Securing RSA against Fault Analysis by Double Addition Chain Exponentiation[⋆]

Matthieu Rivain

Oberthur Technologies & University of Luxembourg
m.rivain@oberthur.com

Abstract. Fault Analysis is a powerful cryptanalytic technique that enables to break cryptographic implementations embedded in portable devices more efficiently than any other technique. For an RSA implemented with the Chinese Remainder Theorem method, one faulty execution suffices to factorize the public modulus and fully recover the private key. It is therefore mandatory to protect embedded implementations of RSA against fault analysis.
This paper provides a new countermeasure against fault analysis for exponentiation and RSA. It consists in a *self-secure* exponentiation algorithm, namely an exponentiation algorithm that provides a direct way to check the result coherence. An RSA implemented with our solution hence avoids the use of an extended modulus (which slows down the computation) as in several other countermeasures. Moreover, our exponentiation algorithm involves 1.65 multiplications per bit of the exponent which is significantly less than the 2 required by other self-secure exponentiations.

1 Introduction

The *physical cryptanalysis* gathers different cryptanalytic techniques taking advantage of the physical properties of cryptographic implementations. Among these, one mainly identifies *side channel analysis* [27, 26] that physically observes cryptographic computations and *fault analysis* [8, 6] that physically disturbs them. The latter consists in exploiting the faulty outputs resulting from erroneous computations in order to retrieve information on the secret key. Fault analysis has been introduced first against RSA and other public key schemes [8] and then against DES [6]. Several works followed that improved fault analysis and generalized it to other algorithms.

A straightforward way to protect any algorithm against fault analysis is by performing twice the computation and by checking that the same result is obtained. In case of inconsistency, an error message is returned thus preventing the exposure of the faulty result. A variant consists in verifying an encryption by a decryption (or *vice versa*). These countermeasures are suitable for fast algorithms such as block ciphers, but when a public key cryptosystem such as RSA must be implemented, a doubling of the execution time becomes prohibitive.

⋆ Updated version of the work published in the proceedings of CT-RSA 2009.

That is why, securing RSA against fault analysis constitutes a challenging issue of embedded cryptography. Several methods have been proposed so far but the number of secure and practical solutions is still quite restricted.

In this paper, we provide a new countermeasure against fault analysis for exponentiation and RSA that constitutes an efficient alternative to the existing solutions. First we introduce preliminaries about RSA, fault analysis and the existing countermeasures (Sect. 2). Then we describe our self-secure exponentiation algorithm (Sect. 3) and the resulting secure RSA-CRT algorithm (Sect. 4). Afterward we analyze the security of our solution (Sect. 5) and we address its resistance *vs* side channel analysis (Sect. 6). Finally, we give an analysis of the time and memory complexities of our solution and we compare them to previous solutions in the literature (Sect. 7).

2 RSA and Fault Analysis

2.1 The RSA Cryptosystem

RSA is nowadays the most widely used public key cryptosystem [33]. An RSA public key is composed of a public modulus N which is the product of two large secret primes p and q and of a public exponent e which is co-prime with the Euler's totient of N namely $\varphi(N) = (p-1) \cdot (q-1)$. The corresponding RSA private key is composed of the public modulus N and the secret exponent d that is defined as the inverse of e modulo $\varphi(N)$.

An RSA signature (or deciphering) s of a message $m < N$ is obtained by computing: $s = m^d \bmod N$. The signature verification (or message ciphering) is the inverse operation that can be performed publicly since, according to Euler's Theorem, we have: $m = s^e \bmod N$.

For efficient implementation of RSA, one makes often use of the Chinese Remainder Theorem (CRT). This theorem implies that $m^d \bmod N$ can be computed from $m^d \bmod p$ and $m^d \bmod q$. The RSA-CRT hence consists in performing the two following exponentiations: $s_p = m^{d_p} \bmod p$ and $s_q = m^{d_q} \bmod q$, where $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$. By Fermat's little Theorem, we have $s_p = m^d \bmod p$ and $s_q = m^d \bmod q$. Therefore, once s_p and s_q have been computed, s can be recovered from s_p and s_q by applying a so-called *recombination step*: $s = \mathsf{CRT}(s_p, s_q)$. Two methods exist for CRT recombination: the one from Gauss and the one from Garner. The less memory consuming is the Garner's recombination that is defined as $\mathsf{CRT}(s_p, s_q) = s_q + q \cdot \big(i_q \cdot (s_p - s_q) \bmod p\big)$, where $i_q = q^{-1} \bmod p$. The whole RSA-CRT is around 4 times faster than the straightforward RSA which makes its use very common, especially in the context of low resource devices were computation time is often critical.

2.2 Fault Analysis Against RSA

The most powerful fault attack against RSA is known as the *Bellcore attack* [8] that targets a CRT implementation. It consists in corrupting one of the two

CRT exponentiations, say the one modulo p. The RSA computation thus results in a faulty signature $\tilde{s}$ that is correct modulo q (*i.e.* $\tilde{s} \equiv s \bmod q$) and corrupted modulo p (*i.e.* $\tilde{s} \not\equiv s \bmod p$). This implies that the difference $\tilde{s} - s$ is a multiple of q but is not a multiple of p, and hence we have $\gcd(\tilde{s} - s, N) = q$. Therefore, a pair signature/faulty signature provides a way to factorize N and consequently to fully break RSA. Actually, a pair message/faulty signature suffices to mount the attack since we have $\gcd(\tilde{s}^e - m, N) = q$ [22]. This way, RSA is broken with a single faulty computation.

RSA implemented in straightforward mode (*i.e.* without CRT) is also vulnerable to fault analysis. Several attacks have been published that assume either a faulty exponent [3], a faulty modulus [5, 12, 35] or a faulty intermediate power [8, 9, 34]. These attacks require several faulty signatures to fully recover the key but still constitute practical threats.

Another kind of fault attacks known as *safe-error attacks* can be distinguished from the ones addressed above. Depending on the algorithm, a fault injection may have no effect for some secret key values and may cause a corruption for others. In that case, simply observing wether the computation was corrupted or not reveals information on the secret key. Such attacks are especially threatening since they bypass classical fault analysis countermeasures that return an error in case of fault detection. Among these attacks, two categories can be distinguished: the *C-safe-error attacks* [41] that target dummy operations and the *M-safe-error attacks* that target registers allocations [40, 24]. Our countermeasure provides an error detection mechanism and does not aim to thwart safe-error attacks. However, as discussed in Sect. 5.2, these last can be simply prevented.

Securing RSA against Fault Analysis. A simple way to protect RSA against fault analysis is by verifying the signature s before returning it, namely by performing the following check: $m \stackrel{?}{=} s^e \bmod N$. This method offers a perfect security against differential fault analysis since a faulty signature is systematically detected. This countermeasure is efficient as long as e is small, but in the opposite case, it implies to perform two exponentiations which doubles the time complexity of RSA. This overhead is clearly prohibitive in the context of low resource devices. Moreover, depending on the context, the public exponent e may not be available (*e.g.* the Javacard API for RSA signature [37]). That is why, many works in the last decade have been dedicated to the search of alternative solutions. We review hereafter the main proposals that can be divided into two families: the *extended modulus based countermeasures* and the *self-secure exponentiations.*

2.3 Extended Modulus Based Countermeasures

We present hereafter different countermeasures that all rely on the use of an extended modulus in order to add redundancy in the computation.

Shamir's Trick and Variants. A first solution to protect RSA with CRT has been proposed by Shamir [36]. It consists in performing the two CRT exponentiations with extended moduli $p \cdot t$ and $q \cdot t$ where t is a small integer. Namely, one computes $s_p^* = m^{d \bmod \varphi(p \cdot t)} \bmod p \cdot t$ and $s_q^* = m^{d \bmod \varphi(q \cdot t)} \bmod q \cdot t$. The consistency of the computation is then checked by verifying that $s_p^* \bmod t$ equals $s_q^* \bmod t$. If no error is detected, the algorithm returns $\mathsf{CRT}(s_p^* \bmod p, s_q^* \bmod q)$. Under its simplest form, this countermeasure does not protect the CRT recombination which enables a successful fault attack [2]. Several works have proposed variants of Shamir's countermeasure in order to deal with this issue [2, 7, 14].

