HAL
open science

Formal Analysis of CRT-RSA Vigilant's Countermeasure Against the BellCoRe Attack: A Pledge for Formal Methods in the Field of Implementation Security

Pablo Rauzy, Sylvain Guilley

To cite this version:

Pablo Rauzy, Sylvain Guilley. Formal Analysis of CRT-RSA Vigilant's Countermeasure Against the BellCoRe Attack: A Pledge for Formal Methods in the Field of Implementation Security. Program Protection and Reverse Engineering Workshop 2014, Jan 2014, San Diego, CA, United States. pp.Article No. 2, 10.1145/2556464.2556466 . hal-00939473v2

HAL Id: hal-00939473

https://hal.science/hal-00939473v2

Submitted on 23 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Analysis of CRT-RSA Vigilant's Countermeasure Against the BellCoRe Attack

A Pledge for Formal Methods in the Field of Implementation Security

Pablo Rauzy — Sylvain Guilley
Institut Mines-Télécom ; Télécom ParisTech ; CNRS LTCI
{*firstname.lastname*}@telecom-paristech.fr

Abstract

In our paper at PROOFS 2013, we formally studied a few known countermeasures to protect CRT-RSA against the BellCoRe fault injection attack. However, we left Vigilant's countermeasure and its alleged repaired version by Coron *et al.* as future work, because the arithmetical framework of our tool was not sufficiently powerful. In this paper we bridge this gap and then use the same methodology to formally study both versions of the countermeasure. We obtain surprising results, which we believe demonstrate the importance of formal analysis in the field of implementation security. Indeed, the original version of Vigilant's countermeasure is actually broken, but not as much as Coron *et al.* thought it was. As a consequence, the repaired version they proposed can be simplified. It can actually be simplified even further as two of the nine modular verifications happen to be unnecessary. Fortunately, we could formally prove the simplified repaired version to be resistant to the BellCoRe attack, which was considered a "challenging issue" by the authors of the countermeasure themselves.

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

1 Introduction

Private information protection is a highly demanded feature, especially in the context of global defiance against most infrastructures, assumed to be controlled by governmental agencies. Properly used cryptography is known to be a key building block for secure information exchange. However, in addition to the threat of cyber-attacks, implementation-level hacks are also to be considered seriously. This article deals specifically with the protection of a *decryption* or *signature* crypto-system (called RSA [RSA78]) in the presence of hardware attacks (*e.g.*, we assume the attacker can alter the RSA computation while it is being executed).

It is known since 1997 [BDL97] 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. Last year we formally studied some of them [RG13]. We were able to formally prove that the unprotected implementation of CRT-RSA as well as the countermeasure proposed by Shamir [Sha99] are broken, and that Aumüller *et al.*'s countermeasure [ABF$^+$02] is resistant to the BellCoRe attack. However, we were not able to study Vigilant's countermeasure [Vig10, Vig08] or its repaired version by Coron *et al.* [CGM$^+$10], because the tool we developed lacked the ability to work with arithmetical properties necessary to handle these countermeasures. In particular, our framework was unaware of the easiness to compute the discrete logarithm in some composite degree residuosity classes. These difficulties were foreseen by Coron *et al.* themselves in the conclusion of their paper [CGM$^+$10]:

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

This is precisely the purpose of this paper. The state-of-the-art of formal proofs of Vigilant's countermeasure is the work of Christofi, Chetali, Goubin, and Vigilant [CCGV13]. However, for tractability reasons, the proof is conducted on reduced versions of the algorithms parameters. One fault model is chosen authoritatively (*viz.* the zeroization of a complete intermediate data), which is a strong assumption. In addition, the verification is conducted on a specific implementation in pseudocode, hence concerns about its portability after compilation into machine-level code.

Contributions. We improved finja (our tool based on symbolic computation in the framework of modular arithmetic [RG13]) to enable the formal study of CRT-RSA Vigilant's countermeasure against the BellCoRe attack. The finja tool allows a full fault coverage of CRT-RSA algorithm, thereby keeping the proof valid even if the code is transformed (*e.g.*, optimized, compiled, partitioned in software/hardware, or equipped with dedicated countermeasures). We show that the original countermeasure [Vig08] is indeed broken, but not as much as Coron *et al.* thought it was when they proposed a *manually* repaired version of it [CGM$^+$10]. We simplify the repaired version accordingly, and formally prove it resistant to 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, *e.g.*, during their transit on buses or when they reside on register banks), with or without forcing at zero, and with possibly faults at various locations. Thanks to the formal analysis, we are able to simplify the countermeasure even further: two of the nine checks are unnecessary.

Organization of the paper. We recall CRT-RSA cryptosystem and the BellCoRe attack in Sec. 2. Vigilant's countermeasure and its variant by Coron *et al.* are exposed in Sec. 3. In Sec. 4 we detail the modular arithmetic framework used by our tool. The results of our analysis are presented in Sec. 5. Finally, conclusions and perspectives are given in Sec. 6. After that, Appx. A details the practical issues related to fault injection analysis, for single- and multiple-fault attacks.

2 CRT-RSA and the BellCoRe Attack

This section recaps known results about fault attacks on CRT-RSA (see also [Koç94], [TW12, Chap. 3] and [JT11, Chap. 7 & 8]). 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 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 Euler's theorem[1]. In this equation, φ is Euler's 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 only pretending to know d. 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 (see PKCS #1 v2.1, §3.1).

