Leakage Assessment in Fault Attacks: A Deep Learning Perspective

Sayandeep Saha[1], Manaar Alam[1], Arnab Bag[1], Debdeep Mukhopadhyay[1] and Pallab Dasgupta[1]

Indian Institute of Technology, Kharagpur, India,
{sahasayandeep,alammanaar,arnabbag,debdeep,pallab}@cse.iitkgp.ernet.in

Abstract. Generic vulnerability assessment of cipher implementations against fault attacks (FA) is a research area which is still largely unexplored. The security assessment for FA becomes especially interesting in the presence of countermeasures, as countermeasure structures are not very well-formalized so far, and on several occasions, they fail to fulfil their sole purpose of preventing FAs. In this paper, we propose a general, simulation-based, statistical yes/no test to assess information leakage in the context of FAs. The fascinating feature of the proposed test is that it is oblivious to the structure of the countermeasure/cipher under test, and detects fault-induced leakage solely by observing the ciphertext distributions. Unlike a recently proposed approach [SKP+19a], which utilizes t-test and its higher-order variants for detecting leakage at different moments of ciphertext distributions, in this work we present a Deep Learning (DL) based leakage assessment method. Our DL-based method is not specific to moment-based leakages only and thus, can expose leakages in several cases where t-test based technique either fails or demands a prohibitively large number of ciphertexts. Experimental evaluation over a representative set of countermeasures establishes that the DL-based method mostly outperforms the t-test based leakage assessment in terms of the number of ciphertexts required. Further, we present a novel analysis technique to interpret the leakages from the DL models, which is highly desirable for a sound vulnerability assessment.
In another vertical of this work, we enhance the leakage assessment test methodology for recently proposed Statistical-Ineffective-Fault-Analysis (SIFA) and establish the efficacy by verifying different countermeasures including a publicly available hardware implementation of a SIFA countermeasure. In the third vertical, we enhance the test for verifying FA-assisted leakages from so-called "non-cryptographic" parts of an implementation. As concrete proof of this, we validate a well-accepted automotive security module called Secure Hardware Extension (SHE) for which the test figured out non-trivial vulnerabilities.

Keywords: Fault attack · Block cipher · Information leakage · Deep Learning

Introduction

Implementation-based attacks are one of the potent threats to modern cryptographic primitives and protocols mainly due to their practicality and diversity. Fault attacks (FA), a class of active implementation-based attacks [BS97, TMA11] have recently gained significant attention from both industry and academia. The core idea of such fault assisted cryptanalysis is to deliberately perturb the computation or control-flow of a system and gain some information about the secret through the faulty system responses. The fascinating feature of FAs is that malicious faults are easy to generate but challenging to prevent. Especially, classical fault tolerance techniques often fall prey against precisely placed and repeatable faults.

There exist several physical means of injecting faults with malicious intentions. Some of the prominent methods include clock-glitching [ADN+10] and under-powering [CML+11] (causes setup-time violation in the underlying circuit), electromagnetic (EM) glitch (causes change in the

transistor state) [DDRT12], laser-based fault injection [CML+11, ADM+10], and Rowhammer bug (DRAM vulnerability; for remote fault injection) [BM16, ZLZ+18]. While the nature and precision of the faults happening in a system usually vary with the injection mechanisms, it is just one aspect of a FA. The key extraction process also critically depends on the underlying algorithm and its implementation. The standard way of performing an FA is to analyze the algorithm along with a logical abstraction of the faults happening in a system known as a *fault model*. Classically, data corruption faults having uniformly random or some statistically biased distribution (even constant valued faults) are exploited in most of the FAs. However, in a more generic scenario, one cannot rule out faults in the control-flow and faults at instruction-level which have also been shown to be fatal for cryptographic implementations on several occasions. Being a practical class of attack, it is imperative that the fault model exploited should be practically realizable. Fortunately (or unfortunately) the enormous improvement of fault injection mechanisms in last few years [CML+11, ADM+10, DDRT12] now allows even strongest fault model assumptions to be reasonably realizable for different implementations. It is thus desirable that a good implementation should ensure security even against the strongest possible fault model.

In this paper, we mainly focus on FAs in the context of block ciphers, which are one of the most widely deployed cryptographic primitives to date. It is somewhat well-understood that existing block cipher constructs alone are not capable of throttling FAs, and suitable countermeasures are required. FA countermeasures are usually incorporated at the algorithm-level [GST12, GMJK15] or at a slightly lower level of abstraction such as in the assembly instructions [MHER14, PCM17] or hardware circuits [SRM19, SJR+19, BKHL19]. One common feature for most of these countermeasures is that they utilize some form of redundancy (time, hardware, or information redundancy) to detect/correct the presence of a fault in the computation. However, the detection/correction operation may or may not be explicit. The most common form of countermeasures, widely referred to as detection countermeasures deploy an explicit check operation to detect the faulty computation and then react by either muting or randomizing the output [GMJK15]. In contrast, infective countermeasures avoid the explicit check operation and introduce a randomized infection function which masks a faulty ciphertext with the aim of making it useless for the adversary [GST12]. In contrast to algorithm-level integration, instruction-level countermeasures implement redundancy at the granularity of assembly codes. One straightforward strategy is to incorporate redundant instructions in the code with the assumption that an attacker may not be able to bypass all of them at once causing an effective corruption [MHER14, PCM17].