Vigilant Scheme. In [38], Vigilant proposed another countermeasure based on a modulus extension. The modulus is multiplied by $t = r^2$ for a small random number r. The message is then formatted as follows: $\hat{m} = \alpha m + \beta \cdot (1+r) \bmod Nt$ where (α, β) is the unique solution in $\{1, \cdots, Nt\}^2$ of the system $\alpha \equiv 1 \bmod N$, $\alpha \equiv 0 \bmod t$, $\beta \equiv 0 \bmod N$ and $\beta \equiv 1 \bmod t$. Then, the exponentiation $s_r = \hat{m}^d \bmod Nt$ is performed. As shown in [38], s_r equals $\alpha m^d + \beta \cdot (1 + dr) \bmod Nt$. Therefore, the signature can be recovered from s_r since it satisfies $s = s_r \bmod N$ and the consistency of the computation can be verified by checking $s_r \equiv 1 + dr \bmod t$. This method can be extended to protect RSA-CRT (see [38] for details).

Security Considerations. The security of an extended modulus based countermeasure is not perfect. For instance, if a faulty message $\widetilde{m}$ satisfies $\widetilde{m} \equiv m \bmod t$ and $\widetilde{m} \not\equiv m \bmod N$, then the exponentiation of this message results in a faulty signature that is not detected. The non-detection probability of an extended modulus based countermeasure is roughly about 2^{-k} where k denotes the bit-length of the modulus extension t. Therefore, the greater k, the more secure the countermeasure. However, the greater k, the slower the exponentiation (see Sect. 7.3). This kind of countermeasure hence offers a time/security tradeoff. A usual choice for k is 64 bits which provides a fair security. However, depending on the application, one may choose $k = 32$ (low security, more efficient exponentiation) or $k = 80$ (strong security, less efficient exponentiation).

2.4 Self-secure Exponentiations

For the countermeasures presented hereafter, the redundancy is not included in the modular operations anymore but at the exponentiation level. Namely, the exponentiation algorithm provides a direct way to check the consistency of the computation.

Giraud Scheme. The Giraud Scheme [18] relies on the use of the Montgomery powering ladder. It uses the fact that this exponentiation algorithm works with a pair of intermediate variables (a_0, a_1) storing values of the form $(m^\alpha, m^{\alpha+1})$. At the end of the exponentiation the pair (a_0, a_1) equals (m^{d-1}, m^d) and the consistency of the computation can be verified by checking wether $a_0 \cdot m$ equals a_1. If a fault is injected during the computation, the coherence between a_0 and a_1 is lost and the fault is detected by the final check.

Boscher *et al.* Scheme. The scheme by Boscher *et al.* [10] is based on the *right-to-left square-and-multiply-always* algorithm [15] which was originally devoted to thwart *simple side channel analysis* (see Sect. 6.1). In [10], the authors observe that this algorithm computes a triplet (a_0, a_1, a_2) that equals $(m^d, m^{2^l-d-1}, m^{2^l})$ at the end of the algorithm, where l denotes the bit-length of d. The principle of their countermeasure is hence to check that $a_0 \cdot a_1 \cdot m$ equals a_2 at the end of the exponentiation. Once again, in case of fault injection, the relation between the a_i's is broken and the fault is detected by the final check.

The main drawback of these two countermeasures is that they both impose the use of an exponentiation algorithm that performs 2 modular multiplications per bit of the exponent while other exponentiation algorithms require an average of 1.5 multiplications per bit of the exponent (and sometimes less).

In the next section, we propose a new self-secure exponentiation. Our method requires around 1.65 multiplications per bit of the exponent in average and hence constitutes an efficient alternative to the existing countermeasures.

3 A New Self-secure Exponentiation Based on Double Addition Chains

3.1 Basic Principle

In the following, we shall call *double exponentiation* an algorithm taking as inputs an element m and a pair of exponents (a, b), and computing the pair of powers (m^a, m^b).

The core idea of our method is to process a double exponentiation to compute the pair $(m^d, m^{\varphi(N)-d})$ modulo N. Then, the consistency of the computation is verified by performing the following check:

$$m^d \cdot m^{\varphi(N)-d} \overset{?}{\equiv} 1 \bmod N \ . \tag{1}$$

If no error occurs during the computation then, due to Euler's Theorem, this check is positive. In that case, the algorithm returns $m^d \bmod N$. On the other hand, if the computation is corrupted, then the result of this check is negative with high probability. In that case, the algorithm returns an error message.

In order to construct a self-secure exponentiation based on aforementioned principle, we need a double exponentiation algorithm. We propose hereafter such an algorithm that is well suited for implementation constrained in memory. Our solution is based on the building of an *addition chain*. This notion, as well as the ensued notion of *addition chain exponentiation* are briefly introduced in the next section (see [25] for more details).

3.2 Addition Chain Exponentiations

At first, we give the definition of an addition chain.

Definition 1. *An* addition chain *for an integer a is a sequence $x_0, x_1, \cdots, x_n$ with $x_0 = 1$ and $x_n = a$ that satisfies the following property: for every k there exist indices $i, j < k$ such that $x_k = x_i + x_j$.*

An addition chain $(x_i)_i$ for an integer a provides a way to evaluate any element m to the power a. Let $m_0 = m$. For k from 1 to n, one computes $m_k = m_i {\cdot} m_j$ where $i, j < k$ are such that $x_k = x_i + x_j$. By induction, the sequence $(m_k)_k$ satisfies: $m_k = m^{x_k}$ for every $k \leq n$ which leads to $m_n = m^{x_n} = m^a$. Such an addition chain exponentiation may require an important amount of memory to store the intermediate powers required for the computation of subsequent powers. This can make the exponentiation unpractical, especially in the context of low resource devices. Therefore, the minimum number of variables required to store the intermediate powers is an important parameter of the addition chain exponentiation. This parameter that directly results from the addition chain will be called the *memory depth* of the chain in the following.

In this paper, an addition chain $x_0, x_1, \cdots, x_n$ with $(x_{n-1}, x_n) = (a, b)$ is called a *double addition chain* for the pair (a, b). A double addition chain for a pair (a, b) provides a way to perform the double exponentiation $m \mapsto (m^a, m^b)$ for any element m.

Remark 1. What we call here double exponentiation shall not be confused with multi-exponentiations (also known as simultaneous exponentiations) that compute a product of powers $\prod_i m_i^{a_i}$ (see for instance [32]). What we call double addition chain is also called *addition sequence* in the general case where possibly more than two powers must be computed [11, 19]. Addition sequences have not been so much investigated. In [11], the authors propose some heuristics but these are not suitable for implementations constrained in memory.

3.3 A Heuristic for Double Addition Chains

In this section, we propose a heuristic to compute a double addition chain with a memory depth of 3 for any pair of natural integers (a, b). This provides us with a double exponentiation algorithm that is well suited for implementations constrained in memory.

Without loss of generality, we assume $a \leq b$. The chain involves a pair of intermediate results (a_i, b_i) that are initialized to $(0, 1)$ and that equal (a, b) once all the additions have been performed. In order to have a memory depth of 3, one single additional variable is used that keeps the value 1 (this amounts to keep the element m in a register for the resulting exponentiation). Therefore, at the i^{th} step of the chain, one can either increment a_i or b_i by 1, double a_i or b_i, or add a_i and b_i together.

