JCEN manuscript No.
(will be inserted by the editor)

Formal verification of an implementation of CRT-RSA algorithm

Maria Christofi · Boutheina Chetali · Louis Goubin · David Vigilant

Received: date / Accepted: date

Abstract Cryptosystems are highly sensitive to physical attacks, which leads security developers to design more and more complex countermeasures. Nonetheless, no proof of flaw absence has been given for any implementation of these countermeasures. This paper aims to formally verify an implementation of one published countermeasure against fault injection attacks. More precisely, the formal verification concerns Vigilant's CRT-RSA countermeasure which is designed to sufficiently protect CRT-RSA implementations against fault attacks. The goal is to formally verify whether any possible fault injection threatening the pseudo-code is detected according to a predefined attack model.

Keywords fault attacks · frama-C · countermeasures · cryptographic implementation · formal verification · RSA-CRT

Maria Christofi
Gemalto, 6, rue de la Verrerie, 92447 Meudon sur Seine, France

Versailles Saint-Quentin-en-Yvelines University
E-mail: maria.christofi@gemalto.com

Boutheina Chetali
Trusted Labs SAS, 5, rue du bailliage, 78000 Versailles, France
E-mail: boutheina.chetali@trusted-labs.com

Louis Goubin
Versailles Saint-Quentin-en-Yvelines University
E-mail: louis.goubin@uvsq.fr

David Vigilant
Gemalto, 6, rue de la Verrerie, 92447 Meudon sur Seine, France
E-mail: david.vigilant@gemalto.com

1 Introduction

Cryptographic implementations may be subject to physical attacks that disturb the execution of the embedded code. These attacks aim to disclose sensitive information or to force malicious behavior of the attacked code. To protect the implementations against this kind of attacks, countermeasures are designed, implemented and tested using several attack scenarios. To increase the level of confidence in the correctness of the countermeasure implementation, specific procedures of code review and cross-review are used. This "manual" verification procedure is itself error prone, and in some case its degree of exhaustivity depends on the time-to-market of the product.

The aim of this work is to provide the crypto-developer with a verification procedure which will improve the current process of correctness of the countermeasure with more automation and confidence. An implementation verification procedure can be seen as a procedure which takes as input an implementation (or a pseudo-code) with the corresponding countermeasure and outputs a "yes" or "no" answer. A "yes" answer means that the set of countermeasures present in the code is efficient enough to detect every possible attack scenario, according to a predefined attack model for the considered implementation, while a "no" answer means that the developer has to improve this set in order to include the missing scenarios.

To the authors' knowledge, the formal verification of implementations of countermeasures has not really been the subject of research until now. Several works have been done on the formal verification of cryptosystems, but generally focused on the correctness of the cryptographic protocol with respect to its specification (like [2]) and more recently to its implementation.

Indeed, such verification increases confidence in the cryptographic implementation and excludes flaws due to the weaknesses of the countermeasures.

The goal of this work is to demonstrate the robustness of the countermeasure with respect to a given attack model. For that, a classical approach consists in proving that an abstract model of the implementation with its countermeasure verifies a set of properties. Then using one of the existing approaches (empirical method, code generation, computational method) to convince the developer that the verified abstract model is a correct abstraction of the original code. The approach we take is to follow the developer view, using the source code of the cryptosystem with its countermeasures. For that, we will use a static analysis based tool which takes the pseudo-code as an input to the formal verification. Moreover, we will focus on a well-

known cryptosystem, RSA, and more precisely the algorithm associated to Vigilant's countermeasure provided in [23].

Attacks based on information gained from the physical implementation of a cryptosystem, other than brute force or theoretical weaknesses in the algorithms, are called *side channel attacks*. Attacks are typically distinguished in passive (such as timing information, power consumption and electromagnetic leaks) and active (such as fault injection) attacks.

This paper considers only fault injection attacks and more precisely single fault injection attacks, i.e. attack scenarios where only one fault is injected. However, the method presented in this paper can be extended to multiple fault injection attacks.

Structure of the paper

In Sect. 2, fault attacks and some of the CRT-RSA countermeasures are reminded. Then the general idea of Vigilant's countermeasure is briefly presented in Sect. 3. Section 4 describes our methodology to formally verify an implementation of a countermeasure and then Sect. 5 presents the formal verification of the pseudo-code of Vigilant's countermeasure as well as its results.

2 Fault attacks and countermeasures on CRT-RSA

This Section is a short introduction to fault injection attacks, and especially attacks targeting the CRT-RSA algorithm.

2.1 Fault injection attacks

Fault attacks consist in tampering with a device in order to have it perform some erroneous operations, hoping that the result of that erroneous behavior will leak information about the involved secret parameters.

The fault attacks in a specific code can be seen either as modifications of a specific variable or as modifications of code instructions (including modifications on the execution flow and logical level modifications). The former one concerns attacks that aim to trouble on the value of a register, while the later one concerns attacks on the instructions of the code. In [4], Bar-El and al. present various methods to induce faults and exploit such errors, and give several examples of both attacks and countermeasures.

Modifying a variable with a fault injection can be seen as adding new instructions that assign an arbitrary value to this variable. In the same vein, modifications of code instructions are simulated by a *goto* instruction. Formalizing modifications of instructions requires the formalization of the program execution, and this will be part of a future extension of this work. A first attempt to model this kind of modifications, and especially the jump attacks, one can find in [6].

The methodology proposed in Sect. 4 aims to guarantee the validity of a countermeasure pseudo-code where the effect of the attack is the value modification of a variable.

Therefore, the level of the details provided in the pseudo-code is relevant with respect to the formalism. For example, the result of a formal verification can be different for a pseudo-code where the smallest manipulated variables are large integers, compared to a pseudo-code where the smallest variables are arrays of bits (or words) in a lower-level implementation. Indeed, the second pseudo-code would contain more steps including all multi-precision integers operations. These extra steps would represent more locations for fault injections. Therefore the formal verification should be applied to a pseudo-code as fine as possible, in order to give the best confidence.

