A Formal Proof of Countermeasures against Fault Injection Attacks on CRT-RSA

Pablo Rauzy, Sylvain Guilley

A Formal Proof of Countermeasures against Fault Injection Attacks on CRT-RSA

Pablo Rauzy, Sylvain Guilley
firstname.lastname@telecom-paristech.fr

Télécom ParisTech

Abstract. In this article, we describe a methodology that aims at either breaking or proving the security of CRT-RSA algorithms against fault injection attacks. In the specific case-study of BellCoRe attacks, our work bridges a gap between formal proofs and implementation-level attacks. We apply our results to three versions of CRT-RSA, namely the naive one, that of Shamir, and that of Aumüller *et al.* Our findings are that many attacks are possible on both the naive and the Shamir implementations, while the implementation of Aumüller *et al.* is resistant to all fault attacks with one fault. However, we show that the countermeasure is not minimal, since two tests out of seven are redundant and can simply be removed.

Keywords: RSA (*Rivest, Shamir, Adleman* [13]), CRT (*Chinese Remainder Theorem*), fault injection, BellCoRe (*Bell Communications Research*) attack, formal proof, OCaml.

1 Introduction

It is known since 1997 that injecting faults during the computation of CRT-RSA could yield to malformed signatures that expose the prime factors (p and q) of the public modulus ($N = p \cdot q$). Notwithstanding, computing without the fourfold acceleration conveyed by the CRT is definitely not an option in practical applications. Therefore, many countermeasures have appeared that consist in step-wise internal checks during the CRT computation. To our best knowledge, none of these countermeasures have been proven formally. Thus without surprise, some of them have been broken, and then patched. The current state-of-the-art in computing CRT-RSA without exposing p and q relies thus on algorithms that have been carefully scrutinized by cryptographers. Nonetheless, neither the hypotheses of the fault attack nor the security itself have been unambiguously modeled.

This is the purpose of this paper. The difficulties are *a priori* multiple: in fault injection attacks, the attacker has an extremely high power because he can fault any variable. Traditional approaches thus seem to fall short in handling this problem. Indeed, there are two canonical methodologies: *formal* and *computational* proofs. Formal proofs (*e.g.*, in the so-called Dolev-Yao model) do not

capture the requirement for faults to preserve some information about one of the two moduli; indeed, it considers the RSA as a black-box with a key pair. Computational proofs are way too complicated since the handled numbers are typically $2,048$ bit long.

The state-of-the-art contains one reference related to the formal proof of CRT-RSA: it is the work of Christofi, Chetali, Goubin and Vigilant [6]. For tractability purposes, the proof is conducted on reduced versions of the algorithms parameters. One fault model is chosen authoritatively (the zeroization of a complete intermediate data), which is a strong assumption. In addition, the verification is conducted on a pseudo-code, hence concerns about its portability after compilation into machine-level code. Another reference related to formal proofs against fault injection attacks is the work of Guo, Mukhopadhyay and Karri. In [8], they explicit an AES implementation that is provably protected against differential fault analyses [3]. The approach is purely combinational, because the faults propagation in AES concerns 32-bit words called columns; consequently, all fatal faults (and thus all innocuous faults) can be enumerated.

Contributions. Our contribution is also to reach a full fault coverage on CRT-RSA algorithm, thereby keeping the proof even if the code is transformed (*e.g.*, compiled or partitioned in software/hardware). To this end we developed tools based of symbolic computation in the framework of modular arithmetic, which enable formal analysis of CRT-RSA and its countermeasures against fault injection attacks. We apply our methods on three implementations of CRT-RSA: an unprotected one, one protected by Shamir countermeasure, and one protected by Aumüller *et al.* countermeasure. We find many possible fault injections which enable BellCoRe attacks on an unprotected implementation of the CRT-RSA computation, as well as on one protected by Shamir countermeasure. We formally prove the security of the Aumüller *et al.* countermeasure against the BellCoRe attack, under a fault model that considers *permanent faults* (in memory) and *transient faults* (one-time faults, even on copies of the secret key parts), with or without forcing at zero, and with possibly faults at various locations. We also simplify Aumüller *et al.* countermeasure by proving that two out of the seven tests it consists of are redundant and can be removed.

Organization of the paper. We recall CRT-RSA cryptosystem and the BellCoRe attack in Sec. 2; still from an historical perspective, we explain how the CRT-RSA implementation has been amended to withstand more or less efficiently the BellCoRe attack. Then, in Sec. 3, we define our approach. Sec. 4, Sec. 5, and Sec. 6 are case studies using the methods developed in Sec. 3 of respectively an unprotected version of the CRT-RSA computation, a version protected by Shamir countermeasure, and a version protected by Aumüller *et al.* countermeasure. Conclusions and perspectives are in Sec. 7. To improve the readability of the article, the longest code portions have been consigned in appendix.

2 CRT-RSA and the "BellCoRe" attack

This section recaps known results about fault attacks on CRT-RSA (see also [12] and [15, Chap. 3]). Its purpose is to settle the notions and the associated notations that will be used in the later sections (that contain novel contributions).