To construct such a chain, we start from the pair (a, b) and go down to the pair $(0, 1)$ by applying the inverse operations. Namely, we define a sequence $(\alpha_i, \beta_i)_i$ such that $(\alpha_0, \beta_0) = (a, b)$ and $(\alpha_n, \beta_n) = (0, 1)$ for some $n \in \mathbb{N}$, and where, for every i, the pair $(\alpha_{i+1}, \beta_{i+1})$ is obtained from (α_i, β_i) by decrementing, by dividing by two and/or by subtracting an element to the other one. In order to limit the memory required to the storage of the chain, we have to restrict the set of possible operations. Our heuristic is the following one:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} \big(\alpha_i, \beta_i/2\big) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \bmod 2 = 0 \\ \big(\alpha_i, (\beta_i - 1)/2\big) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \bmod 2 = 1 \\ \big(\beta_i - \alpha_i, \alpha_i\big) & \text{if } \alpha_i > \beta_i/2 \end{cases} \qquad (2)$$

Proposition 1. *If $\alpha_0, \beta_0 \in \mathbb{N}^*$ are such that $\alpha_0 \leq \beta_0$ then the sequence $(\alpha_i, \beta_i)_i$ satisfies the following properties:*

1. *For every i, we have $\alpha_i \leq \beta_i$.*
2. *There exists $n \in \mathbb{N}$ such that $(\alpha_n, \beta_n) = (0, 1)$.*

Proof. The first property is straightforward: it is true for $i = 0$ and it is preserved by every step. The second one is demonstrated as follows. For every i such that $\alpha_i > 0$, we have $\alpha_{i+1} \leq \beta_{i+1} \leq \beta_i$ and $\alpha_{i+1} + \beta_{i+1} < \alpha_i + \beta_i$. This implies that there exists $n' \in \mathbb{N}$ such that $\alpha_{n'} > 0$ and $\alpha_{n'+1} \leq 0$. From (2), one deduces $\alpha_{n'} = \beta_{n'} > 0$ and $\alpha_{n'+1} = 0$. Denoting x the natural integer such that $(\alpha_{n'+1}, \beta_{n'+1}) = (0, x)$, we finally get $(\alpha_{n'+\lceil \log x \rceil}, \beta_{n'+\lceil \log x \rceil}) = (0, 1)$. $\diamond$

At this point, we need a binary representation for the sequence of additions to perform for the processing of the sequence $(a_i, b_i)_i$. Let us denote by n the natural integer satisfying $(\alpha_n, \beta_n) = (0, 1)$. We define τ and ν as the n-bit vectors whose coordinates satisfy:

$$\tau_i = \begin{cases} 0 & \text{if } \alpha_{n-i} \leq \beta_{n-i}/2 \\ 1 & \text{if } \alpha_{n-i} > \beta_{n-i}/2 \end{cases} \qquad (3)$$

and

$$\nu_i = \beta_{n-i} \bmod 2 \ . \qquad (4)$$

The sequence $(a_i, b_i)_i$ can be computed from τ and ν by initializing (a_0, b_0) to $(0, 1)$ and by iterating:

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i, 2b_i) & \text{if } \tau_{i+1} = 0 \text{ and } \nu_{i+1} = 0 \\ (a_i, 2b_i + 1) & \text{if } \tau_{i+1} = 0 \text{ and } \nu_{i+1} = 1 \\ (b_i, a_i + b_i) & \text{if } \tau_{i+1} = 1 \end{cases}$$

One can verify that $(a_i, b_i) = (\alpha_{n-i}, \beta_{n-i})$ holds for every i which yields $(a_n, b_n) = (a, b)$.

Let us remark that the whole sequence ν is not necessary for processing this addition chain (and the resulting exponentiation). Indeed, only the bits ν_i for which τ_i equals 0 are required. Therefore, the exponentiation algorithm shall make use of a single compressed sequence ω in order to avoid memory loss. We simply define ω as the sequence obtained from τ by inserting every bit ν_i for which $\tau_i = 0$ between τ_i and τ_{i+1}. In the sequel, we shall denote by n^* the bit-length of ω. Moreover, when we will need to make appear the relationship between the pair (a, b) and ω, we will use the notation $\omega(a, b)$.

The sequence $\omega(a,b)$ thus constitutes the binary representation of the double addition chain for the pair of exponents (a,b). To process the relying double exponentiation one must pre-compute ω. This is done by computing the pair (α_i, β_i) for every $i \in \{1, \cdots, n\}$. The following algorithm details such a computation. It makes use of two registers R_0 and R_1 that store the intermediate results α_i and β_i. It makes also use of a Boolean variable γ such that α_i is stored in $R_{\gamma\oplus 1}$ and β_i is stored in R_γ.

Algorithm 1 Double addition chain computation – ChainCompute

INPUT: A pair of natural integers (a,b) s.t. $a \leq b$
OUTPUT: The chain $\omega(a,b)$

```
1. $R_0 \leftarrow a$; $R_1 \leftarrow b$; $\gamma \leftarrow 1$; $j \leftarrow n^*$
2. while $(R_{\gamma\oplus 1}, R_\gamma) \neq (0,1)$ do
3.     if $(R_\gamma/2 > R_{\gamma\oplus 1})$
4.         then $\omega_{j-1} \leftarrow 0$; $\omega_j \leftarrow R_\gamma \bmod 2$; $R_\gamma \leftarrow R_\gamma/2$; $j \leftarrow j-2$
5.         else $\omega_j \leftarrow 1$; $R_\gamma \leftarrow R_\gamma - R_{\gamma\oplus 1}$; $\gamma \leftarrow \gamma \oplus 1$; $j \leftarrow j-1$
6. end while
7. return $\omega$
```

Remark 2. The length n^* is *a priori* unknown before the computation of the chain. However, as shown in Sect. 7.2, it is upper bounded by $2.2\lceil \log b \rceil$ (with high probability). For a practical implementation of Algorithm 1, one may use a buffer of $2.2\lceil \log b \rceil$ bits to store ω, initializing j by the final bit index of this buffer.

The following algorithm describes the resulting double modular exponentiation algorithm. It makes use of two registers R_0 and R_1 that store the intermediate results m^{a_i} and m^{b_i} and one more register to hold m. It makes also use of a Boolean variable γ such that m^{a_i} is stored in $R_{\gamma\oplus 1}$ and m^{b_i} is stored in R_γ.

Algorithm 2 Double modular exponentiation – DoubleExp

INPUT: An element $m \in \mathbb{Z}_N$, a chain $\omega(a,b)$ s.t. $a \leq b$, a modulus N
OUTPUT: The pair of modular powers $(m^a \bmod N, m^b \bmod N)$

```
1. $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $\gamma \leftarrow 1$
2. for $i = 1$ to $n^*$ do
3.     if $(\omega_i = 0)$ then
4.         $R_\gamma \leftarrow R_\gamma^2 \bmod N$; $i \leftarrow i+1$
5.         if $(\omega_i = 1)$ then $R_\gamma \leftarrow R_\gamma \cdot m \bmod N$
6.     else
7.         $R_{\gamma\oplus 1} \leftarrow R_{\gamma\oplus 1} \cdot R_\gamma \bmod N$; $\gamma \leftarrow \gamma \oplus 1$
8. end for
9. return $(R_{\gamma\oplus 1}, R_\gamma)$
```

3.4 The Secure Exponentiation Algorithm

Following the principle described in Sect. 3.1, Algorithm 2 provides a way to perform a modular exponentiation secure against fault analysis. The resulting secure modular exponentiation is depicted in the following algorithm.

Algorithm 3 Secure modular exponentiation

```
INPUT: A message $m$, a secret exponent $d$, a modulus $N$ and its Euler's totient $\varphi(N)$
OUTPUT: The modular power $m^d \bmod N$

1. $\omega \leftarrow \mathsf{ChainCompute}\big(d,\ 2\varphi(N) - d\big)$
2. $(s, c) \leftarrow \mathsf{DoubleExp}\big(m,\ \omega,\ N\big)$
3. if $s \cdot c \bmod N \neq 1$ then return "error";  else return $s$
```

Remark 3. For the chain computation (Step 1), $\varphi(N) - d$ is replaced by $2\varphi(N) - d$ in order to fit the constraint $a \leq b$ imposed by the chain computation algorithm. This does not affect the result of the double exponentiation in Step 2 since we have $m^{\varphi(N)-d} \equiv m^{2\varphi(N)-d} \bmod N$.

4 A New Secure RSA-CRT Algorithm

For an RSA computation, the secure modular exponentiation proposed above can be extended to be performed in CRT mode. Two double exponentiations are performed separately in order to compute the pairs (s_p, c_p) and (s_q, c_q) where $c_p = m^{p-1-d_p} \bmod p$ and $c_q = m^{q-1-d_q} \bmod q$. Then the signature s is recovered from s_p and s_q by CRT recombination and its value is checked modulo p (resp. q) using c_p (resp. c_q) according to (1).

