High-Level Simulation for Multiple Fault Injection Evaluation

Maxime Puys*, Lionel Rivière[1,2], Julien Bringer[1], and Thanh-ha Le[1]

[1] SAFRAN Morpho, France –`first.name@morpho.com`
[2] Télécom Paristech, France – `first.name@telecom-paristech.fr`

Abstract. Faults injection attacks have become a hot topic in the domain of smartcards. This work exposes a source code-base simulation approach designed to evaluate the robustness of high-level secured implementations against single and multiple fault injections. In addition to an unprotected CRT-RSA implementation, we successfully attacked two countermeasures with the high-level simulation under the *data* fault model. We define a filtering criterion that operates on found attacks and we refine our simulation analysis accordingly. We introduce a broader fault model that consists in skipping C lines of code and exhibit benefits of such high-level fault model in term of simulation performance and attack coverage.

1 Introduction

Effects of physical attacks on secure implementations were first described in 1997. In particular, fault injection attacks aim at modifying a program's state using an external event such as laser beams, voltage glitches or electromagnetic waves. Among the numerous possibilities opened by such attacks, an attacker can perform a Differential Fault Analysis (DFA) [1,2] and retrieve secret information such as embedded cryptographic keys.

Smartcard-based products, which are widespread in the daily life, can be a profitable target for attackers. Banking or biometric credentials for instance are stored on such devices. They may be sensitive to such fault attacks and therefore require a high security certification standards such as the Common Criteria [3]. Their compliance to those standards are evaluated by governmental agencies that deliver certificates accordingly. One of the very first step of a certification process is the security code review. In order to bring out vulnerabilities, evaluators and developers perform high-level security code reviews at the source code level. Suspected points are then audited at the assembly level, but not systematically. However, mostly manual, this task is time consuming and error prone. A code reviewer could miss critical errors that might lead to a major security breach.

These two reasons encourage the development of automated high-level code analysis to help evaluators and developers. Complete and exhaustive approaches

* Work done while the author was in internship at Morpho

are often used at the binary level where all impacts of a fault can be considered (such as code operation modification) but this takes a long time to perform. In order to rapidly point out vulnerabilities in the security evaluation process, the high-level fault simulation becomes definitely useful. It constitutes a complementary step to other following low-level simulated or practical analyses. From the research point of view, on the one hand, we must provide fault models as accurate as possible on which simulation can rely with a high level of confidence. On the other hand, a comprehensive understanding of fault properties would allow us to better analyze its impacts on software and to design more efficient countermeasures accordingly.

Contributions. We propose an efficient high-level approach to analyze source codes against multiple fault injections. To achieve this end, we built tools to exhaustively explore a data fault model defined a the C variable level. Helped by an oracle, they can for each combination of faults tell if an attack worked on the program. Thanks to the testing approach we are able to produce detailed counterexamples or state on the program's robustness with respect to the chosen fault model. We demonstrate the validity of our approach on three implementations of the CRT-RSA [4] algorithm in the signature process by performing BellCoRe attacks [1]. Moreover, we propose two attack classification criteria aiming at regrouping them, which becomes time-saving when dealing with multiple faults on realistic implementations. Finally, we study the results of the analysis with a line-skip fault model, lighter than the data fault model.

Organization of the paper. In section 2, we define useful terms widely used to express several concepts and security notions. We also recall the CRT-RSA algorithm and the BellCoRe attack. Section 3 further explains the considered fault model and the approach we use to evaluate implementations. Section 4 shows the results of our tests on the three CRT-RSA implementations. Finally, section 5 defines two classification criteria and discusses outcomes obtained from the three examples under the line-skip fault model.

2 State-of-the-art and definitions

2.1 Terminology

This section proposes succinct definitions of four notions: an attack, a fault, a security breach and a vulnerability. We explain how these notions are related and we will refer to these definitions all along this paper.