A fault can then be characterized by different aspects, like the number of affected bits, but also error location, time of occurrence and persistence. The different fault models are summarized in Table 1.

2.2 Countermeasures on CRT-RSA

Focusing now to the CRT-RSA algorithm, as a signing procedure and some already known countermeasures used to protect it.

Let $N = p \cdot q$ be a product of two large prime numbers. To sign a message m, one first computes $S_p = m^d \mod p$ and $S_q = m^d \mod q$ and then uses the Chinese Remainder Theorem (CRT) to build the signature $S = m^d \mod N$ (this is done by computing $S = (S_p \cdot q \cdot (q^{-1} \mod p) + S_q \cdot p \cdot (p^{-1} \mod q)) \mod N$).

CRT-RSA is especially susceptible to software or hardware errors. Boneh, DeMillo and Lipton were the first to present a fault attack on RSA in both standard and CRT mode [7]. In the case of the CRT-RSA algorithm, if a fault is induced during the computation of S_p (respectively S_q), then an erroneous value S'_p (resp. S'_q) is used during the CRT-recombination leading to an erroneous signature S'. As $S \equiv S_p \mod p$ and $S \equiv S_q \mod q$, we now have $S' \equiv S \mod q$ (resp. $S' \equiv S \mod p$), but $S' \not\equiv S \mod p$ (resp. $S' \not\equiv S \mod q$). Therefore, if $p \nmid (S - S')$ then the secret parameter

Table 1: Fault models

	Precise Bit Fault Model	**Single Bit Fault Model**	**Byte Fault Model**	**Random Fault Model**	**Arbitrary Fault Model**
control on location	complete (chosen bit)	loose (chosen variable)	loose	loose	loose/no
control on timing	precise	no	no	no	no
number of affected bits	1	1	8	random	random
fault type	bit set or reset	bit flip	random	random	unknown
persistence	permanent and transient	permanent and transient	permanent and transient	permanent and transient	permanent and transient

q can be easily obtained by computing $\gcd(S - S', N)$. The other secret parameters of the private key $p, d_p(= e^{-1} \mod (p-1)), d_q(= e^{-1} \mod (q-1)), i_q(= q^{-1} \mod p)$ can then easily be computed.

An improvement of this attack comes later on by Lenstra in [19]. He claims that if a fault is induced during the computation of S_p then $S'^e \equiv m \mod q$ but $S'^e \not\equiv m \mod p$. Therefore the secret parameter q can be obtained by computing $\gcd(S'^e - m, N)$. The advantage of this attack comparing to the previous one is that now only one execution of the cryptographic algorithm is required to recover the private key.

However, for the above attacks, the attacker needs to know the whole message. Some efforts have already been done for attacks without the need of knowing it. As for example, the one of Coron and al. in [12].

An obvious countermeasure against these attacks is to verify the signature by using the public key (e, N). Usually e is small (for example $2^{16}+1$), but this method may be very costly when e is large as it implies a second exponentiation. Moreover, the public exponent is not always available.

Since the publication of this attack, a large variety of countermeasures have been published in the field. The first method was proposed by Shamir in [22]. Shamir suggests to choose a small integer r, then compute $S_{pr} = m^d \mod pr$ and $S_{qr} = m^d \mod qr$ and ensure the integrity of these two exponentiations by testing whether $S_{pr} \equiv S_{qr} \mod r$ before combining S_{pr} and S_{qr} with the CRT formula. However, Aumüller and al. in [3] show that this method does not protect the CRT recombination and propose an implementation that also protects the CRT recombination. As opposed to Shamir's method, only d_p and d_q (and not d) are required. This solution gives good performance, as comparing to the classical CRT-RSA implementation, only two extra exponentiations and a few modular reductions are required. The main disadvantage of this method: it requires an extra prime parameter. There are already many improvements of Shamir's method, such as the one proposed by Vigilant in [23]. After some flaws discovered, [11] presents an improvement of this algorithm giving two possible attacks and the corresponding countermeasures. The first attack concerns a fault that changes the last "*mod* N" operation, while the second one concerns the way that $p-1$ (resp. $q-1$) is computed/stocked. The first attack does not apply to the case of our model (due to the impossibility of implementing a "*mod* 0" operation, see later on for more details about the model used). The second attack demands a different implementation than the one presented in the original paper [23].

Another protection has been proposed by Giraud in [15] in which the fault detection comes from the exponentiation algorithm. Actually, by using the Montgomery powering ladder to compute $m^d \mod N$, both values $m^d \mod n$ and $m^{d-1} \mod N$ are available at the end of the computation. These values can then be used to verify the integrity of the exponentiation. In [8], Boscher and al. also proposed a countermeasure where the detection comes from another exponentiation algorithm. Finally, Rivain proposed a detection method based on addition chains in [21].

Examples of pseudo-codes for implementing the countermeasures were only provided by Aumüller and al. in [3] and by Vigilant in [23]. This paper studies the pseudo-code provided by Vigilant. As said in Sect. 1, the results of our method are specific to the implementation verified and can be different for different implementations of the same algorithm. As we want to verify the original implementation of Vigilant's countermeasure against fault attacks, we will not take into account the modifications provided in [11].

3 Vigilant's CRT-RSA countermeasure

Vigilant's countermeasure is a method to protect a modular exponentiation against fault attacks. This method can be efficiently used for protecting CRT-RSA on embedded devices, since it does not require the public exponent, neither precomputation, nor extra parameters.