Algorithm 4 Secure RSA-CRT

```
INPUT: A message $m$, the secret exponents $d_p$ and $d_q$, the secret primes $p$ and $q$
OUTPUT: The modular power $m^d \bmod p \cdot q$

1. $\omega_p \leftarrow \mathsf{ChainCompute}\big(d_p,\ 2(p-1) - d_p\big)$
2. $(s_p, c_p) \leftarrow \mathsf{DoubleExp}(m \bmod p,\ \omega_p,\ p)$
3. $\omega_q \leftarrow \mathsf{ChainCompute}\big(d_q,\ 2(q-1) - d_q\big)$
4. $(s_q, c_q) \leftarrow \mathsf{DoubleExp}(m \bmod q,\ \omega_q,\ q)$
5. $s \leftarrow \mathsf{CRT}(s_p, s_q)$
6. if $(s \cdot c_p \bmod p \neq 1$  or  $s \cdot c_q \bmod q \neq 1)$  then return "error"  else return $s$
```

Remark 4. We assume that $m \bmod p$ (resp. $m \bmod q$) cannot be corrupted before the beginning of the double exponentiation. This is mandatory for the security of Algorithm 4, since such a corruption would not be detected and would enable the Bellcore attack. In practice, this can be ensure by computing a cyclic redundancy code for $m \bmod p$ (resp. $m \bmod q$) at the beginning of the RSA-CRT algorithm. Then, at the beginning of the double exponentiation algorithm, $m \bmod p$ (resp. $m \bmod q$) is recomputed from m and its integrity is checked once it has been loaded in two different registers (m and R_1 in Algorithm 2). Any corruption occurring after this check shall be detected by the final check.

Remark 5. The chains ω_p and ω_q can be either computed on-the-fly as depicted in Algorithm 4 (Steps 1 and 3) or pre-computed and stored in non-volatile memory. The first solution has the advantages of preserving the classical RSA-CRT parameters and of enabling the exponent blinding countermeasure (see Sect. 6.2). The second solution has the advantage of avoiding the timing and memory overhead induced by the chain computations.

5 Security Against Fault Analysis

In this section, we analyze the security of our method against fault analysis. We start with a few remarks of practical purpose, then we investigate the detection probability of a fault injection and finally we address safe-error attacks.

Remark 6. We assume that the Boolean γ cannot be modified at the end of Algorithm 2. This is mandatory for the security of the solution since such a modification would result in a swapping of the two registers which would not be detected. In practice, this can be ensured by doubling the variable γ.

Remark 7. We assume that one cannot switch the last bit(s) of the chain ω (resp. ω_p, ω_q) from 1 to 00 (or *vice cersa*). This would provoke an undetected error. Such a switch can be prevented in practice by checking that the loop index i matches the chain length n^* at the end of Algorithm 2.

Remark 8. In Algorithms 3 and 4, we assume that the integrity of the chain computation parameters is checked before executing the chain computation algorithm. This avoids any attack that would corrupt d (resp. d_p, d_q) before the computation of $2\varphi(N) - d$ (resp. $2(p-1) - d_p$, $2(q-1) - d_q$).

Remark 9. Some papers claim that coherence checks using conditional branches should be avoided to strengthen fault analysis security [42, 14]. The argument behind this assertion is that the coherence check could be easily skipped by corrupting the status register. An alternative solution to direct coherence checking is to use an *infection procedure* that renders the erroneous signature harmless in case of fault detection [42]. However, most of the proposed countermeasures have security flaws due to ineffective infection methods (for instance [7, 14] have been broken in [39, 4]). Moreover, the infection procedure can also be skipped as it has been practically demonstrated in [23]. In [16], a simple solution is proposed that performs a coherence check without conditional branches in a way that is secure against operations skipping. We suggest to use this solution for the coherence checks performed in Algorithm 3 (Step 3) and Algorithm 4 (Step 6).

5.1 Fault Detection

We analyze hereafter the different fault attacks that can be attempted on our secure exponentiation algorithm and we investigate the corresponding detection probability. We only focus on transient faults, namely faults whose effect lasts for one computation. Permanent fault attacks are easily thwarted by the addition of some cyclic redundancy codes to check the parameters integrity.

We use the generic notation M to denote the involved modulus that may equal N (for a straightforward RSA), p or q (for a RSA-CRT) and we denote by $\mathrm{ord}_M(m)$ the order of an element m in $\mathbb{Z}_M^*$. When the fault causes the corruption of an intermediate variable v, we denote the corrupted variable by $\widetilde{v}$ and the error by ε such that $\widetilde{v} = v + \varepsilon$. We analyze here the condition about ε for a non-detection and we bound the probability $\mathcal{P}$ of non-detection in the *uniform fault model i.e.* assuming that ε is uniformly distributed.

For our analysis, the following lemma shall be useful (see the proof in Appendix A).

Lemma 1. *Let M be an integer greater than* 30*. Let m be a random variable uniformly distributed over $\mathbb{Z}_M^*$ and let u be a random variable uniformly distributed over $\{1, \cdots, \varphi(M)\}$ and independent of m. We have:*

$$\mathrm{P}\left(\mathrm{ord}_M(m) | u\right) < \frac{2}{M^{1/3}} \ . \tag{5}$$

For the sake of simplicity, we approximate hereafter a uniform distribution over $\mathbb{Z}_M$ by a uniform distribution over $\mathbb{Z}_M^*$. This approximation is sound in our context since M is a large prime or an RSA modulus.

Corruption of one of the two exponents. Among the exponents a and b, one equals d and the other one equals $\varphi(M) - d$. On the one hand, if $\varphi(M) - d$ is corrupted, then the result of the exponentiation remains correct (*i.e.* it equals $m^d \bmod M$) and the attack failed whatever the result of the final check (which is however very likely to detect the fault). On the other hand, if d is corrupted, we show hereafter that the final check will detect the error with high probability.

In fact, the error is not detected if and only if we have $m^{\widetilde{d}} \cdot m^{\varphi(M)-d} \equiv 1 \bmod M$ that is $m^{\varepsilon} \equiv 1 \bmod M$. This occurs if and only if ε is a multiple of the order of m. Therefore, the probability of non-detection can be expressed as $\mathcal{P} = \mathrm{P}\left(\mathrm{ord}_M(m) | \varepsilon\right)$. Hence, the lower the order of m, the higher the probability of non-detection. Since a potential attacker does not know $\varphi(M)$, he cannot chose m in a way that affect its order. For this reason, m can be considered uniformly distributed over $\mathbb{Z}_M$. Therefore, in the uniform fault model, Lemma 1 implies $\mathcal{P} < 2/M^{1/3}$.

Remark 10. The bound provided by Lemma 1 is not tight at all but it is sufficient to show that $\mathcal{P}$ is negligible. For instance, if M satisfies $\log M \geq 244$, which is necessary (but not sufficient) for the security of RSA (even for RSA-CRT where $\log N = 2 \log M$), $\mathcal{P}$ is strictly lower than 2^{-80} which is negligible.

Corruption of the message or an intermediate power. From the definition of the double addition chain given in Sect. 3.3, one can see that for every $i \in \{1, \cdots, n\}$, the pair (a_n, b_n) can be expressed as a linear transformation of the triplet $(a_i, b_i, 1)$. Let us denote by α_i^a, β_i^a, δ_i^a the three coefficients of the expression of a_n, namely $a_n = \alpha_i^a a_i + \beta_i^a b_i + \delta_i^a$. By analogy, we denote by α_i^b, β_i^b, δ_i^b the coefficients in the expression of b_n.