A **security breach** is the deviation of a program from its expected behavior in terms of security. A **vulnerability** names the presence of an error or lack of security in the program that might lead to a security breach. This term will be defined more precisely in section 5 using our classification criteria. A **fault** represents an external event changing the program's state. Effects of fault and its behavior is formalized through a fault model. A fault model is characterized by its parameters such as the attacker *control* or the *fault persistence*. The spatial *control* parameter for instance, reflects the ability of the attacker to accurately locate its target in the source code whereas the timing *control* reflects his ability to tamper with its target at a particular moment during an ongoing computation. The *fault persistence* reflect the duration of a fault. For instance, for a given attacked variable v, the fault persistence is said transient when a single evaluation of v differs from its expected value. It is said permanent when v keeps a wrong state for each remaining evaluations. An **attack** is the exploitation of a vulnerability by a fault. In [5], authors refined this definition with the presence of a goal for the attacker. However, we will confine to the basic definition.

2.2 CRT-RSA

We choose to analyze the well known asymmetrical CRT-RSA algorithm [4]. Let's say that Alice wants to send the message m to Bob. She has to sign m using her RSA private key (d, N) and then she computes the signature $S = m^d \mod N$. The idea behind the CRT algorithm is to replace the costly modular exponentiation of RSA with two sub-exponentiations with half the size of the original exponent. This roughly speeds up the computation by a factor of four.

Hence, the RSA private key d is split in two parts d_p and d_q. The inverse of q modulo p is denoted i_q. We obtain:

$$\begin{aligned} d_p &= d \mod (p-1) \\ d_q &= d \mod (q-1) \\ i_q &= q^{-1} \mod p \end{aligned}$$

The two modular sub-exponentiation are realized as following :

$$\begin{aligned} S_p &= m^{d_p} \mod p \\ S_q &= m^{d_q} \mod q \\ S &= S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p) \end{aligned}$$

This last step *recombines* the two sub-signatures S_p and S_q in the final signature S. It can be performed using either Gauss or Garner's formula. The latter is the most used because as it provides better memory performances. This is the one we presented in the algorithm.

2.3 BellCoRe attack on CRT-RSA

This attack has been discovered in 1997 by Boneh, DeMillo and Lipton [1] from BellCoRe (Bell Communications Research). An attacker is able to retrieve a prime factor p or q of N if he is able to inject a fault in the signature computation in order to obtain a faulty signature $\widehat{S}$ such as:

1. $|\widehat{S}| \neq |S|$
2. And $|\widehat{S} \mod p| = |S \mod p|$ or $|\widehat{S} \mod q| = |S \mod q|$

The attacker is then able to retrieve either p or q by computing $gcd(N, S - \widehat{S})$. In 1999, Joye, Lenstra and Quisquater [6] showed that this attack can use only the faulty signature $\widehat{S}$ and the message m and retrieve either p or q by computing $gcd(N, m - \widehat{S}^e)$ with an overwhelming probability.

2.4 Code Security Properties

Security metrics aim at defining quantitative and objective criteria in order to gauge various aspect of security. It can be considered in multiple ways [7] and takes several form [8,9]. In the context of smartcards implementation robustness testing against fault injection and considering a manual security code review, they are designed to facilitate decision-making and improve the code robustness. Measuring how the targeted implementation deviates from its functional specification under fault injection constitutes the *correctness* aspect of security. It is the assurance that the targeted function carries out its task precisely to the specification with the expected behavior. Another metric to assess the *efficiency* of a simulation tool is needed in order to confront specific tools.

Measurement requires realistic assumptions and inputs to attain reliable results [10]. The qualitative and quantitative properties of a security objective must be defined. In our case, the security objective is to preserve the correctness of an ongoing RSA ciphering or signature under fault injections to avoid BellCoRe attacks. The quantitative aspect of such an objective lies in the number of deviation from the expected behavior. According to Section 2.1, it corresponds to the number of security breaches. The qualitative aspect corresponds to the nature of the attack (the fault model), and the way it deviates from the reference.

The classification of deviant cases permit to define the criticality level of found attacks. Thereby, we can measure the code sensitivity or robustness of a targeted code under fault attack, and determine potential vulnerabilities according to a measurement system, a fault model and a simulation tool.

2.5 Existing Works

In [11,12], Christofi et al. propose a formal method to validate cryptographic implementations against first order fault injections relying on theorem proving. The fault targets a C variable and sets it to zero. Their studies lead them to the implementation of a *Frama-C* plugin named *TL-Face* and using the *Jessie* plugin in order to solve *weakest precondition* problems. A case study has been made on the Vigilant implementation of CRT-RSA, revealing possible BellCoRe attacks.

