From Code Review to Fault Injection Attacks: Filling the Gap using Fault Model Inference

Louis Dureuil[1,2,3]*, Marie-Laure Potet[1,3]**, Philippe de Choudens[1,2], Cécile Dumas[1,2], and Jessy Clédière[1,2]

[1] Univ. Grenoble Alpes, F-38000 Grenoble, France.
[2] CEA, LETI, MINATEC Campus, F-38054 Grenoble, France.
`{louis.dureuil,philippe.de.choudens,cecile.dumas,jessy.clediere}@cea.fr`
[3] CNRS, VERIMAG, F-38000 Grenoble, France.
`{louis.dureuil,marie-laure.potet}@imag.fr`

Abstract. We propose an end-to-end approach to evaluate the robustness of smartcard embedded applications against perturbation attacks. Key to this approach is the fault model inference phase, a method to determine a precise fault model according to the attacked hardware and to the attacker's equipment, taking into account the probability of occurrence of the faults. Together with a fault injection simulator, it allows to compute a predictive metrics, the vulnerability rate, which gives a first estimation of the robustness of the application. Our approach is backed up by experiments and tools that validate its potential for prediction.

Keywords: smartcard, perturbation attack, fault injection, fault model, vulnerability rating, attack potential, electromagnetic attacks

1 Introduction

1.1 Context

Secure devices (smartcards, security tokens, and in the near future mobile phones) are subject to drastic security requirements and certification processes. They must be protected against high level attack potential as described in [9] (i.e. multiple attackers with a high level of expertise, using sophisticated equipments, etc.). As a result, norms (for instance, the Common Criteria) require the vulnerability analysis to follow the state-of-the-art in terms of attacks.[4] Nowadays, a very studied class of attack is perturbation attack, which is performed using electrical glitches, focalised light [6] or electromagnetic injectors [12].[5] Progress

* This work has been partially supported by the project SERTIF (ANR-14-ASTR-0003-01).

** This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025).

[4] We target here the AVA class, dedicated to vulnerability assessment.

[5] Sometimes referred to as "EM probes".

in perturbation techniques allow multiple attacks over the course of a single execution [10] (also known as high-order attacks).

Perturbation attacks typically result in fault injection, which modifies the data and/or control flow of the execution. Fault injection can be exploited to access or modify secure assets on the card and to produce faulty ciphertexts in cryptographic contexts [7]. Codes must be hardened using software countermeasures (redundant tests, integrity counters, *etc.*) in addition to the already mandatory hardware countermeasures. Qualifying the resulting robustness of embedded software against fault injection is a very challenging task. It is nowadays a mainly hand-crafted process that requires various skills from the implied people. Furthermore, the ever-evolving state of the art requires periodic re-evaluations of previously certified embedded software.

1.2 Perturbation attack and Fault model

When assessing robustness, evaluators limit the type of faults they consider to a specific fault model. Fault models vary greatly in the literature, with volatile or non-volatile bit set or reset, register corruption or modifications of a byte or a word in memory [2, 6, 16], and higher-level effects such as test inversion, data reassignment and instruction replacement [4, 11, 13]. The variability of fault models can be explained by several factors: the memory technologies used in the card, its logic circuits, the available hardware countermeasures, the equipment of the attacker and the attack parameters: for instance, in EM injectors the model depends on the angle between the injector and the plane of the card [12].

1.3 Evaluation process

Assessing the robustness of an embedded software against fault injection spans several subprocesses. *Code analysis* aims to detect vulnerabilities in the software from the source and assembly codes and looks for attack paths using a given fault model. *Penetration tests* consist in performing perturbation attacks on the card according to the hardware technology and with some knowledge about the behaviour of the application (for instance, obtained via power consumption analysis).

Based on the results of these two processes, an *attack potential* is determined, according to a scoring grid [9]. *Attack potential* takes into account several factors such as *elapsed time*, the attacker's *expertise*, *knowledge of the target of evaluation (TOE)*, *equipment* and the *availability of open samples*. For each factor, a table establishes the correspondence between possible levels for this factor and identification and exploitability ratings. For instance, the *knowledge of the TOE* can be *public*, *restricted*, *sensitive*, *critical* or *very critical*. The final rating combines the values of identification and exploitability associated to each factor.

1.4 Open problems

Hardening and evaluating embedded software against fault injection is a hot topic as demonstrated by the number of recent studies dedicated to this subject.

As a consequence of the first attack against RSA [7], countermeasures are proposed in cryptographic contexts [6, 10], some of which are formally proved to be robust against a given fault model [8]. Unfortunately, due to the possibility for multiple fault injection, countermeasures can also be attacked [14]. Thus, adding the suitable set of countermeasures becomes a very complex task. Moreover, as pointed out by S. Mangard in his keynote at CARDIS 2014,[6] a significant part of the vulnerabilities discovered by evaluators implies non-cryptographic code.

Another research direction is the development of automated tools simulating fault injection, at the source code [3, 5, 13] or at the binary levels [4, 11]. These tools can be combined with a proof-based approach in order to qualify the robustness of the considered applications, as shown in [3, 8, 13]. Fault injection simulators provide exhaustiveness and reproducibility. But, in return, these tools generally produce a very large number of potential attacks, which requires manual examination to decide if the attacks are actually a serious threat. Indeed, these tools being dedicated to some specific fault models, the attacks they detect are not necessarily achievable on a given hardware component.