If the message m is corrupted at the i^{th} step of the exponentiation, this last returns the following pair of powers: $\left(m^a(m^{-1} \cdot \widetilde{m})^{\delta_i^a}, m^b(m^{-1} \cdot \widetilde{m})^{\delta_i^b}\right)$ modulo M. The error is not detected if and only if we have $(m^{-1} \cdot \widetilde{m})^{\delta_i^a + \delta_i^b} \equiv 1 \bmod M$, that is $(1 + \varepsilon \cdot m^{-1})^{\delta_i^a + \delta_i^b} \equiv 1 \bmod M$. This occurs if and only if the order of $m' = 1 + \varepsilon \cdot m^{-1}$ divides $\delta_i^a + \delta_i^b$. Therefore, the probability of non-detection can be expressed as $\mathcal{P} = \mathrm{P}\left(\mathrm{ord}_M(m') | \delta_i^a + \delta_i^b\right)$. Following the same reasoning, a corruption of the intermediate power m^{a_i} (resp. m^{b_i}) is not detected with a probability $\mathcal{P} = \mathrm{P}\left(\mathrm{ord}_M(m') | \alpha_i^a + \alpha_i^b\right)$ where $m' = 1 + e \cdot m^{-a_i}$ (resp. $\mathcal{P} = \mathrm{P}\left(\mathrm{ord}_M(m') | \beta_i^a + \beta_i^b\right)$ where $m' = 1 + \varepsilon \cdot m^{-b_i}$).

Since a and b are unknown to the attacker, this one cannot chose the value of $\delta_i^a + \delta_i^b$, $\alpha_i^a + \alpha_i^b$ or $\beta_i^a + \beta_i^b$ since these directly ensue from a and b. That is why, we make the heuristic assumption that $\mathcal{P}$ equals $\mathrm{P}\left(\mathrm{ord}_M(m') | u\right)$ where u is uniformly distributed over $\{1, \cdots, \varphi(M)\}$. In the uniform fault model, we have the uniformity of m' that holds from the one-to-one relationship between ε and m' for every $m \neq 0$. Consequently, Lemma 1 implies $\mathcal{P} < 2/M^{1/3}$ and p is negligible.

Corruption of the chain. A faulty chain $\widetilde{w}$ results in a faulty pair of powers $(m^{\widetilde{a}}, m^{\widetilde{b}})$. The error is not detected if and only if the order of m divides $\widetilde{a} + \widetilde{b}$, hence the non-detection probability can be expressed as $\mathcal{P} = \mathrm{P}\left(\mathrm{ord}_M(m) | \widetilde{a} + \widetilde{b}\right)$.

As shown in Sect. 7.2, the expected bit-length of the chain ω yielding a pair of l-bit exponents (a, b) is of $2l$. This suggests an almost bijective relationship between the chains space and the exponents pairs space. In the uniform fault model, we can therefore consider that $\widetilde{a}$ and $\widetilde{b}$ are uniformly distributed which, by Lemma 1, implies $\mathcal{P} < 2/M^{1/3}$.

Corruption of the modulus. If the modulus M is corrupted at the i^{th} step of the exponentiation, then this last results in the two following powers: $m_1^{\alpha_i^a} \cdot m_2^{\beta_i^a} \cdot m^{\delta_i^a} \bmod \widetilde{M}$ and $m_1^{\alpha_i^b} \cdot m_2^{\beta_i^b} \cdot m^{\delta_i^b} \bmod \widetilde{M}$ where $m_1 = m^{a_i} \bmod M$ and $m_2 = m^{b_i} \bmod M$. Therefore, the error is not detected if and only if we have $m_1^{\alpha_i^a + \alpha_i^b} \cdot m_2^{\beta_i^a + \beta_i^b} \cdot m^{\delta_i^a + \delta_i^b} \bmod \widetilde{M} = 1$.

In the uniform fault model, the faulty modulus $\widetilde{M}$ is uniformly distributed over $[0, 2^l[$ where l denotes the bit-length of M. Therefore, the probability of non-detection $\mathcal{P}$ is close to $\mathrm{P}\left(u_1 \bmod u_2 = 1\right)$ where u_1 and u_2 are uniform (and independent) random variables over $[0, 2^l[$. This probability equals $2^{-l} \sum_{i=1}^{2^l - 1} (1/i)$

which is strictly lower than 2^{-80} for every $l \geq 86$. The probability of non-detection $\mathcal{P}$ is hence negligible in our context.

5.2 Safe-error Attacks

As recalled in Sect. 2.2, safe-error attacks divide into two categories: C-safe-error attacks [41] and M-safe-error attacks [40, 24].

To prevent C-safe-error attacks one must ensure that no dummy operation is conditionally performed depending on the secret key. Our secure exponentiation does not perform any dummy operation and is hence secure against C-safe-error attacks. When the chain is computed on-the-fly, it must be done in an atomic way in order to thwart simple side channel analysis (see Sect. 6.1). The atomic version of the chain computation algorithm (see Appendix B) makes use of dummy operations and is hence vulnerable to C-safe-error attacks. In that case, these can be thwarted by using the exponent blinding countermeasure (see Sect. 6.2).

To prevent M-safe-error attacks one can either randomize the exponent (using for instance the exponent blinding) or randomize the indices of the registers that are addressed by some exponent bits (or chain bits in our context). When the chain is pre-computed, the exponent cannot be randomized and the *registers indices randomization* introduced hereafter shall be used. The principle is to randomly chose the registers to store the different variables among the different used registers. For instance, in Algorithm 1, a random bit r is picked up so that the registers R_0 and R_1 are switched if r equals 1. In the description of Algorithm 1 this amounts to replace R_γ by $R_{\gamma \oplus r}$. In this way, a M-safe-error attack will imply a faulty output once out of two, independently of the performed operation. This simple countermeasure thwarts the attacks recently published in [24].

6 Toward Side Channel Analysis Resistance

In this section, we address the resistance of our exponentiation algorithm against the two main kinds of side channel analysis (SCA): *simple SCA* and *differential SCA*.

6.1 Simple Side Channel Analysis

Simple SCA [26] exploits the fact that the operation flow of a cryptographic algorithm may depend on the secret key. Different operations may induce different patterns in the side channel leakage which provides secret information to any attacker able to eavesdrop this leakage. To thwart simple SCA, an algorithm must be *atomic* [13], namely, it must have the same operation flow whatever the secret key.

The chain computation algorithm (Algorithm 2) and the double exponentiation algorithm (Algorithm 1) may be vulnerable to simple SCA. To circumvent this weakness, we provide atomic versions of these algorithms in Appendix B.

6.2 Differential Side Channel Analysis

Differential SCA [26] exploits the fact that the side channel leakage reveals information about some key-dependent intermediate variables of the computation. Since its first publication, several improvements of differential SCA have been proposed, especially to attack modular exponentiation [1, 17, 20, 30]. In order to thwart differential SCA, one usually makes use of randomization techniques. The message randomization as well as the modulus randomization are usual countermeasures that can be straightforwardly combined with our method. The exponent is usually randomized using the blinding technique that consists in performing the exponentiation to the power $d' = d + r \cdot \varphi(N)$ for a small random number r [27, 30, 15]. This technique cannot be straightforwardly applied while using our secure exponentiation algorithm since we have $d' > \varphi(N)$ for every $r > 0$. Therefore, we propose the following simple adaptation: in Step 1 of Algorithm 3, the exponent a is set to $d + r_1 \cdot \varphi(N)$ and the exponent b is set to $r_2 \cdot \varphi(N) - d$ where r_1 and r_2 are two small random numbers with $r_2 \geq r_1 + 2$. Then the rest of the secure exponentiation algorithm does not change. Since $m^{d+r_1 \cdot \varphi(N)} \equiv m^d \bmod N$, the desired signature is computed and since $m^{d+r_1 \cdot \varphi(N)} \cdot m^{r_2 \cdot \varphi(N)-d} \equiv 1 \bmod N$, the final check is correctly carried out.

Remark 11. If the chain ω is pre-computed, the exponent blinding cannot be used. In that case, another kind of randomization (message, modulus) shall be used. However, these do not prevent a differential SCA targeting the chain itself (as for instance the SEMD attack of [30] or the address-bit DPA [20]). To deal with this issue, we suggest to use a Boolean masking such as proposed in [21].