In [13], Rauzy et al. study the effects of a first order fault on several CRT-RSA implementations. Their fault model consists in replacing any intermediate value with either zero or a random value. No mathematical property such as co-primality or equivalent modulo is considered. Their analyses lead to the implementation of an `OCaml` tool testing exhaustively every possible faults. It has been used to compare an unprotected implementation of CRT-RSA with the

Shamir [14] one and the Aumüller one [15]. In [16], they extend their approach on Vigilant and Coron's counter-measures and provide high-order attacks. Note that their approach targets values that are used in mathematical operations only.

In [17], Kauffmann-Tourkestansky works on a first order fault model targeting control flow in order to skip instructions (using NOP or JUMP instructions). He uses a mutation analysis of C source codes and tries to fill the gap between high-level and low-level implementations.

In [18], Heydemann et al. also focus on an instruction skip based fault model. They propose a set of counter-measures applicable to every instruction of the Thumbs2 instruction set of the ARM language [19]. They suppose that it is hard for an attacker to reproduce twice the same fault in a few cycles delay and give a way to duplicate each instruction. Finally, they prove that this mutated program (with all its instruction duplicated) has the same behavior than the original using the Vis model-checking tool [20].

In [21], Berthier et al. propose a brand new approach for evaluating smart-cards security against first order fault injection. It consists in embedding the fault simulator itself directly on the smartcard. This way, faults are tested on the final product, which is more reliable than on a software model. Moreover, it also enables the possibility to study the behavior of the card after injections using side-channel analysis. Their fault model targets byte skipping of an arbitrary length. They put it into practice on an implementation of a DES cipher, revealing for instance a fault that skip a function call, which compromised the security of the implementation.

In [5], Potet et al. study the effects of a test inversion based fault model. The analysis is objective guided, in term of reaching or not basic blocks. The fault model is exhaustively explored by a mutation approach. Moreover, in order to take into account higher order faults, that would cause path explosion (and in their case, mutants number explosion), they only create one higher-level mutant. It embeds on its own the *possibility* of injecting each possible fault or not. The paths are then covered by the concolic execution tool Klee [22].

Those existing works emphasize the importance of considering both data and control flow fault models in secure implementation robustness evaluation. With our high-level fault simulation approach presented in the next section, we consider both models while ensuring a very efficient detection and high performances.

3 High-Level Simulation Approach

Section 3.1 defines two reachable fault models considered with fault simulator, namely the data and the line-skip fault models. Section 3.2 defines the testing protocol used to evaluate implementations.

3.1 Mechanisms

The fault simulator operates at the source code level, considered fault models are defined with this granularity accordingly.

Granularity:	: C variable
(Spatial | temporal) control	: Complete | Limited
Persistence:	: Permanent or transient
Multiplicity:	: First order or higher
Type:	: Set to 0 or 1

Fault model 1: Data

This model is a subset of the one proposed in [23,13]. Both of them allow the attacker to perform permanent and transient faults on every variable and intermediate values. On the opposite, our model only allow permanent faults on variable and transient faults on intermediate values, which is more realistic. It also assumes that an attacker can not modify the secret key, the message to sign or the signature, which should pass integrity checks at any time.

The *instruction-skip* fault model is explored in previous works [21,24,25], but almost exclusively at the assembly code level. To our knowledge, no experiment targets CRT-RSA under such fault model in the literature. As there is no assembly instruction notion at the C code level, we propose a high-level extension of the *instruction-skip* fault model as follows:

Granularity	: C code line
Skip Width	: One C code line
(Spatial | temporal) control	: Complete | Limited
Persistence	: Transient
Multiplicity	: First order or higher
Type	: Skip of lines in C source code

Fault model 2: Line-skip

In practice, the fault simulator has been designed to permit arbitrary width line skips. However, a C line of code is often represented by several lines of assembly. Knowing that in the current state of the art, it remains difficult to skip multiple assembly lines, an attacker will unlikely be able to skip multiple C lines of code. Then, we will only consider faults with a width of one line.

Table 1 summarizes the differences of our fault models with state of the art, on several general criteria in order to show the diversity of existing analysis on this topic. HL denotes High-Level abstraction and LL denotes Low-Level abstraction.

The simulator mostly relies on a testing approach in the sense that the targeted implementation is not modified between simulations. However, a single mutation might be needed to decompose computations and let the intermediate values appear as one-time variables.