1.5 Our approach

In this paper, our contributions are the following: we propose an end-to-end approach, and tools, to respond to these challenges. The proposed approach introduces a preparatory phase designed to infer the most suitable fault model according to the considered hardware, independently of a given application. Fault models take into account the probability of occurrence of faults. The second step is fault injection simulation, which explores the consequences of the obtained fault model on the application. Lastly, we propose a predictive vulnerability rate, based on the results provided by both the inference and fault injection simulation phases, allowing us to classify attacks and to measure robustness. As we will see, this rating gives a partial measure of the attack potential, requiring no penetration testing campaign. Figure 1 summarizes the approach.

The proposed approach allows the evaluator to fine-tune the fault model in order to improve the result of fault injection simulation, and makes the fault model reusable between applications using the same card. Although an alternative method to find fault models through experiments has been proposed in [15], it is not part of an end-to-end process, which is the specificity of our approach. In particular, to our knowledge, no other attempt has been made to combine the results of a fault model deduced at the card level with fault injection simulation, with the goal of producing a vulnerability rating at the application level.

The remainder of this paper is organized as follows: section 2 proposes a formalization of the fault models which can be produced by the inference phase; section 3 proposes a methodology for fault model inference and illustrates it on a case study; section 4 presents CELTIC, a fault injection simulator, and defines the vulnerability rate and sensibility of a location; finally, section 5 presents

[6] "The `if` statement that surrounds the cryptographic implementations".

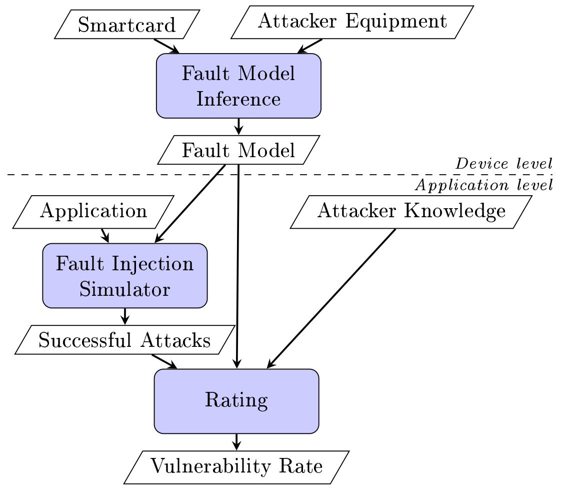

Fig. 1. The overall approach

experiments we conducted to evaluate the vulnerability rate, and gives some future perspectives.

2 Fault model formalization

Fault models are a key part of our approach: they must be easy to specify as they serve as input of fault injection simulation; they must be low-level enough to be described as an output of fault model inference; lastly we aim to retain enough expressiveness to be compatible with classic fault models such as instruction skip or volatile bit (re)set.

2.1 Fault and Fault Model

With this in mind, we define a location, an instant, a fault and a fault model. The location and instant express the classic time and space characteristics of a fault.

Definition 1. *Location $\ell = (storage\_type, id)$: An abstract storage unit of type storage_type (e.g., non-volatile memory such as EEPROM, volatile memory such as RAM, registers) uniquely identified by id (an address in memory, a register name or number, etc.).*

Definition 2. *Instant i: An abstract value that identifies when a fault occurs. It can be expressed in seconds from the beginning of the execution, in numbers of loads from a given location, or number of loads/stores from any location.*

Definition 3. *Fault (i, ℓ, a, b): A replacement b of the value a returned by a load from location ℓ at the instant i.*

Definition 4. *Fault model $FM_{d,e}$: A set of sequences of faults that can actually be injected during the execution of any program on a device d with the perturbation equipment e.*

While this definition of a fault may seem restrictive, it covers classic fault models as well as combination thereof, and faults on code and data:

– *Volatile bit (re)set or byte change on code or data.* Set of faults that modify a load of the address corresponding to the code or data to the specified value.
– *Instruction skip (NOP).* In this model, instructions are skipped, i.e., replaced by NOP (no operation) instructions. The NOP fault model is a set of sequences of faults that replace the loads of the original opcodes and operands of the skipped instructions with NOP opcodes.
– *Non-volatile faults.* Set of sequences of faults that replace the original value returned by all loads from the affected location with the modified value until the next store to the affected location.

2.2 Probabilistic Fault Model

Probabilistic fault models refine fault models by adding two additional key pieces of information: The *probability of occurrence of each kind of fault* and their *relation to the attack parameters.* This way, we aim to capture the notion of plausibility of a fault.

Definition 5. *Attack Parameters p: A tuple of physical quantities that the attacker can measure and choose in a given range of values. We denote as $\mathcal{P}$ the space of the attack parameters.*

Definition 6. *Probabilistic Fault Model $\mathcal{M}_{d,e}$:*

$$Pr(F = f \mid p) \tag{1}$$

where F is a random variable valued in the domain of faults, and represents the fault injected during an attack on device d with equipment e, where f denotes a specific sequence of faults, and p the attack parameters.

In section 3.2, we conduct fault model inference on a commercial, ARMv7-M, secure smartcard (denoted card A) using EM injection.[7] In EM injection, the attack parameters are a tuple $p = (\theta, x, y, z, t)$, where θ is the angle between the probe and the plane of the card, x, y and z give the spatial localization of the probe relatively to the card, and t is the delay before the EM field is applied. Table 1 presents the resulting probabilistic fault model for card A with $\mathcal{P} = \{(-90°, x_0, y_0, z_0, t_0 + j\delta) \mid j \in \mathbb{N}\}$, with x_0, y_0, z_0, t_0 and δ chosen constants.

[7] Our EM injector is made of small copper wire loops (100μm), driven by a 500A current during 10ns.

