Algorithmic Countermeasures Against Fault Attacks and Power Analysis for RSA-CRT

Ágnes Kiss, Juliane Krämer, Pablo Rauzy, Jean-Pierre Seifert

Algorithmic Countermeasures Against Fault Attacks and Power Analysis for RSA-CRT

Ágnes Kiss[1], Juliane Krämer[1,2], Pablo Rauzy[3], and Jean-Pierre Seifert[2]

[1] TU Darmstadt, Darmstadt, Germany
agnes.kiss@crisp-da.de, jkraemer@cdc.informatik.tu-darmstadt.de
[2] TU Berlin, Berlin, Germany
{juliane, jpseifert}@sec.t-labs.tu-berlin.de
[3] Inria, CITI lab, France
pablo.rauzy@inria.fr

Abstract. In this work, we analyze all existing RSA-CRT countermeasures against the Bellcore attack that use binary self-secure exponentiation algorithms. We test their security against a powerful adversary by simulating fault injections in a fault model that includes random, zeroing, and skipping faults at all possible fault locations. We find that most of the countermeasures are vulnerable and do not provide sufficient security against all attacks in this fault model. After investigating how additional measures can be included to counter all possible fault injections, we present three countermeasures which prevent both power analysis and many kinds of fault attacks.

Keywords: Bellcore attack, RSA-CRT, modular exponentiation, power analysis

1 Introduction

In a fault attack, an adversary is able to induce errors into the computation of a cryptographic algorithm and thereby to gain information about the secret key or other secret information used in the algorithm. The first fault attack [4] targets an RSA implementation using the Chinese remainder theorem, RSA-CRT, and is known as the Bellcore attack. The Bellcore attack aroused great interest and led to many publications about fault attacks on RSA-CRT,e.g., [1, 6, 9, 11, 22]. Countermeasures to prevent the Bellcore attack can be categorized into two families: the first one relies on a modification of the RSA modulus and the second one uses self-secure exponentiation. The countermeasures in the first family were recently analyzed [21], and a formal proof of their (in)security was provided.

We complement the work of [21] by comprehensively analyzing the countermeasures in the second family, i.e., those based on self-secure exponentiation. These countermeasures use specific algorithms that include redundancy within the exponentiations. The first such method is based on the Montgomery ladder [9]. This was adapted to the right-to-left version of the square-and-multiply-always algorithm [5,6] and to double exponentiation [18,22]. We test the security of these methods using an automated testing framework. We use the same fault

model as in [21], but extend it to meet the particularities of self-secure exponentiation algorithms. We reveal that the countermeasures have certain vulnerabilities in this extended fault model. Based on these findings, we improve the countermeasures and present three self-secure exponentiation methods that are secure against fault injections, safe-error attacks, and power analyses. We note that non-algorithmic level countermeasures are not in the scope of this paper.

Our Contribution: In this paper, we ***test the security*** of the self-secure exponentiation countermeasures ***against the Bellcore attack*** by simulating random, zeroing, and skipping faults at all possible fault locations (Section 4). Thereafter, we ***propose secure countermeasures***, step-by-step achieving protection against all fault injections and resistance to power analysis and safe-error attacks. We present one countermeasure for each of the exponentiation algorithms used as self-secure exponentiation: the *Montgomery ladder*, the *square-and-multiply-always* algorithm, and the *double exponentiation* method. Despite the natural overhead caused by the included measures against all the considered attack types, ***our algorithms remain highly efficient*** (Section 5).

2 Background

In this section, we give the necessary background information for our work.

2.1 The Bellcore Attack on RSA-CRT

We use the standard notation for RSA [23]: M denotes the message, $N = pq$ the public modulus with secret primes p and q, $\varphi(N) = (p-1)(q-1)$. The public exponent e with $\gcd(e, \varphi(N)) = 1$ is chosen along with the secret exponent d, where $e \cdot d \equiv 1 \bmod \varphi(N)$. The signature is calculated $S = M^d \bmod N$, and $S^e \equiv (M^d)^e \equiv M \bmod N$. The calculation can be speeded up by a factor of four using the RSA-CRT implementation [20]. Two smaller exponentiations $S_p = M^{d_p} \bmod p$ and $S_q = M^{d_q} \bmod q$ are performed with exponents $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, and recombined with the method $S = \mathrm{CRT}(S_p, S_q) = ((S_p - S_q) i_q \bmod p) q + S_q$, where $i_q = q^{-1} \bmod p$. The public key of RSA-CRT is (e, N) and the private key includes p, q, d_p, d_q and i_q.

A *fault attack* is a physical attack where the attacker is able to induce faults into the execution of the algorithm. The first attack on RSA-CRT was proposed by Bellcore researchers [4]. The fault is induced into the calculation of strictly one of the intermediate signatures, resulting in $\widehat{S_p}$ (or $\widehat{S_q}$). If $\widehat{S_p}$ (or $\widehat{S_q}$) is used during recombination, a faulty signature $\widehat{S}$ is returned. With high probability q (or p) can be deduced as $\gcd(S - \widehat{S}, N)$ [4] or as $\gcd(\widehat{S}^e - M \bmod N, N)$ [11].

During the discussion of fault attacks, the precise description of the *fault model* is essential: it includes the assumptions on the adversary's abilities. The Bellcore attack targeting an unprotected implementation uses one fault injection and loose assumptions in the fault model, i.e., a very weak attacker. The attacker is only assumed to alter an intermediate signature, which can be achieved by an arbitrary modification of any variable throughout the exponentiation, i.e., affecting any bit or any byte results in a successful attack.

2.2 Safe-Error Attacks

Classical fault attacks exploit the corrupted result or the difference between a correct and faulty results. However, it was noted in [26] that secret information may leak depending on if a fault has effect on the result of the computation or not. The techniques that exploit such behavior are called safe-error (SE) attacks.

Computational safe-error attacks (C-SE) [27] target dummy operations. If the result remains error-free although a fault was induced, it affects a dummy operation and thus, information about the secret key can be revealed.

Memory safe-error attacks (M-SE) [26] assume a more powerful attacker. Knowing how the internal variables are processed in the memory throughout a certain step of the algorithm, one may be able to derive the secret key [26]. Memory safe-error attacks are prevented by randomizing the targeted variables.

2.3 Power Analysis Methods

Simple power analysis (SPA) studies the power consumption of a single execution of the algorithm. If the execution depends on the value of the secret key, the adversary is able to obtain information by analyzing the power trace.

Differential power analysis (DPA) is a natural extension of SPA [16]. When performing a DPA, an attacker collects several power trace measurements of the executions of the same algorithm and uses statistical methods to reveal the secret key. Prevention generally requires randomization of variables.

2.4 Algorithms for Regular Exponentiation

Classical modular exponentiation algorithms are vulnerable to SPA, since the power consumption of the different operations can be differentiated [17]. To prevent SPA, regularity of the modular exponentiation algorithms is required. It means that the same operations are performed independently from the value of the exponent. Below, we recapitulate the two most widely used binary methods.

Square-and-Multiply-Always: The right-to-left exponentiation algorithm was modified in [7] to the square-and-multiply-always method, shown in Alg. 1a.