Faults are injected using the well known Gnu Project Debugger (GDB) [26] that makes it possible to pause the execution via breakpoints, change any variable's value and then resume the execution. Moreover, we enhanced the control and efficiency of our simulator by providing automation through Python scripts.

Reference	Abstraction level	Type of fault model	Persistence	High order
[11] [12]	HL (C)	Data-flow	Permanent	
[13] [16]	HL (OCaml)	Data-flow	Transient	✓
[17]	HL (C) / LL	Control-flow	Transient	
[18] [24]	LL	Control-flow	Transient	
[21]	LL	Control-flow	Transient	
[5]	Intermediate (LLVM)	Control-flow	Transient	✓
Our approach	HL (C)	Data-flow Control-flow	Permanent Transient	✓

Table 1. Comparison of our approach with state-of-the-art

3.2 Test Protocol

Our testing protocol currently targets any single function in an implementation. Several parameters can be tuned to specify the fault model such as the fault multiplicity or which variable to attack or not. If not specified, every global variable of the file, local variable and parameter of the function will be faulted at each line of the function.

GDB commands are controlled by Python scripts, which, for each combination of faults (aka targeted variables, injection lines and new values set), will request GDB to:

1. Execute the target;
2. Set breakpoints where the fault shall be injected;
3. Inject the faults;
4. Get the system state post execution;
5. Repeat this sequence until the fault model is exhaustively explored.

This will spot possible attacks with respect to an oracle defined by the user. As we will see, there might be many of them.

We choose to allow the simulator to change the value of a variable even if this variable is not used at the targeted line. It means that every variable will be forced to every possible value (zero or one) at each line. This possibility could be realistic for example in a system where the values of the variables are stored in memory and loaded each time they are read. In a system where this assertion does not stand (basically all system with registers), only the realistic attacks will be included in the total set of attacks found.

Instinctively, such flexibility will create a relation between some attacks in a way that they can be regrouped as one generic attack and several ways to reproduce it. Thus, in section 5, we will define precisely what we call redundant attacks. Then we will detail classification criteria in order to regroup such attacks.

4 Case Study

In this section we present a case study in order to show the validity of our tool. We concentrate on the process of signature using CRT-RSA. The objective is to ask a system to sign a random message m with its own key (p, q, d_p, d_q, i_q, N) and to obtain a prime factor p or q of N using a BellCoRe attack.

We will study the results of the simulator on an unprotected CRT-RSA implementation and on one using the Aumüller et al. counter-measure [15]. Both of them will be tested with the data fault model and the line-skip fault model explained in Section 3.1.

4.1 Study of an unprotected implementation of CRT-RSA

```
1 int CRT_RsaSign(int $M$, int $p$, int $q$, int $d_p$, int $d_q$, int $i_q$)
2     $S_p = M^{d_p} \mod p$                              /* Signature modulo p */
3     $S_q = M^{d_q} \mod q$                              /* Signature modulo q */
4     $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$  /* Recombining */
5     return $S$
```

Listing 1.1. Unprotected implementation of CRT-RSA

Simulation 1 targets the unprotected CRT-RSA given in Listing 1.1 with first order attacks under the data fault model. For the rest of this paper, we will describe each simulation experiment with the following structure:

Target function	: Unprotected CRT_RsaSign
Success oracle	: Success of a BellCoRe attack on the signature
Fault model	: Data
Fault multiplicity	: 1 (first order)
Result	: **11 attacks found**

Simulation 1: Data model on unprotected CRT-RSA (first order faults)

By $\mathcal{O}(S, \widehat{S}, N)$ we denote the success oracle that returns true if a BellCoRe attack succeed with the given parameters, false otherwise.