Protecting an exponentiation $S = m^d \mod N$ against fault attacks consists in computing $m^d \mod N$ in $\mathbb{Z}_{N \cdot r^2}$ where r is a small random integer co-prime with N. The message m is transformed into m' such that:

$$m' \equiv \begin{cases} m \mod N \\ 1 + r \mod r^2 \end{cases}$$

This implies that

$$S' = m'^d \mod Nr^2 \equiv \begin{cases} m^d \mod N \\ 1 + d \cdot r \mod r^2 \end{cases}$$

So, a consistency check of the result S' can be performed modulo r^2 from d and r. If the verification $S' \mod r^2 = 1 + d \cdot r \mod r^2$ is successful, then the final result $S = S' \mod N$ is returned.

This secure exponentiation can be applied to RSA with CRT. The principle is to perform two exponentiations modulo $p \cdot r^2$ and $q \cdot r^2$ (so we obtain S_p and S_q respectively) and then perform a final consistency check after recombination, guaranteeing that no error occurred during the computation of S_p or S_q and during the recombination.

Algorithm 1 presents the pseudo-code of Vigilant's implementation as provided in [23].

This implementation has many advantages:

- no need of special hypotheses for r. However, in [23] we can find some recommendations about r, such that $i_q \neq 0 \mod r$, r should be odd, at least a 32-bit random integer and as large as possible
- no precomputation is needed
- only p, q, d_p, d_q, i_q, and m are needed for the calculation

Algorithm 1 Vigilant's CRT-RSA implementation code

Input: message m, e, key (p, q, d_p, d_q, i_q)
32-bit random integer r
64-bit random integers R_1, R_2, R_3, R_4
Output: signature $S = m^d \mod N$

$p' = p \cdot r^2$
$m_p = m \mod p'$
$i_{pr} = p^{-1} \mod r^2$
$\beta_p = p \cdot i_{pr}$
$\alpha_p = (1 - \beta_p) \mod p'$
$\hat{m}_p = (\alpha_p \cdot m_p + \beta_p \cdot (1 + r)) \mod p'$
if $(\hat{m}_p \neq m \mod p)$ **then return** error
$d'_p = d_p + R_1 \cdot (p - 1)$
$S_{pr} = \hat{m}_p^{d'_p} \mod p'$
if $(d'_p \neq d_p \mod (p - 1))$ **then return** error
if $(\beta_p \cdot S_{pr} \neq \beta_p \cdot (1 + d'_p \cdot r) \mod p')$ **then return** error
$S'_p = S_{pr} - \beta_p \cdot (1 + d'_p \cdot r) - R_3$
$q' = q \cdot r^2$
$m_q = m \mod q'$
$i_{qr} = q^{-1} \mod r^2$
$\beta_q = q \cdot i_{qr}$
$\alpha_q = (1 - \beta_q) \mod q'$
$\hat{m}_q = (\alpha_q \cdot m_q + \beta_q \cdot (1 + r)) \mod q'$
if $(\hat{m}_q \neq m \mod q)$ **then return** error
if $(m_p \mod r^2 \neq m_q \mod r^2)$ **then return** error
$d'_q = d_q + R_2 \cdot (q - 1)$
$S_{qr} = \hat{m}_q^{d'_q} \mod q'$
if $(d'_q \neq d_q \mod (q - 1))$ **then return** error
if $(\beta_q \cdot S_{qr} \neq \beta_q \cdot (1 + d'_q \cdot r) \mod q')$ **then return** error
$S'_q = S_{qr} - \beta_q \cdot (1 + d'_q \cdot r) - R_4$
$S = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
$N = p \cdot q$
if $(N \cdot [S - R_4 - q \cdot i_q \cdot (R_3 - R_4)] \neq 0 \mod Nr^2)$ **then return** error
if $(q \cdot i_q \neq 1 \mod p)$ **then return** error
return $S \mod N$

4 Formal verification of implementations of countermeasures

The aim of this work is to formally verify the resistance of the pseudo-code described in Algorithm 1 against fault attacks. The goal is to build a formal environment that will allow the cryptographic engineer to introduce his secure code and check the validity of his countermeasure. The main steps of the verification procedure to follow are:

1. define the implementation that we want to verify with the corresponding set of countermeasures
2. choose a fault model
3. simulate every possible injected fault with respect to this fault model
4. inject this fault model to the original source code implementation
5. model the property corresponding to the verification
6. use a tool to generate some proof obligations corresponding to the property to prove
7. prove these obligations (either automatically or using a proof assistant)

For the verification part of our work, we use a static analysis based tool, named frama-C [14], that will allow to perform an analysis of the source code without executing it. The source code will then correspond to the implementation of the cryptosystem with its countermeasures along with a simulation of the chosen fault model.

4.1 Frama-C

Frama-C [14] is an open source extensible platform dedicated to source code analysis of C software. The frama-C platform gathers several static analysis techniques into a single collaborative extensible framework. The

collaborative approach of frama-C allows static analyzers to build upon the results already computed by other analyzers in the framework.

In addition, frama-C verifies some "safety properties" like the division by zero or loop's termination and correctness.

One of the advantages of frama-C, against other tools of static analysis or even bug-finding tools, is that it allows its user to manipulate functional specifications, and to prove that the source code satisfies these specifications written in a dedicated language ACSL [1] (ANSI/ISO C Specification Language, a behavioral specification language for C programs). ACSL is a language of annotations, threatened as standard comments by the C compiler, that allows the user to express the above specifications in such a way that they do not affect a normal execution of the implementation but they are verified by frama-C.

Frama-C is a plugin system. In order to perform a verification, we use Jessie [17], the deductive verification plugin of C programs annotated with ACSL. It uses internally the languages and tools of the Why platform [24] [1]. The Jessie plugin uses Hoare-style [16] weakest precondition computations to formally prove ACSL properties. The generated verification conditions can be submitted to external automatic provers such as Simplify, Alt-Ergo, Z3, CVC3.