Algorithm 1 SPA-resistant modular exponentiation methods

(1a) Square-and-multiply-always [7]

input: $M \neq 0$, $d = (d_{n-1}, \ldots, d_0)_2$, x
output: $M^d \bmod x$

1: $R_0 := 1, R_1 := 1, R_2 := M$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_2 \bmod x$
4: $\quad R_2 := R_2^2 \bmod x$
5: **end for**
6: **return** R_0

(1b) Montgomery ladder [13]

input: $M \neq 0$, $d = (d_{n-1}, \ldots, d_0)_2$, x
output: $M^d \bmod x$

1: $R_0 := 1, R_1 := M$
2: **for** $i = n-1$ **to** 0 **do**
3: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_{d_i} \bmod x$
4: $\quad R_{d_i} := R_{d_i}^2 \bmod x$
5: **end for**
6: **return** R_0

By introducing dummy operations in register R_1 (line 3), one squaring and one multiplication is performed at each iteration.

Montgomery Ladder: The powering ladder, shown in Alg. 1b, was proposed in [19] and its correctness discussed in [13]. The algorithm is regular without including dummy operations and is resistant to safe-error attacks [13].

3 Countermeasures Against the Bellcore Attack

To counter the Bellcore attack, *straightforward countermeasures* aim to verify the integrity of the computation before returning the result, e.g., by repeating the computation and comparing the results. Due to the inefficiency of such measures, several improved countermeasures appeared starting from 1999.

3.1 Two Families of Countermeasures

The advanced countermeasures were divided into two families according to the difference in their nature [21]: *Shamir's family* and Giraud's family. We refer to the latter as *self-secure exponentiation countermeasures.*

Shamir's family consists of the countermeasures that prevent the Bellcore attack by multiplicatively extending the modulus x with a random number s. They rely on the fact that an invariant, inherited from the calculations modulo the extended modulus, i.e., modulo $x \cdot s$, must hold modulo s. Shamir's algorithm from [24] motivated researchers to develop such countermeasures, e.g., [1,12,21].

The idea of **self-secure exponentiation countermeasures** was proposed in [9]. If the exponentiation algorithm returns more than one power of a given input and keeps a *coherence* between its registers throughout the exponentiation, an invariant can be formulated that must hold at the end of the algorithm. However, it is claimed to be lost if a fault injection takes place.

3.2 Self-Secure Exponentiation Countermeasures

In this section, we recapitulate the existing self-secure exponentiation countermeasures. The algorithmsare provided in Appendix A in Alg. 5–10.

The first countermeasure was proposed by **Giraud** [9]. It exploits the fact that while using the *Montgomery ladder*, the temporary registers R_0 and R_1 are of the form ($M^{k-1} \bmod x$, $M^k \bmod x$) for some integer k after each iteration of Alg. 1b. After two exponentiations that result in the pairs ($S'_p = M^{d_p-1} \bmod p$, $S_p = M^{d_p} \bmod p$) and ($S'_q = M^{d_q-1} \bmod q$, $S_q = M^{d_q} \bmod q$), and two recombinations $S' = \mathrm{CRT}(S'_p, S'_q) = M^{d-1} \bmod pq$ and $S = \mathrm{CRT}(S_p, S_q) = S^d \bmod pq$, the invariant $M \cdot S' \equiv S \bmod pq$ holds. Giraud claims that in case of a fault attack within the exponentiation, the coherence is lost for S_p, S'_p (or S_q, S'_q) and thus for S and S'. Despite its advantages, the Montgomery ladder exponentiation remains vulnerable to DPA [16] ($\mathrm{DPA}^{\mathrm{exp}}$). **Fumaroli and Vigilant** blinded the base element with a small random number r [8], using one more register R_2 in the exponentiation. Besides being more memory-costly, this method was proven

Table 1: Self-secure exponentiation countermeasures. CRT, check, inv., reg., mult., and sq. denote the number of CRT recombinations, checking procedures, inversions, additional large registers, multiplications, and squaring operations respectively, in terms of the bit-length n of the exponent. PA and SE denote the resistance against power analysis and safe-error attacks. ✓ means that there are included countermeasures, × refers to the lack of them.

Countermeasure			**Efficiency criteria**						**Physical attacks**				
Author(s)	Ref.		CRT	Check	Inv.	Reg.	Mult.	Sq.	PA			SE	
	Ref.	Alg.	Total			Per exp.			SPA	exp DPA	CRT DPA	C-SE	M-SE
Giraud	[9]	5	2	4	0	3	n	n	✓	×	✓	✓	✓
Fumaroli,Vigilant	[8]	6	2	4	$2^{(p,q)}$	4	$n+3$	$2n$	✓	✓	×	✓	✓
Boscher et al.'07	[6]	7	3	5	0	4	n	n	✓	×	×	✓	×
Boscher et al.'09	[5]	7	3	5	$1^{(pq)}$	4	$n+2$	n	✓	✓	×	✓	✓
Rivain	[22]	8	1	2	0	2	$\sim 1.65n$		×	×	×	✓	×
Rivain (SCA)	[22]	9	1	2	0	3	$\sim 1.65n$	0	✓	✓	×	✓	✓
Le et al.	[18]	10	1	2	0	3	$\sim 0.67n$	n	×	×	×	✓	×

to be insecure against fault attacks [14], due to the lack of coherence between R_2 and the other registers. Moreover, it remains vulnerable to the DPA attack on the CRT recombination from [25] ($\mathrm{DPA}^{\mathrm{CRT}}$).

The *square-and-multiply-always algorithm* (Alg. 1a), uses dummy operations to prevent SPA. **Boscher et al.** in 2007 proposed a self-secure exponentiation countermeasure based on this algorithm [6]. In the end of the execution, R_0 holds the value $M^d \mod x$, R_1 holds $M^{2^n-d-1} \bmod x$, while R_2 only depends on the binary length n of the exponent, and equals to $M^{2^n} \bmod x$. Thus, the coherence $M \cdot R_0 \cdot R_1 \equiv R_2 \bmod x$ is kept throughout the algorithm. Boscher et al. in 2009 [5], modified the method in order to achieve resistance against DPA on the exponentiation without significant overhead. 2^w-ary versions of the algorithmwere proposed [2, 10].

Rivain proposed a solution that uses *double exponentiation* [22]. Such a method receives the base M, two exponents d_1, d_2, the modulus x, and outputs both $M^{d_1} \bmod x$ and $M^{d_2} \bmod x$. It makes use of a double addition chain for the pair (d_1, d_2), by means of which the two modular exponentiations are performed at once, using altogether $1.65n$ operations on average. We assume this chain to be precomputed. **Le et al.** presented a double exponentiation algorithm, that does not rely on precomputation [18]. The binary exponentiation works as two parallel executions of the right-to-left exponentiation and uses register R_0 for calculations with d_1 and register R_1 for calculations with d_2. $M^{2^n} \bmod x$ is computed only once and is stored in R_2.

Table 1 summarizes the different properties of the self-secure exponentiation countermeasures. We consider the security and efficiency of the methods, since measures against physical attacks imply overhead. When discussing efficiency, we describe the following relevant properties to achieve low time and memory con-

sumption: number of registers containing large values that are used additionally to the input *registers* (M, d, x) during the exponentiation, number of *multiplications*, *squaring operations* and *inversions* using large registers. We summarize if they include protection against physical attacks such as *power analysis* on the exponentiation and the CRT recombination and *safe-error attacks*.

4 Security of Self-Secure Exponentiation Methods

The security of self-secure exponentiation countermeasures relies mainly on the *exponentiation algorithms*. Each method has an invariant that holds throughout its execution, which is claimed to be lost in case a fault is injected. Accordingly, the modular exponentiation methods have to be tested against fault attacks. In this section, we recapitulate the fault model that we adopt, briefly describe our methodology and discuss our results.

4.1 Simulating Fault Injections Against Self-Secure Exponentiation Countermeasures

The designers of the countermeasures provide either formal and informal explanations for their security assumptions and their fault models differ from each other. To the best of our knowledge, we are the first to simulate all possible fault injections in a common fault model.