Data attack example 1 The unprotected implementation of CRT-RSA is prone to numerous attacks. For instance, forcing the value of S_p to zero prior to the execution of line 4 reveals the prime factor q of N. Indeed,

$$S - \widehat{S} = q \cdot ((i_q \cdot (S_p - S_q) \mod p) - (i_q \cdot (-S_q) \mod p)) \text{ and } gcd(N, S - \widehat{S}) = q$$

Data attack example 2 An even clearer attack consists in zeroing the whole intermediate value $q \cdot (i_q \cdot (S_p - S_q) \mod p)$ prior to the execution of line 4 will result in $\widehat{S} = S_q$, thus we have a BellCoRe attack:

$$|\widehat{S}| \neq |S| \text{ and } |\widehat{S} \mod q| = |S \mod q|$$

Target function	: Unprotected CRT_RsaSign
Success oracle	: Success of a BellCoRe attack on the signature
Fault model	: Line-skip
Fault multiplicity	: 1 (first order)
Result	: **4 attacks found**

SIMULATION 2: Line-skip model on unprotected CRT-RSA (first order faults)

Simulation 2 targets the same unprotected CRT-RSA with first order attacks according to the line-skip fault model. Results are shown below:

Line attack example 1 Skipping the line computing $i_q \cdot (S_p - S_q) \mod p$ (on line 4 of listing 1.1) led to $\widehat{S} = S_q + q \cdot (S_p - S_q)$ which allows a BellCoRe attack.

The line-skip fault model detected four vulnerable lines. Listing 1.1 shows a generic code of the naive RSA where several computations are gathered on few lines. There is a strong dependency between the implementation and the attack success rate with the line skip fault model. The latter can also recover several attacks found by the data fault model and count them as a single one, which explains the lower number of found attacks in Simulation 2.

For instance, if we consider a variable a that is used in an attacked exponentiation, with the data fault model, we can set it either to 0 or 1. However, with the line-skip fault model, the attack output will depend on the initialization value of a. Therefore, if a was initialized to 0, we would recover a *set-to-zero* data fault model. Moreover, if a was not initialized, we would recover a *random* data fault model and finally, if a was initialized to a constant value, we would recover a *set-to-value* data fault model. Even if the two latter data fault models are not directly considered by our simulation, the line-skip fault model can detect them.

4.2 Study of the Shamir implementation of CRT-RSA

The counter-measure of Shamir [14] introduces a new factor r co-primed with p and q, random and small (less than 64 bits). Computations are thus performed modulo $p \cdot r$ (resp. modulo $q \cdot r$), which allows to retrieve the result by reducing modulo p (resp. modulo q). A verification is possible by reducing modulo r.

Our simulator shows that a first order fault is enough to break Shamir's implementation. For example, forcing the value of S_p to zero allows us to obtain the exact same attack than in Simulation 1, as the integrity test only relies on $S_p^{'}$ and $S_q^{'}$.

4.3 Study of the Aumüller implementation of CRT-RSA

The counter-measure of Aumüller [15] has been developed in order to enhance the version of Shamir against first order attacks. It stills introduce a new factor t co-primed with p and q. However, the computation of d_p and d_q is performed outside of the function which removes the use of d. Moreover, the ending verification introduced by Shamir is now *asymmetrical* and intermediate verification are also added. The Aumüller implementation of CRT-RSA is given in listing 1.2.

On the Aumüller implementation of CRT-RSA, our results matches the one exposed in [13]. No first order attack is found by setting any variable or intermediate data to zero.

```
int CRT_RsaSign(int $M$, int $p$, int $q$, int $d_p$, int $d_q$, int $i_q$)
    t = rand()

    $p' = p \cdot t$
    $d'_p = d_p + random_1 \cdot (p-1)$
    $S'_p = M^{d'_p} \mod p'$                /* Signature modulo p' */

    if (($p' \mod p \neq 0$) or ($d'_p \not\equiv d_p \mod (p-1)$)) {
        $takeCounterMeasure()$ }

    $q' = q \cdot t$
    $d'_q = d_q + random_2 \cdot (q-1)$
    $S'_q = M^{d'_q} \mod q'$                /* Signature modulo q' */

    if (($q' \mod q \neq 0$) or ($d'_q \not\equiv d_q \mod (q-1)$)) {
        $takeCounterMeasure()$ }

    $S_p = S'_p \mod p$
    $S_q = S'_q \mod q$
    $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$   /* Recombining */

    if (($S - S'_p \not\equiv 0 \mod p$) or ($S - S'_q \not\equiv 0 \mod q$)) {
        $takeCounterMeasure()$ }

    $S_{pt} = S'_p \mod t$
    $S_{qt} = S'_q \mod t$
    $d_{pt} = d'_p \mod (t-1)$
    $d_{pt} = d'_p \mod (t-1)$

    if ($S_{pt}^{d_{qt}} \not\equiv S_{qt}^{d_{pt}} \mod t$) { $takeCounterMeasure()$ }
    else { return $S$ }
```

Listing 1.2. Aumüller implementation of CRT-RSA

Simulation 3 targets the CRT-RSA implementation protected with the Aumüller countermeasure shown in Listing 1.2 above. Second order attacks are performed according to the data fault model, we obtain:

The Aumüller implementation is not robust against second order attacks using this data fault model. The first fault is used to corrupt the computation while the second avoids the counter-measure to be triggered.