Table 1. Probabilistic Fault Model for card A under EM perturbation

Fault Sequence	Probability
$< (i_j, \ell_j, a, 0) > \mid a \neq 0$	4.8%
$< (i_j, \ell_j, a, b) > \mid a \neq 0 \wedge \frac{\lvert a-b \rvert}{a} \leqslant 1\%$	1.8%
$< (i_j, \ell_j, a, b) > \mid a \neq 0 \wedge 1\% < \frac{\lvert a-b \rvert}{a} \leqslant 20\%$	1.6%
$< (i_j, \ell_j, a, b) > \mid a \neq 0 \wedge b \neq 0 \wedge \frac{\lvert a-b \rvert}{a} > 20\%$	1.3%
$< (i_j, \ell_j, a, 0), (i_{j+1}, \ell_{j+1}, a', 0) > \mid a \neq 0 \wedge a' \neq 0$	0.5%
$< \varnothing >$ (No fault observed)	90%

3 Fault Model Inference

Probabilistic fault model inference is performed in three steps.

Step 1. Parameter discovery: we determine a space $\mathcal{P}_0$ of attack parameters where faults occur reasonably often and whose size is small enough to carry the rest of the process.

Step 2. Raw fault model construction: we perform many perturbation attacks for each parameter $p \in \mathcal{P}_0$ on the target device running a specific program called the fault detection program.

Step 3. Fault model generalization: we manually infer a more general fault model extending parameters and values.

We detail each of these steps and illustrate them in this section. In the traditional approach, testers perform step 1 identically, while step 2 is conducted directly on the tested application and step 3 is missing, which leads to suboptimal code reviews.

3.1 Fault Detection Program

In [15], the authors propose a method called fault model extraction to establish a fault model from observations of the result of perturbation attacks on a specific test program. But in our understanding, little is done to ensure that the interpretation matches the fault that is actually injected. This is however a difficult problem, because an observation can result from various faults (for instance, a fault on data can result from a volatile bit reset in memory, or from a faulty store instruction). While the exact fault is not of interest when attacking a single application (only the success of the attack matters), it becomes crucial to eliminate context-dependent results when working at the device level.

Our specifically designed fault detection program is a first step in this direction. It directly outputs the fault injected during an execution under perturbation attack, and uses a sentinel to give us confidence that this observation matches the actual injected fault.

Listing 1.1 is an excerpt of the fault detection program, that targets an ARMv7-M architecture and aims at detecting EEPROM faults. Initially, `r0` points to the start of an EEPROM buffer, `r1` to the start of a RAM buffer, `r2` and `r3` to different parts of the output buffer. The program performs a copy of the EEPROM buffer to the output buffer and a copy of the RAM buffer to the output buffer. This program is put in RAM and ran from there. We then perform perturbation attacks to see the faults injected in EEPROM copied to the output buffer. This way, we can see how many EEPROM locations are perturbed as well as the injected values.

The copy of the RAM buffer acts as the sentinel. If the RAM copy is faulty, then it means that the attack perturbed the RAM or registers. In such case, we cannot guarantee the integrity of the code of the fault detection program, and we must discard the result.

This fault detection program can easily be adapted to other devices and architectures as long as they allow execution from RAM. Once the EEPROM fault model has been established, we can swap the roles of RAM and EEPROM in the program to establish the RAM fault model.

```
; main_loop:
  58:    ldrb    r5, [r0, #0] ; r5 <- @EEPROM
  5a:    strb    r5, [r2, #0] ; r5 -> @IO_EEPROM
  5c:    ldrb    r5, [r1, #0] ; r5 <- @RAM
  5e:    strb    r5, [r3, #0] ; r5 -> @IO_RAM
  60:    add.w   r0, r0, #1 ; @EEPROM += 1
  64:    add.w   r1, r1, #1 ; @RAM += 1
  68:    add.w   r2, r2, #1 ; @IO_EEPROM += 1
  6c:    add.w   r3, r3, #1 ; @IO_RAM += 1
```

Listing 1.1. Fault detection program for card A

We now apply the three steps of fault model inference to card A.

3.2 Case Study/Step 1: Parameter Discovery

We tested the influence of the angle θ and the position (x, y) of the injector relatively to the surface of the chip, with z at a fixed value z_0. Regarding the influence of θ, and in accordance with the state of the art [12], we found a majority of bitset faults with the injector parallel to the card (e.g., $\theta = 0°$), and a majority of bit reset faults the injector orthogonal ($-90°$). Figure 2 shows an overlay at (x, y) positions where faults occurred for a fixed θ angle.

3.3 Case Study/Step 2: Raw Fault Model Construction

We attacked the device running our fault detection program repeatedly. We denote as a the value in the EEPROM buffer in the fault detection program. We chose $\mathcal{P}_0 = \{(-90°, x_0, y_0, z_0, t_0 + 10k) \mid k \in \{0, \ldots, 39\}\}$, with 300 values a randomly chosen in [0, 65535] (we additionally tested the special values 0 and 0xFFFF). For each pair (p, a) of parameter and input value, we performed 30 repetitions, for a total of $30 \times 40 \times 300 = 360000$ repetitions which resulted in a 10 days process. From the raw results, we generated Fig. 3, a heat map of the probability that a value a be replaced by the value b.