2.1 CRT-RSA

RSA is both an *encryption* and a *signature* scheme. It relies on the identity that for all message $0 \leq m < N$, $(m^d)^e \equiv m \mod N$, where $d \equiv e^{-1} \mod \varphi(N)$, by the Euler theorem. In this equation, φ is the Euler totient function, equal to $\varphi(N) = (p-1)\cdot(q-1)$ when $N = p \cdot q$ is a composite number, product of two primes p and q. For example, if Alice generates the signature $S = m^d \mod N$, then Bob can verify it by computing $S^e \mod N$, which must be equal to m unless Alice is pretending to know d although she does not. Therefore (N, d) is called the private key, and (N, e) the public key. In this paper, we are not concerned about the key generation step of RSA, and simply assume that d is an unknown number in $[\![1, \varphi(N) = (p-1)\cdot(q-1)[\![$. Actually, d can also be chosen equal to the smallest value $e^{-1} \mod \lambda(n)$, where $\lambda(n) = \frac{(p-1)\cdot(q-1)}{\gcd(p-1,q-1)}$ is the Carmichael function. The computation of $m^d \mod N$ can be speeded-up by a factor four by using the Chinese Remainder Theorem (CRT). Indeed, figures modulo p and q are twice as short as those modulo N. For example, for $2,048$ bit RSA, p and q are $1,024$ bit long. The CRT-RSA consists in computing $S_p = m^d \mod p$ and $S_q = m^d \mod q$, which can be recombined into S with a limited overhead. Due to the little Fermat theorem (special case of the Euler theorem when the modulus is a prime), $S_p = (m \mod p)^{d \mod (p-1)} \mod p$. This means that in the computation of S_p, the processed data have $1,024$ bit, and the exponent itself has $1,024$ bits (instead of $2,048$ bits). Thus the multiplication is four times faster and the exponentiation eight times faster. However, as there are two such exponentiations (modulo p and q), the overall CRT-RSA is roughly speaking four times faster than RSA computed modulo N.

This acceleration justifies that CRT-RSA is always used if the factorization of N as $p \cdot q$ is known. In CRT-RSA, the private key is a more rich structure than simply (N, d): it is actually comprised of the 5-tuple (p, q, d_p, d_q, i_q), where:

- $d_p \doteq d \mod (p-1)$,
- $d_q \doteq d \mod (q-1)$,
- $i_q \doteq q^{-1} \mod p$.

The "naive" CRT-RSA algorithm is presented in Alg. 1. It is straightforward to check that the signature computed at line 3 belongs to $[\![0, p\cdot q-1]\!]$. Consequently, no reduction modulo N is necessary before returning S.

2.2 BellCoRe attack on CRT-RSA

In 1997, an dreadful remark has been made by Boneh, DeMillo and Lipton [4], three staff of BellCoRe: Alg. 1 could reveal the secret primes p and q if the

Algorithm 1: Naive CRT-RSA

Input : Message m, key (p, q, d_p, d_q, i_q)
Output: Signature $m^d \mod N$

1 $S_p = m^{d_p} \mod p$ /* Signature modulo p */
2 $S_q = m^{d_q} \mod q$ /* Signature modulo q */
3 $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ /* Recombination */
4 **return** S

computation is faulted, even in a very random way. The attack can be expressed as the following proposition.

Proposition 1 (Orignal BellCoRe attack). *If the intermediate variable S_p (resp. S_q) is returned faulted as $\widehat{S_p}$ (resp. $\widehat{S_q}$)[1], then the attacker gets an erroneous signature $\widehat{S}$, and is able to recover p (resp. q) as* $\gcd(N, S - \widehat{S})$.

Proof. For all integer x, $\gcd(N, x)$ can only take 4 values:

- 1, if N and x are coprime,
- p, if x is a multiple of p,
- q, if x is a multiple of q,
- N, if x is a multiple of both p and q, *i.e.*, of N.

In Alg. 1, if S_p is faulted (*i.e.*, replaced by $\widehat{S_p} \neq S_p$), then $S - \widehat{S} = q \cdot \Big((i_q \cdot (S_p - S_q) \mod p) - (i_q \cdot (\widehat{S_p} - S_q) \mod p)\Big)$, and thus $\gcd(N, S - \widehat{S}) = q$. If S_q is faulted (*i.e.*, replaced by $\widehat{S_q} \neq S_q$), then $S - \widehat{S} \equiv (S_q - \widehat{S_q}) - (q \mod p) \cdot i_q \cdot (S_q - \widehat{S_q}) \equiv 0 \mod p$ because $(q \mod p) \cdot i_q \equiv 1 \mod p$. Thus $S - \widehat{S}$ is a multiple of p. Additionally, $S - \widehat{S}$ is not a multiple of q. So, $\gcd(N, S - \widehat{S}) = p$. □

This version of the BellCoRe attack requires that two identical messages with the same key can be signed; indeed, one signature yields the genuine S while the other one is perturbed, and thus returns $\widehat{S}$. Little later, the BellCoRe attack has been improved by Joye, Lenstra and Quisquater [10]. This time, the attacker can recover p or q with one only faulty signature, provided the input m of RSA is known.

Proposition 2 (One faulty signature BellCoRe attack). *If the intermediate variable S_p (resp. S_q) is returned faulted as $\widehat{S_p}$ (resp. $\widehat{S_q}$), then the attacker gets an erroneous signature $\widehat{S}$, and is able to recover p (resp. q) as* $\gcd(N, m - \widehat{S}^e)$ *(*with an overwhelming probability*).*

[1] In other papers related to faults, the faulted variables (such as X) are noted either with a star (X^*) or a tilde ($\tilde{X}$); in this paper, we use a hat, as it can stretch, hence cover the adequate portion of the variable. For instance, it allows to make an unambiguous difference between a faulted data raised at some power and a fault on a data raised at a given power (contrast $\widehat{X}^e$ with $\widehat{X^e}$).

Proof. By proposition 1, if a fault occurs during the computation of S_p, then $\gcd(N, S - \widehat{S}) = q$ (*most likely*). This means that:

- $S \not\equiv \widehat{S} \mod p$, and thus $S^e \not\equiv \widehat{S}^e \mod p$ (*indeed, if the congruence was true, we would have* $e|p-1$*, which is very unlikely*);
- $S \equiv \widehat{S} \mod q$, and thus $S^e \equiv \widehat{S}^e \mod q$;

As $S^e \equiv m \mod N$, this proves the result. A symmetrical reasoning can be done if the fault occurs during the computation of S_q. □

2.3 Protection of CRT-RSA against BellCoRe attacks

Several protections against the BellCoRe attacks have been proposed. A non-exhaustive list is given below, and then, the most salient features of these countermeasures are described:

- Naive;
- Obvious countermeasures: no CRT, or with signature verification;
- Shamir [14];
- Aumüller *et al.* [1];
- Vigilant, original [16] and with some corrections by Coron *et al.* [7];
- Kim *et al.* [11].