Fault Model: We adopt the generic fault model of [21]. Therefore, we simulate three types of fault injections: *random* and *zeroing faults* in case of which the affected variable is changed to a random value and null, respectively, and *skipping faults* which cause instruction skips, i.e., jumps over some lines of the pseudocode. We take the following fault types into consideration: faults on local variables, on input parameters, and on conditional tests. An adversary is able to target any variable, but cannot specify the bits his fault affects. When inducing a random fault, he does not know its concrete value. Since refined methods appear for performing instruction skips in practice (e.g. [3]), we consider it as a possible threat when discussing physical attacks. The injection of skipping faults was observed as practical in [21], but was covered by means of random and zeroing faults. This does not hold for self-secure exponentiation. When considering skipping faults, we count the number of lines that have to be skipped in the pseudocode. In the Montgomery ladder shown in Alg. 1b, the pair (R_0, R_1) is of the form $(M^{k-1} \bmod x, M^k \bmod x)$ at each iteration, which coherence is assumed to be lost in case of a fault injection. However, an adversary might skip two consecutive lines (3-4) at any iteration of the loop. The invariant holds for the corrupted $\widehat{R_0}$ and $\widehat{R_1}$ and thus, the fault is not detected.

Our Framework: In case of self-secure exponentiation countermeasures, the underlying *exponentiation algorithm* has to be tested and checked that the invariant is lost if a fault is injected. When simulating the attacks, we needed features that the tool used for the analysis of Shamir's family lacked [21]: redefinition of variables and support for loops. Therefore, we created our own

Table 2: Results of our fault injection tests on the exponentiation algorithms, assuming that the checking procedures are protected We note that we rely on the original fault models of the countermeasures from column Ref., recapitulated in Appendix A. ✓ denotes that our tests did not reveal any vulnerability against the fault type, M and d_1, d_2 denote the vulnerability of the message and the exponents in the exponentiation algorithm, respectively. When considering skipping faults, we indicate which lines are skipped together to achieve a successful attack. The register numbering R_i, $i \in \{0, 1, 2\}$ and the lines are according to the algorithms in column Alg.

Countermeasure			**Fault injection attacks**				
Author(s)	Ref.	Alg.	Random	Zeroing		Skipping	
Fault number			1	1	2	1	2
Giraud	[9]	5	✓	M, R_0, R_1		✓	(4-5)
Fumaroli, Vigilant	[8]	6	R_2	M, R_0, R_1, R_2		(7)	(5-6) or 2·(7)
Boscher et al.'07	[6]	7	✓	✓	✓	✓	(6-7)
Boscher et al.'09	[5]	7	✓	✓	✓	✓	(6-7)
Rivain	[22]	8	M	✓	✓	✓	✓
Rivain (SCA)	[22]	9	M	✓	✓	✓	✓
Le et al.	[18]	10	M	✓	d_1, d_2	✓	✓

framework in Java. A manual step of our method was identifying the possible fault injection locations within the exponentiation algorithms. After this manual step, the simulation of multiple fault injections in all possible combination of fault locations was fully automated, for all the three fault types. A simulation results in a successful Bellcore attack if a corrupted signature is returned. For more details on our simulation framework, the reader is referred to the full version [15].

4.2 Simulation Results

The results of our fault injection simulations are shown in Table 2. While performing the tests with multiple faults, we considered protected checking procedures, since skipping or zeroing any of the checks would enable a successful Bellcore attack. When considering faults on the checking procedures, a method can be protected against n fault injections by repeating each check n times.

Random Faults: If a countermeasure is protected against one random fault injection, it cannot be broken with more than one random faults either. This is due to the fact that a random fault cannot induce a verification skip [21]. Our results confirm that in case of the algorithms that use the *Montgomery ladder* or the *square-and-multiply-always algorithm*, the intermediate secret exponent and the loop counter have to be protected against random faults. [6, 8, 9] use the *checksum* of the exponent to verify its integrity and thwart the attack. It was revealed in [14], that the introduction of **register $\mathbf{R_2}$** in Fumaroli and Vigilant's countermeasure [8] made it vulnerable to any random fault on R_2 at any iteration

of the algorithm. This is due to the fact that R_2 is calculated independently of the other two registers, which are multiplied with its final value. In case of the countermeasures using *double exponentiation*, a possible random fault is the corruption of the intermediate **message M**, resulting in $\widehat{M}$. Rivain identified this vulnerability and suggested to compute a cyclic redundancy code [22].

Zeroing Faults: Without a specific checking procedure against zeroing faults, the exponentiation algorithms (Section 2.4) are vulnerable. According to [9], it is unlikely to be able to zero a large buffer in practice. However, as [6,21], we take zeroing faults into consideration but note that their injection is very difficult to achieve in practice. In case of the methods that use the *Montgomery ladder* and the *square-and-multiply-always exponentiation*, if the **message M** in the beginning of the algorithms is zeroed, zeroes are returned. The same holds for any of the **registers** $\mathbf{R_0}, \mathbf{R_1}$ in the method using the Montgomery ladder and for $\mathbf{R_2}$ in Fumaroli and Vigilant's and Boscher et al.'s methods. Then, the checking procedure holds even though the recombination is computed with only one of the intermediate signatures. Giraud considered this vulnerability impossible, while Boscher et al. included checks against it. The two countermeasures that use *double exponentiation* are not vulnerable to a single zeroing fault. In the case of Rivain's method [22], the exponent is given by the addition chain, which we assume to be protected. For the algorithm by Le et al. [18], two zeroing faults on the **exponents** $\mathbf{d_1}, \mathbf{d_2}$ are necessary to conduct a Bellcore attack. If any other values are zeroed, the coherence check does not hold and the fault is detected.

Skipping Faults: Our simulations show that only the method by Fumaroli and Vigilant [8] is vulnerable to the instruction skip of **one line**, the calculation of register R_2, which has a similar effect as the random fault on R_2. When two lines are skipped together, both regular, SPA-resistant algorithms, i.e., the *Montgomery ladder* and the *square-and-multiply-always* methods are vulnerable. By skipping **two consecutive lines within the loop**, they preserve the coherence between the variables even though the results are corrupted. Even if the loop counter i is protected, skipping faults result in successful Bellcore attacks.

5 PA-SE-FA-Resistant Self-Secure Exponentiation Countermeasures

We propose a secure countermeasure for each of the exponentiation algorithms that are used for constructing self-secure exponentiation methods. We claim that our proposed countermeasures are secure against *power analysis* (PA), *safe-error* (SE) attacks, and *fault attacks* (FA) and remain highly efficient. For the verification of the resistance against fault injection attacks, we applied our framework from Section 4.1 on the proposed algorithms. We discuss the implied overhead by the introduced protection against physical attacks. FA_i^j denotes fault attacks of type j (r, z, s denote random, zeroing and skipping faults, resp.), against variable(s) i.