For more complex situations, interactive theorem provers, like Coq, PVS, Isabelle/HOL, can be used to establish the validity of the verification conditions.

The aim of the work presented in this paper is the verification of a C code including a cryptographic implementation and a simulation of all possible fault attack scenarios. For this, Jessie needs as input this transformed code and outputs the proof obligations to verify using an automatic or an interactive prover. The user is then free to exploit these results.

4.2 Fault model

As this is a first attempt to formally verify a cryptographic implementation, we have chosen a quite simple fault model which is still realistic, but different from the one described in [23]. The two models are clearly not equivalent. However, the verification procedure is still the same for other models and this extension will be part of our future work.

In the original fault model, the attacker can:

- inject only one fault per execution
- modify a value in memory obtaining a totally random result uncorrelated to the original value (known as permanent fault)
- modify a value when it is handled in local registers, without modifying the global value in memory. The handled value obtained is fully random from the attacker point of view and uncorrelated to the original value (known as transient fault)

but the attacker cannot:

- modify the code execution. Processor instructions cannot be replaced or removed while executing code
- inject a permanent fault in the input elements, the message m or the key (p, q, d_p, d_q, i_q)
- change the boolean result of a conditional check. An expression "if $a = b$" has a result true or false that cannot be modified.

Our fault model is based on the above with three differences. We consider that the attacker :

- can modify the value in memory but by only setting the value to 0 (in the case of the pseudo-code, this corresponds to set the whole variable to 0)
- can inject a permanent fault in the input elements, the message m as well as the key (p, q, d_p, d_q, i_q)
- cannot inject a fault in m at the very beginning (i.e. before line 1 of the Algorithm 1) of the implementation.

4.3 Fault injection simulation

Once the fault model is defined, it must be injected in the initial code of the implementation. This simulation consists in setting the value of the "attacked" variable to 0, for every possible fault (with respect to variable's location). Obviously such a modeling creates a lot of cases to verify. The number of the cases increases according to the number of the code instructions and the variables used in it. Thus, for codes that describe real cryptographic implementations, this modeling may become very huge and so, quite inefficient.

For that we introduce an optimization by defining some equivalence classes between attacks that have the same effects. To do so, we use the notions of *read* and *write* for any variable used in the code. The general idea is to characterize every line of the original code by a *read, write, read/write,* $\emptyset$ type according to the actions occurred to the variables appeared in it. The *read* (resp. *write*) type means that the considered code line reads (resp. writes) the variable. The *read/write* type means that the code line performs a read and a write operation (as for example, for the variable *var* in

[1] WHY is a general-purpose verification condition generator, which is used as a back-end by other verification tools but which can also be used directly to verify programs. WHY produces verification conditions from annotated programs given as input.

the instruction $var = var + 1$). The $\emptyset$ means that no operation is performed concerning this variable.

Definition 1 Let i be the number of the line and *var* the variable that we want to check, then Type(var,i) define this characterization for the variable *var* on line *i*.

We then determine the next use of a variable *var* with the help of the following definition.

Definition 2 Let consider a m-line code for our implementation, i the line of the code that we want to test and *var* the name of the variable that we want to verify. Then,

$$NextType(var,i) = \begin{cases} Type(var,j) \text{ , where } j \text{ is the minimal value, greater than } i \text{ such that } Type(var,j) \neq \emptyset \\ \emptyset \text{ , if } i = m \end{cases}$$

The different types are illustrated in a simple example in Table 2.

Attacks on code with sequential control flow. To simplify, let's first focus to a code without any loops nor conditionals. For such a code, the equivalence classes correspond to the minimal code to verify in order to ensure a security property. In fact, the class of the original source code includes all the attacked codes for which the attack is useless. Formally, we have:

Lemma 1 *If* $NextType(var,i) \in \{write, \emptyset\}$*, then an attack on var injected at line i is useless and equivalent to the original source code.*

Obviously, if the next use of *var* is "write", the operation performed will have no effect to the value of *var* stored in memory. Contrary to the cases that the next use of *var* is "read" or "read/write" where the following lemma is applied:

Lemma 2 *If* $NextType(var,i) \in \{read, read/write\}$ *and j the line that represents the next use of the variable var, then an attack on var injected at the interval* $[i,j]$ *has exactly the same effect on var than an attack injected at line j, but has no effect between lines i and* $j-1$.

The aim of these two lemmas is to separate the useful attacks from the useless ones, i.e. the attacks that have an effect on the code from the ones that have no effect. It reduces the number of the attacks so that only useful attacks are kept. These two lemmas are summarized to the following theorem (we recall that for the moment we have a code with no loop and no conditionals):

Theorem 1 *If there are n read and read/write operations on the code for one variable, the minimal number of faults with different effect for this variable is* $n+1$ *(i.e. one attack for every read and read/write operation plus the original code-without faults injection).*

Attacks on code with conditionals and loops. Let us now consider the case of a source code with conditionals and loops. The type of any line can be defined in the same way as for any other line of a non conditional code, thus Lemma 1 remains valid.

We first deal with the conditional instructions (an if-then-else structure). This part of code can be decomposed in three parts: the condition, the then-block and the else-block (which can be empty). It is possible to inject attacks at either the condition or the then/else-block.

As in the case of code with sequential control flow, we inject an attack before any *read* operation of every variable.

However, as we want to minimize the number of injected attacks, if no *read* operation happens during the if-condition and a *read* operation happens in both the then and the else block, instead of injecting an attack at both corresponding lines, we can inject an attack before the if-condition when no operation is performed between the if-condition and both these *reads*. An example is given in Fig. 1. This can be done only in the case of fault models where the fault is always of the same nature. An example of this kind of fault model is the one studied in this paper, which consists on setting a value to 0. An example of fault model that we cannot apply this optimization is the fault model which sets a value to a random value. This is because every fault injection can correspond to another random value.

```
1:  int  example_if(int x, int y){
2:           if (y > 0 )                  // condition
3:             {y = x; }                  // then-block
4:           else y = −x;                 // else-block
5:           return y;
6:  }
```

Fig. 1: Code example with conditionals (considering attacks on variable x). For the fault model consisting on setting a value on 0, instead of injecting two attacks in both lines 3 and 4, we can inject one and only attack in line 2.