[1]We use the usual convention in all mathematical equations, namely that the "mod" operator has the lowest binding precedence, *i.e.*, $a \times b \mod c \times d$ represents the element $a \times b$ in $\mathbb{Z}_{c \times d}$.

2.2 CRT-RSA

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 $2,048$ bits RSA, p and q are $1,024$ bits 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 $1,024$ bits, 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:

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

The 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.

Algorithm 1: Unprotected 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

2.3 The BellCoRe Attack

In 1997, an dreadful remark has been made by Boneh, DeMillo and Lipton [BDL97], three staff of BellCoRe: Alg. 1 could reveal the secret primes p and q if the line 1 or 2 of the computation is faulted, even in a very random way. The attack can be expressed as the following proposition.

Proposition 1 (Original BellCoRe attack)**.** *If the intermediate variable S_p (resp. S_q) is returned faulted as $\widehat{S_p}$ (resp. $\widehat{S_q}$)*[2]*, 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 any 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,

[2]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 $\widehat{X^e}$).

- 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
$S - \widehat{S} = q \cdot ((i_q \cdot (S_p - S_q) \mod p) - (i_q \cdot (\widehat{S_p} - S_q) \mod p))$, 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 $(q \mod p) \cdot i_q \equiv 1 \mod p$, and thus $S - \widehat{S}$ is a multiple of p. Additionally, $S - \widehat{S}$ is not a multiple of q.
So, $\gcd(N, S - \widehat{S}) = p$. □

3 Vigilant Countermeasure Against the BellCoRe Attack

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 several other countermeasures have been proposed.

Until recently, none of these countermeasures had been proved. In 2013, Christofi, Chetali, Goubin, and Vigilant [CCGV13] formally proved an implementation of Vigilant's countermeasure. However, 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 specific implementation in pseudocode, hence concerns about its portability after compilation into machine-level code. The same year, we formally studied [RG13] the ones of Shamir [Sha99] and Aumüller *et al.* [ABF+02] using a tool based on the framework of modular arithmetic, thus offering a full fault coverage on CRT-RSA algorithm, thereby keeping the proof valid even if the code is transformed (*e.g.*, optimized, compiled or partitioned in software/hardware). We proved the former to be broken and the latter to be resistant to the BellCoRe attack.

3.1 Vigilant's Original Countermeasure

In his paper at CHES 2008, Vigilant [Vig08] proposed a new method to protect CRT-RSA. Its principle is to compute $M^d \mod N$ in $\mathbb{Z}_{Nr^2}$ where r is a random integer that is coprime with N. Then M is transformed into M^* such that

$$M^* \equiv \begin{cases} M \mod N, \\ 1 + r \mod r^2, \end{cases}$$

which implies that

$$S^* = M^{*d} \mod Nr^2 \equiv \begin{cases} M^d \mod N, \\ 1 + dr \mod r^2. \end{cases}$$

The latter results are based on the binomial theorem, which states that:

$$\begin{aligned} (1+r)^d &= \sum_{k=0}^{d} \binom{d}{k} r^k \\ &= 1 + dr + \binom{d}{2} r^2 + \text{higher powers of } r, \end{aligned} \tag{1}$$

which simplifies to $1 + dr$ in the ring $\mathbb{Z}_{r^2}$. This property can be seen as a special case of an easy computation of a discrete logarithm in a composite degree residuosity class (refer for instance to [Pai99]). Therefore the result S^* can be checked for consistency modulo r^2. If the verification $S^* \stackrel{?}{=} 1 + dr \mod r^2$ succeeds, then

the final result $S = S^* \mod N$ is returned. Concerning the random numbers used in the countermeasure, its author recommend using a 32 bits integer for r, and 64 bits for the R_is. The original algorithm for Vigilant's countermeasure is presented in Alg. 49.

3.2 Coron *et al.* Repaired Version

At FDTC 2010, Coron *et al.* [CGM+10] proposed some corrections to Vigilant's original countermeasure. They claimed to have found three weaknesses in integrity verifications:

1. one for the computation modulo $p - 1$;
2. one for the computation modulo $q - 1$;
3. and one in the final check modulo Nr^2.

They propose a fix for each of these weaknesses. To correct weakness #1, lines 11 to 15 in Alg. 49 becomes:

Algorithm 3: Vigilant's CRT-RSA: Coron *et al.* fix #1

```
1 p_minusone = p − 1
2 d'_p = d_p + R_1 · p_minusone
3 S_pr = M'_p^{d'_p} mod p'
4 p_minusone = p − 1
5 if d'_p ≢ d_p mod p_minusone then
6 |  return error
7 end
```

Similarly, for weakness #2, lines 32 to 36 in Alg. 49:

Algorithm 4: Vigilant's CRT-RSA: Coron *et al.* fix #2

```
1 q_minusone = q − 1
2 d'_q = d_q + R_2 · q_minusone
3 S_qr = M'_q^{d'_q} mod q'
4 q_minusone = q − 1
5 if d'_q ≢ d_q mod q_minusone then
6 |  return error
7 end
```

These two fixes immediately looked suspicious to us. Indeed, Coron *et al.* suppose that the computations of $p - 1$ and $q - 1$ are factored as an optimization, while they are not in the original version of Vigilant's countermeasure. This kind of optimization, called "common subexpression elimination", is very classical but is, as for most speed-oriented optimization, not a good idea in security code. We will see in Sec. 5 that the original algorithm, which did not include this optimization, was not at fault here.