Algorithm 2 PA-SE-FA method with the Montgomery ladder

(2a) MonExp$(M, d, x, r, r_{\text{inv}}, s)$

input: M,$d=(d_{n-1},\ldots,d_0)_2$, x, r, r_{inv}, s
output: $(r^{2^n} \cdot M^d \bmod sx$, $r^{2^n} \cdot M^{d+1} \bmod sx$, $r_{\text{inv}}^{2^n} \bmod sx)$

1: $x := s \cdot x$ ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{\text{r, z}}_{d,i}$
2: $R_0 := r$
3: $R_1 := r \cdot M \bmod x$
4: $R_2 := r_{\text{inv}} \bmod x$
5: **for** i from $n-1$ to 0 **do**
6: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_{d_i} \bmod x$
7: $\quad R_{d_i} := R_{d_i}^2 \bmod x$
8: $\quad R_2 := R_2^2 \bmod x$
9: **end for**
10: **return** (R_0, R_1, R_2)

(2b) RSA-CRT

input: $M \neq 0, p, q, d_p, d_q, i_q$, $D = p \oplus q \oplus d_p \oplus d_q \oplus i_q$
output: $M^d \bmod pq$ or error

1: Pick k-bit random prime s, such that $ps \nmid M, qs \nmid M$ ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{\text{r, z}}_{d,i}$
2: Pick random integer $r \in \mathbb{Z}^*_{pqs}$ ▷ $\text{FA}^{\text{r}}_{R_2}$, $\text{FA}^{\text{s}}_{(8)}$
3: $r_{\text{inv}} := r^{-1} \bmod pqs$ ▷ $\text{FA}^{\text{r}}_{R_2}$, $\text{FA}^{\text{s}}_{(8)}$
4: $(S_p, S'_p, R_p) :=$MonExp$(M \bmod sp, d_p, p, r, r_{\text{inv}}, s)$
5: $(S_q, S'_q, R_q) :=$MonExp$(M \bmod sq, d_q, q, r, r_{\text{inv}}, s)$
6: **if** $S_p \cdot S_q = 0$ **then** ▷ $\text{FA}^{\text{z}}_{M,R_0,R_1,R_2}$
7: $\quad$ **return** error
8: **end if**
9: $S := \text{CRT}_{\text{blinded}}(S_p, S_q)$ ▷ DPA_{CRT}
10: $S' := \text{CRT}_{\text{blinded}}(S'_p, S'_q)$ ▷ DPA_{CRT}
11: $R := \text{CRT}_{\text{blinded}}(R_p, R_q)$ ▷ $\text{FA}^{\text{r}}_{R_2}$, $\text{FA}^{\text{s}}_{(8)}$
12: $S := R \cdot S \bmod pq$ ▷ $\text{FA}^{\text{r}}_{R_2}$, $\text{FA}^{\text{s}}_{(8)}$
13: **if** $M \cdot S \not\equiv R \cdot S' \bmod pq$ **then**
14: $\quad$ **return** error
15: **end if**
16: $S_{ps} = (S_p \bmod s)^{d_q \bmod (s-1)} \bmod s$
17: $S_{qs} = (S_q \bmod s)^{d_p \bmod (s-1)} \bmod s$
18: **if** $S_{ps} \neq S_{qs}$ **then**
19: $\quad$ **return** error ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{\text{r, z}}_{d,i}$
20: **end if**
21: **if** $p \oplus q \oplus d_p \oplus d_q \oplus i_q \neq D$ **then**
22: $\quad$ **return** error ▷ $\text{FA}^{\text{r, z}}_{p,q,i_q,d_p,d_q}$
23: **end if**
24: **return** S

5.1 Countermeasure using the Montgomery Ladder

Fumaroli and Vigilant's countermeasure [8] (Alg. 6) which aimed to improve Giraud's method [9] (Alg. 5) was proven to be vulnerable to random fault attacks [14]. Alg. 2 presents our secure method with the ***Montgomery ladder***.

To prevent fault attacks on **register $\mathbf{R_2}$** ($\mathbf{FA^r_{R_2}}$, $\mathbf{FA^s_{(8)}}$), we return the blinded registers R_0 and R_1 and perform the multiplication with the inverse contained in R_2. This multiplication happens modulo pq, after the blinded CRT recombinations of all the three registers in lines 9–11 in Alg. 2b.

To achieve prevention against **skipping faults** ($\mathbf{FA^s_{(6\text{-}7)}}$), we include a check for verifying the integrity of the exponentiations. Since the coherence in the regular exponentiation algorithms is not lost when skipping faults are injected, we create a hybrid countermeasure with a technique used in Shamir's family

by Aumüller et al. [1]. We conclude the necessity of the modulus extension to prevent skipping faults and multiply the modulus with a k-bit random prime s. S_p and S_q are calculated modulo $p \cdot s$ and $q \cdot s$, respectively, and the signature is recombined to $S = M^d \bmod pq$ using the blinded recombination from [9]:

$$S = \mathrm{CRT}_{\mathrm{blinded}}(S_p, S_q) = (((S_p - S_q) \bmod sp) \cdot i_q \bmod sp) \cdot q + S_q \bmod pq. \quad (1)$$

To verify that no instruction was skipped, two small exponentiations modulo the k-bit number s with the k-bit exponents are performed as in lines 16–17. If a skipping fault occurs and the value of S_p or S_q is corrupted, the check in line 18 does not hold with probability 2^{-k}. Besides protecting against skipping faults, this measure detects faults on the **exponent** and **loop counter i** ($\mathbf{FA^{r,z}_{d,i}}$) of the exponentiation algorithm, without an additional large register. If the small exponentiations are calculated using the Montgomery ladder (Alg. 1b), then besides the k-bit message, exponent, and modulus, two k-bit registers, k multiplications and squarings are used. However, a checksum as an input has to be included to detect the corruption of p, q, i_q, d_p or d_q in Alg. 2b in line 21.

We note that the blinded CRT recombination recapitulated in Eq. 1 also prevents the **DPA** attack on the CRT recombination ($\mathbf{DPA_{CRT}}$) from [25].

To avoid **zeroing faults** ($\mathbf{FA^{z}_{M,R_0,R_1,R_2}}$), we check that none of the values returned by the exponentiation is zero. We perform this before the CRT recombinations in Alg. 2b, by verifying $S_p \cdot S_q \neq 0$ in line 6. In order to make sure that this check does not violate the correctness of the algorithm when the message is a multiple of ps or qs, we choose s such that $ps \nmid M$ and $qs \nmid M$.

Alg. 2 presents the algorithm that is based on the Montgomery ladder and is protected against power analysis (PA), safe-error (SE), and fault attacks (FA). For eliminating the revealed vulnerabilities against fault injection attacks, we included an additional CRT recombination, transformed two small inversions to one of doubled size, included one large input register D, two times k multiplications and k squaring operations on k-bit registers, where k is the security parameter that defines the probability of undetected skipping faults as 2^{-k}. We note that since modular inversion and prime generation imply significant costs, lines (1-3) can be precomputed (without the assumption $ps \nmid M, qs \nmid M$) and s, r and r_{inv} can be provided as inputs to Alg. 2b.

5.2 Countermeasure using the Square-and-Multiply-Always Exp.

Boscher et al. described a ***square-and-multiply always algorithm*** that is resistant to SPA, DPA, and SE [5] (Alg. 7). The algorithm includes a technique against the exponent modification, and the check $R_2 \neq 0$ in the end of the exponentiation to detect **zeroing faults** ($\mathbf{FA^{z}_{M,R_2}}$) [6]. Instead of this check in both exponentiations, we suggest to verify $S_p \cdot S_q \neq 0$ in Alg. 3b as in Alg. 2b.

Against **skipping faults** ($\mathbf{FA^{s}_{(6-7)}}$) we suggest the same measure as in Alg. 2: blinding the modulus and performing two small exponentiations in the RSA-CRT algorithm. For retrieving the signature, the CRT recombination in Eq. 1 is used. Though not mentioned in [5], the random value r in Alg. 3b should not be

Algorithm 3 PA-SE-FA method with the square-and-multiply-always exp.