Obvious countermeasures Fault attacks on RSA can be thwarted simply by refraining from implementing the CRT. If this is not affordable, then the signature can be verified before being outputted. Such protection is efficient in practice, but is criticized for two reasons. First of all, it requires an access to e; second, the performances are incurred by the extra exponentiation needed for the verification. This explains why other countermeasures have been devised.

Shamir The CRT-RSA algorithm of Shamir builds on top of the CRT and introduces, in addition to the two primes p and q, a third factor r. This factor r is random and small (less than 64 bit long), and thus co-prime with p and q. The computations are carried out modulo $p' = p \cdot r$ (resp. modulo $q' = q \cdot r$), which allows for a retrieval of the intended results by reducing them modulo p (resp. modulo q), and for a verification by a reduction modulo r. Alg. 2 describes one version of Shamir's countermeasure.

Aumüller The CRT-RSA algorithm of Aumüller *et al.* is a variation of that of Shamir, that is primarily intended to fix two shortcomings. First it removes the need for d in the signature process, and second, it also checks the recombination step. The countermeasure, given in Alg. 3, introduces, in addition to p and q, a third prime t. The computations are done modulo $p' = p \cdot t$ (resp. modulo $q' = q \cdot t$), which allows for a retrieval of the intended results by reducing them modulo p (resp. modulo q), and for a verification by a reduction modulo t.

Algorithm 2: Shamir CRT-RSA

Input : Message m, key (p, q, d, i_q),
32-bit random prime r
Output: Signature $m^d \mod N$,
or error if some fault injection has been detected.

1 $p' = p \cdot r$
2 $d_p = d \mod (p-1) \cdot (r-1)$
3 $S'_p = m^{d_p} \mod p'$ `/* Signature modulo` p' `*/`
4 $q' = q \cdot r$
5 $d_q = d \mod (q-1) \cdot (r-1)$
6 $S'_q = m^{d_q} \mod q'$ `/* Signature modulo` q' `*/`
7 $S_p = S'_p \mod p$
8 $S_q = S'_q \mod q$
9 $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ `/* Same as in line 3 of Alg. 1 */`
10 **if** $S'_p \not\equiv S'_q \mod r$ **then**
11 | **return** error
12 **else**
13 | **return** S
14 **end**

However, the verification is more subtle than for the case of Shamir. In Shamir's CRT-RSA (Alg. 2), the verification is *symmetrical*, in that the computations modulo $p \cdot r$ and $q \cdot r$ operate on the same object, namely m^d. In Aumüller *et al.*'s CRT-RSA (Alg. 3), the verification is *asymmetrical*, since the computations modulo $p \cdot t$ and $q \cdot t$ operate on two different objects, namely $m^{d_p \mod (t-1)}$ and $m^{d_q \mod (t-1)}$. The verification consists in an identity that resembles that of ElGamal for instance: $(m^{d_p \mod (t-1)})^{d_q \mod (t-1)} \equiv (m^{d_q \mod (t-1)})^{d_p \mod (t-1)} \mod t$. Specifically, if we note S'_p the signature modulo p', then $S_p = S \mod p$ is equal to $S'_p \mod p$. Furthermore, let us denote $S_{pt} = S'_p \mod t$, $S_{qt} = S'_q \mod t$, $d_{pt} = d_p \mod (t-1)$ and $d_{qt} = d_q \mod (t-1)$. It can be checked that those figures satisfy the identity: $S_{pt}^{d_{qt}} \equiv S_{qt}^{d_{pt}} \mod t$, because both terms are equal to $m^{d_{pt} \cdot d_{qt}} \mod t$. The prime t is referred to as a security parameter, as the probability to pass the test (at line 23 of Alg. 3) is equal to $1/t$ (*i.e.*, about 2^{-32}), assuming a uniform distribution of the faults. Indeed, this is the probability to find a large number that, once reduced modulo t, matches a predefined value.

Alg. 3 does some verifications during the computations, and reports an error in case a fault injection can cause a malformed signature susceptible of unveiling p and q. More precisely, an error is returned in either of these seven cases:

1. p' is not a multiple of p (*because this would amount to faulting* p *in the naive algorithm*)
2. $d'_p = d_p + \mathsf{random}_1 \cdot (p-1)$ is not equal to $d_p \mod (p-1)$ (*because this would amount to faulting* d_p *in the naive algorithm*)

Algorithm 3: Aumüller CRT-RSA

Input : Message m, key (p, q, d_p, d_q, i_q),
32-bit random prime t
Output: Signature $m^d \mod N$,
or error if some fault injection has been detected.

1 $p' = p \cdot t$
2 $d'_p = d_p + \mathsf{random}_1 \cdot (p-1)$ `/* Against DPA, not fault attacks */`
3 $S'_p = m^{d'_p} \mod p'$ `/* Signature modulo` p' `*/`
4 **if** $(p' \mod p \neq 0)$ **or** $(d'_p \not\equiv d_p \mod (p-1))$ **then**
5 | **return** error
6 **end**
7 $q' = q \cdot t$
8 $d'_q = d_q + \mathsf{random}_2 \cdot (q-1)$ `/* Against DPA, not fault attacks */`
9 $S'_q = m^{d'_q} \mod q'$ `/* Signature modulo` q' `*/`
10 **if** $(q' \mod q \neq 0)$ **or** $(d'_q \not\equiv d_q \mod (q-1))$ **then**
11 | **return** error
12 **end**
13 $S_p = S'_p \mod p$
14 $S_q = S'_q \mod q$
15 $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ `/* Same as in line 3 of Alg. 1 */`
16 **if** $(S - S'_p \not\equiv 0 \mod p)$ **or** $(S - S'_q \not\equiv 0 \mod q)$ **then**
17 | **return** error
18 **end**
19 $S_{pt} = S'_p \mod t$
20 $S_{qt} = S'_q \mod t$
21 $d_{pt} = d'_p \mod (t-1)$
22 $d_{qt} = d'_q \mod (t-1)$
23 **if** $S_{pt}^{d_{qt}} \not\equiv S_{qt}^{d_{pt}} \mod t$ **then**
24 | **return** error
25 **else**
26 | **return** S
27 **end**