The third fix proposed by Coron *et al.* consists in transforming lines 42 to 45 of Alg. 49 into:

Algorithm 5: Vigilant's CRT-RSA: Coron *et al.* fix #3

```
1 N = p · q
2 if p · q · (S − R_4 − q · iq · (R_3 − R_4)) ≢ 0 mod Nr^2 then
3 |  return error
4 end
```

What has changed is the recomputation of $p \cdot q$ instead of reusing N in the verification. The idea is that if p or q is faulted when N is computed then error will be returned. In the original countermeasure, if p or q were to be faulted, the faulted version of N would appear in the condition twice, simplifying itself out and thus hiding the fault.

Algorithm 2: Vigilant's CRT-RSA

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

Choose random numbers r, R_1, R_2, R_3, and R_4.

$p' = pr^2$
$M_p = M \mod p'$
$i_{pr} = p^{-1} \mod r^2$
$B_p = p \cdot i_{pr}$
$A_p = 1 - B_p \mod p'$
$M'_p = A_p M_p + B_p \cdot (1 + r) \mod p'$

if $M'_p \not\equiv M \mod p$ **then**
  return error
end

$d'_p = d_p + R_1 \cdot (p - 1)$
$S_{pr} = {M'_p}^{d'_p} \mod p'$

if $d'_p \not\equiv d_p \mod p - 1$ **then**
  return error
end

if $B_p S_{pr} \not\equiv B_p \cdot (1 + d'_p r) \mod p'$ **then**
  return error
end

$S'_p = S_{pr} - B_p \cdot (1 + d'_p r - R_3)$

$q' = qr^2$
$M_q = M \mod q'$
$i_{qr} = q^{-1} \mod r^2$
$B_q = q \cdot i_{qr}$
$A_q = 1 - B_q \mod q'$
$M'_q = A_q M_q + B_q \cdot (1 + r) \mod q'$

if $M'_q \not\equiv M \mod q$ **then**
  return error
end

if $M_p \not\equiv M_q \mod r^2$ **then**
  return error
end

$d'_q = d_q + R_2 \cdot (q - 1)$
$S_{qr} = {M'_q}^{d'_q} \mod q'$

if $d'_q \not\equiv d_q \mod q - 1$ **then**
  return error
end

if $B_q S_{qr} \not\equiv B_q \cdot (1 + d'_q r) \mod q'$ **then**
  return error
end

$S'_q = S_{qr} - B_q \cdot (1 + d'_q r - R_4)$

$S = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
$N = pq$

if $N \cdot (S - R_4 - q \cdot i_q \cdot (R_3 - R_4)) \not\equiv 0 \mod Nr^2$ **then**
  return error
end

if $q \cdot i_q \not\equiv 1 \mod p$ **then**
  return error
end

return $S \mod N$

4 Formal Methods

Our goal is to prove that the proposed countermeasures work, *i.e.*, that they deliver a result that does not 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.

4.1 CRT-RSA and Fault Injection

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 but which we consider to be random, 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 Def. 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 reliably to intermediate variables, due to protections and noise in the side-channel leakage (*e.g.*, power consumption, electromagnetic emanation). Remark that our model only takes into account fault injection on data; the control flow is supposed not to be modifiable.

Remark 1. We notice that the fault injection model of Def. 2 corresponds to that of Vigilant [Vig08], 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. For instance a fault in the implementation of the multiplication (or the exponentiation) is either inoffensive, and we do not need to care about it, or it affects the result of the multiplication (or the exponentiation), and our model takes it into account without going into the details of how the multiplication (or exponentiation) is computed.

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.

Remark 2. The algorithm only exhibits 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.

4.2 How finja Works

Our tool[3], finja, works within the framework of modular arithmetic, which is the mathematical framework of CRT-RSA computations. The general idea is to represent the computation term as a tree which encodes the computation properties. Our tool then does *symbolic computation* to simplify the term. This is done by *term rewriting* it, using rules from arithmetic and the properties encoded in the tree. Fault injections in the computation term are simulated by changing the properties of a subterm, thus impacting the simplification process. An attack success condition is also given and used on the term resulting from the simplification to check whether the corresponding attack works on it. For each possible fault injection, if the attack success condition (which may reference the original term as well as the faulted one) can be simplified to *true* then the attack is said to work with this fault, otherwise the computation is protected against it. The outputs of our tool are in HTML form: easily readable reports are produced, which contains all the information about the possible fault injections and their outcome.

4.2.1 Computation Term

The computation is expressed in a convenient statement-based input language. This language's Backus Normal Form is given in Fig. 1.

A computation term is defined by a list of statements finished by a `return` statement. Each statement can either:

- declare a variable with no properties (line 3);
- declare a variable which is a prime number (line 4);
- declare a variable by assigning it a value (line 5), in this case the properties of the variable are the properties of the assigned expression;
- perform a verification (line 6).

