Notes in Computer Science., Springer (2008) 380–395

5. Blömer, J., Otto, M., Seifert, J.P.: A New RSA-CRT Algorithm Secure against Bellcore Attacks. In Jajodia, S., Atluri, V., Jaeger, T., eds.: ACM Conference on Computer and Communications Security – CCS'03, ACM Press (2003) 311–320
6. Boneh, D., DeMillo, R., Lipton, R.: On the Importance of Checking Cryptographic Protocols for Faults. In Fumy, W., ed.: Advances in Cryptology, International Conference on the Theory and Application of Cryptographic Techniques – EUROCRYPT '97. Volume 1233 of Lecture Notes in Computer Science., Springer (1997) 37–51
7. Boreale, M.: Attacking Right-to-Left Modular Exponentiation with Timely Random Faults. In Breveglieri, L., Koren, I., Naccache, D., Seifert, J.P., eds.: Fault Diagnosis and Tolerance in Cryptography – FDTC 2006. Volume 4236 of Lecture Notes in Computer Science. (2006) 24–35
8. Boscher, A., Naciri, R., Prouff, E.: CRT RSA Algorithm Protected against Fault Attacks. In Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.J., eds.: Information Security Theory and Practices – WISTP 2007. Volume 4462 of Lecture Notes in Computer Science. (2007) 229–243
9. Brier, E., Chevallier-Mames, B., Ciet, M., Clavier, C.: Why one should also secure rsa public key elements. In Goubin, L., Matsui, M., eds.: Cryptographic Hardware and Embedded Systems – CHES 2006. Volume 4249 of Lecture Notes in Computer Science. (2006) 324–338
10. Ciet, M., Joye, M.: Practical Fault Countermeasures for Chinese Remaindering Based RSA. In Breveglieri, L., Koren, I., eds.: Workshop on Fault Diagnosis and Tolerance in Cryptography – FDTC'05. (2005) 124–132
11. Coron, J.S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Koç, Ç., Paar, C., eds.: Cryptographic Hardware and Embedded Systems – CHES '99. Volume 1717 of Lecture Notes in Computer Science. (1999) 292–302
12. Downey, P., Leong, B., Sethi, R.: Computing Sequences with Addition Chains. SIAM Journal on Computing **10**(3) (1981) 638–646
13. Garner, H.L.: The residue number system. Electronic Computers, IRE Transactions on (2) (1959) 140–147
14. Giraud, C.: An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. IEEE Transactions on Computers **55**(9) (September 2006) 1116–1120
15. Joye, M.: Highly Regular m-Ary Powering Ladders. In Jr., M.J.J., Rijmen, V., Safavi-Naini, R., eds.: Selected Areas in Cryptography, 16th Annual International Workshop – SAC 2009. Volume 5867 of Lecture Notes in Computer Science., Springer (2009) 350–363
16. Joye, M.: A Method for Preventing "Skipping" Attacks. In: 2012 IEEE Symposium on Security and Privacy Workshops, IEEE Computer Society (2012) 12–15
17. Joye, M., Karroumi, M.: Memory-Efficient Fault Countermeasures. In Prouff, E., ed.: Smart Card Research and Advanced Applications, 10th IFIP WG 8.8/11.2 International Conference – CARDIS 2011. Volume 7079 of Lecture Notes in Computer Science., Springer (2011) 84–101
18. Koc, C.K.: Analysis of Sliding Window Techniques for Exponentiation. Computers and Mathematics with Applications **30** (1995) 17–24
19. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. 1st edn. CRC Press, Inc. (1996)
20. Montgomery, P.L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Mathematics of Computation **48**(177) (1987) 243–264

21. Quisquater, J.J., Couvreur, C.: Fast decipherment algorithm for rsa public-key cryptosystem. Electronics Letters **18**(21) (1982) 905–907
22. Rivain, M.: Securing RSA against Fault Analysis by Double Addition Chain Exponentiation. In Fischlin, M., ed.: Topics in Cryptology, The Cryptographers' Track at the RSA Conference 2009 - CT-RSA 2009. Volume 5473 of Lecture Notes in Computer Science., Springer (2009) 459–480
23. Rivest, R., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM **21**(2) (1978) 120–126
24. Schmidt, J., Herbst, C.: A Practical Fault Attack on Square and Multiply. In Breveglieri, L., Gueron, S., Koren, I., Naccache, D., Seifert, J.P., eds.: Fault Diagnosis and Tolerance in Cryptography – FDTC 2008. (2008) 53–58
25. Seifert, J.P.: On authenticated computing and rsa-based authentication. In Atluri, V., Meadows, C., Juels, A., eds.: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005, ACM (2005) 122–127
26. Shamir, A.: Improved Method and Apparatus for Protecting Public Key Schemes from Timing and Fault Attacks. Patent WO9852319 (November 1998) Also presented to EUROCRYPT'97 rump session.
27. Sun Microsystems: Application Programming Interface – Java Card^{TM} Plateform, Version 2.2.2 (March 2006) http://java.sun.com/products/javacard/specs.html.
28. Vigilant, D.: RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In Oswald, E., Rohatgi, P., eds.: CHES. Volume 5154 of Lecture Notes in Computer Science., Springer (2008) 130–145
29. Yao, A.C.C.: On the evaluation of powers. SIAM Journal on Computing **5**(1) (1976) 100–103
30. Yen, S.M., Joye, M.: Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. IEEE Transactions on Computers **49**(9) (2000) 967–970