3. q' is not a multiple of q (*because this would amount to faulting q in the naive algorithm*)
4. $d'_q = d_q + \mathsf{random}_2 \cdot (q-1)$ is not equal to $d_q \mod (q-1)$ (*because this would amount to faulting d_q in the naive algorithm*)
5. $S - S'_p \mod p$ is nonzero (*because this would amount to faulting the recombination modulo p in the naive algorithm*)
6. $S - S'_q \mod q$ is nonzero (*because this would amount to faulting the recombination modulo q in the naive algorithm*)
7. $S_{pt}^{d_q} \mod t$ is not equal to $S_{qt}^{d_p} \mod t$ (*this checks simultaneously for the integrity of S'_p and S'_q*)

Notice that the last verification could not have been done on the naive algorithm, and constitutes the added value for the Aumüller algorithm. These seven cases are *informally* assumed to protect the algorithm against the BellCoRe attacks. The criteria for fault detection is not to detect all faults; for instance, a fault on the final return of S (line 26) is not detected. However, of course, such a fault is not exploitable by a BellCoRe attack.

Remark 1. Some parts of the Aumüller algorithm are actually not intended to protect against fault injection attacks, but against side-channel analysis, such as differential power analysis (DPA). This is the case of lines 2 and 8 in Alg. 3. They can be removed if a minimalist protection against only fault injection attacks is looked for; but as they do not introduce weaknesses, they are simply kept as such.

Vigilant The CRT-RSA algorithm of Vigilant [16] also considers computations in a larger ring than $\mathbb{Z}_p$ (abbreviation for $\mathbb{Z}/p\mathbb{Z}$) and $\mathbb{Z}_q$, to enable verifications. In this case, a small random number r is cast, and computations are carried out in $\mathbb{Z}_{p\times r^2}$ and $\mathbb{Z}_{q\times r^2}$. In addition, the computations are now conducted not on the plain message m, but on an encoded message m', built using the CRT as the solution of those two requirements:

i: $m' \equiv m \mod N$, and
ii: $m' \equiv 1 + r \mod r^2$.

This system of equations has a single solution modulo $N \times r^2$, because N and r^2 are coprime. Such a representation allows to conduct in parallel the functional CRT-RSA (line i) and a verification (line ii). The verification is elegant, as it leverages this remarkable equality: $(1+r)^{d_p} = \sum_{i=0}^{d_p} \binom{d_p}{i} \cdot r^i \equiv 1 + d_p \cdot r \mod r^2$. Thus, as opposed to Aumüller *et al.*'s CRT-RSA, that requires one exponentiation (line 23 of Alg. 3), the verification of Vigilant's algorithm adds only one affine computation (namely $1 + d_p \mod r^2$).

The original description of Vigilant's algorithm involves some trivial computations on p and q, such as $p-1$, $q-1$ and $p \times q$. Those can be faulted, in such a way the BellCoRe attack becomes possible despite all the tests. Thus, a patch by Coron *et al.* has been released in [7] to avoid the reuse of $\widehat{p-1}$, $\widehat{q-1}$ and $\widehat{p \cdot q}$ in the algorithm.

Kim The authors Kim, Kim, Han and Hong propose in [11] a CRT-RSA algorithm that is based on a collaboration between a customized modular exponentiation and verifications at the recombination level based on Boolean operations. The underlying protection concepts being radically different from the algorithms of Shamir, Aumüller and Vigilant, we choose not to detail this interesting countermeasure.

In this paper, we will focus on three implementations, namely the naive one (Sec. 4), the one protected by Shamir countermeasure (Sec. 5), and the one with Aumüller *et al.* countermeasure (Sec. 6).

3 Formal Methods

For all the countermeasures presented in the previous section (Sec. 2), we can see that no formal proof of resistance against attacks is claimed. Informal arguments are given, that convince that for some attack scenarii, the attack attempts are detected hence harmless. Also, an analysis of the probability that an attack succeeds is carried out, however, this analysis strongly relies on assumptions on the faults distribution. Last but not least, the algorithms include protections against both passive side-channel attacks (SPA, DPA) and against active side-channel attacks, which makes it difficult to analyze for instance the minimal code to be added for the countermeasure to be correct.

Our goal is to prove that the proposed countermeasures work, *i.e.*, that they deliver a result that does leak information about neither p nor q (if the implementation is subject to fault injection) exploitable in a BellCoRe attack. In addition, we wish to reach this goal with the two following assumptions:

- our proof applies to a very general attacker model, and
- our proof applies to any implementation that is a (strict) refinement of the abstract algorithm.

First, we must define what computation is done, and what is our threat model.

Definition 1 (CRT-RSA). *The CRT-RSA computation takes as input a message m, assumed known by the attacker, and a secret key (p, q, d_p, d_q, i_q). Then, the implementation is free to instantiate any variable, but must return a result equal to: $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$, where:*

- $S_p = m^{d_p} \mod p$, *and*
- $S_q = m^{d_q} \mod q$.

Definition 2 (fault injection). *An attacker is able to request RSA computations, as per Definition 1. During the computation, the attacker can modify any intermediate value by setting it to either a random value or zero. At the end of the computation the attacker can read the result.*

Of course, the attacker cannot read the intermediate values used during the computation, since the secret key and potentially the modulus factors are used. Such "whitebox" attack would be too powerful; nonetheless, it is very hard in practice for an attacker to be able to access intermediate variables, due to protections and noise in the side-channel leakage (*e.g.*, power consumption, electromagnetic emanation). Remark that our model only take into account fault injection on data; the control flow is supposed not to be modifiable.