As can be seen in lines 9 to 15, an expression can be:

- zero, one, or an already declared variable;
- the sum (or difference) of two expressions;
- the product of two expressions;

[3] `http://pablo.rauzy.name/sensi/finja.html`

```
1  term    ::= ( stmt )* 'return' mp_expr ';'
2  stmt    ::= ( decl | assign | verif ) ';'
3  decl    ::= 'noprop' mp_var ( ',' mp_var )*
4            | 'prime' mp_var ( ',' mp_var )*
5  assign  ::= var ':=' mp_expr
6  verif   ::= 'if' mp_cond 'abort with' mp_expr
7  mp_expr ::= '{' expr '}' | expr
8  expr    ::= '(' mp_expr ')'
9            | '0' | '1' | var
10           | '-' mp_expr
11           | mp_expr '+' mp_expr
12           | mp_expr '-' mp_expr
13           | mp_expr '*' mp_expr
14           | mp_expr '^' mp_expr
15           | mp_expr 'mod' mp_expr
16 mp_cond ::= '{' cond '}' | cond
17 cond    ::= '(' mp_cond ')'
18           | mp_expr '=' mp_expr
19           | mp_expr '!=' mp_expr
20           | mp_expr '=[' mp_expr ']' mp_expr
21           | mp_expr '!=[' mp_expr ']' mp_expr
22           | mp_cond '/\' mp_cond
23           | mp_cond '\/' mp_cond
24 mp_var  ::= '{' var '}' | var
25 var     ::= [a-zA-Z][a-zA-Z0-9_']*
```

Figure 1: BNF of our tool's input language.

- the exponentiation of an expression by another;
- the modulus of an expression by another.

The condition in a verification can be (lines 18 to 23):

- the equality or inequality of two expressions;
- the equivalence or non-equivalence of two expressions modulo another (lines 20 and 21);
- the conjunction or disjunction of two conditions.

Optionally, variables (when declared using the `prime` or `noprop` keywords), expressions, and conditions can be protected (lines 3, 4, 7, and 16, `mp` stands for "maybe protected") from fault injection by surrounding them with curly braces. This is useful for instance when it is necessary to express the properties of a variable which cannot be faulted in the studied attack model. For example, in CRT-RSA, the definitions of variables d_p, d_q, and i_q are protected because they are seen as input of the computation.

Finally, line 25 gives the regular expression that variable names must match (they start with a letter and then can contain letters, numbers, underscore, and simple quote).

After it is read by our tool, the computation expressed in this input language is transformed into a tree (just like the abstract syntax tree in a compiler). This tree encodes the arithmetical properties of each of the intermediate variable, and thus its dependencies on previous variables. Properties of intermediate variables can be everything that is expressible in the input language. For instance, being null or being the product of other terms (and thus, being a multiple of each of them), are possible properties.

An example of usage can be found in Appx. B, where the original version of Vigilant's countermeasure is written in this language. Note the evident similarity with the pseudocode of Alg. 49.

4.2.2 Fault Injection

A fault injection on an intermediate variable is represented by changing the properties of the subterm (a node and its whole subtree in the tree representing the computation term) that represents it. In the case of a fault which forces at zero, then the whole subterm is replaced by a term which only has the property of being null. In the case of a randomizing fault, by a term which have no properties.

Our tool simulates *all the possible fault injections* of the attack model it is launched with. The parameters allow to choose:

- *how many faults* have to be injected (however, the number of tests to be done is impacted by a factorial growth with this parameter, as is the time needed to finish the computation of the proof);
- *the type* of each fault (*randomizing* or *zeroing*);
- if *transient* faults are possible or if only *permanent* faults should be performed.

4.2.3 Attack Success Condition

The attack success condition is expressed using the same condition language as presented in Sec. 4.2.1. It can use any variable introduced in the computation term, plus two special variables `_` and `@` which are respectively bound to the expression returned by the computation term as given by the user and to the expression returned by the computation with the fault injections. This success condition is checked for each possible faulted computation term.

4.2.4 Simplification Process

The simplification is implemented as a recursive traversal of the term tree, based on pattern-matching. It works just like a naive interpreter would, except it does symbolic computation only, by rewriting the term using rules from arithmetic. Simplifications are carried out in the $\mathbb{Z}$ ring, and its $\mathbb{Z}_N$ subrings. The tool knows how to deal with most of the $\mathbb{Z}$ ring axioms:

- the neutral elements (0 for sums, 1 for products);
- the absorbing element (0, for products);
- inverses and opposites;
- associativity and commutativity.

However, it does not implement distributivity as it is not confluent. Associativity is implemented by flattening as much as possible ("removing" all unnecessary parentheses), and commutativity is implemented by applying a stable sorting algorithm on the terms of products or sums.

The tool also knows about most of the properties that are particular to $\mathbb{Z}_N$ subrings and applies them when simplifying a term modulo N:

- identity:
  - $(a \mod N) \mod N = a \mod N$,
  - $N^k \mod N = 0$;
- inverse:
  - $(a \mod N) \times (a^{-1} \mod N) \mod N = 1$,
  - $(a \mod N) + (-a \mod N) \mod N = 0$;
- associativity and commutativity:

- $(b \mod N) + (a \mod N) \mod N = a + b \mod N$,
- $(a \mod N) \times (b \mod N) \mod N = a \times b \mod N$;

- subrings: $(a \mod N \times m) \mod N = a \mod N$.

In addition to those properties a few theorems are implemented to manage more complicated cases where the properties are not enough when conducting symbolic computations:

- Fermat's little theorem;
- its generalization, Euler's theorem;
- Chinese remainder theorem;
- a particular case of the binomial theorem (see Sec. 3.1).