Table 2: Code example

1:	int example(int a, int b){		
2:	int x = 0;	// Type (x,2) = write	On line 2, the use of x is of type "write".
3:	a = a + 1;	// Type (a,3) = read/write	On line 3, the use of a is of type "read" and "write".
		// NextType (x,3) = write	On line 3, the next use of x is of type "write" (on line 4).
		// NextType (a,3) = read	On line 3, the next use of a is of type "read" (on line 4).
4:	x = a + b;	// NextType (x,4) = $\emptyset$	On line 4, there is no next use of x.
5:	}		

In the same vein, for the loop instructions, we inject an attack before any *read* operation of every variable.

4.4 Adding the fault model to the implementation

Before starting the verification, the simulated fault model will be added to the original code. For that, an additional variable is used, named f, which represents the faults. All possible attacks are finally introduced in such a way that this part of code will be executed once the corresponding simulated attack occurs.

As an example, one can see Figure 2. In this example, a fault consists on setting the value of a variable to 0. The lines 1, 6, 11 and 12 of the transformed code are equivalent to the initial code, while both the lines 3 and 8 represent attacks to the variable x, and lines 4 and 9 attacks to the variable y. Lines 2 to 5 describe all possible attacks for the instruction at the line 6, while lines 7 to 10 describe all possible attacks for the return statement at line 11.

Similarly, all possible attacks (w.r.t. the fault model) can be simulated and induced into the original code. The automatic generation of this simulation and its injection into the original code is already implemented (even if some improvements are still necessary).

4.5 Modeling the main property

The goal of the verification is to prove, for a given implementation, the validity of a set of countermeasures with respect to a set of attacks (for a given attack model). In other words, given an implementation and a set of countermeasures, we want to prove whether any attack by fault injection (w.r.t. the attack model) is detected (an error flag is raised).

For the fault model studied in this paper, this means that the output of any execution of the given code is either the expected result or the error flag. As we cannot know in advance the expected result, we have to express it in terms of a function using the entry variables. The property to prove is then summarized to the Theorem 2.

Theorem 2 *Let $f \in \{0\} \cup F$, where F is the set of faults for the current implementation and $f = 0$ the original execution of the implementation (without injected faults). Let also res be the output of the implementation, $x_1, ..., x_n$ be the n variables of the input of the implementation and g a function. Then :*

$$[(f = 0) \Rightarrow (res = g(x_1, ..., x_n))] \; AND$$
$$[(\forall f \in F) \Rightarrow ((res = ERROR) \; OR \; (res = g(x_1, ..., x_n)))]$$

When the output is the error flag, it means that the countermeasures are robust in the sense that they detect any fault injection (according to the model).

The second part of the above equation may be changed according to the chosen fault model. That is because if we use a fault model where we know the effect of the fault, for example a fault model that set the value of a variable to 0 or to a known constant, the valid result at the end of the execution will give us the correct value of this variable, and so, this fault witnesses the real value of a variable (which can also be a secret variable). Whereas a fault model that we do not know the effect of the fault, like a fault model that set the value of a variable to a random value unknown to the attacker, a valid result at the end of the execution will witness nothing, as we would not know the faulty value. So, in the first case, we need to ensure the whole property, while in the second one, a fault is detected when the error flag is up.

5 Formal verification of the pseudo-code of Vigilant's countermeasure

The following section describes the use of the presented approach to the pseudo-code of Vigilant's countermeasure. The verification is based on the procedure described in Sect. 4.

```
1:  int  example(int x, int y, int f){
2:           x = y;
3:           return x;
4:  }
```

(a) initial code

```
1:  int  example(int x, int y, int f){
2:           switch(f){
3:             case 1 : x = 0; break;
4:             case 2 : y = 0; break;
5:           }
6:           x = y;
7:           switch (f) {
8:             case 3: x = 0; break;
9:             case 4: y = 0; break;
10:          }
11:          return x;
12: }
```

(b) transformed code

Fig. 2: An example of a fault injection in the code

As described in Sect. 4.2, the fault model we use is the following:
An attacker can:

- inject only one fault per execution
- modify the value in memory by setting the value to 0
- inject both transient and permanent faults to any variable

but (s)he cannot:

- modify the code execution
- inject a fault in m at the very beginning (that is before line 1 of the Algorithm 1) of the implementation.
- inject a fault in S at the very end (i.e. after line 31 of the Algorithm 1) of the implementation
- change the boolean result of a conditional check. An expression "if $a = b$" has a result true or false that cannot be modified.

For the pseudo-code of Vigilant's CRT-RSA algorithm presented in [23], under the above assumptions and using the procedure described in this paper, 95 possible faults are obtained. These faults are presented in the Appendix A.

Some additional hypotheses have to be made:

- $m \mod p \not\equiv 0$ and $m \mod q \not\equiv 0$
- r is odd and $i_q \not\equiv 0 \mod r$ as it is recommended in [23]
- $\gcd(p, r^2) = 1$ and $\gcd(q, r^2) = 1$, for the efficiency of the computation of i_{pr} and i_{qr} respectively

Once every possible fault is injected using the method described in Sect. 4 and with respect to the above fault model, we call the frama-c platform with the jessie plugin to run the verification procedure of the property of Theorem 2.