As a side remark, we notice that the fault injection model of Definition 2 corresponds to that of Vigilant ([16]), with the exception that the conditional tests can also be faulted. To summarize, an attacker can:

- modify a value in the global memory (*permanent fault*), and
- modify a value in a local register or bus (*transient fault*),

but cannot

- inject a permanent fault in the input data (message and secret key), nor
- modify the control flow graph.

The independence of the proofs on the algorithm implementation demands that the algorithm is described at a high level. The two properties that characterize the relevant level are as follows:

1. The description should be low level enough for the attack to work if protections are not implemented.
2. Any additional intermediate variable that would appear during refinement could be the target of an attack, but such a fault would propagate to an intermediate variable of the high level description, thereby having the same effect.

From those requirements, we deduce that:

1. The RSA description must exhibit the computation modulo p and q and the CRT recombination; typically, a completely blackbox description, where the computations would be realized in one go without intermediate variables, is not conceivable.
2. However, it can remain abstract, especially for the computational parts[2].

In our approach, the protections must thus be considered as an augmentation of the unprotected code, *i.e.*, a derived version of the code where additional variables are used. The possibility of an attack on the unprotected code attests that the algorithm is described at the adequate level, while the impossibility of an attack (to be proven) on the protected code shows that added protections are useful in terms of resistance to attacks.

[2] For instance a fault in the implementation of the multiplication (or the exponentiation) is either inoffensive, and we don't need to care about it, or it affects the result of the multiplication (or the exponentiation), and our model take it into account without going into the details of how the multiplication (or exponentiation) is computed.

Remark 2. The algorithm only exhibit evidence of safety. If after a fault injection, the algorithm does not simplify to an error detection, then it might only reveal that some simplification is missing. However, if it does not claim safety, it produces a *simplified* occurrence of a possible weakness to be investigated further.

Several tools are *a priori* suitable for a formalization of CRT-RSA. PARI/GP is a specialized computer algebra system, primarily aimed at solving number theory problems. Although PARI/GP can do a fair amount of symbolic manipulation, it remains limited compared to systems like Axiom, Magma, Maple, Mathematica, Maxima, or Reduce. Those last software also fall short to implement automatically number theoretic results like the Euler theorem. This explains why we developed from scratch a system to reason on modular numbers from a formal point of view. Our system is not general, in that it cannot for instance factorize terms in an expression. However, it is simply able to simplify recursively what is simplifiable from a set of unambiguous rules. This behavior happens to be suitable to the problem of resistance to fault attacks, because the redundancy that is added in the computation is meant to be simplified at the end (if no fault happened).

We describe the computation by a recursively defined term and we model it in OCaml [9][3] with an algebraic data type:

```
type term =
| Zero                          (* the constant zero *)
| One                           (* the constant one *)
| Named of string               (* a named number with no properties *)
| Num of int                    (* a number with no properties *)
| Prime of string               (* a named prime number *)
| Sum of term list              (* a sum of terms *)
| Prod of term list             (* a product of terms *)
| Opp of term                   (* the opposite of a term *)
| Inv of term                   (* the inverse of a term *)
| Mod of term * term            (* the remainder of division of a term by another *)
| Pow of term * term            (* the exponentiation of a term by another *)
| Let of string * term * term   (* the definition of a variable *)
| Let_ of string * term * term  (* the safe definition of a variable *)
| Var of string                 (* a reference to a variable *)
| If of term * term * term      (* a condition *)
```

There is no difference between `Named` and `Num` other than cosmetic, for display purpose. The `Num` constructor takes an `int` to ensure that OCaml sees each of them as a different value[4].

The definition of a variable (`Let`) consists of a string for the name of that variable, a term for its value, and a term in which the variable is defined (it writes

[3] We can only guarantee the validity of our tools by the simplicity of its code, and by making it free software (in the near future) so that it can be subject to review.

[4] Using a constructor with no argument, `Random` for instance, is not possible because OCaml would return *true* when comparing a `Random` with a `Random`.

like the `let x = v in e` form, which binds `x` to the value of the expression `v` in the expression `e`, in the OCaml programming language).

The safe definition of a variable (`Let_`) is the same thing except that we assume that there will be no fault in the value term.

An `If` conditional consists of three terms. Its value is the value of the second term if the first one is not zero, or of the third term otherwise[5].

Such a description of the computation, while abstracting the computational parts, allows to simplify the defined terms using rules from arithmetic and properties that we can deduce on the terms, such as being null, being null modulo another term, or being a multiple of another term.

We implement simplification as an OCaml function based on pattern-matching on the term. It applies most of the rules from arithmetic in the $\mathbb{Z}$ ring, and from modular arithmetic in the $\mathbb{Z}/n\mathbb{Z}$ rings. We omit factorization and expansion as they are not confluent operations in general. We also implement a few theorems such as the little Fermat's theorem and its generalization, *i.e.*, Euler's theorem. Of course we cannot do integer factorization to compute φ in our model so we raise an exception to handle cases where the exponent is not a product of prime numbers, but this actually never happens in well formed CRT-RSA computations, including computations with Shamir or Aumüller *et al.* countermeasure.

The simplification function is a recursive traversal of the term tree, and each step is very simple and easily verifiable, thus making it trustworthy. In particular, it is able to prove Proposition 1 and 2.

Injecting a fault in a computation amounts to replacing a subterm by zero or by a number with no properties, according to Definition 2. In the former case, the fault consists in zeroizing an intermediate variable (like in [6]). In the latter case, the fault consists in assuming that all the properties of the subterm are lost. Indeed, numbers with remarkable properties are extremely rare and thus the probability to create a property by a randomizing fault is negligible.