The tool also implements a lemma that is used for modular (in)equality verifications (note that a or b can be 0): $a \times x \stackrel{?}{\equiv} b \times x \mod N \times x$ is simplified to the equivalent test $a \stackrel{?}{\equiv} b \mod N$.

5 Analysis of Vigilant's Countermeasure

This section presents and discusses the results of our formal analysis of Vigilant's countermeasure.

5.1 Original and Repaired Version

The original version of Vigilant's countermeasure in our tool's input language can be found in Appx. B.

In a single fault attack model, we found that the countermeasure has a single point of failure: if transient fault are possible, then if p or q are randomized in the computation of N at line 52, then a BellCoRe attack works.

The repaired version by Coron *et al.* thus does unnecessary modifications, since it fixes three spotted weaknesses (as seen in Sec. 3.2) while there is only one to fix. Only implementing the modification of Alg. 5 is sufficient to be provably protected against the BellCoRe attack.

After that, we went on to see what would happen in a multiple fault model. What we found is the object of the next section.

5.2 Our Fixed and Simplified Version

Attacking with multiple faults when at least one of the fault is a zeroing fault, means that some verification can be skipped by having their condition zeroed. We found that two of the tests could be skipped without impacting the protection of the computation (the lines 38 and 57). Both tests have been removed from our fixed and simplified version, which can be found in Appx. C. We have proved that removing any other verification makes the computation vulnerable to single fault injection attacks, thereby proving the *minimality* of the obtained countermeasure.

Vigilant's original fault model did not include the possibility to inject fault in the conditions of the verifications. If we protect them from fault injection, we found that it is still possible to perform a BellCoRe attack exploiting two faults; Woudenberg *et al.* [vWWM11] showed that two-fault attacks are realistic. Both faults must be zeroing faults. One on R_3 (resp. R_4), and one on S'_p (resp. S'_q). It works because the presence of R_3 (resp. R_4) in the last verification is not compensated by its presence in S'_p (resp. S'_q). This shows that even security-oriented optimizations cannot be applied mindlessly in security code, which once again proves the importance of formal methods.

5.3 Comparison with Aumüller *et al.*'s Countermeasure

Now that we have (in Appx. C) a version of Vigilant's countermeasure formally proven resistant to BellCoRe attack, it is interesting to compare it with the only other one in this case, namely Aumüller *et al.*'s countermeasure. Both countermeasures were developed without the help of formal methods, by trial-and-error engineering, accumulating layers of intermediate computations and verifications to patch weaknesses found at the previous iteration. This was shown, unsurprisingly, to be an ineffective development method. On one hand, there is no guarantee that all weaknesses are eliminated, and there was actually a point of failure left in the original version of Vigilant's countermeasure. On the other hand, there is no guarantee that the countermeasure is minimal. Vigilant's countermeasure could be simplified by formal analysis, we removed two out of its nine verifications. This proves that "visual" code verification as is general practice in many test/certification labs, even for such a concise algorithm is far from easy in practice. Moreover, the difficulty of this task is increased by the multiplicity of the fault model (permanent vs. transient, randomizing vs. zeroing, single or double).

Apart from these saddening similarities in the development process, there are notable distinctions between the two countermeasures. Vigilant's method exposes the secret values p and q at two different places, namely during recombination and reduction from modulo pqr^2 to pq. Generally speaking, it is thus less safe than Aumüller *et al.*'s method. In terms of computational complexity the two countermeasures are roughly equivalent. Aumüller *et al.*'s one does 4 exponentiations instead of 2 for Vigilant's, but the 2 additional ones are with 16 bits numbers, which is entirely negligible compared to the prime factors (p and q are $1,024$ bits numbers). Vigilant's countermeasure requires more random numbers, which may be costly but is certainly negligible too compared to the exponentiations.

6 Conclusions and Perspectives

We have formally proven the resistance of a fixed version of Vigilant's CRT-RSA countermeasure against the BellCoRe fault injection attack. Our research allowed us to remove two out of nine verifications that were useless, thereby simplifying the protected computation of CRT-RSA while keeping it formally proved. Doing so, we believe that we have shown the importance of formal methods in the field of implementation security. Not only for the development of trustable devices, but also as an *optimization enabler*, both *for speed and security*. Indeed, Coron *et al.*'s repaired version of Vigilant's countermeasure included fixes for weaknesses that have been introduced by a hasty speed-oriented optimization, (which is never a good idea in the security field), and two faults attacks are possible only because of random numbers that have been introduced to reinforce the countermeasure.

As a first perspective, we would like to further improve our tool. It is currently only able to inject faults in the data. It would be interesting to be able to take into account faults on instructions such as studied by Heydemann *et al.* [HMER13]. It would be interesting as well to be able to automatically refine some properties of the variables, for instance to study attacks using a chosen message as input of the algorithm. Also, multiple fault analyses can take up to several dozens of minutes to compute. Since each attack is independent, our tool would greatly benefit (2 to 8 times speed-up on any modern multi-core computer) from a parallelization of the computations. Finally, it is worthwhile to note that our tool is *ad hoc* in the sense that we directly inject the necessary knowledge into its core. It would be interesting to see if general purpose tool such as EasyCrypt [BGZB09] could be a good fit for this kind of work.

Acknowledgements

The authors wish to thank David Vigilant for his insightful answers to our questions. We also sincerely acknowledge the SIGPLAN PAC program of the ACM for a financial sponsor.