Target function	: Aumüller CRT_RsaSign
Success oracle	: Success of a BellCoRe attack on the signature
Fault model	: Data
Fault multiplicity	: 2 (second order)
Result	: **802 attacks found**

Simulation 3: Data model on Aumüller CRT-RSA (second order faults)

Data attack example 3 A *Set-to-one* fault on $(p-1)$ before the execution of line 5 sets up a BellCoRe attack. Secondly, performing the same fault on $(p-1)$ before the execution of line 8 avoids triggering the counter-measure.

Data attack example 4 The attack presented for unprotected and Shamir implementations consisting in setting the whole intermediate value $q\cdot(i_q\cdot(S_p-S_q) \mod p)$ to zero before the execution of line 18 stills enables a BellCoRe attack. However, it will also trigger the counter-measure of line 20 (through the test $S - S_p^{'} \not\equiv 0 \mod p$). A second fault will disable it by setting the intermediate value $S - S_p$ to zero.

Such a huge number of attacks (802 in Simulation 3) makes results impossible to analyze by hand. Moreover it is obvious that most of these attacks found can be regrouped into some generic attacks with different ways to reproduce them. This example definitely shows the necessity of classification criteria and metrics.

Simulation 4 targets the same protected CRT-RSA with the Aumüller countermeasure but second order attacks are performed according to the line-skip fault model. Results are shown below:

Target function	: Aumüller CRT_RsaSign
Success oracle	: Success of a BellCoRe attack on the signature
Fault model	: Line-skip
Fault multiplicity	: 2 (second order)
Result	: **13 attacks found**

Simulation 4: Line-skip model on Aumüller CRT-RSA (second order faults)

Line attack example 2 Skipping the line computing $i_q \cdot (S_p - S_q) \mod p$ (on line 18 of listing 1.2) led to $\widehat{S} = S_q + q \cdot (S_p - S_q)$ which allows a BellCoRe attack. It also triggers the counter-measure on line 20 which can be easily skipped by our fault model at the second order. This attack is very similar to the Data attack example 4.

Thirteen attacks are spotted by the line-skip fault model. Interestingly, the number of attacks found by the line-skip fault model in Simulation 4 is drastically lower to the one found by the data fault model in simulation 3. This can be explained knowing that it only depends on the lines while the data fault model also depends on variables and values. We will present deeper analyses of the link between these two models in section 5.

To the best of our knowledge, some studies showed that the Aumüller implementation of CRT-RSA is weak against multiple fault injection such as [27] but none provides detailed experimental results.

5 Advanced Analysis

Criteria. We recall that a successful attack is the exploitation of a code vulnerability, which is induced by the fault injection. According to the transient value modification fault model, we provide a variable-centric criteria $\mathcal{C}_{val}$ by which we measure the code sensitivity. $\mathcal{C}_{val}$ only depends on the value taken by the targeted variable regardless the attacked line of code. It is defined as:

$$\mathcal{C}_{val}(targeted\_var, value) := \#\{line \mid \mathcal{O}(S, \widehat{S}, n) = true\}$$

For a given couple $(targeted\_variable, value)$, $\mathcal{C}_{val}$ describes the number of successful attacks obtained by setting $targeted\_variable$ to $value$ regardless of the lines. We define as non-redundant, the successful attack that has the greatest injection line number.

	Unprotected (order 1)	Shamir (order 1)	Aumüller (order 1)	Aumüller (order 2)
Attacks found	11	15	0	802
Non-redundant attack	9	11	0	85

Table 2. Found Attacks with the Fault Simulation with Data Fault Model

Table 2 summarizes the experimental results obtained by the fault simulator on CRT-RSA implementations with the data fault model on C variables. The first line shows how many attacks the simulator has found on each implementation. The second line displays how many non-redundant attacks are found. When the same targeted variable is modified to the same value at several different lines of the code, it describes the same attack. We call such groups of lines a vulnerability. This is why the second line of Table 2 report a lower number of found attack.