The results of this verification indicate some cases of faults (the underlined cases in Algorithm 2 of Appendix A) that are not detected by the given countermeasures.
"Sensitive" cases are separated in three main categories:

- The first category contains cases with success probability one (that means that such a fault will never be detected). These cases (cases 19, 36, 60 and 77 in Algorithm 2) correspond to faults on the random values R_1, R_2, R_3 and R_4 and concern the randomization of some variables. In these cases, the output is the real signature and no information about the secret values is obtained. Hence, these cases are of a real interest as we can expect the same behavior whenever a random value appears. However, whenever we obtain the valid signature, the attacks presented in Sect. 2.2 are no more applicable. (Depending on the fault model this can give some information to the attacker about the attacked variable)
- The second (and the bigger) one contains cases with a weak success probability. (Noting $|x|$ the size of x)
  - For the cases 6, 8, 13, 27, 29, 33, 34, 41, 44, 46, 51, 68, 70, 74, 75, 79, 82, 87, 88 and 91, the probability that an injected fault is undetectable is $2^{-2|r|+1}$.
  - For the cases 22, 28 and 32, this probability is $2^{-(|p'|-1)} ln 2$.
  - For the cases 63, 69 and 73, this probability is $2^{-(|q'|-1)} ln 2$.

  We notice here that frama-C tool cannot manipulate probabilities. The probabilities mentioned here are manually calculated (see Appendix B for more details).
- The last category contains cases with a high success probability (in this case 1) and where the output is a faulty signature. These are the most dangerous cases as we can extract information about the secret values. These cases are: 18 and 59 and correspond to permanent faults on d_p and d_q during the computation of d'_p and d'_q respectively. In case 18

(respectively 59), we obtain a faulty signature modulo p (resp. modulo q) and the right one modulo q (resp. modulo p). So it will be easy for the attacker to compute q (resp. p) and then the other secret parameters. Indeed as already said, our fault model allows permanent faults on d_p and d_q, contrary to the original fault model. This fault model difference is of prime importance for our results here.

6 Related work

To our knowledge, the use of frama-C for the verification of countermeasures is novel, but other uses of frama-C already exist. In [13], one can find the results of a formal verification of source code of a model of automaton in SAM language and its C language implementation, obtained using frama-C and Caveat. In [9], one can find a formal proof of correctness of the key commands of the SCHUR software, which is an interactive program for calculating with characters of Lie groups and symmetric functions. Another example of a use of frama-C is [5] which is about verification of some interval security properties for smart card C codes using value analysis.

Other verification techniques, such as model checking, are also quite common to verify temporal properties in programs. In [18], such a verification concerning safety properties can be found, while in [10], one can find the results of a verification of a real system using MOPS - a tool for software model checking security-critical applications-. Although model checking is fully automated, it is limited to simple implementations due to the exhaustive exploration of the model.

Another remarkable effort on verifying programs with the presence of faults is made in [20] (thank to the anonymous reviewer for this citation), where the authors have developed a new logic for reasoning about faults.

7 Conclusion and perspectives

Vigilant's countermeasure is a countermeasure protecting modular exponentiations against fault attacks that was applied to CRT-RSA. In this paper, we have presented the results of the formal verification of the resistance of the pseudo-code provided in [23] against fault attacks, with respect to the fault model described above.

The obtained results are very promising. The approach has been developed with a simple fault model. The goal now is first to evaluate the pertinence of this fault model with the crypto-developers. Then, we plan to extend this method to other fault models and to double fault attacks. This work will continue along with experimentations on other cryptographic countermeasures. More practical steps are also planned, such as improving the automation in order to provide crypto-developers with a full validation environment.

Acknowledgements The authors would like to thank Pascal Paillier for his useful contribution to this work.

References

1. ACSL. `http://frama-c.com/acsl.html`
2. Aizatulin, M., Dupressoir, F., Gordon, A.D., Jürjens, J.: Verifying Cryptographic Code in C: Some Experience and the Csec Challenge. In: Formal Aspects of Security and Trust - 8th International Workshop, FAST 2011, Leuven, Belgium, September 12-14, 2011. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 7140, pp. 1–20. Springer (2012)
3. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In: CHES, *Lecture Notes in Computer Science*, vol. 2523, pp. 260–275. Springer (2003)
4. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. IACR Cryptology ePrint Archive **2004**, 100 (2004)
5. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.F.: Attack model for verification of interval security properties for smart card C codes. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS '10, pp. 2:1–2:12. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1814217.1814219. URL `http://doi.acm.org/10.1145/1814217.1814219`
6. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.F.: Simulating physical attacks in smart card C codes: the jump attack case. In: e-Smart (2011)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. J. Cryptology **14**(2), 101–119 (2001)
8. Boscher, A., Naciri, R., Prouff, E.: CRT RSA Algorithm Protected Against Fault Attacks. In: WISTP, *Lecture Notes in Computer Science*, vol. 4462, pp. 229–243. Springer (2007)
9. Butelle, F., Hivert, F., Mayero, M., Toumazet, F.: Formal Proof of SCHUR Conjugate Function. In: AISC/MKM/Calculemus, *Lecture Notes in Computer Science*, vol. 6167, pp. 158–171. Springer (2010)
10. Chen, H., Dean, D., Wagner, D.: Model Checking One Million Lines of C Code. In: NDSS. The Internet Society (2004)
11. Coron, J.S., Giraud, C., Morin, N., Piret, G., Vigilant, D.: Fault Attacks and Countermeasures on Vigilant's RSA-CRT Algorithm. In: FDTC, pp. 89–96. IEEE Computer Society (2010)
12. Coron, J.S., Naccache, D., Tibouchi, M.: Fault attacks against emv signatures. In: CT-RSA, *Lecture Notes in Computer Science*, vol. 5985, pp. 208–220. Springer (2010)