In our model, a fault can occur at any place in the computation. This is modeled by creating a faulted version of the term for each possible fault. To compute the nth faulted version, we traverse the term tree incrementing a counter at each recursive call, and when this counter's value is n, we return the fault (either `Zero` or `Num(`n`)`). When the computation of the faulted version is finished, the faulted term is compared to the original one. If they are the same, it means that the recursive traversing of the term tree was complete and that n is too large, which means we have generated all possible faulted versions of the term.

The faulted versions of the term we want to study are then used to check whether the properties required for the BellCoRe attack to be effective are respected.

Example 1. If we have the following term t which represents the computation $a + b \times c$: `Sum([ Named("a") ; Prod([ Named("b") ; Named("c") ]) ])`, it can be faulted in five different ways (using the randomizing fault):

[5] Remark: we do not apply any restrictions on the possible fault injections in any of the three arguments of the `If` constructor.

1. `Num(1)`, the final result is faulted;
2. `Sum([ Num(2) ; Prod([ Named("b") ; Named("c") ]) ])`, a is faulted;
3. `Sum([ Named("a") ; Num(3) ])`, the result of $b \times c$ is faulted;
4. `Sum([ Named("a") ; Prod([ Num(4) ; Named("c") ]) ])`, b is faulted;
5. `Sum([ Named("a") ; Prod([ Named("b") ; Num(5) ]) ])`, c is faulted.

If the properties that interest us is to know whether t is congruent with a modulo b, we can check if `Mod(Sum([ t, Opp(a) ]), b)` simplifies to `Zero`. Of course it will be true for t, but it will only be true for the fifth version of faulted t. If we had used the zeroing fault, it would also have been true for the third and fourth versions.

4 Study of an Unprotected CRT-RSA Computation

Here is the description of the naive CRT-RSA computation (Alg. 1). For readability reasons, `Named("x")` and `Prime("x")` have been replaced by `x`. `p` and `q` are prime numbers; `m` and `e` are numbers with no properties):

```
Let_("dp", Mod(Pow(e, Opp(One)), Sum([ p ; Opp(One) ])),
Let_("dq", Mod(Pow(e, Opp(One)), Sum([ q ; Opp(One) ])),
Let_("iq", Mod(Pow(q, Opp(One)), p),
Let("sp", Mod(Pow(m, Var("dp")), p),
Let("sq", Mod(Pow(m, Var("dq")), q),
Sum([ Var("sq")
    ; Prod([ q
           ; Mod(Prod([ Var("iq")
                      ; Sum([ Var("sp") ; Opp(Var("sq")) ]) ]),
                 p) ]) ]) )))))
```

The first three lines define d_p, d_q, and i_q. As we can see we use `Let_` rather than `Let` for these definitions, so the computation of the values of these variables cannot be faulted (since they are seen as inputs of the algorithm). After that, S_p and S_q are computed and then recombined in the last expression, as in Definition 1.

To test if the BellCoRe attack works on a faulted version $\widehat{S}$, we perform the following tests (we note $|S|$ for the simplified version of S):

1. Is $|S|$ equal to $|\widehat{S}|$?
2. Is $|S \mod p|$ equal to $|\widehat{S} \mod p|$?
3. Is $|S \mod q|$ equal to $|\widehat{S} \mod q|$?

If the first test is false and at least one of the second and third is true, we have a BellCoRe attack, as seen in Sec. 2.

There are 27 different possible faults in our model of the unprotected CRT-RSA, 17 of which allows a BellCoRe attack using the randomizing fault, and 19 with the zeroing fault. These results are obtained almost instantaneously by our tool.

As an example, replacing the intermediate variable holding the value of $i_q \cdot (S_p - S_q) \mod p$ in the final expression with zero or a random value makes the first and second tests false, and the last one true, and thus allows a BellCoRe attack.

5 Study of the Shamir Countermeasure

The description of the computation of CRT-RSA with Shamir countermeasure (Alg. 2) can be found in App. A.

Using the same method as for the unprotected implementation of CRT-RSA, we can prove that on the 75 different possible faults, 21 allows a BellCoRe attack, whether using a randomizing fault or a zeroing fault (these results are obtained almost instantaneously by our tool). This is not really surprising, as the test which is done on line 10 of Alg. 2 does not verify if a fault is injected during the computations of S_p and S_q, nor during their recombination in S. For instance zeroing or randomizing the intermediate variable holding the result of $S_p - S_q$ during the computation of S (line 9 of Alg. 2) result in a BellCoRe attack.

During our study of this countermeasure, we remarked that if the attacker can modify the value of an intermediate variable only for one use of this variable (*transient fault*), we can do more attacks. In practice, it would translate into faulting the variable when it is read (*e.g.*, in a register or on a bus), rather than in (persistent) memory. This behavior could also be the effect of a fault injection in cache, which is later replaced with the good value when it is read from memory again. To the authors knowledge, these are not impossible situations. Nonetheless, growing the power of the attacker to take that into account break some very important assumptions that are classical (sometimes even implicit) in the literature. It does not matter that the parts of the secret key are stored in a secure "key container" if their values can be a faulted at read time. Indeed, allowing this kind of fault enable even more BellCoRe attacks on a CRT-RSA computation protected by the Shamir countermeasure. For instance, if the value of p is randomized for the computation of the value of S_p (line 7 of Alg. 2), then we have $S \neq \widehat{S}$, but also $S \equiv \widehat{S} \mod q$, which enables a BellCoRe attack, as seen in Sec. 2.

It is often asserted that the countermeasure of Shamir is unpractical due to its need for d (as mentioned in [1] and [16]), and because there is a possible fault attack on the recombination, *i.e.*, line 9 of Alg. 2 (as mentioned in [16]). However, the attack on the recombination can easily be checked, by testing that $S - S_p \not\equiv 0 \mod p$ and $S - S_q \not\equiv 0 \mod q$ before returning the result. Notwithstanding, to our best knowledge, it is difficult to detect the attack our tool found (described in the previous paragraph), and so the existence of this attack (new, in the sense it has not been described previously) is a compelling reason for not implementing Shamir's CRT-RSA.