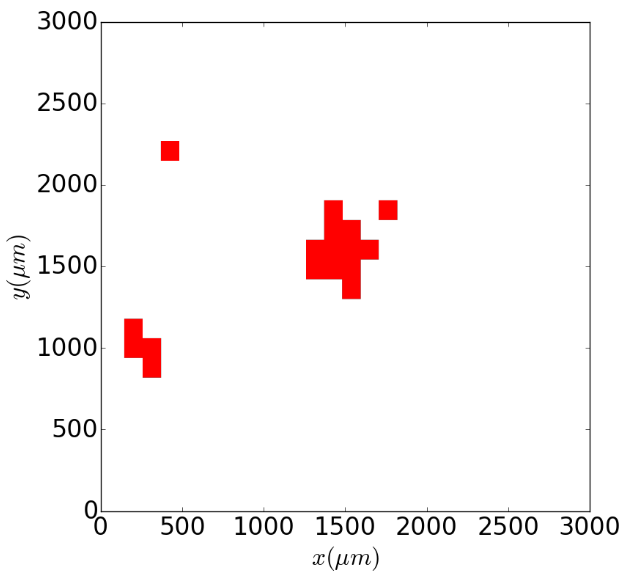

Fig. 2. (x, y) positions where an EEPROM fault is injected with $\theta = -90^\circ$

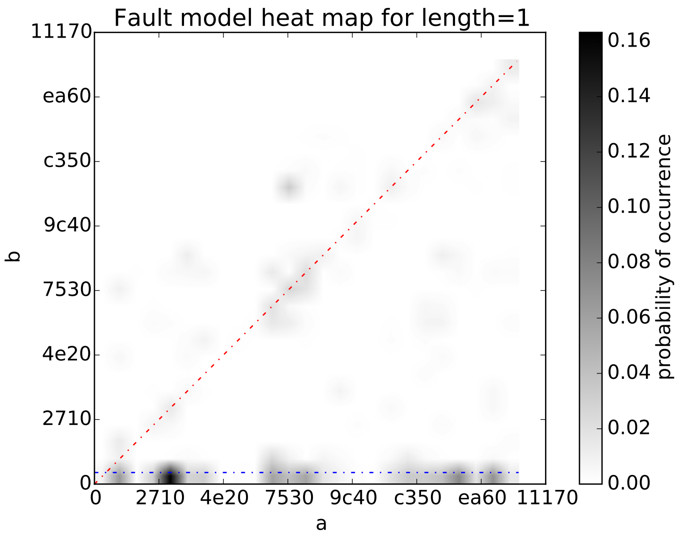

Fig. 3. Heat map of the probability of replacing a with b

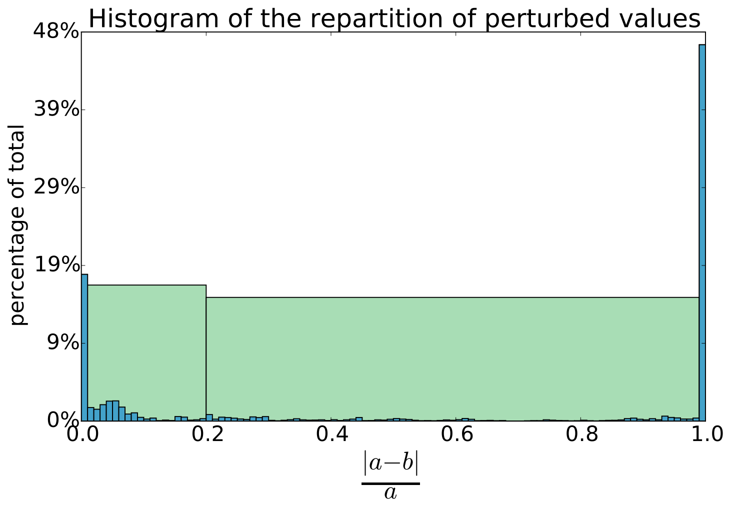

Fig. 4. Histogram of the repartition of perturbed values as a function of their distance to the original values

3.4 Case Study/Step 3: Fault Model Generalization

Fault model generalization is a necessary manual step taken to extend the fault model from the tested parameters to a bigger space of parameters and to generalize the model from our fault detection program to any application. To extend the fault model from the tested values to any value, we noticed in Fig. 3 that a majority of faults reset the original halfword a to 0, but a significant part of the faults result in slight alteration of a, i.e., the perturbed value b is such that the relative difference $d(a, b) = \frac{|a-b|}{a}$ is small.[8] We used this knowledge to build the bins of the histogram in Fig. 4, which illustrates the repartition of the probabilities of occurrence of the faults as a function of $d(a, b)$. We can see the prevalence of the zero ($d(a, 0) = 1$), and of the smallest values of the function (when $d(a, b) < 0.01$), that match the probabilities summarized in Table 1 of section 2.2.

To generalize the space of attack parameters to: $\mathcal{P} = \{(-90°, x_0, y_0, z_0, t_0 + j\delta) \mid j \in \mathbb{N}\}$, we observed the probability of fault injection as a function of time (Fig. 5) and noticed a periodicity $\delta = 720$ns.

Moreover, we observed a relation between the position of the perturbed halfword in the EEPROM buffer and the time parameter. Specifically, if $t = t_0 + j\delta$, then the j^{th} halfword of the buffer is perturbed. This property is essential to generalize the inferred fault model to any application under test as it ensures that there exists a range of time parameters such that any EEPROM location

[8] The *hamming distance* was considered, but gave seemingly less relevant results, with only 16 possible values.