13. Duprat, S., Gaufillet, P., Lamiel, V.M., Passarello, F.: Formal verification of SAM state machine implementation. In: Embedded Real Time Software and Systems (ERTS'10) (2010)
14. frama-c. `http://frama-c.com/`
15. Giraud, C.: An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. IEEE Trans. Computers **55**(9), 1116–1120 (2006)
16. Hoare, C.A.R.: An axiomatic basis for computer programming (reprint). Commun. ACM **26**(1), 53–56 (1983)
17. Jessie. `http://krakatoa.lri.fr/#jessie`
18. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3), 291–314 (2001)
19. Lenstra, A.: Memo on RSA signature generation in the presence of faults. manuscript (1996)
20. Meola, M.L., Walker, D.: Faulty logic: Reasoning about fault tolerant programs. In: Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6012, pp. 468–487. Springer (2010)
21. Rivain, M.: Securing RSA against Fault Analysis by Double Addition Chain Exponentiation. IACR Cryptology ePrint Archive **2009**, 165 (2009)
22. Shamir, A.: Improved method and apparatus for protecting public key schemes from timing and fault attacks. Patent number: WO9852319 (1998)
23. Vigilant, D.: RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In: CHES, *Lecture Notes in Computer Science*, vol. 5154, pp. 130–145. Springer (2008)
24. Why. `http://why.lri.fr/`

A Vigilant's CRT-RSA implementation code with fault simulation

Algorithm 2 Vigilant's CRT-RSA implementation code with fault simulation

Input: message m, e, key (p, q, d_p, d_q, i_q)
32-bit random integer r
64-bit random integers R_1, R_2, R_3, R_4
an integer f
Output: signature $S = m^d \mod N$

switch (f) {
  case 1 : $p = 0$
  case 2 : $r = 0$
}
$p' = p \cdot r^2$
switch (f) {
  case 3 : $p' = 0$
}
$m_p = m \mod p'$
switch (f) {
  case 4 : $p = 0$
  case 5 : $r = 0$
}
$i_{pr} = p^{-1} \mod r^2$
switch (f) {
  case 6 : $i_{pr} = 0$
  case 7 : $p = 0$
}
$\beta_p = p \cdot i_{pr}$
switch (f) {
  case 8 : $\beta_p = 0$
  case 9 : $p' = 0$
}
$\alpha_p = (1 - \beta_p) \mod p'$
switch (f) {
  case 10 : $\alpha_p = 0$
  case 11 : $m_p = 0$
  case 12 : $p' = 0$
  case 13 : $\beta_p = 0$
  case 14 : $r = 0$
}
$\hat{m}_p = (\alpha_p \cdot m_p + \beta_p \cdot (1 + r)) \mod p'$
switch (f) {
  case 15 : $\hat{m}_p = 0$
  case 16 : $m = 0$
  case 17 : $p = 0$
}
if $(\hat{m}_p \neq m \mod p)$ **then return** error
switch (f) {
  case 18 : $d_p = 0$
  case 19 : $R_1 = 0$
  case 20 : $p = 0$
}
$d'_p = d_p + R_1 \cdot (p - 1)$
switch (f) {
  case 21 : $d'_p = 0$
  case 22 : $\hat{m}_p = 0$
  case 23 : $p' = 0$
}