7 Complexity Analysis

In this section we analyze the time complexity and the memory complexity of our proposal. In the sequel, we shall denote by l the bit-length of the exponentiation inputs. Namely for a straightforward RSA we have $l = \lceil \log N \rceil$ and for a RSA-CRT we have $l = \lceil \log N/2 \rceil$.

7.1 Time Complexity

Our secure exponentiation is mainly composed of the chain computation and the double exponentiation. The chain computation loop is shorter than the exponentiation loop and it involves simple operations (*e.g.* substraction, division by 2) whose time complexities are negligible compared to a modular multiplication. Therefore, the time complexity of our proposal mainly depends on the number of multiplications performed by the double exponentiation algorithm (all the more so as the chain may be pre-computed). We shall denote this number by m and we shall define the *multiplications-per-bit ratio* as the coefficient θ satisfying $m = \theta l$.

Some practical values for the expectation and the standard deviation of θ are given in Table 7.1 that were obtained by simulations. For $l \in \{512, \cdots, 1024\}$, the expected multiplications-per-bit ratio is around 1.65. Compared to the classical *square-and-multiply* algorithm, our exponentiation hence requires 10% more multiplications, implying a 10% overhead in average, which is a fair cost for fault analysis resistance. Moreover, the time complexity of our exponentiation is steadier than the one of the square-and-multiply since the standard deviation $\sigma(\theta)$ is lower than $1/5$ and decreasing for $l \geq 512$ while, for the square-and-multiply algorithm, it is constant to $1/4$.

Table 1. Expectation and standard deviation of the double exponentiation multiplications-per-bit ratio.

	$l = 512$	$l = 640$	$l = 768$	$l = 896$	$l = 1024$
$\mathrm{E}(\theta)$	1.65	1.66	1.66	1.66	1.66
$\sigma(\theta)$	0.020	0.017	0.017	0.016	0.014

7.2 Memory Complexity

Our double exponentiation algorithm requires three l-bit registers to store the message and the pair of powers. If the chain ω is computed on-the-fly, it requires an additional buffer is necessary to store it.

We performed simulations to derive the practical values of the expectation and the standard deviation of the chain length n^*. For the expectation, we obtained $\mathrm{E}(n^*) \approx 2.03\ l$ for $l \in \{512, \cdots, 1024\}$. For the standard deviation, the obtained values are summarized in Table 2. Approximating the distribution of n^* by a Gaussian, we get $\mathrm{P}(n^* > \mathrm{E}(n^*) + k\sigma(n^*)) = \big(1 - \mathrm{erf}\big(k/\sqrt{2}\big)\big)/2$ where $\mathrm{erf}(\cdot)$ denotes the error function. For $k = 10$ and for $l \in \{512, \cdots, 1024\}$, this probability is lower than 2^{-80}. Consequently, for $l \in \{512, \cdots, 1024\}$, the probability to have $n^* > 2.2\ l$ is negligible in practice, hence ω can be stored in a $(2.2\ l)$-bit buffer.

Table 2. Standard deviation of the chain bit-length.

	$l = 512$	$l = 640$	$l = 768$	$l = 896$	$l = 1024$
$\sigma(n^*)$	$0.015\ l$	$0.013\ l$	$0.011\ l$	$0.010\ l$	$0.010\ l$

On the whole, our secure exponentiation requires $5.2\ l$ bits of memory when the chain is computed on-the-fly and it requires $3\ l$ bits of memory when the chain is pre-computed.

For our secure RSA-CRT (see Algorithm 4), the peak of memory consumption is reached in the second exponentiation while s_p and c_p must be kept in memory. This makes a total memory consumption of 7.2 l bits with on-the-fly chain computation and of 5 l bits with pre-computed chain.

7.3 Comparison With Previous Solutions

We analyze hereafter the complexity of previous countermeasures in the literature. As explained in Sect. 2, these can be divided in two categories: the extended modulus based countermeasures and the self-secure exponentiations.

Extended Modulus Based Countermeasures. The time complexity of an extended modulus based countermeasure (such as the Shamir's trick or the Vigilant Scheme) is around the complexity of the main exponentiation loop(s) since the additional computations are negligible. However, such countermeasures are not free in terms of timing since the use of an extended modulus slows down the exponentiation. In fact, the time complexity of a modular multiplication can be written as $l^2 t_0$ where t_0 denotes a constant time that depends on the device architecture. Denoting by k the bit-length of the modulus extension, an extended modulus exponentiation has a time complexity of $m(l + k)^2 t_0$ while a normal exponentiation has a time complexity of $ml^2 t_0$. Besides, the modulus extension implies an increase of the exponentiation execution time by a factor $(1 + k/l)^2$. As an illustration, Table 7.3 gives several values of the induced overhead according to the modulus length and to the extension length. For instance,

Table 3. Time overhead (in %) for an extended modulus based modular exponentiation.

	$l = 512$	$l = 768$	$l = 1024$
$k = 32$ (low security)	13	9	6
$k = 64$ (fair security)	27	17	13
$k = 80$ (strong security)	34	22	16

an RSA 1024 implemented in CRT ($l = 512$) with extended modulus providing a fair level of security ($k = 64$) is about 27% slower than an unprotected one. This time overhead is sizeable; in particular it is significantly greater than the 10% overhead induced by our countermeasure. However, extended modulus based countermeasures enables the use of exponentiation algorithms faster that the square-and-multiply such as the q-ary or the sliding windows methods (see for instance [29]). Roughly, a q-ary exponentiation has a multiplications-per-bit ratio of $1 + (2^q - 1)/(q2^q)$ which is lower than or equal to 1.5, but it has a higher memory complexity since it requires $2^{q-1} + 1$ registers. The use of a sliding window allows to slightly improve the time complexity of a q-ary method [28].

The memory complexity of an exponentiation with modulus extension is of $n_r(l+k)$ where n_r denotes the number of registers required by the exponentiation algorithm. For an RSA-CRT, the memory complexity depends on the used countermeasure. For the Vigilant Scheme, the memory consumption peak occurs during the second exponentiation while the values S'_p, i_{qr}, r, R_3 and R_4 must hold in memory (see [38]). This results in a memory consumption of $n_r(l+k)+(l+k)+3.5\ k=(n_r+1)\cdot l+(n_r+4.5)\cdot k$ bits.

Remark 12. We do not detail the memory complexity of the other extended modulus based countermeasures since, for most of them, it is close to the memory complexity of the Vigilant Scheme.

Previous Self-secure Exponentiations. The Giraud Scheme and the Boscher *et al.* Scheme both have a multiplications-per-bit ratio constant to 2. This implies an average time overhead of 33% compared to the square-and-multiply algorithm and of 21% compared to our exponentiation. However, both of these schemes do not require additional computations contrary to the extended modulus based countermeasures or to our scheme when the chain is computed on-the-fly. Although these additional computations are theoretically negligible, they may induce an overhead for a practical implementation depending on the device architecture.

In terms of memory, we shall focus on the Giraud Scheme since it is less consuming than the Boscher *et al.* Scheme. The secure exponentiation requires two l-bit registers. For the RSA-CRT, the peak of memory consumption is reached during the two recombinations. For instance, the first recombination requires (at least) $3l$ bits of memory while m, S_p and S_q must hold in memory (see [18]) which makes a total complexity of $7l$ bits.

Comparison with our Solution. Table 4 provides a comparison between the Giraud Scheme, the Vigilant Scheme and ours for an RSA 1024 with CRT (*i.e.* $l=512$). For the Vigilant Scheme, we assume a modulus extension of $\{64,80\}$ bits and a q-ary sliding window exponentiation for $q=1,2$ or 3 [29]. The results

Table 4. Memory and time complexities of different fault analysis countermeasures for an RSA 1024 with CRT.

Countermeasure	Time ($10^6 \cdot t_0$)	Memory (Kb)
Vigilant [38] ($q=1$)	$\{511,484\}$	{2.4, 2.3}
Vigilant [38] ($q=2$)	$\{468,444\}$	{2.6, 2.5}
Vigilant [38] ($q=3$)	$\{440,417\}$	{3.7, 3.6}
Giraud [18]	537	3.5
This paper	443	2.5 (+1.1)

given in Table 4 shows that our countermeasure is currently one of the most competitive solution to thwart fault analysis for an RSA 1024 with CRT.