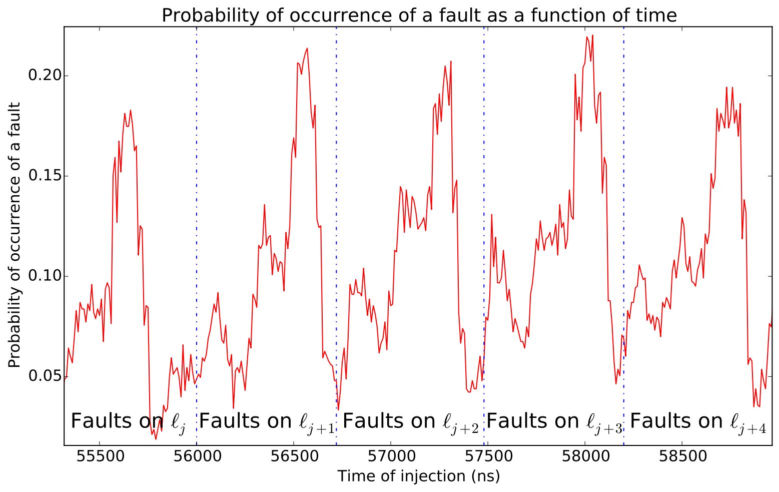

Fig. 5. Probability of fault as a function of time

accessed at some point in the application can be perturbed. Although it does not explicitly give us the correct time parameter, we see in section 4.2 that only its existence is required for vulnerability rating. To our knowledge, such properties expressing the relation between the attack parameters and the space-time characteristics of the fault have not been described in the literature.

3.5 Variability of fault models

Table 2 summarizes the probabilistic fault model we inferred on a secure smartcard B that uses a proprietary CISC instruction set. We attacked with a laser, and chose $\mathcal{P} = \{(\text{power} = 1\text{W}, \text{spot size} = 20\mu m, \lambda = 980nm, x = x_0, y = y_0, d = 100ns, t = t0 + j\delta)\ \}$, where λ is the wavelength of the laser, and d the duration of a laser shot. Comparing the two fault models, we can notice that card B puts values at 0 with much more consistency, but can perturb between 1 and 6 consecutive bytes whereas faults in card A typically perturb a single halfword. This illustrates the variability of fault models, which is linked to the attacked hardware and to the equipment of the attacker.

4 Assessing Robustness at the Application Level

4.1 Automatic Code Analysis

The next step of our approach (see Fig. 1) consists in simulating fault injection at the application level. To do so, we designed and implemented CELTIC,[9] a

[9] CEsti-LeTi Integrated Circuit

Table 2. Fault model for card B

Faults	Probability
$< (i_j, \ell_j, a, 0) >$	4.32%
$< (i_j, \ell_j, a_0, 0), (i_{j+1}, \ell_{j+1}, a_1, 0) >$	2.93%
$< (i_j, \ell_j, a_0, 0), (i_{j+1}, \ell_{j+1}, a_1, 0), (i_{j+2}, \ell_{j+2}, a_2, 0) >$	3.13%
$< (i_j, \ell_j, a_0, 0), \ldots, (i_{j+3}, \ell_{j+3}, a_3, 0) >$	2.98%
$< (i_j, \ell_j, a_0, 0), \ldots, (i_{j+4}, \ell_{j+4}, a_4, 0) >)$	6.56%
$< (i_j, \ell_j, a_0, 0), \ldots, (i_{j+5}, \ell_{j+5}, a_5, 0) >$	2.48%
$< \varnothing >$ (No fault injected)	77.58%

simulator of native smartcard binaries, able to simulate fault injection. CELTIC was implemented in C++.

Listing 1.2 provides a pseudo algorithm for CELTIC. The simulator starts classically with a *golden run* (variable `Xref` in listing 1.2) of the tested native smartcard application, i.e., a run without fault injection (function `simulateWithoutFault` in listing 1.2). The *golden run* allows to gather information which is used to identify all locations that are accessed during the execution, where faults are susceptible of being injected. CELTIC can be configured to use any probabilistic fault model $\mathcal{M}_{d,e}$ inferred following the process of section 3, according to the definition given in section 2.1. Each possible sequence of faults $f =< (i_0, \ell_0, a_0, b_0), \ldots, (i_k, \ell_k, a_k, b_k) >$ is generated from all sequences of accesses $< (i_0, \ell_0, a_0), \ldots, (i_k, \ell_k, a_k) >$ in the *golden run* that match the faults described by $\mathcal{M}_{d,e}$ (function `findAllFaults` in listing 1.2). For each f, the algorithm performs an attack, i.e., a simulation where the matching sequence is replaced with the sequence $< (i_0, \ell_0, b_0), \ldots, (i_k, \ell_k, b_k) >$ (function `simulateWithFaults` in listing 1.2). A user-provided oracle on the state of the simulated processor allows to filter the successful attacks (function `isAttackSuccessful` in listing 1.2).[10] We denote as $\mathcal{F}_S$ (variable `successful` in listing 1.2) the set of faults leading to successful attacks, and $\mathcal{F}$ (variable `attacks` in listing 1.2) the set of all performed attacks. $\mathcal{F}_S$ is used as an input to compute the vulnerability rate, according to the probability attached to each kind of faults in the fault model.

```
def celtic(program, faultModel, isAttackSuccessful):
    attacks = set()
    successful = set()
    Xref = simulateWithoutFault(program)
    for f in findAllFaults(Xref, faultModel):
            attacks.add(f)
                Xfault = simulateWithFaults(program, f)
                if isAttackSuccessful(Xfault):
                        successful.add(f)
    return successful, attacks
```

Listing 1.2. Pseudo-algorithm of CELTIC

[10] For instance, in a PIN verification one can check that the authentication token is `true` even though the provided PIN is wrong.