Line Skip Fault Model. We now provide a line-centric criteria $\mathcal{C}_{line}$ that for each line of the implementation under testing returns a boolean value depending on the presence of an attack with an injection on this line. In the end, for a given implementation, this criteria returns the set of lines where injecting a fault results in an attack. It is defined as follows:

$$\mathcal{C}_{line}(l, m) := \begin{cases} true & \text{if } \exists\ \mathcal{A}(l, m) \mid \mathcal{O}(S, \widehat{S}, N) = true \\ false & otherwise \end{cases}$$

For a given line l, $\mathcal{C}_{line}$ returns true if there exist a successful attack $\mathcal{A}$ on line l according to the fault model m denoted as $\mathcal{A}(l, m)$. For each implementation that we studied, we retrieve vulnerable lines filtered by $\mathcal{C}_{line}$ under the data fault model. Then, we check how many of these lines are also vulnerable under the line-skip fault model. For readability purpose, we define

$$Vulnerable\ lines_{model} := \{l \mid C_{line}(l, model) = true\}$$

Thereby, we compute the following ratio:

$$Gain := \frac{\#\{Vulnerable\ lines_{data} - Vulnerable\ lines_{lskip}\}}{\#\{Vulnerable\ lines_{lskip}\}}$$

It happened that for each of our implementations, this gain ratio was 100% even considering redundant attacks. This means that each vulnerable line found with the data fault model is also vulnerable with the line-skip fault model.

Moreover, as we can see in table 3, the first (resp. second) line displays the time taken by our tool for each implementation using the data (resp. line-skip) fault model. The third line is the ratio of the first and second lines. We can here conclude that simulations using the line-skip fault model are much quicker than using the data fault model. This is especially true when dealing with higher order faults.

	Unprotected (order 1)	Shamir (order 1)	Aumüller (order 1)	Aumüller (order 2)
Data	3,53 s	38,75 s	143,93 s	1361,45 mn
Line Skip	2,18 s	2,39 s	14,31 s	7,41 mn
Gain ratio	162 %	1621 %	1006 %	18373 %

Table 3. Line-skip Fault Model And Fault Simulation Timing Performances

In our experiments, every attack found by the data fault model seems to be reproducible using the line-skip fault model. As mentioned in section 4.1, the general link between the two fault models is not straightforward. On one hand, the general effects of a line skip on a variable is a *Set-to-last-value*. It differs from *Set-to-0/1*. If we consider that an attacker has no control on intermediate values, the line-skip fault model can be seen as a *uncontrolled* value fault model. On the other hand, considering a single line of code, the data fault model targets only one variable while the line-skip fault model targets every modified variables of the line. Despite these differences, it still becomes really useful to run line-skip based simulations prior to data based simulation at high-level as it produces similar effects. Moreover, the large simulation performance improvement justifies this choice.

Finally, high-level line-skip allows discover attacks completely invulnerable to data fault models such as removing a `break` statement in a `switch` or adding/removing loop iterations. As a matter of fact, the last has been put into practice in [28] where the authors are able to increase the leakage of an AES key for side channels analysis. Generally, these high-level faults models can be considered as the consequence of low-level faults encompassing several of them.

Representativeness Discussion. Attacking the source code structure can break the data or the control flow as the targeted source code could represent one or several instructions, which can manipulate data or not. Post compilation, a human readable C variable or loop counter will consist in a memory address for the machine. An `if-then-else` expression, a `switch` or `break` statement or an arithmetic operation will consist in a hardware encoded instruction for the machine. A possible way to improve our approach could be the use of Abstract Syntax Trees (AST) instead of C lines. This Intermediate Representation (IR) would abstract the code structure but keep its semantic and therefore improve our approach accuracy and representativeness.

6 Conclusion

We propose an approach to evaluate the robustness of secured implementations against multiple fault injections. This approach works on high-level source codes (such as C). We propose a first fault model relying on data modification with the granularity of a C variable. This fault model has been automated in a Python tool that is able to try it exhaustively on a given implementation and log out the functions outputs. Helped by oracles, it can tell whether a combination of faults results in a attack or not.