References

[ABF+02] 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.

[BDL97] 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.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[CCGV13] 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.

[CGM+10] 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.

[HMER13] Karine Heydemann, Nicolas Moro, Emmanuelle Encrenaz, and Bruno Robisson. Formal Verification of a Software Countermeasure Against Instruction Skip Attacks. Cryptology ePrint Archive, Report 2013/679, 2013. `http://eprint.iacr.org/2013/679`.

[JT11] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer LNCS, March 2011. `http://joye.site88.net/FAbook.html`. DOI: 10.1007/978-3-642-29656-7 ; ISBN 978-3-642-29655-0.

[KJJ96] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[Koç94] Çetin Kaya Koç. High-Speed RSA Implementation, November 1994. Version 2, `ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf`.

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, May 2-6 1999. Prague, Czech Republic.

[RG13] Pablo Rauzy and Sylvain Guilley. A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA. Cryptology ePrint Archive, Report 2013/506, 2013. `http://eprint.iacr.org/2013/506`.

[RSA78] 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.

[Sha99] 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.

[TW12] Mohammad Tehranipoor and Cliff Wang, editors. *Introduction to Hardware Security and Trust*. Springer, 2012. ISBN 978-1-4419-8079-3.

[Vig08] 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.

[Vig10] David Vigilant. Countermeasure securing exponentiation based cryptography, Feb 17 2010. European patent, EP 2154604 A1.

[vWWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors, *FDTC*, pages 91–99. IEEE, 2011.

A Practical Feasibility of the Identified Attacks

In this section, we discuss the key features that an attacker must control for his attacks to succeed. We first describe in Sec. A.1 the timing characteristics of a CRT-RSA computation implemented on a smartcard. Second, we analyze in Sec. A.2 the performance of a fault injection bench. Third, we analyse in Sec. A.3 which fault injection is realistic or, at the contrary, challenging practically speaking.

A.1 CRT-RSA Algorithm in a Smartcard

We consider a smartcard implementation of CRT-RSA. It consists in an arithmetic hardware accelerator, capable of computing multiplications and additions. It is thus termed a MAC (Multiplier-Accumulator Circuit). This MAC can manipulate data on 64 bits. In 65 nm and lower technologies, such a MAC can run at the frequency of 100 MHz (after logical synthesis in a low power standard cell library). Let us consider that N is a 2048 bits number, *i.e.*, p and q hold on 1024 bits. The computations are carried out in the so-called Montgomery representation. For instance, when computing a modular multiplication, the multiplication and the reductions are interleaved. Thus, we derive some approximate timings for the unitary operations.

- One schoolbook addition on $\mathbb{Z}_p$ (or equivalently on $\mathbb{Z}_q$) of two large numbers takes $1024/64 = 16$ clock cycles, *i.e.*, 160 ns;
- One schoolbook squaring or multiplication on $\mathbb{Z}_p$ of two large numbers takes $2 \times (1024/64)^2 = 512$ clock cycles, *i.e.*, 5.12 μs; The factor two takes into the account the reduction steps.
- One schoolbook exponentiation on $\mathbb{Z}_p$ of one large number by another large number takes $512 \times (1024 \times 3/2) = 786432$ clock cycles, *i.e.*, about 7.8 ms; This estimation assumes the square-and-multiply exponentiation algorithm, in which there are as many squarings as bits in p, but only half (in average) multiplications.

Besides, we notice that some operations in Alg. 49 are hybrid: they involve one large number and one small number (*e.g.*, 64 bits number, of the size of r, R_1, R_2, R_3 or R_4). Obviously, those operations unfold more quickly than operations with two large numbers. Some operations in Alg. 3 can even be implemented to be extremely fast. For instance, the subtraction on line 1 or 4 of Alg. 3 (resp. Alg. 4) can be imagined to be done in one clock cycle, *i.e.*, 10 ns. Indeed, as p and q are odd (because they are large prime numbers), this "decrementation" consists simply in resetting the least significant bit of the operand p (resp. q).

So, concluding, the operations in the CRT-RSA algorithm can be of diverse durations, ranging 6 orders of magnitude, from 10 ns for operations computable in one clock cycle, to about 8 ms for one modular exponentiation.

A.2 Fault Injection Setup

An efficient fault injection method consists in the production of a transient electromagnetic (EM) pulse of high energy in the vicinity of the circuit that runs the CRT-RSA. This kind of fault is termed EMI (EM

Injection). Some setups can feature the possibility of multiple fault injections. Simple, double and triple EM pulses are illustrated respectively in Fig. 2, 3 and 4.