Remark 13. The time complexity for the Vigilant Scheme with sliding widow is computed as follows. A q-ary exponentiation performs an average of $l \cdot \big(1 + (2^q - 1)/(q2^q)\big)$ multiplications [29] and the use of a sliding window yields an improvement of about 5% for $l = 512$ [28]. Therefore, the time complexity of one exponentiation is estimated to $0.95 \cdot (l + k)^2 t_0 \cdot l \cdot \big(1 + (2^q - 1)/(q2^q)\big)$. Concerning the memory complexity, the sliding window method requires a total of $n_r = 2^{q-1} + 1$ registers.

8 Conclusion

In this paper, we have described a new countermeasure to protect exponentiation and RSA against fault analysis. The core idea of our method is to introduce redundancy in the computation by performing a double exponentiation. To do so, we proposed a double exponentiation algorithm that is based on the computation of an addition chain. We analyzed the security of our solution *vs* fault analysis and we showed how it can be protected against side channel analysis. We also studied the time and memory complexities of our countermeasure which showed that it offers an efficient alternative to the existing schemes. A direction for further research would be to investigate more efficient double exponentiation algorithms and time-memory tradeoffs.

Acknowledgements

I would like to thank Jean-Sébastien Coron, Emmanuelle Dottax, Christophe Giraud, Gilles Piret and Emmanuel Prouff for helpful comments. I am also especially grateful to one of the anonymous reviewers of CT-RSA 2009 for valuable suggestions.

References

1. F. Amiel, B. Feix, and K. Villegas. Power Analysis for Secret Recovering and Reverse Engineering of Public Key Algorithms. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *SAC 2007*, LNCS, pages 110–125. Springer, 2007.
2. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In B. Kaliski Jr., Ç. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 260–275. Springer, 2002.
3. F. Bao, R. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, and T.-H. Ngair. Breaking Public Key Cryptosystems an Tamper Resistance Devices in the Presence of Transient Fault. In *5th Security Protocols Workshop*, volume 1361 of *LNCS*, pages 115–124. Springer, 1997.

4. A. Berzati, C. Canovas, and L. Goubin. (In)security Against Fault Injection Attacks for CRT-RSA Implementations. In L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, editors, *FDTC 2008*, pages 101–107. IEEE Computer Society, 2008.
5. A. Berzati, C. Canovas, and L. Goubin. Perturbating RSA Public Keys: An Improved Attack. In E. Oswald and P. Rohatgi, editors, *CHES 2007*, volume 5154 of *LNCS*, pages 380–395. Springer, 2008.
6. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystem. In B. Kalisky Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.
7. J. Blömer, M. Otto, and J.-P. Seifert. A New RSA-CRT Algorithm Secure against Bellcore Attacks. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *CCS'03*, pages 311–320. ACM Press, 2003.
8. D. Boneh, R. DeMillo, and R. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In W. Fumy, editor, *EUROCRYPT '97*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.
9. M. Boreale. Attacking Right-to-Left Modular Exponentiation with Timely Random Faults. In L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, editors, *FDTC 2006*, volume 4236 of *LNCS*, pages 24–35. Springer, 2006.
10. A. Boscher, R. Naciri, and E. Prouff. CRT RSA Algorithm Protected against Fault Attacks. In D. Sauveron, K. Markantonakis, A. Bilas, and J.-J. Quisquater, editors, *WISTP 2007*, volume 4462 of *LNCS*, pages 229–243. Springer, 2007.
11. J. Bos and M. Coster. Addition chain heuristics. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 400–407. Springer, 1989.
12. E. Brier, B. Chevallier-Mames, M. Ciet, and C. Clavier. Why One Should Also Secure RSA Public Key Elements. In L. Goubin and M. Matsui, editors, *CHES 2006*, volume 4249 of *LNCS*, pages 324–338. Springer, 2006.
13. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
14. M. Ciet and M. Joye. Practical Fault Countermeasures for Chinese Remaindering Based RSA. In L. Breveglieri and I. Koren, editors, *FDTC'05*, pages 124–132, 2005.
15. J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. Koç and C. Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.
16. E. Dottax, C. Giraud, M. Rivain, and Y. Sierra. On Second-Order Fault Analysis Resistance for CRT-RSA Implementations. Cryptology ePrint Archive, Report 2009/24, 2009. http://eprint.iacr.org/2009/024.
17. P.-A. Fouque and F. Valette. The Doubling Attack: Why Upwards is Better than Downwards. In C. Walter, Ç. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 269–280. Springer, 2003.
18. C. Giraud. An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. *IEEE Transactions on Computers*, 55(9):1116–1120, Sept. 2006.
19. D. M. Gordon. A Survey of Fast Exponentiation Methods. *J. Algorithms*, 27(1):129–146, 1998.
20. K. Itoh, T. Izu, and M. Takenak. Address-bit Differential Power Analysis of Cryptographic Schemes OK-ECDH and OK-ECDSA. In B. Kaliski Jr., Ç. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 129–143. Springer, 2002.

21. K. Itoh, T. Izu, and M. Takenaka. A Practical Countermeasure against Address-Bit Differential Power Analysis. In C. Walter, Ç. Koç, and C. Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 382–396. Springer, 2003.
22. M. Joye, A. Lenstra, and J.-J. Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *Journal of Cryptology*, 12(4):241–245, 1999.
23. C. H. Kim and J.-J. Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In D. Sauveron, K. Markantonakis, A. Bilas, and J.-J. Quisquater, editors, *WISTP 2007*, volume 4462 of *LNCS*, pages 215–228. Springer, 2007.
24. C. H. Kim, J. H. Shin, J.-J. Quisquater, and P. J. Lee. Safe-error attack on spa-fa resistant exponentiations using a hw modular multiplier. In K.-H. Nam and G. Rhee, editors, *ICISC*, volume 4817 of *LNCS*. Springer, 2007.
25. D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, third edition, 1988.
26. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
27. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
28. Ç. Koç Analysis of the Sliding Window Techniques for Exponentiation. *Computer & Mathematics with applications*, 30(10):17–24, 1995.
29. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
30. T. Messerges, E. Dabbish, and R. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcard. In Ç. Koç and C. Paar, editors, *CHES '99*, volume 1717 of *LNCS*, pages 144–157. Springer, 1999.
31. D. S. Mitrinovic, J. Sándor, and B. Crstici. *Handbook of Number Theory*. Springer, 1995.
32. B. Möller. Algorithms for Multi-exponentiation. In S. Vaudenay and A. Youssef, editors, *SAC 2001*, volume 2259 of *LNCS*, pages 165–180. Springer, 2001.
33. R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
34. J. Schmidt and C. Herbst. A Practical Fault Attack on Square and Multiply. In L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, editors, *FDTC 2008*, pages 53–58. IEEE Computer Society, 2008.
35. J.-P. Seifert. On Authenticated Computing and RSA-based Authentication. In V. Atluri, C. Meadows, and A. Juels, editors, *ACM CCS 2005*, pages 122–127. ACM, 2005.
36. A. Shamir. Improved Method and Apparatus for Protecting Public Key Schemes from Timing and Fault Attacks. Publication number: WO9852319, Nov. 1998.
37. Sun Microsystems. Application Programming Interface – Java Card$^{\mathrm{TM}}$ Plateform, Version 2.2.2, Mar. 2006. http://java.sun.com/products/javacard/specs.html.
38. D. Vigilant. RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In E. Oswald and P. Rohatgi, editors, *CHES 2007*, volume 5154 of *LNCS*, pages 130–145. Springer, 2008.
39. D. Wagner. Cryptanalysis of a Provable Secure CRT-RSA Algorithm. In B. Pfitzmann and P. Liu, editors, *CCS'04*, pages 82–91. ACM Press, 2004.
40. S.-M. Yen and M. Joye. Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.

41. S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. A Countermeasure against one Physical Cryptanalysis May Benefit Another Attack. In K. Kim, editor, *ICISC 2001*, volume 2288 of *LNCS*, pages 414–427. Springer, 2001.
42. S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. RSA Speedup with Residue Number System Immune against Hardware Fault Cryptanalysis. *IEEE Transactions on Computers*, 52(4):461–472, 2003.