(3a) SqExp$(M, d, x, r, r_{\text{inv}}, s)$

input: M,$d=(d_{n-1},\ldots,d_0)_2$, x, r, r_{inv}, s
output: $(r \cdot M^d \bmod sx$, $r_{\text{inv}} \cdot M^{2^n-d-1} \bmod sx$, $M^{2^n} \bmod sx)$

1: $x := s \cdot x$ ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{r,z}_{d,i}$
2: $R_0 := r$
3: $R_1 := r_{\text{inv}}$
4: $R_2 := M$
5: **for** i from 0 to $n-1$ **do**
6: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_2 \bmod x$
7: $\quad R_2 := R_2^2 \bmod x$
8: **end for**
9: **return** (R_0, R_1, R_2)

(3b) RSA-CRT

input: $M \neq 0, p, q, d_p, d_q, i_q$, $D = p \oplus q \oplus d_p \oplus d_q \oplus i_q$
output: $M^d \bmod pq$ or error

1: Pick k-bit random prime s such that $ps \nmid M, qs \nmid M$ ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{r,z}_{d,i}$
2: Pick random integer $r \in \mathbb{Z}^*_{pqs}$ ▷ $\text{FA}^{\text{r}}_{R_2}$, $\text{FA}^{\text{s}}_{(8)}$
3: $r_{\text{inv}} := r^{-1} \bmod pqs$
4: $(S_p, S'_p, T_p) := \text{SqExp}(M \bmod sp, d_p, p, r, r_{\text{inv}}, s)$
5: $(S_q, S'_q, T_q) := \text{SqExp}(M \bmod sq, d_q, q, r, r_{\text{inv}}, s)$
6: **if** $S_p \cdot S_q = 0$ **then** ▷ $\text{FA}^{\text{z}}_{M,R_2}$
7: $\quad$ **return** error
8: **end if**
9: $S := \text{CRT}_{\text{blinded}}(S_p, S_q)$
10: $S' := \text{CRT}_{\text{blinded}}(S'_p, S'_q)$
11: $T := \text{CRT}_{\text{blinded}}(T_p, T_q)$
12: **if** $M \cdot S \cdot S' \not\equiv T \bmod pq$ **then**
13: $\quad$ **return** error
14: **end if**
15: $S_{ps} = (r_{\text{inv}} S_p \bmod s)^{d_q \bmod (s-1)} \bmod s$
16: $S_{qs} = (r_{\text{inv}} S_q \bmod s)^{d_p \bmod (s-1)} \bmod s$
17: **if** $S_{ps} \neq S_{qs}$ **then**
18: $\quad$ **return** error ▷ $\text{FA}^{\text{s}}_{(6-7)}$, $\text{FA}^{r,z}_{d,i}$
19: **end if**
20: **if** $p \oplus q \oplus d_p \oplus d_q \oplus i_q \neq D$ **then**
21: $\quad$ **return** error ▷ $\text{FA}^{\text{r, z}}_{p,q,i_q,d_p,d_q}$
22: **end if**
23: **return** $r_{\text{inv}} \cdot S \bmod pq$

too small to avoid the following SPA during the computation of Alg. 3a: if an adversary is allowed to input the message $M = 1$, the value of register R_2 remains 1 for the whole computation. Therefore, the multiplication in line 6 would only depend on the bits of the secret exponent d, multiplied either with a small number (r) or with a large number (r_{inv}). This could result in differences in the power consumption trace and therefore we chose r to be an at least $(n+k)$-bit integer, where n is the bitlength of p and of q, since it is used for operations of that size in Alg 3a.

Our PA-SE-FA-resistant algorithm with the square-and-multiply-always exponentiation is depicted in Alg. 3. To eliminate the identified vulnerabilities, we included one large input register D along with two times k multiplications and k squaring operations on k-bit registers, in a similar manner as in Alg. 2.

Algorithm 4 PA-SE-FA method with double exponentiation

(4a) DoubleExp(M, d_1, d_2, x, s)

input: $M \neq 0$,
$d_1 = (d_{1,n-1}, \ldots, d_{1,0})_2$,
$d_2 = (d_{2,n-1}, \ldots, d_{2,0})_2, x, s$
output: $(M^{d_1} \bmod xs$,
$M^{d_2} \bmod xs)$

1: $x := s \cdot x$ ▷ DPA_{CRT}
2: $R_{(0,1)} := 1$ ▷ SPA
3: $R_{(1,1)} := 1$ ▷ SPA
4: $R_{(0,2)} := 1$ ▷ SPA
5: $R_{(1,2)} := 1$ ▷ SPA
6: $R_2 := M$
7: **for** $i = 0$ **to** $n-1$ **do** ▷ SPA
8: $R_{(\overline{d_{1,i}},1)} := R_{(\overline{d_{1,i}},1)} \cdot R_2 \bmod x$
9: $R_{(\overline{d_{2,i}},2)} := R_{(\overline{d_{2,i}},2)} \cdot R_2 \bmod x$
10: $R_2 := R_2^2 \bmod x$
11: **end for**
12: **if** $R_{(0,1)}R_{(1,1)} \not\equiv R_{(0,2)}R_{(1,2)} \bmod x$ **then** ▷ C SE
13: **return** error
14: **end if**
15: **return** $(R_{(0,1)}, R_{(0,2)})$

(4b) RSA-CRT

input: M, p, q, d_p, d_q, i_q
output: $M^d \bmod pq$ or error

1: Pick small $r_1, r_2 \in \mathbb{Z}$ $r_2 \geq r_1 + 2$
2: Pick k-bit random prime s
3: $(S_p, c_p) :=$ ▷ $\text{DPA}, \text{M-SE}, \text{FA}^{\text{r}}_M, \text{FA}^{\text{z}}_{(d_1,d_2)}$
DoubleExp$(M \bmod p, d_p{+}r_1(p-1), r_2(p-1) - d_p{-}1, p, s)$
4: $(S_q, c_q) :=$ ▷ $\text{DPA}, \text{M-SE}, \text{FA}^{\text{r}}_M, \text{FA}^{\text{z}}_{(d_1,d_2)}$
DoubleExp$(M \bmod q, d_q{+}r_1(q-1), r_2(q-1) - d_q{-}1, q, s)$
5: $S := \text{CRT}_{\text{blinded}}(S_p, S_q)$ ▷ DPA_{CRT}
6: **if** $M \cdot S \cdot c_p \not\equiv 1 \bmod p$ **then**
7: **return** error ▷ $\text{FA}^{\text{r}}_M, \text{FA}^{\text{z}}_{(d_1,d_2)}$
8: **end if**
9: **if** $M \cdot S \cdot c_q \not\equiv 1 \bmod q$ **then**
10: **return** error ▷ $\text{FA}^{\text{r}}_M, \text{FA}^{\text{z}}_{(d_1,d_2)}$
11: **end if**
12: **return** $S \bmod pq$

5.3 Countermeasure using Double Exponentiation

Rivain proposed the first countermeasure that uses ***double exponentiation*** [22] (Alg. 8). He included modifications by means of which it becomes SPA-DPA-SE-resistant, still requiring the precomputation of the addition chain (Alg. 9). Our aim is to consider measures in the insecure but more efficient algorithm by Le et al. [18] (Alg. 10), which does not include precomputation but ignores protection against PA and SE.

Firstly, we transform the algorithm to become resistant to **SPA**. We use two additional registers with dummy operations in order to achieve regularity. Thus, the algorithm requires the use of altogether 5 registers: $R_{(0,1)}$ and $R_{(1,1)}$ belonging to exponent d_1, $R_{(0,2)}$ and $R_{(1,2)}$ belonging to exponent d_2, and R_2 used as before. Since for every bit of the exponents the same operations have to performed, this results in altogether $2n$ multiplications and n squaring operations.