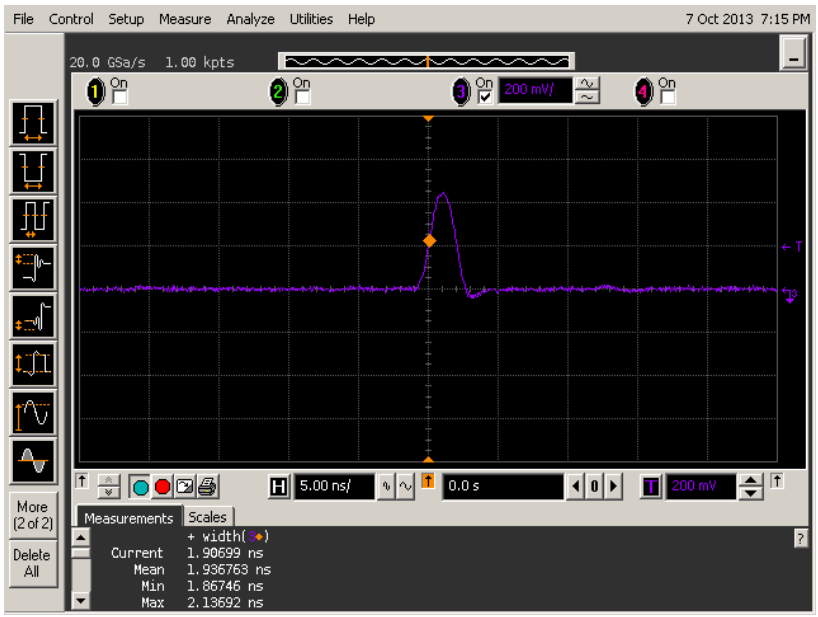

Figure 2: First-order fault injection attack — proof of concept

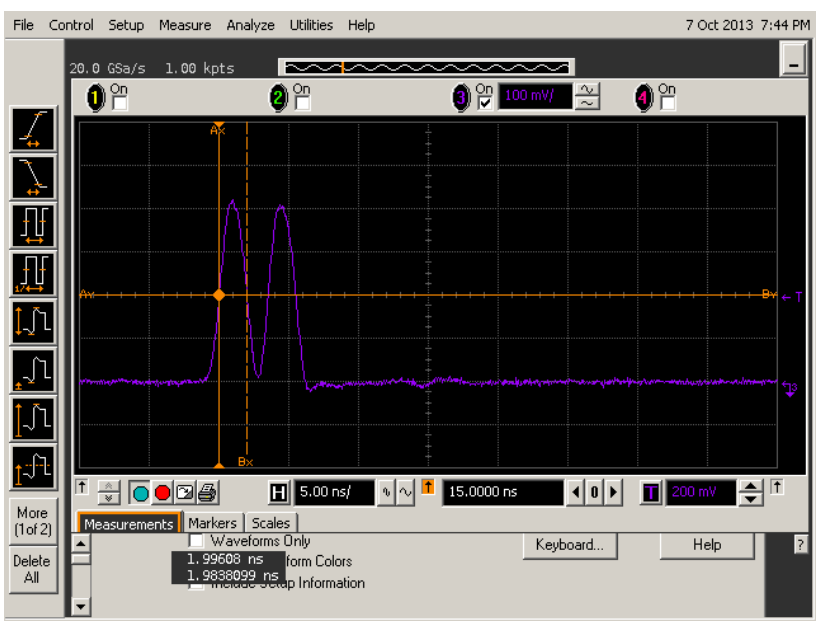

Figure 3: Second-order fault injection attack — proof of concept

We detail below the technical characteristics of a fairly high-end bench[4]:

[4]We credit Laurent Sauvage for this information, that comes from the specification of one plug-in of the Smart-SIC Analyzer tool of Secure-IC S.A.S.

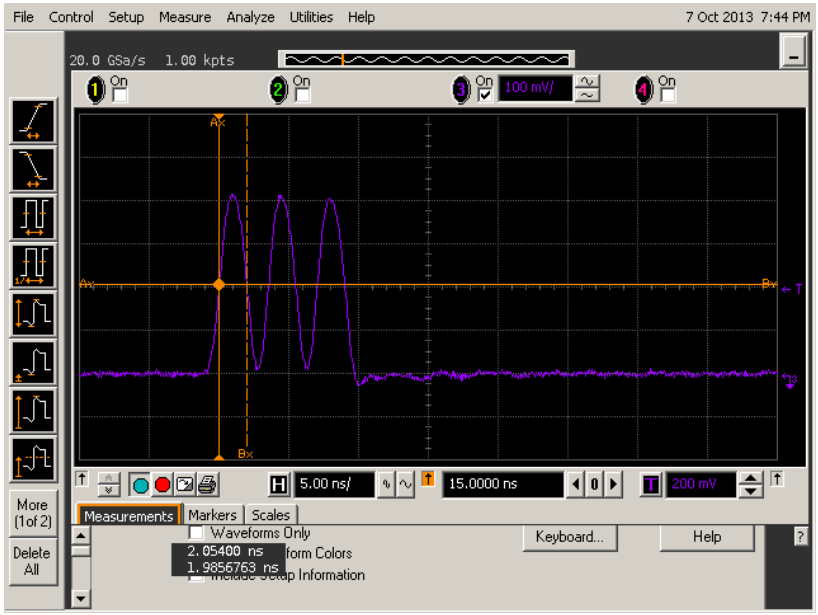

Figure 4: Third-order fault injection attack — proof of concept

- Pulse amplitude: ..... 0—500 mV.
- Pulse duration: ..... 2 ns.
- Repeatability: ..... 2 ns $\iff$ 500 MHz.

A.3 Practicality of the Attacks Identified by finja

The *pulse amplitude* (or "*energy*") shall be sufficiently high for the EMI to have an effect, and not too large to avoid chip damage. A typical method to set the adequate level of energy is to run the CRT-RSA computations iteratively, while increasing the energy from zero, with small increments (*e.g.*, a few millivolts). After a given number of iterations, the CRT-RSA signature will be faulty or the circuit under attack will refuse to give an answer. From this value on, the EMI starts to have an effect on the chip. Perhaps this fault injection is probabilistic, meaning that it does not always have an effect. So, it is safe to increase the energy level of a few percents to raise the injection probability of success as close to one as possible.