4.2 Vulnerability Rate

At its core, the vulnerability rate of the product describes how easy it is for the attacker to perform a successful attack. We propose the following definition:

Definition 7. *Vulnerability Rate $\mathcal{V}$: Let $\mathcal{P}$ be the space of attack parameters, $\mathcal{M}_{d,e}$ be a probabilistic fault model, and $\mathcal{F}_S$ be the set of successful attacks.*

$$\begin{aligned} \mathcal{V} &= Pr(\textit{Attack is successful}) \\ &= \sum_{p \in \mathcal{P}} Pr(\textit{Attack is successful} \mid p) \cdot Pr(p) \\ \mathcal{V} &= \underbrace{\sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} Pr(F = f \mid p)}_{\textit{Fault Model \& CELTIC}} \cdot \underbrace{Pr(p)}_{\textit{Attacker choice}} \end{aligned} \tag{2}$$

where $Pr(p)$ is the probability that the attacker chooses the parameters p for the perturbation attack.

Remark 1. Since we sum the space of attack parameters $\mathcal{P}$, we do not need to know the explicit attack parameters p that contribute to $\mathcal{V}$.

$Pr(p)$ depends on the attacker model. In the general case it can follow any law of probability, but we propose several typical models that suit the practice of the evaluators:

Equiprobable attacker. An attacker without knowledge does not favor any attack, i.e., $\forall p \in \mathcal{P}$, $Pr(p) = \frac{1}{|\mathcal{P}|}$:

$$\mathcal{V}_{equi} = \frac{\sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} Pr(F = f \mid p)}{\mid \mathcal{P} \mid} \tag{3}$$

Realistic Attacker. In practice, the attacker has some knowledge of the parameters to use, for instance through side-channel information on the attacked application, and will use this knowledge to apply the equiprobable model on a reduced space $\mathcal{P}'$ of attack parameter values, that still contains the attack parameters p that contribute to $\mathcal{V}$.

All-knowing attacker. An all-knowing, ideal attacker attacks only with the parameters p_{max} such that $\sum_{f \in \mathcal{F}_S} Pr(F = f \mid p_{max})$ is maximal, which is equivalent to a degenerated equiprobable attacker where $\mathcal{P}' = \{p_{max}\}$.

Remark 2. $\mathcal{V}_{max} = \sum_{f \in \mathcal{F}_S} Pr(F = f \mid p_{max}) \geqslant \mathcal{V}_{equi}$ for any considered $\mathcal{P}'$.

4.3 Sensibility of a single location

Automatic tools tend to output many vulnerabilities, some of which are less relevant than others. Therefore, existing tools [5] classically regroup successful attacks according to their location to find the most vulnerable locations. Vulnerability rate can be restricted by location for this purpose:

Definition 8. *Sensibility $\mathcal{S}_\ell$ of the location ℓ: The vulnerability rate restricted to the successful sequences of faults that involve ℓ, denoted as $\mathcal{F}_S^\ell$:*

$$\mathcal{S}_\ell = \sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S^\ell} Pr(F = f \mid p) \cdot Pr(p) \tag{4}$$

5 Experimentation and Conclusion

5.1 Experimental Comparison with the Traditional Approach

Rating comparison. We compared the vulnerability rating $\mathcal{V}$ with the rating $\mathcal{T}$ obtained in the traditional approach, using the physical success rate φ as reference. Our goal is to show the benefits in accuracy of using a probabilistic fault model obtained from fault model inference. To do so, we conducted experimental penetration tests on the cards A and B on several implementations of classic commands of various robustness.

We calculated the empirical success rate φ as the ratio of the number of physical successful attacks to the total number of performed attacks. To determine $\mathcal{V}$ we used CELTIC with the inferred fault model of Table 1. We used the *realistic* attacker model with sets of attack parameters $\mathcal{P}'$ adapted to each command. We also computed $\mathcal{T} = \frac{|\mathcal{F}'_S|}{|\mathcal{F}'|}$, the "traditional" success rate offered by existing tools [4, 5], where $\mathcal{F}'_S$ and $\mathcal{F}'$ denote respectively the set of successful attacks and the set of all possible attacks found by CELTIC with an arbitrary exhaustive byte replacement fault model. We chose arbitrarily the fault model to reflect the practice of the "traditional" approach, and our choice was the exhaustive byte replacement because it model a situation with zero knowledge of the values that can be injected. Table 3 summarizes the results of the various ratings.

Table 3. Rating criteria of several implementations on various cards

Card	Command	$\mathcal{V}$	$\mathcal{T}$	φ	$\mid \mathcal{P}' \mid$
A	VerifyPIN	2.35×10^{-5}	3.2×10^{-2}	3.40×10^{-5}	5883
A	SecureVerifyPIN	2.08×10^{-6}	8.5×10^{-5}	0	5000
A	GetChallenge	2.01×10^{-5}	1.75×10^{-3}	2.94×10^{-5}	6800
A	SecureGetChallenge	7.1×10^{-7}	2.74×10^{-6}	0	3000
B	GetChallenge	1.1×10^{-3}	1.2×10^{-3}	1.4×10^{-3}	231
B	SecureGetChallenge	0	2.14×10^{-4}	0	833

For all less secure implementations, our vulnerability rate $\mathcal{V}$ has the same order of magnitude as φ.