6 Study of the Aumüller *et al.* Countermeasure

The description of the computation of CRT-RSA with Aumüller *et al.* countermeasure (Alg. 3) is quite large and can thus be found in the App. B.

Using the same method as before, we can prove that on the 145 different possible faults, *none* allows a BellCoRe attack, whether the fault is zero or random (these results are obtained in very few seconds by our tools). This is a proof that the Aumüller *et al.* countermeasure works when there is one fault[6].

We also tried to remove some of the tests that the countermeasure consists of. It appears that *two of them are unnecessary*: the first ones of lines 4 and 10 in the Aumüller CRT-RSA as presented in Alg. 3. These two tests are actually redundant with the two tests of line 16 and the test of line 23 of Alg. 3 which also verify the integrity of p' and q' by using them indirectly. Removing these tests is not very useful in terms of performances, but their uselessness shows the need for formal studies in the field of implementation security, even if they might appear unnatural at first.

Since it allowed more attacks on the Shamir countermeasure, we also tested the Aumüller *et al.* countermeasure against *transient fault* such as described in Sec. 5. It happens that Aumüller *et al.* is resistant against such fault injections too.

Our methods also confirmed that the computation of d_p, d_q, and i_q (in terms of p, q, and d) must not be part of the algorithm. The countermeasure effectively needs these three variables to be inputs of the algorithm to work properly. For instance there is a BellCoRe attack if d_q happens to be zeroed. However, even with d_p, d_q, and i_q as inputs, we can still attempt to attack a CRT-RSA implementation protected by the Aumüller *et al.* countermeasure by doing more than one fault.

Our results are as follows. With more than one fault it is obvious that the countermeasure can be dodged if one of the fault is a zeroing of an intermediate variable which is used as condition in one of the useful tests. However we were able to prove that Aumüller *et al.* countermeasure is still efficient if there is two or even three randomizing faults. The computations for two and three faults took respectively a few minutes and a few dozen of minutes.

7 Conclusions and Perspectives

We have formally proven the resistance of the Aumüller *et al.* countermeasure against the BellCoRe attack by fault injection on CRT-RSA. To our knowledge, it is the first time that a formal proof of security is done for a BellCoRe countermeasure.

[6] This result is worthwhile some emphasis: the genuine algorithm of Aumüller is thus *proved* resistant against single fault attacks. At the opposite, the CRT-RSA algorithm of Vigilant is not immune to single fault attacks (refer to [7]), and the corrections suggested in the same paper by Coron *et al.* have not yet been proved.

The proof enables us to show that two out of seven of the tests done by Aumüller *et al.* countermeasure are unnecessary, thereby simplifying the protected computation of CRT-RSA.

During our research, we have raised several questions about the assumptions traditionally made by countermeasures. The possibility of fault at read time is, in particular, responsible for many vulnerabilities. The possibility of such fault means that part of the secret key can be faulted (even if only for one computation). It allows interesting BellCoRe attacks on a computation of CRT-RSA protected by Shamir countermeasure.

The first of these two points demonstrates the lack of formal studies of fault injection attack and their countermeasures, while the second one shows the importance of formal proofs in the field of implementation security.

As a first perspective, we would like to address the hardening of software codes of CRT-RSA under the threat of a bug attack. This attack has been introduced by Biham, Carmeli and Shamir [2] at CRYPTO 2008. It assumes that a hardware has been trapped in such a way that there exists two integers a and b, for which the multiplication is incorrect. In this situation, Biham, Carmeli and Shamir mount an explicit attack scenario where the knowledge of a and b is leveraged to produce a faulted result, that can lead to a remote BellCoRe attack. For sure, testing for the correct functionality of the multiplication operation is impractical (it would amount to an exhaustive verification of 2^{128} multiplications on 64 bit computer architectures). Thus, it can be imagined to use a countermeasure, like that of Aumüller, to detect a fault (caused logically). Our aim would be to assess in which respect our fault analysis formal framework allows to validate the security of the protection. Indeed, a fundamental difference is that the fault is not necessarily injected at *one* random place, but can potentially show up at *several* places.

As another perspective, we would like to handle the repaired countermeasure of Vigilant [7] and the countermeasure of Kim [11]. Regarding Vigilant, the difficulty our verification framework in OCaml shall overcome is to decide how to inject the remarkable identity $(1+r)^{d_p} \equiv 1 + d_p \cdot r \mod r^2$: either it is kept as such such, like an *ad hoc* theorem (but we need to make sure it is called only at relevant places, since it is not confluent), or it is made more general (but we must ascertain that the verification remains tractable). However, this effort is worthwhile, because the authors themselves say in the conclusion of their article [7] that:

> *"Formal proof of the FA-resistance of Vigilant's scheme including our countermeasures is still an open (and challenging) issue."*

Regarding the CRT-RSA algorithm from Kim, the computation is very detailed (it goes down to the multiplication level), and involves Boolean operations (and, xor, *etc.*), so more expertise about both arithmetic and logic must be added to our software.

Eventually, we wish to answer a question raised by Vigilant [16] about the prime t involved in Aumüller *et al.* countermeasure:

"Is it fixed or picked at random in a fixed table?"

The underlying issue is that of *replay* attacks on CRT-RSA, that are more complicated to handle; indeed, they would require a formal system such as ProVerif [5], that is able to prove interactive protocols.

Concerning the tools we developed during our research, they currently only allow to study fault injection in the data, and not in the control flow, it would be interesting to enable formal study of fault injections affecting the control flow. We would also like to make these tools usable by anyone, which will require the creation of a better DSL for describing computations and attacks, as well as a nice user interface to our code, which is still in "research code" stage for now.