The *pulse duration* is an important parameter, because it conditions the accuracy in time of the stress. As mentioned in Sec. A.1, there is a great variability in the duration of the operations. Therefore, some attacks will be more difficult to set up successfully. For exemple, the two corrections (Alg. 3 and 5) finally repair a flaw that is fairly difficult to fault, because the operations last:

- 1 clock cycle, *i.e.*, 10 ns, and
- 512 clock cycles, *i.e.*, 5.12 μs respectively.

Implementations of CRT-RSA are generally working in constant time, to prevent timing attacks [KJJ96]. Those attacks are indeed to be taken very seriously, as they can even be launched without a physical access to the device. Paradoxically, this helps carry out timely attacks on focused stages in the algorithm. So, with the setup described in Sec. A.2, such fault attacks remain possible. Also, multiple faults can be injected easily if the algorithm timing is deterministic, because the *repetition rate* of the setup is greater than once per clock cycle. Indeed, a new pulse can be generated every other period of 2 ns, *i.e.*, at a maximal rate of 500 MHz.

However, a bench of lower quality would deny some exploits, for instance those that require: (a) to fault an operation of small duration; (b) for high-order fault attacks, operations that are too close one from each other.

B Vigilant's Original Countermeasure

```
noprop error, M, e, r, R1, R2, R3, R4 ;
prime {p}, {q} ;

dp := { e^-1 mod (p-1) } ;
dq := { e^-1 mod (q-1) } ;
iq := { q^-1 mod p } ;

--- p part ---
p' := p * r * r ;
Mp := M mod p' ;
ipr := p^-1 mod (r * r) ;
Bp := p * ipr ;
Ap := 1 - Bp mod p' ;
M'p := Ap * Mp + Bp * (1 + r) mod p' ;

if M'p !=[p] M abort with error ;

d'p := dp + R1 * (p - 1) ;
Spr := M'p^d'p mod p' ;

if d'p !=[p - 1] dp abort with error ;

if Bp * Spr !=[p'] Bp * (1 + d'p * r)
  abort with error ;

S'p := Spr - Bp * (1 + d'p * r - R3) ;

--- q part ---
q' := q * r * r ;
Mq := M mod q' ;
iqr := q^-1 mod (r * r) ;
Bq := q * iqr ;
Aq := 1 - Bq mod q' ;
M'q := Aq * Mq + Bq * (1 + r) mod q' ;

if M'q !=[q] M abort with error ;

if Mp !=[r * r] Mq abort with error ;

d'q := dq + R2 * (q - 1) ;
Sqr := M'q^d'q mod q' ;

if d'q !=[q - 1] dq abort with error ;

if Bq * Sqr !=[q'] Bq * (1 + d'q * r)
  abort with error ;

S'q := Sqr - Bq * (1 + d'q * r - R4) ;

--- recombination ---
S := S'q + q * (iq * (S'p - S'q) mod p') ;
N := p * q ;

if N * (S - R4 - q * (iq * (R3 - R4)))
  !=[N * r * r] 0 abort with error ;

if q * iq !=[p] 1 abort with error ;

return S mod N ;

%%

_ != @ /\ ( _ =[p] @ \/ _ =[q] @ )
```

C Fixed Vigilant's Countermeasure

```
noprop error, M, e, r, R1, R2, R3, R4 ;
prime {p}, {q} ;

dp := { e^-1 mod (p-1) } ;
dq := { e^-1 mod (q-1) } ;
iq := { q^-1 mod p } ;

--- p part ---
p' := p * r * r ;
Mp := M mod p' ;
ipr := p^-1 mod (r * r) ;
Bp := p * ipr ;
Ap := 1 - Bp mod p' ;
M'p := Ap * Mp + Bp * (1 + r) mod p' ;

if M'p !=[p] M abort with error ;

d'p := dp + R1 * (p - 1) ;
Spr := M'p^d'p mod p' ;

if d'p !=[p - 1] dp abort with error ;

if Bp * Spr !=[p'] Bp * (1 + d'p * r)
  abort with error ;

S'p := Spr - Bp * (1 + d'p * r - R3) ;

--- q part ---
q' := q * r * r ;
Mq := M mod q' ;
iqr := q^-1 mod (r * r) ;
Bq := q * iqr ;
Aq := 1 - Bq mod q' ;
M'q := Aq * Mq + Bq * (1 + r) mod q' ;

if M'q !=[q] M abort with error ;

-- useless verification removed

d'q := dq + R2 * (q - 1) ;
Sqr := M'q^d'q mod q' ;

if d'q !=[q - 1] dq abort with error ;

if Bq * Sqr !=[q'] Bq * (1 + d'q * r)
  abort with error ;

S'q := Sqr - Bq * (1 + d'q * r - R4) ;

--- recombination ---
S := S'q + q * (iq * (S'p - S'q) mod p') ;
N := p * q ;

if p * q * (S - R4 - q * (iq * (R3 - R4)))
  !=[N * r * r] 0 abort with error ;

-- useless verification removed

return S mod N ;

%%

_ != @ /\ ( _ =[p] @ \/ _ =[q] @ )
```