Introducing regularity includes dummy operations. Registers $R_{(1,1)}$ and $R_{(1,2)}$ are unused and thus all the multiplications that assign values to them are dummy operations. To avoid **computational safe-error attacks** (**C-SE**) on these operations, in the end of the exponentiation we include the check whether $R_{(0,1)} \cdot R_{(1,1)} \equiv R_{(0,2)} \cdot R_{(1,2)} \bmod x$. Since both the products corresponding to the two exponents are $M^{2^n-1} \bmod x$, this holds if the values are not corrupted. With this, we verify the correctness of the dummy values.

Method		**Efficiency criteria**								**Fault injection attacks**					**Other**	
Ref.	Alg.	CRT	Check	Inv.	Reg	k-bit	Reg	Mult.	Sq.	Ran	Zeroing		Skipping		PA	SE
		Total					Per exp.				1	2	1	2		
[8]	6	2	4	$2^{(p,q)}$	0	0	4	$n+3$	$2n$	R_2	M,$R_\forall$		(7)	(5-6),2(7)	✓	✓
	2	**3**	4	$\mathbf{1}^{(\mathbf{pqs})}$	**1**	**4k**	**3**	$n+2$	$2n$	✔	✔	✔	✔	✔	✓	✓
[5,6]	7	3	5	$1^{(pq)}$	0	0	4	$n+2$	n	✓	✓	✓	✓	(6-7)	✓	✓
	3	3	4	$1^{(pqs)}$	**1**	**4k**	**3**	$n+1$	n	✓	✓	✓	✓	✔	✓	✓
[22]	9	1	2	0	0	0	3	$1.65n$	0	M	✓	✓	✓	✓	✓	✓
[18]	10	1	2	0	0	0	3	$1.65n$		M	✓	d_1,d_2	✓	✓	×	×
	4	1	**4**	0	0	0	**5**	$\mathbf{2}n+3$	n	✔	✓	✔	✓	✓	✔	✔

Table 3: Comparison of our PA-SE-FA self-secure exponentiation countermeasures with previous methods. The notation is consistent with that of Table 1 and Table 2, k-bit denoting the included k-bit operations (squaring and multiplication). We highlight with bold checkmarks (✔) those vulnerabilities that we eliminated in our secure countermeasures and we bold the additional resources needed to be used in order to achieve security against all the considered attacks.

To achieve resistance against **differential power analysis** on the exponentiation ($\mathbf{DPA_{exp}}$) and **memory safe-error attacks** (**M-SE**), we include the exponent blinding method of Rivain in the RSA-CRT algorithm [22]. Against DPA on the CRT recombination ($\mathbf{DPA_{CRT}}$), we apply the blinded CRT recombination method with extended modulus from [9]. For the description of r_1 and r_2 and the correctness of the blinding method, the reader is referred to [9,22].

To detect any randomizing fault on the **message M** ($\mathbf{FA^r_M}$), we include its value in the coherence checks as it was seen in case of the countermeasures from [5, 6, 8, 9]. We decrease the value of the exponents used for the calculation of c_p and c_q by one, and multiply the results with M, during the verification in lines 7 and 10 of Alg. 4b. For instance, if S_p and c_p are calculated by means of a corrupted $\widehat{M}$, the verification $M \cdot \widehat{M}^{d_p+r_1\varphi(p)} \cdot \widehat{M}^{r_2\varphi(p)-d_p-1} \equiv 1 \bmod p$ does not hold with high probability. With this, the zeroing faults on **exponents $\mathbf{d_1}$ and $\mathbf{d_2}$** ($\mathbf{FA^z_{(d_1,d_2)}}$) are also thwarted, the algorithm returns $(1, 1)$ in case of two null exponents, and the modified check does not hold anymore.

Our PA-SE-FA-resistant countermeasure using double exponentiation is depicted in Alg. 4. Though the modified countermeasure is less memory-efficient than Le et al.'s algorithm, we note its advantage against physical attacks.

6 Conclusion

In this paper, we analyzed the existing self-secure exponentiation countermeasures against the Bellcore attack on RSA-CRT. Using our framework, we simulated all possible fault injections considering random and zeroing faults as well as instruction skips on the lines of pseudocode. We found that all the coun-

termeasures using regular exponentiation algorithms lacked protection against some kind of faults or power analyses.

We presented three countermeasures, one for each exponentiation algorithm used for designing self-secure exponentiation countermeasures(cf. Table 3). All the three methods are based on regular algorithms to prevent *simple power analysis* (SPA), include randomization to be resistant to *differential power analysis* (DPA) and *memory safe-error* (M-SE) attacks, and eliminate dummy operations which could be exploited by *computational safe-error* (C-SE) attacks. Measures are included against all considered *fault injection attacks* (FA) as well. We verified that we eliminated the previous vulnerabilities of the methods without introducing new ones by applying our simulation framework on the pseudocode of the improved algorithms. To prevent skipping faults, we included additional checks into two of our methods, inspired by a countermeasure in Shamir's family, resulting in hybrid methods. We included prevention against fault attacks on the previously vulnerable register in the countermeasure that uses the Montgomery ladder. Our proposed solution that uses double exponentiation includes protection against power analyses and safe-error attacks in the algorithm where it was not considered.

We note that the vulnerability of the message corruption and of the DPA on the CRT recombination in Rivain's SPA-resistant method can be eliminated in a similar algorithmic manner as in Section 5.3, gaining another, the most efficient secure software countermeasure when precomputation is allowed. When precomputation is not allowed, our proposed solution using the square-and-multiply-always algorithm is the most efficient algorithmic countermeasure.

Acknowledgments

This work has been co-funded by the DFG as part of projects P1 and S5 within the CRC 1119 CROSSING and by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.: Fault attacks on RSA with CRT: concrete results and practical countermeasures. In: Cryptographic Hardware and Embedded Systems, (CHES '02). pp. 260–275. Springer (2003)
2. Baek, Y.: Regular 2^w-ary right-to-left exponentiation algorithm with very efficient DPA and FA countermeasures. Int. J. Inf. Sec. 9(5), 363–370 (2010)
3. Blömer, J., Gomes Da Silva, R., Gunther, P., Kramer, J., Seifert, J.P.: A practical second-order fault attack against a real-world pairing implementation. In: Fault Diagnosis and Tolerance in Cryptography (FDTC '14). pp. 123–136. IEEE (2014)
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Advances in Cryptology – EUROCRYPT 1997. pp. 37–51. Springer (1997)
5. Boscher, A., Handschuh, H., Trichina, E.: Blinded fault resistant exponentiation revisited. In: Fault Diagnosis and Tolerance in Cryptography (FDTC '09). pp. 3–9. IEEE (2009)