References

1. Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In Burton S. Kaliski, Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2002.
2. Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In *CRYPTO*, volume 5157 of *LNCS*, pages 221–240. Springer, 2008. Santa Barbara, CA, USA.
3. Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO*, volume 1294 of *LNCS*, pages 513–525. Springer, August 1997. Santa Barbara, California, USA. DOI: 10.1007/BFb0052259.
4. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Proceedings of Eurocrypt'97*, volume 1233 of *LNCS*, pages 37–51. Springer, May 11-15 1997. Konstanz, Germany. DOI: 10.1007/3-540-69053-0_4.
5. Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.
6. Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. Formal verification of an implementation of CRT-RSA Vigilant's algorithm. *Journal of Cryptographic Engineering*, 3(3), 2013. DOI: 10.1007/s13389-013-0049-3.
7. Jean-Sébastien Coron, Christophe Giraud, Nicolas Morin, Gilles Piret, and David Vigilant. Fault Attacks and Countermeasures on Vigilant's RSA-CRT Algorithm. In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *FDTC*, pages 89–96. IEEE Computer Society, 2010.
8. Xiaofei Guo, Debdeep Mukhopadhyay, and Ramesh Karri. Provably secure concurrent error detection against differential fault analysis. Cryptology ePrint Archive, Report 2012/552, 2012. http://eprint.iacr.org/2012/552/.
9. INRIA. OCaml, a variant of the Caml language. http://caml.inria.fr/ocaml/index.en.html.
10. Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *J. Cryptology*, 12(4):241–245, 1999.
11. Sung-Kyoung Kim, Tae Hyun Kim, Dong-Guk Han, and Seokhie Hong. An efficient CRT-RSA algorithm secure against power and fault attacks. *J. Syst. Softw.*, 84:1660–1669, October 2011.
12. Çetin Kaya Koç. High-Speed RSA Implementation, November 1994. Version 2, ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf.
13. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
14. Adi Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks, November 1999. Patent Number 5,991,415; also presented at the rump session of EUROCRYPT '97.
15. Mohammad Tehranipoor and Cliff Wang, editors. *Introduction to Hardware Security and Trust*. Springer, 2012. ISBN 978-1-4419-8079-3.
16. David Vigilant. RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

A Description of Shamir Implementation of CRT-RSA

For readability purpose, `Named("x")` and `Prime("x")` have been replaced by `x`; `p`, `q`, and `r` are prime numbers; `m`, `d`, and `ERROR` are numbers with no properties):

```
Let_("iq", Mod(Pow(q, Opp(One)), p),
Let("p'", Prod([ p ; r ]),
Let("dp", Mod(d, Prod([ Sum([ p ; Opp(One) ]) ; Sum([ r ; Opp(One) ]) ])),
Let("S'p", Mod(Pow(m, Var("dp")), Var("p")),
Let("q'", Prod([ q ; r ]),
Let("dq", Mod(d, Prod([ Sum([ q ; Opp(One) ]) ; Sum([ r ; Opp(One) ]) ])),
Let("S'q", Mod(Pow(m, Var("dq")), Var("q")),
Let("Sp", Mod(Var("S'p"), p),
Let("Sq", Mod(Var("S'q"), q),
Let("S", Sum([ Var("Sq")
             ; Prod([ q
                    ; Mod(Prod([ Var("iq")
                               ; Sum([ Var("Sp") ; Opp(Var("Sq")) ]) ]),
                          p) ]) ]),
If(Mod(Sum([ Var("S'p") ; Opp(Var("S'q")) ]), r),
   ERROR,
Var("S")))))))))))))
```

B Description of Aumüller *et al.* Implementation of CRT-RSA

For readability purpose `Named("x")` and `Prime("x")` have been replaced by `x` (`m`, `e`, `Random1`, `Random2`, and `ERROR` are numbers with no properties; and `p`, `q`, and `t` are prime numbers).

```
Let_("dp", Mod(Pow(e, Opp(One)), Sum([ p ; Opp(One) ])),
Let_("dq", Mod(Pow(e, Opp(One)), Sum([ q ; Opp(One) ])),
Let_("iq", Mod(Pow(q, Opp(One)), p),
Let("p'", Prod([ p ; t ]),
Let("d'p", Sum([ Var("dp") ; Prod([ Random1 ; Sum([ p ; Opp(One) ]) ]) ]),
Let("s'p", Mod(Pow(m, Var("d'p")), Var("p")),
If(Mod(Var("p'"), p),
   ERROR,
If(Mod(Sum([ Var("d'p") ; Opp(Var("dp")) ]), Sum([ p ; Opp(One) ])),
   ERROR,
Let("q'", Prod([ q ; t ]),
Let("d'q", Sum([ Var("dq") ; Prod([ Random2 ; Sum([ q ; Opp(One) ]) ]) ]),
Let("s'q", Mod(Pow(m, Var("d'q")), Var("q")),
If(Mod(Var("q'"), q),
   ERROR,
If(Mod(Sum([ Var("d'q") ; Opp(Var("dq")) ]), Sum([ q ; Opp(One) ])),
   ERROR,
Let("sp", Mod(Var("s'p"), p),
Let("sq", Mod(Var("s'q"), q),
```

```
Let("S", Sum([ Var("sq")
             ; Prod([ q
                    ; Mod(Prod([ Var("iq")
                               ; Sum([ Var("sp")
                                     ; Opp(Var("sq")) ]) ]), p) ]) ]),
If(Mod(Sum([ Var("S") ; Opp(Var("s'p")) ]), p),
   ERROR,
If(Mod(Sum([ Var("S") ; Opp(Var("s'q")) ]), q),
   ERROR,
Let("spt", Mod(Var("s'p"), t),
Let("sqt", Mod(Var("s'q"), t),
Let("dpt", Mod(Var("d'p"), Sum([ t ; Opp(One) ])),
Let("dqt", Mod(Var("d'q"), Sum([ t ; Opp(One) ])),
If(Mod(Sum([ Pow(Var("spt"), Var("dqt"))
           ; Opp(Pow(Var("sqt"), Var("dpt"))) ]), t),
   ERROR,
Var("S"))))))))))))))))))))))))))
```