For all implementations except GetChallenge on card B, $\mathcal{T}$ predicts a much higher probability of success than φ. The difference between $\mathcal{T}$ and φ is dependent on the inconsistencies between the arbitrarily chosen fault model and the one that can actually be achieved on card. In SecureGetChallenge on card B, both $\mathcal{V} = \varphi = 0$, whereas in other secure implementations, $\varphi = 0 \neq \mathcal{V}$. The difference comes from the approximations of $\mathcal{V}$, which in turn are the results of approximations in the fault model inference process. Another source of approximation is the choice of $\mathcal{P}$, in particular the chosen range of time values in perturbation attacks are supposed to cover exactly the execution time of the code, which is difficult to ensure in practice. Lastly, φ is also approximated, in the sense that the individual experiments may have been too short to expose the vulnerabilities.[11]

Prediction of Elapsed Time in Attack Potential. The *elapsed time* factor of the attack potential is classically obtained as the inverse of the multiplication of the number of attacks per second s—in our case, determined experimentally at 1.27 attack·s^{-1} with EM and at 3.30 attack·s^{-1} with laser—by the empirical success rate φ: $(s \times \varphi)^{-1}$. This process requires however that the evaluator performs the physical perturbation attacks on the application. It would be interesting to predict the *elapsed time* factor using $\mathcal{V}$. We calculated $(s \times \mathcal{V})^{-1}$ and $(s \times \mathcal{T})^{-1}$ using the ratings obtained from our previous experiments. The results are summarized in Table 4, along with the score that the attack would receive according to the *elapsed time* factor (ET) in attack potential. From the results we can see that the score for $\mathcal{T}$ does not match φ, being lower in almost all cases. On the other hand, $\mathcal{V}$ has scores similar to φ, and therefore gives a good approximation of the *elapsed time* factor.

Table 4. Comparison of (expected) exploitability times

Card	Command	$(s \times \mathcal{V})^{-1}$	$\rightarrow$ ET	$(s \times \mathcal{T})^{-1}$	$\rightarrow$ ET	$(s \times \varphi)^{-1}$	$\rightarrow$ ET
A	VerifyPIN	8 hours	3	24 seconds	0	6 hours	3
A	SecureVerifyPIN	1 week	4	2.5 hours	3	> *3 days*[11]	$\geqslant 4$
A	GetChallenge	10 hours	3	7 min	0	7.4 hours	3
A	SecureGetChallenge	2 weeks	6	3.5 days	4	> *3 days*[11]	$\geqslant 4$
B	GetChallenge	5 min	0	5 min	0	5 min	0
B	SecureGetChallenge	unpractical	*	20 min	3	> *3 days*[11]	$\geqslant 4$

Sensibility of the locations. Figure 6 compares the sensibilities of the locations $\ell = (EEPROM, address)$with $address \in [0x100030, 0x10006c]$, $\mathcal{S}_\ell$ (de-

[11] Each experiment lasted for no more than 3 days.

fined in section 4.3) with the classic criteria $\mathcal{T}_\ell = \frac{|\mathcal{F}_S^\ell|}{|\mathcal{F}^\ell|}$ (normalized to be comparable with $\mathcal{S}_\ell$) on VerifyPIN on card A (EM injection). Our goal is to demonstrate the importance of the weight given to each fault according to a probabilistic fault model on the detection of "sensible" spots in the code. We observe a false positive at address 0x100042, where $\mathcal{T}_\ell$ is high while $\mathcal{S}_\ell = 0$. Indeed, faults at address 0x100042 are faults where the original value is $a = 0$, which we excluded in our probabilistic fault model for card A (see Table 1). Conversely, we observe two "false negatives", i.e, $\mathcal{T}_\ell \ll \mathcal{S}_\ell$ at addresses 0x100064 and 0x100066. Faults at this location have a modified value $b = 0$, which has a high probability of occurrence per the probabilistic fault model for card A. An arbitrary fault model may result in missed vulnerable spots of the code, and conversely spots may be falsely reported as vulnerable. In terms of code analysis, it may lead to vulnerabilities inexplicably difficult to patch, or to the introduction of unneeded countermeasures.

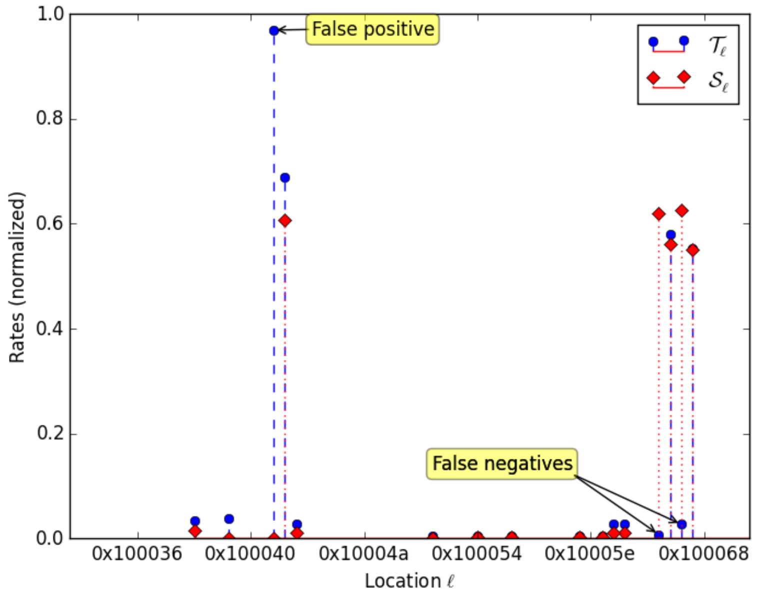

Fig. 6. Comparison of normalized $\mathcal{T}_\ell$ and $\mathcal{S}_\ell$

5.2 Discussion

Our approach is an end-to-end process that aims at qualifying the robustness of a smartcard product. This goal is achieved by splitting the assessment of robustness between the probabilistic fault model, that handles the faults at the device level, and the fault injection simulator that handles the consequences of the faults at the application level. The approach shifts the responsibility of performing effective perturbation attacks from the application level to the device level. For evaluators, this means an initial investment to perform the fault model inference process, which becomes worthwhile when evaluating several applications using the same device. For developers, it allows to compare the robustness of several implementations without resorting to physical tests (assuming the fault models are available).