6. Boscher, A., Naciri, R., Prouff, E.: CRT RSA algorithm protected against fault attacks. In: Information Security Theory and Practices. pp. 229–243. Springer (2007)
7. Coron, J.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Cryptographic Hardware and Embedded Systems (CHES '99). pp. 292–302. Springer (1999)
8. Fumaroli, G., Vigilant, D.: Blinded fault resistant exponentiation. In: Fault Diagnosis and Tolerance in Cryptography (FDTC '06). pp. 62–70. Springer (2006)
9. Giraud, C.: An RSA implementation resistant to fault attacks and to simple power analysis. IEEE Trans. Computers 55(9), 1116–1120 (2006)
10. Joye, M., Karroumi, M.: Memory-efficient fault countermeasures. In: Smart Card Research and Advanced Applications. pp. 84–101. Springer (2011)
11. Joye, M., Lenstra, A.K., Quisquater, J.: Chinese remaindering based cryptosystems in the presence of faults. J. Cryptology 12(4), 241–245 (1999)
12. Joye, M., Paillier, P., Yen, S.M.: Secure evaluation of modular functions (2001)
13. Joye, M., Yen, S.: The Montgomery powering ladder. In: Cryptographic Hardware and Embedded Systems, CHES 2002. pp. 291–302. Springer (2003)
14. Kim, C.H., Quisquater, J.: How can we overcome both side channel analysis and fault attacks on RSA-CRT? In: Fault Diagnosis and Tolerance in Cryptography (FDTC '07). pp. 21–29. IEEE (2007)
15. Kiss, A., Krämer, J., Rauzy, P., Seifert, J.P.: Algorithmic countermeasures against fault attacks and power analysis for RSA-CRT. Cryptology ePrint Archive, Report 2016/238 (2016), `http://eprint.iacr.org/2016/238`
16. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in Cryptology – CRYPTO 1999. pp. 388–397. Springer (1999)
17. Krämer, J., Nedospasov, D., Seifert, J.: Weaknesses in current RSA signature schemes. In: Information Security and Cryptology, (ICISC '11). pp. 155–168. Springer (2012)
18. Le, D., Rivain, M., Tan, C.H.: On double exponentiation for securing RSA against fault analysis. In: Topics in Cryptology – CT-RSA '14. pp. 152–168. Springer (2014)
19. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of computation 48(177), 243–264 (1987)
20. Quisquater, J.J., Couvreur, C.: Fast decipherment algorithm for RSA public-key cryptosystem. Electronics letters 18(21), 905–907 (1982)
21. Rauzy, P., Guilley, S.: Countermeasures against high-order fault-injection attacks on CRT-RSA. In: Fault Diagnosis and Tolerance in Cryptography (FDTC '14). pp. 68–82. IEEE (2014)
22. Rivain, M.: Securing RSA against fault analysis by double addition chain exponentiation. In: Topics in Cryptology – CT-RSA '09. pp. 459–480. Springer (2009)
23. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
24. Shamir, A.: Method and apparatus for protecting public key schemes from timing and fault attacks (1999), US Patent 5,991,415
25. Witteman, M.: A DPA attack on RSA in CRT mode (2009)
26. Yen, S., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Computers 49(9), 967–970 (2000)
27. Yen, S., Kim, S., Lim, S., Moon, S.: A countermeasure against one physical cryptanalysis may benefit another attack. In: Information Security and Cryptology (ICISC '01). pp. 414–427. Springer (2002)
28. Yen, S., Lien, W., Moon, S., Ha, J.: Power analysis by exploiting chosen message and internal collisions – vulnerability of checking mechanism for RSA-decryption. In: Progress in Cryptology – Mycrypt 2005. vol. 3715, pp. 183–195. Springer (2005)

A Self-Secure Exponentiation Countermeasures

Algorithm 5 Giraud's countermeasure [9]
PA attack model: SPA, chosen message SPA from [28].
Fault model: Random faults on variables and input parameters. Zeroing attacks, disruption of checking are regarded as impossible in practice. For the integrity check of d, i, we assume that an additional register is used in Table 1.

(5a) Modular exp.: GirExp(M, d, x, r)

input: $M, d = (d_{n-1}, \ldots, d_0)_2$ odd, x, r
output: $(M^{d-1} \bmod r \cdot x, M^d \bmod r \cdot x)$

1: $x_r := r \cdot x$
2: $R_0 := M$, $R_1 := R_0^2 \bmod x_r$
3: **for** i from $n-2$ to 1 **do**
4: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_{d_i} \bmod x_r$
5: $\quad R_{d_i} := R_{d_i}^2 \bmod x_r$
6: **end for**
7: $R_1 := R_1 \cdot R_0 \bmod x_r$
8: $R_0 := R_0^2 \bmod x_r$
9: **if** i or d disturbed **then**
10: $\quad$ **return** error
11: **end if**
12: **return** (R_0, R_1)

(5b) Giraud's RSA-CRT

input: M, p, q, d_p, d_q, i_q
output: $M^d \bmod pq$ or error

1: Pick k-bit random prime r
2: (S'_p,S_p) :=GirExp$(M \bmod p,d_p,p,r)$
3: (S'_q,S_q) :=GirExp$(M \bmod q,d_q,q,r)$
4: $S := \mathrm{CRT}_{\mathrm{blinded}}(S_p, S_q)$
5: $S' := \mathrm{CRT}_{\mathrm{blinded}}(S'_p, S'_q)$
6: $S' := M \cdot S' \bmod (p \cdot q)$
7: **if** $S' \neq S$ **then return** error
8: **end if**
9: **if** p, q or i_q disturbed **then**
10: $\quad$ **return** error
11: **end if**
12: **return** S

Algorithm 6 Fumaroli and Vigilant's countermeasure [8]
Attack model: SPA, DPA, against which blinding is included.
Fault model: That of Giraud's [9].

(6a) Modular exp.: FumVigExp(M, d, x)

input: $M \neq 0, d = (d_{n-1}, \ldots, d_0)_2$, x
output: $(M^d \bmod x, M^{d+1} \bmod x)$

1: Pick k-bit random prime r
2: $R_0 := r$, $R_1 := rM \bmod x$
3: $R_2 := r^{-1} \bmod x$, $D := 0$
4: **for** i from $n-1$ to 0 **do**
5: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_{d_i} \bmod x$
6: $\quad R_{d_i} := R_{d_i}^2 \bmod x$
7: $\quad R_2 := R_2^2 \bmod x$
8: $\quad D := D + d_i$,
9: $\quad D := D \cdot 2$
10: **end for**
11: $D := D/2$
12: $R_2 := R_2 \oplus D \oplus d$
13: **return** $(R_2 \cdot R_0 \bmod x, R_2 \cdot R_1 \bmod x)$

(6b) Fumaroli and Vigilant's RSA-CRT

input: $M \neq 0, p, q, d_p, d_q, i_q$
output: $M^d \bmod pq$ or error

1: (S_p,S'_p) :=FumVigExp$(M \bmod p,d_p,p)$
2: (S_q,S'_q) :=FumVigExp$(M \bmod q,d_q,q)$
3: $S := \mathrm{CRT}(S_p, S_q)$
4: $S' := \mathrm{CRT}(S'_p, S'_q)$
5: **if** $S \cdot M \bmod p \cdot q \not\equiv S'$ **then**
6: $\quad$ **return** error
7: **end if**
8: **if** p, q or i_q disturbed **then**
9: $\quad$ **return** error
10: **end if**
11: **return** S

Algorithm 7 Boscher et al's countermeasure 2007 [6], **modifications 2009** [5]
Attack model: Regularity against SPA, **blinding against DPA**.
Fault model: One fault per execution [6], on local variables, input parameters.