We demonstrate the validity of our tool on three examples of CRT-RSA implementations and obtained results according to the current state-of-the-art. Moreover, we proposed a second fault model relying on line skipping that is faster. In our experiments, we found that it covers entirely the attacks found by the data fault model with a huge speed gain.

Finally, we proposed a set of criteria and metrics in order to regroup attacks found and quantify in term of security the robustness of an implementation. Further work is to refine the link we found between our data and control-flow fault models.

References

1. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.
2. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
3. CSE, SCSSI, BSI, NLNCSA, CESG, NIST, and NSA. Common Criteria 2. https://www.commoncriteriaportal.org.
4. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.
5. M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: a symbolic approach for evaluation the robustness of secured codes against control flow fault injection. In *ICST*, 2014.
6. M. Joye, A. K. Lenstra, and J.-J. Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *J. Cryptology*, 12(4):241–245, 1999.
7. R.-S. Miani, M. Cukier, B. B. Zarpelão, and L. de Souza Mendes. Relationships Between Information Security Metrics: An Empirical Study. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13, pages 22:1–22:4, New York, NY, USA, 2013. ACM.
8. R. B. Vaughn, R. R. Henning, and A. Siraj. Information Assurance Measures and Metrics - State of Practice and Proposed Taxonomy. In *HICSS*, page 331, 2003.
9. R. Savola. Towards a taxonomy for information security metrics. In G. Karjoth and K. Stølen, editors, *QoP*, pages 28–30. ACM, 2007.
10. W. Jansen. *Directions in security metrics research.* DIANE Publishing, 2010. NISTIR 7564.

11. M. Christofi. *Preuves de sécurité outillées d'implémentation cryptographiques*. PhD thesis, Laboratoire PRiSM, Université de Versailles Saint Quentin-en-Yvelines, France, 2013.
12. M. Christofi, B. Chetali, L. Goubin, and D. Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptographic Engineering*, 3(3):157–167, 2013.
13. P. Rauzy and S. Guilley. A Formal Proof of Countermeasures against Fault Injection Attacks on CRT-RSA. volume 2013, page 506, 2013.
14. A. Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. Patent Number 5,991,415, Novembre 1999. also presented at the rump session of EUROCRYPT '97.
15. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. 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.
16. P. Rauzy and S. Guilley. Formal Analysis of CRT-RSA Vigilant's Countermeasure Against the BellCoRe Attack: A Pledge for Formal Methods in the Field of Implementation Security. In S. Jagannathan and P. Sewell, editors, *PPREW@POPL*, page 2. ACM, 2014.
17. X. Kauffmann-Tourkestansky. *Analyses securitaires de code de carte a puce sous attaques physiques simulees*. PhD thesis, Université d'Orléans, 2012.
18. K. Heydemann, N. Moro, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. In *PROOFS 2013*, Santa-Barbara, États-Unis, Aot 2013.
19. ARM Architecture Reference Manual - Thumb-2 Supplement, 2005.
20. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.. Ranjan, S. Sarwary, T. R. Staple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In Rajeev Alur and ThomasA. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer Berlin Heidelberg, 1996.
21. M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: Embedded Fault Injection Simulator on Smartcard. In J. Jürjens, F. Piessens, and N. Bielova, editors, *ESSoS*, volume 8364 of *LNCS*, pages 222–229. Springer, 2014.
22. The KLEE symbolic virtual machine. http://klee.llvm.org/.
23. D. Vigilant. Rsa with crt: A new cost-effective solution to thwart fault attacks. In *Proceeding Sof the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.
24. N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *FDTC*, pages 77–88. IEEE, 2013.
25. V. K. Kosuri and N. Fazal. FPGA Modeling of Fault-Injection Attacks on Cryptographic Devices. In *IJERA*, volume 3, pages 937–943, 2013.
26. http://www.sourceware.org/gdb/.
27. S.-K. Kim, T. H. Kim, D.-G. Han, and S. Hong. An Efficient CRT-RSA Algorithm Secure Against Power and Fault Attacks. *J. Syst. Softw.*, 84(10):1660–1669, Octobre 2011.
28. A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria. Electromagnetic Glitch on the AES Round Counter. In E. Prouff, editor, *COSADE*, volume 7864 of *LNCS*, pages 17–31. Springer, 2013.