A Proof of Lemma 1

Proof. By the law of total probability, we have:

$$\mathrm{P}\left(\mathrm{ord}_M(m)|u\right) = \sum_{\lambda \in \mathcal{D}(\varphi(M))} \mathrm{P}\left(\lambda|u\right) \mathrm{P}\left(\mathrm{ord}_M(m) = \lambda\right) \ , \tag{6}$$

where $\mathcal{D}$ is the function mapping a natural integer to the set of its divisors. On the one hand, the probability $\mathrm{P}\left(\lambda|u\right)$ equals $1/\lambda$. On the other hand, for every $\lambda \in \mathcal{D}(\varphi(M))$, there are $\varphi(\lambda)$ elements of order λ in $\mathbb{Z}_M^*$ which leads to $\mathrm{P}\left(\mathrm{ord}_M(m) = \lambda\right) = \varphi(\lambda)/\varphi(M)$. On the whole, (6) can be rewritten as:

$$\mathrm{P}\left(\mathrm{ord}_M(m)|u\right) = \frac{1}{\varphi(M)} \sum_{\lambda \in \mathcal{D}(\varphi(M))} \frac{\varphi(\lambda)}{\lambda} \ . \tag{7}$$

Since $\varphi(\lambda)/\lambda$ is strictly lower than or equal to 1, we have $\mathrm{P}\left(\mathrm{ord}_M(m)|u\right) \leq d(\varphi(M))/\varphi(M)$ where $d(\cdot)$ denotes the divisor function (*i.e.* the function that maps a natural integer to the quantity of its distinct divisors). It is well known that the divisor function satisfies $d(x) < 2\sqrt{x}$ for every x [31] which implies $\mathrm{P}\left(\mathrm{ord}_M(m)|u\right) < 2/\sqrt{\varphi(M)}$. Since we have $\varphi(M) > n^{2/3}$ for every $M > 30$ [31], we get (5). ◇

B Atomic Algorithms

Atomic Chain Computation. Looking at the chain computation algorithm, we observe that the main operations (namely operations on large registers) performed at each loop iteration are a division by two and possibly a substraction (depending on the value of τ_i). To render the algorithm atomic both operations must be performed at each loop iteration. The following algorithm describes the atomic version of the chain computation. It makes use of three registers: R_0, R_1 and R_2 which are used to store the values of α_i and β_i as well as a temporary value. It also uses three indices $i_\alpha, i_\beta, i_{tmp} \in \{0,1,2\}$ such that α_i is stored in R_{i_α}, β_i is stored in R_{i_β} and the temporary value is stored in $R_{i_{tmp}}$.

Algorithm 5 Atomic double addition chain computation

INPUT: A pair of natural integer (a, b) s.t. $a \leq b$

OUTPUT: The chain $\omega(a, b)$

```
1. $R_{i_\alpha} \leftarrow a$; $R_{i_\beta} \leftarrow b$; $j \leftarrow n^*$
2. while $(R_{i_\alpha}, R_{i_\beta}) \neq (0, 1)$ do
3.     $R_{i_{tmp}} \leftarrow R_{i_\beta} - R_{i_\alpha}$
4.     $v \leftarrow R_{i_\beta} \bmod 2$
5.     $R_{i_\beta} \leftarrow R_{i_\beta}/2$
6.     $t \leftarrow (R_{i_\beta} \leq R_{i_\alpha})$
7.     $\omega_{j-1} \leftarrow t$; $\omega_j \leftarrow t \vee v$
8.     $(i_\alpha, i_\beta, i_{tmp}) \leftarrow \big(t \wedge (i_{tmp}, i_\alpha, i_\beta)\big) \vee \big((t \oplus 1) \wedge (i_\alpha, i_\beta, i_{tmp})\big)$
9.     $j \leftarrow j - 1 - (t \oplus 1)$
10. end while
11. return $\omega$
```

Notations. In Step 6, the notation $t \leftarrow (R_{i_\beta} \leq R_{i_\alpha})$ is used to denote the operation that compares the two values in R_{i_β} and R_{i_α} and that returns the binary value t satisfying $t = 1$ if $R_{i_\beta} \leq R_{i_\alpha}$ and $t = 0$ otherwise. In Steps 8 and 9, the logical AND is extended to the $\{0, 1\} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ operator performing a logical AND between the left argument and each coordinate of the right argument.

Looking at Algorithm 5, we see that, at each loop iteration, the Boolean values t and v represent the values of τ_i and ν_i. One can verify that if $t = 0$ then these values are stored in (ω_{j-1}, ω_j) and j is decremented by two while if $t = 1$ then t is stored in ω_j and j is decremented by 1. Moreover, if $t = 0$ then Steps 8 and 9 have no effect while if $t = 1$ then Step 8 ensures that the indices of the different registers are permuted so that (α_i, β_i) is correctly updated.

Although Algorithm 5 requires three l-bit registers and a $(2.2\ l)$-bit buffer to store ω (see Sect. 7), its memory consumption can be reduced to $4.2\ l$ bits using the following trick. During the computation of the $1.2\ l$ high order bits of ω, the l low order bits allocated for ω are used as one of the three necessary l-bit registers. Once the $1.2\ l$ high order bits of ω have been computed, the intermediate values α_i and β_i have a bit-length lower than $l/2$. Therefore, the three registers can be allocated on less than $2l$ bits and the low order part of the buffer for ω can be freed.

Atomic Double Exponentiation. The following algorithm describes the atomic version of the double modular exponentiation. It makes use of two registers $R_{(0,0)}$ and $R_{(0,1)}$ that are used to store the intermediate results m^{a_i} and m^{b_i} and one more register $R_{(1,0)}$ to store m. It makes also use of two Boolean variables γ and μ. The Boolean γ indicates that m^{a_i} is stored in $R_{(0,\gamma\oplus1)}$ and that m^{b_i} is stored in $R_{(0,\gamma)}$. And the Boolean μ indicates wether the next modular multiplication is a multiplication by m ($\mu = 0$) or not ($\mu = 1$).

Algorithm 6 Atomic double modular exponentiation

Input: An element $m \in \mathbb{Z}_N$, a chain $\omega(a, b)$ s.t. $a \leq b$, a modulus N

Output: The pair of modular power $(m^a \bmod N, m^b \bmod N)$

```
1. $R_{(0,0)} \leftarrow 1$; $R_{(0,1)} \leftarrow m$; $R_{(1,0)} \leftarrow m$
2. $\gamma \leftarrow 1$; $\mu \leftarrow 1$; $i \leftarrow 0$
3. while $i < n$ do
4.     $t \leftarrow \omega_i \wedge \mu$; $v \leftarrow \omega_{i+1} \wedge \mu$
5.     $R_{(0,\gamma \oplus t)} \leftarrow R_{(0,\gamma \oplus t)} \cdot R_{((\mu \oplus 1),\gamma \wedge \mu)} \bmod N$
6.     $\mu \leftarrow t \vee (v \oplus 1)$; $\gamma \leftarrow \gamma \oplus t$
7.     $i \leftarrow i + \mu + \mu \wedge (t \oplus 1)$
8. end while
9. return $(R_{\gamma \oplus 1}, R_\gamma)$
```

While $\mu = 1$, the Boolean t is evaluated to τ_i and, if $\tau_i = 1$, the Boolean v is evaluated to ν_i. Then, while $t = 1$ or $v = 0$ each loop iteration corresponds to a step performing one single multiplication which is done in Step 5. If $t = 0$ and $\nu = 1$, the step must perform two multiplications: $R_{(0,\gamma)}$ by $R_{(0,\gamma)}$ and $R_{(0,\gamma)}$ by $R_{(1,0)}$. The first one is performed in Step 5 afterward the Boolean μ is evaluated to 0 thus indicating that the next loop must perform the multiplication by $R_{(1,0)}$. In that case, i is not incremented and the next loop iteration performs the desired multiplication before evaluating μ to 1 and normally carrying on the computation.