(7a) Modular exp: BosExp(M,d,x,$\mathbf{r}$,$\mathbf{r_{inv}}$)

input: M,$d = (d_{n-1}, \ldots, d_0)_2$,$x$,$\mathbf{r}$,$\mathbf{r_{inv}}$
output: ($\mathbf{r} \cdot M^d \bmod x$, $\mathbf{r_{inv}} \cdot M^{2^n-d-1} \bmod x$, $M^{2^n} \bmod x$)

1: $R_0 := 1 \cdot \mathbf{r}$
2: $R_1 := 1 \cdot \mathbf{r_{inv}}$
3: $R_2 := M$
4: $D := 0$
5: **for** i from 0 to $n-1$ **do**
6: $\quad R_{\overline{d_i}} := R_{\overline{d_i}} \cdot R_2 \bmod x$
7: $\quad R_2 := R_2^2 \bmod x$
8: $\quad D := D + 2^n \cdot d_i$
9: $\quad D := D/2$
10: **end for**
11: **if** $(D \neq d)$ or $(R_2 = 0)$ **then**
12: $\quad$ **return** error
13: **end if**
14: **return** (R_0, R_1, R_2)

(7b) Boscher et al.'s RSA-CRT

input: $M \neq 0, p, q, d_p, d_q, i_q$
output: $M^d \bmod pq$ or error

1: **Pick a k-bit random integer r**
2: $\mathbf{r_{inv}} := \mathbf{r^{-1}}$ **mod pq**
3: $(S_p, S'_p, T_p) :=$ BosExp($M \bmod p, d_p, p, \mathbf{r}, \mathbf{r_{inv}}$)
4: $(S_q, S'_q, T_q) :=$ BosExp($M \bmod q, d_q, q, \mathbf{r}, \mathbf{r_{inv}}$)
5: $S := \mathrm{CRT}(S_p, S_q)$
6: $S' := \mathrm{CRT}(S'_p, S'_q)$
7: $T := \mathrm{CRT}(T_p, T_q)$
8: **if** $M \cdot S \cdot S' \not\equiv T \bmod pq$ **then**
9: $\quad$ **return** error
10: **end if**
11: **return** $\mathbf{r_{inv}} \cdot S \bmod pq$

Algorithm 8 Rivain's countermeasure [22]
The addition chain is precomputed with ChainCompute(d_1, d_2) from [22] and stored in memory or is computed on-the-fly.

(8a) Double exp.: RivExp(M,$\omega(d_1,d_2)$,x)

input: M, $\omega(d_1,d_2)$ n-bits chain, $d_1 \leq d_2$, x
output: $(M^{d_1} \bmod x, M^{d_2} \bmod x)$

1: $R_0 := 1$, $R_1 := M$, $\gamma := 1$, $i := 1$
2: **for** $i = 1$ to n **do**
3: $\quad$ **if** $(\omega_i = 0)$ **then**
4: $\quad\quad R_\gamma := R_\gamma^2 \bmod x$
5: $\quad\quad i := i + 1$
6: $\quad\quad$ **if** $(\omega_i = 1)$ **then**
7: $\quad\quad\quad R_\gamma := R_\gamma \cdot M \bmod x$
8: $\quad\quad$ **end if**
9: $\quad$ **else**
10: $\quad\quad R_{\gamma\oplus1} := R_{\gamma\oplus1} \cdot R_\gamma \bmod x$
11: $\quad\quad \gamma := \gamma \oplus 1$
12: $\quad$ **end if**
13: **end for**
14: **return** $(R_{\gamma\oplus1}, R_\gamma)$

(8b) Rivain's RSA-CRT

input: M, p, q, d_p, d_q, i_q
output: $M^d \bmod pq$ or error

1: $\omega_p := \mathrm{ChainCompute}(d_p, 2(p-1)-d_p)$
2: $(S_p, c_p) := \mathrm{RivExp}(M \bmod p, \omega_p, p)$
3: $\omega_q := \mathrm{ChainCompute}(d_q, 2(q-1)-d_q)$
4: $(S_q, c_q) := \mathrm{RivExp}(M \bmod q, \omega_q, q)$
5: $S := \mathrm{CRT}(S_p, S_q)$
6: **if** $S \cdot c_p \not\equiv 1 \bmod p$ **then**
7: $\quad$ **return** error
8: **end if**
9: **if** $S \cdot c_q \not\equiv 1 \bmod q$ **then**
10: $\quad$ **return** error
11: **end if**
12: **return** S

Algorithm 9 Rivain's PA-resistant countermeasure [22]
Attack model: Regular SecRivExp and ChainCom against SPA, blinding against DPA. This blinding method can only be used if the double addition chain is computed on-the-fly.
Fault model: M is assumed to be protected, transient faults, i.e., faults whose effect lasts for one computation, are considered.

(9a) Double exp: SecRivExp(M,$\omega(d_1,d_2)$,x)

input: $M \neq 0$, $\omega(d_1, d_2)$ n-bits, $d_1 \leq d_2$, x
output: $(M^{d_1} \bmod x, M^{d_2} \bmod x)$

$R_{(0,0)} := 1, R_{(0,1)} := M$,
$R_{(1,0)} := M$
$\gamma := 1, \mu := 1, i := 0$
while $i < n$ **do**
  $t := \omega_i \wedge \mu$
  $v := \omega_{i+1} \wedge \mu$
  $R_{(0,\gamma\oplus t)} := R_{(0,\gamma\oplus t)} \cdot R_{((\mu\oplus 1),\gamma\wedge\mu)} \bmod x$
  $\mu := t \vee (v \oplus 1)$
  $\gamma := \gamma \oplus t$
  $i := i + \mu + \mu \wedge (t \oplus 1)$
end while
return $(R_{\gamma\oplus 1}, R_\gamma)$

(9b) RSA-CRT

input: M, p, q, d_p, d_q, i_q
output: $M^d \bmod pq$ or error

Pick small $r_1, r_2 \in \mathbb{Z}$ $r_2 \geq r_1 + 2$
$\omega_p :=$ ChainCom(d_p+$r_1(p-1)$,$r_2(p-1) - d_p$)
$(S_p, c_p) :=$ SecRivExp($M \bmod p, \omega_p, p$)
$\omega_q :=$ ChainCom(d_q+$r_1(q-1)$,$r_2(q-1) - d_q$)
$(S_q, c_q) :=$ SecRivExp($M \bmod q, \omega_q, q$)
$S := \mathrm{CRT}(S_p, S_q)$
if $S \cdot c_p \not\equiv 1 \bmod p$ **then**
  return error
end if
if $S \cdot c_q \not\equiv 1 \bmod q$ **then**
  return error
end if
return $S \bmod pq$

Algorithm 10 Le et al.'s binary countermeasure [18]
Attack model: No side-channel attacks are discussed in [18].
Fault model: Same as that of Rivain [22].

(10a) Double exp.: LeExp(M, d_1, d_2, x)

input: $M \neq 0, d_1 = (d_{1,n-1}, \ldots, d_{1,0})_2$ $d_2 = (d_{2,n-1}, \ldots, d_{2,0})_2, x$
output: $(M^{d_1} \bmod x, M^{d_2} \bmod x)$

$R_0 := 1$, $R_1 := 1$, $R_2 := M$
for $i = 0$ **to** $n - 1$ **do**
  if $d_{1,i} = 1$ **then**
    $R_0 := R_0 \cdot R_2 \bmod x$
  end if
  if $d_{2,i} = 1$ **then**
    $R_1 := R_1 \cdot R_2 \bmod x$
  end if
  $R_2 := R_2^2 \bmod x$
end for
return (R_0, R_1)

(10b) Rivain's RSA-CRT

input: $M \neq 0, p, q, d_p, d_q, i_q$
output: $M^d \bmod pq$ or error

$(S_p, c_p) :=$ LeExp($M \bmod p$, $d_p, 2(p-1) - d_p, p$)
$(S_q, c_q) :=$ LeExp($M \bmod q$, $d_q, 2(q-1) - d_q, q$)
$S := \mathrm{CRT}(S_p, S_q)$
if $S \cdot c_p \not\equiv 1 \bmod p$ **then**
  return error
end if
if $S \cdot c_q \not\equiv 1 \bmod q$ **then**
  return error
end if
return S