55: $S_{pr} = \hat{m}_p^{d'_p} \mod p'$
56: **switch** (f) {
57: **case** 24 : $d'_p = 0$
58: **case** 25 : $d_p = 0$
59: **case** 26 : $p = 0$
60: }
61: **if** $(d'_p \neq d_p \mod (p-1))$ **then return** error
62: <u>case 27 :</u> $\beta_p = 0$
63: <u>case 28 :</u> $S_{pr} = 0$
64: <u>case 29 :</u> $d'_p = 0$
65: **case** 30 : $r = 0$
66: **case** 31 : $p' = 0$
67: **if** $(\beta_p \cdot S_{pr} \neq \beta_p \cdot (1 + d'_p \cdot r) \mod p')$ **then return** error
68: **switch** (f) {
69: <u>case 32 :</u> $S_{pr} = 0$
70: <u>case 33 :</u> $\beta_p = 0$
71: <u>case 34 :</u> $d'_p = 0$
72: **case** 35 : $r = 0$
73: <u>case 36 :</u> $R_3 = 0$
74: **case** 37 : $p' = 0$
75: }
76: $S'_p = (S_{pr} - \beta_p \cdot (1 + d'_p \cdot r) - R_3) \mod p'$
77: **switch** (f) {
78: **case** 38 : $q = 0$
79: **case** 39 : $r = 0$
80: }
81: $q' = q \cdot r^2$
82: **switch** (f) {
83: **case** 40 : $q' = 0$
84: <u>case 41 :</u> $m = 0$
85: }
86: $m_q = m \mod q'$
87: **switch** (f) {
88: **case** 42 : $q = 0$
89: **case** 43 : $r = 0$
90: }
91: $i_{qr} = q^{-1} \mod r^2$
92: **switch** (f) {
93: <u>case 44 :</u> $i_{qr} = 0$
94: **case** 45 : $q = 0$
95: }
96: $\beta_q = q \cdot i_{qr}$
97: **switch** (f) {
98: <u>case 46 :</u> $\beta_q = 0$
99: **case** 47 : $q' = 0$
100: }
101: $\alpha_q = (1 - \beta_q) \mod q'$
102: **switch** (f) {
103: **case** 48 : $\alpha_q = 0$
104: **case** 49 : $m_q = 0$
105: **case** 50 : $q' = 0$
106: <u>case 51 :</u> $\beta_q = 0$
107: **case** 52 : $r = 0$
108: }
109: $\hat{m}_q = (\alpha_q \cdot m_q + \beta_q \cdot (1 + r)) \mod q'$
110: **switch** (f) {
111: **case** 53 : $\hat{m}_q = 0$
112: **case** 54 : $m = 0$
113: **case** 55 : $q = 0$
114: }
115: **if** $(\hat{m}_q \neq m \mod q)$ **then return** error
116: **case** 56 : $m_p = 0$
117: **case** 57 : $m_q = 0$
118: **case** 58 : $r = 0$
119: }
120: **if** $(m_p \mod r^2 \neq m_q \mod r^2)$ **then return** error
121: **switch** (f) {
122: <u>case 59 :</u> $d_q = 0$
123: <u>case 60 :</u> $R_2 = 0$
124: **case** 61 : $q = 0$
125: }
126: $d'_q = d_q + R_2 \cdot (q - 1)$
127: **switch** (f) {
128: **case** 62 : $d'_q = 0$
129: <u>case 63 :</u> $\hat{m}_q = 0$
130: **case** 64 : $q' = 0$
131: }
132: $S_{qr} = \hat{m}_q^{d'_q} \mod q'$
133: **switch** (f) {
134: **case** 65 : $d'_q = 0$
135: **case** 66 : $d_q = 0$
136: **case** 67 : $q = 0$
137: }
138: **if** $(d'_q \neq d_q \mod (q-1))$ **then return** error
139: <u>case 68 :</u> $\beta_q = 0$
140: <u>case 69 :</u> $S_{qr} = 0$
141: <u>case 70 :</u> $d'_q = 0$
142: **case** 71 : $r = 0$
143: **case** 72 : $q' = 0$
144: **if** $(\beta_q \cdot S_{qr} \neq \beta_q \cdot (1 + d'_q \cdot r) \mod q')$ **then return** error
145: **switch** (f) {
146: <u>case 73 :</u> $S_{qr} = 0$
147: <u>case 74 :</u> $\beta_q = 0$
148: <u>case 75 :</u> $d'_q = 0$
149: **case** 76 : $r = 0$
150: <u>case 77 :</u> $R_4 = 0$
151: **case** 78 : $q' = 0$
152: }
153: $S'_q = (S_{qr} - \beta_q \cdot (1 + d'_q \cdot r) - R_4) \mod q$
154: **switch** (f) {
155: <u>case 79 :</u> $S'_q = 0$
156: **case** 80 : $q = 0$
157: **case** 81 : $i_q = 0$
158: <u>case 82 :</u> $S'_p = 0$
159: **case** 83 : $p' = 0$
160: }
161: $S = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
162: **switch** (f) {
163: **case** 84 : $p = 0$
164: **case** 85 : $q = 0$
165: }
166: $N = p \cdot q$
167: **switch** (f) {
168: **case** 86 : $N = 0$
169: <u>case 87 :</u> $S = 0$
170: <u>case 88 :</u> $R_4 = 0$
171: **case** 89 : $q = 0$
172: **case** 90 : $i_q = 0$
173: <u>case 91 :</u> $R_3 = 0$
174: **case** 92 : $r = 0$
175: }
176: **if** $(N \cdot [S - R_4 - q \cdot i_q \cdot (R_3 - R_4)] \neq 0 \mod N \cdot r^2)$ **then return** error
177: **case** 93 : $q = 0$
178: **case** 94 : $i_q = 0$
179: **case** 95 : $p = 0$
180: **if** $(q \cdot i_q \neq 1 \mod p)$ **then return** error
181: **return** $S \mod N$

B Details concerning the success probabilities of fault attacks

In this Appendix, we would like to give more details about the computation of the probabilities presented in Sect. 5. Noting $|x|$ the size of x.

Assume that the attacker modifies value A ($A = B \mod C$) and that C is a uniform, t-bit integer. We suppose that C is odd (r is odd according to the recommendations in Sect. 5, as well as p and q) and we force $2^{t-1} < C < 2^t$. Note $S = \{C : 2^{t-1} < C < 2^t \ and \ C = 1 \mod 2\}$.

We note U the event that the fault is undetected and F the event of taking an element c in S s.t. $c = C$. So, $Pr[U|F]$ is the probability that an event is undetected assuming F. Since the final result will depend only on the initial values which are uniformly distributed (the only exception may be the message m. To avoid this case, we can assume that the message used is the message obtained after a padding - like OAEP-. So the resulted m will also be uniformly distributed), we know that:

$$Pr[U|F] = \frac{1}{C} \ and \ Pr[F] = \frac{1}{|S|}$$

and then

$$Pr[U] = \sum_{C \in S} (Pr[U|F] \cdot Pr[F]) = \frac{1}{|S|} \cdot \sum_{C \in S} \frac{1}{C}$$

Let $\overline{S} = \{C : 2^{t-1} < C < 2^t \ and \ C = 0 \mod 2\}$, the

$$\sum_{C \in S \cup \overline{S}} \frac{1}{C} = [lnC]_{2^{t-1}}^{2^t} = ln(2^t) - ln(2^{t-1}) = ln2$$

We consider approximately that $|S| = |\overline{S}|$. Then:

$$Pr[U] = \frac{1}{|S|} \cdot \sum_{C \in S} \frac{1}{C} \approx \frac{1}{|S|} \cdot \frac{1}{2} \cdot \sum_{C \in S \cup \overline{S}} \frac{1}{C} = \frac{1}{|S|} \cdot \frac{ln2}{2} = \frac{1}{2^{t-2}} \cdot \frac{ln2}{2} = 2^{-(t-1)} ln2$$

This is the obtained probability for the faults: 22, 28 and 32 with $t = |p'|$, 63, 69 and 73 with $t = |q'|$.

Supposing now, that the attacker modifies a value A ($A = B \mod C^2$). Following the same reasoning, we conclude that :

$$Pr[U] \approx 2^{-2t+1}$$

This is the obtained probability for the faults: 6, 8, 13, 27, 29, 33, 34, 41, 44, 46, 51, 68, 70, 74, 75, 79, 82, 87, 88 and 91 with $t = |r|$.