The approach provides the vulnerability rate $\mathcal{V}$, which allows to predict the classic exploitability time factor of *attack potential* [9]. The attacker model $Pr(p)$ can be fine-tuned according to the *Expertise* and the *Knowledge of the TOE* of the attacker. The fault model $\mathcal{M}_{d,e}$ takes into account the equipment of the attacker. The relation between the *attack potential* and our approach is summarized in Table 5. We also defined a metrics $\mathcal{S}_\ell$ to compute the sensibility of individual attack paths and better assess the relevance of the vulnerable code spots detected by our fault injection simulator.

Table 5. Factors of Attack Potential given by our approach

Factor	$\mathcal{V}$
Elapsed Time	✓
Expertise	Partial ($Pr(p)$)
Knowledge of the TOE	Partial ($Pr(p)$)
Access to the TOE	✗
Equipment	Partial ($\mathcal{M}_{d,e}$)
Open Samples	✗

Our approach introduces some approximations that influence the proposed vulnerability rate. Similarly to other approaches, the parameter discovery step relies on the choices of the evaluator (some interesting parameter values might be accidentally ignored). Specific to our approach is the third step of manual generalization, which can also lead to approximations in the resulting model. Since our approach also models the attacker, the choice of an unrealistic model of $Pr(p)$ also leads to approximations in $\mathcal{V}$. However, it still constitutes an improvement over the complete lack of model.

Furthermore, while our fault detection program works flawlessly with faults on the data, it cannot capture the extra mechanisms in use in microprocessors such as instruction caches or instruction pipelines, and therefore approximates faults on the code. Moreover, at the time of writing, no program is able to detect faults on registers with certainty (a sentinel is difficult to design when all instructions typically manipulates registers).

5.3 Perspectives

It would be interesting to compare our results with other fault observation means, such as fault model extraction [15]. For instance, adding their fault detection program to the fault model inference process would improve the confidence in the resulting fault model.

The inference process of section 3 considers the case where a single perturbation attack occurs during the execution of the command. How should we extend this process to the now common multiple fault attacks? It is not obvious that it is possible to reuse the fault models by assuming that perturbation attacks are independent events. For instance, some extra technical difficulties can arise when synchronizing several laser shots. Further tests are required to conclude on the impact of fault model reuse in multiple fault scenarios.

Moreover, the fault injection simulator of section 4.1 is not able to cope with the combinatory explosion associated with handling multiple fault attacks. Some tools propose heuristics to reduce the combinatorics by simulating the effects of the faults at a higher level [5, 13], but they lose the ability to compute a vulnerability rate in the process, and they suffer from side-effects due to the compiler [1] as they work at the source level. We are currently investigating several approaches to handle the combinatorics of mutliple fault simulation while retaining the rating properties.

References

1. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32:23:1–23:84, August 2010.
2. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In *Proceedings of the IEEE*, volume 94, pages 370–382. IEEE, 2006.
3. G. Barthe, F. Dupressoir, P.-A. Fouque, B. Grégoire, and J.-C. Zapalowicz. Synthesis of fault attacks on cryptographic implementations. In *CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1027, New York, NY, USA, 2014. ACM.
4. M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: Embedded fault injection simulator on smartcard. In *ESSoS*, volume 8364 of *Lecture Notes in Computer Science*, pages 222–229. Springer, 2014.
5. P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J. Lalande. High level model of control flow attacks for smart card functional security. In *2012 Seventh International Conference on Availability, Reliability and Security (ARES 2012)*, pages 224–229. IEEE, 2012.
6. J. Blömer, M. Otto, and J.-P. Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *CCS '03*, pages 311–320, New York, NY, USA, 2003. ACM.
7. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology—EUROCRYPT'97*, pages 37–51. Springer, 1997.

8. M. Christofi, B. Chetali, L. Goubin, and D. Vigilant. Formal verification of a CRT-RSA implementation against fault attacks. *Journal of Cryptographic Engineering*, 3(3):157–167, 2013.
9. JIL. Application of Attack Potential to Smartcards. Technical Report Version 2.9, Joint Interpretation Library, January 2013.
10. C. H. Kim and J-J. Quisquater. Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 215–228. Springer, 2007.
11. J-B. Machemie, C. Mazin, J-L. Lanet, and J. Cartigny. SmartCM a smart card fault injection simulator. In *IEEE International Workshop on Information Forensics and Security*. IEEE, 2011.
12. S. Ordas, L. Guillaume-Sage, K. Tobich, J-M. Dutertre, and Ph. Maurine. Evidence of a larger EM-induced fault model. In *CARDIS 2014, Smart Card Research and Advanced Application Conference*, volume 8968 of *Lecture Notes in Computer Science*. Springer, 2015.
13. M-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 213–222. IEEE, 2014.
14. P. Rauzy and S. Guilley. Countermeasures Against High-Order Fault-Injection Attacks on CRT-RSA. In *FDTC'14 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 68–82. IEEE, Sept 2014.
15. L. Rivière, Z. Najm, P. Rauzy, J-L. Danger, J. Bringer, and L. Sauvage. High precision fault injections on the instruction cache of ARMv7-M architectures. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 62–67. IEEE, 2015.
16. I. Verbauwhede, D. Karaklajic, and J. Schmidt. The Fault Attack Jungle - A Classification Model to Guide You. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 3–8. IEEE, 2011.