Very recently, a new form of FAs called Statistical Ineffective Fault Analysis (SIFA) has come in prominence. SIFA typically exploits the fact that certain faults during an injection campaign may remain "ineffective" resulting in correct ciphertexts [DEK+18, DEG+18]. The so-called "ineffectivity" of faults are caused by multiple circuit-level features (such as *fault activation* and *fault propagation* through combinational nets), which eventually make the ineffective faults dependent upon the intermediate state values, and allows key recovery from them[1]. As a result, recent FA countermeasures have started taking special precautions even at the granularity of bits within hardware designs. Recent proposals suggest the use of bit-level error correction to circumvent SIFA attacks [SRM19, SJR+19].

Unfortunately, many of these existing FA countermeasures have been found insecure even against the fault models they were designed to protect against. For example, the infective countermeasure proposed in [GST12] was shown to be vulnerable to all state-of-the-art FAs [TBM14], within two years of its introduction. Instruction-level redundancies were found to be vulnerable against clock glitches causing multiple skips in [YGS+16], despite being formally proven to be secure at [MHER14] against single instruction-skips. Recently, all of them have been found vulnerable against ineffective faults. A key cause behind such design failures is that there exists no general mechanism for security assessment in the context of fault attacks, and manual analysis is the only

[1] Activation and propagation of faults are well-known concepts in the logic testing community. While a given wire within a circuit is targeted with a fault, the *fault activation* implies whether the given fault can actually alter the value in the wire or not. On the other hand, *fault propagation* defines the phenomenon of observing the fault at the output of the circuit.

existing solution to date. While block ciphers are thoroughly analyzed in open forums against different classes of attacks[1], countermeasures are often engineered in-house, considering several other aspects like resource/performance constraints and time to market. Consequently, in most of the cases, they are analyzed by the design team itself, or by security certification facilities as an end product which may leave critical loopholes unobserved. Considering economic factors like time to market, and the cost of manual testing, it is evident that countermeasure assessment methodologies must be formalized and should be automated as well. However, devising a generic test for evaluating against FA is challenging due to several reasons. Firstly, FAs are inherently algorithm-specific, and a large number of ciphers, as well as countermeasure design strategies, are there in existence. A good testing methodology thus should be sufficiently generic to the mathematical and implementation details of the design under test (DUT). Secondly, the evaluation technique must have a sound theoretical background.

Our Contributions:

In this paper, we introduce a Deep Learning (DL) assisted, simple and automated yes/no test methodology for assessing the security provided by an FA countermeasure called Deep Learning Fault Attack Leakage Assessment Test (abbreviated as DL-FALAT). In short, DL-FALAT tries to detect potential information leakage in ciphertext distributions of a block cipher under the influence of faults. The ciphertexts here are interpreted as the manifestation of leakage. The root of this approach lies in the theory of *non-interference* [CHM04]. Major strengths of DL-FALAT lie in its simplicity and the feature of not exploiting any nontrivial information regarding a countermeasure or cipher algorithm. From this perspective, DL-FALAT bears some similarity with the TVLA methodology for SCA [SM15]. However, one should note that the underlying philosophy of DL-FALAT is significantly different from that of TVLA given the leakage model and adversary observables in FA are very different from that of SCA[2]. Also, DL-FALAT enables a semi-black box test methodology in a pre-deployment stage by means of fault simulation. The semi-black box model of testing here allows the evaluator to inject faults at different points within the implementation code, and to change the keys in certain situations. However, any nontrivial detail regarding the implementation (such as the details of the level of redundancy, error-detection code, or the infection function) is not required. One should note that precise control of the faults is impractical at a device level, and straightforward leakage assessment from a final implementation may leave some vulnerable corner cases unexplored, just due to the fact that certain faults may not take place for the specific implementation and injection mechanism. It is not also practical to assume that a testing facility would have access to all existing injection setups, many of which demands special human expertise. We, therefore, suggest that leakage assessment should be performed in a simulated environment so that every possible fault type can be explored. Fault simulation-based leakage assessment is a natural choice in this case, as the leakage here happens through ciphertext and contains zero environmental and measurement noise [3].

Application of Deep Learning: The DL-FALAT substantially improves upon a recently proposed prior approach called ALAFA [SKP+19b]. Unlike ALAFA [SKP+19a], which applies Welch's t-test for leakage assessment, (and hence, is limited to leakages in the moments of a distribution) the DL component in our test methodology can freely combine as many as points as necessary for classification and hence becomes a natural choice for detecting multivariate leakages.

[1]Recently, significant progress has been made in automating the discovery of FAs on unprotected implementations [SMD18, HBZL19, KRH17]

[2]Unlike the leakage in SCA, the manifestation of leakage in FA is specific to the cipher and the countermeasure algorithms under consideration. More precisely, the leakage function in FA is complex and varies significantly between attack strategies, ciphers and countermeasure algorithms (unlike SCA leakage functions which are usually specified by Hamming weight/distance). For example, in a typical differential fault analysis attack, the leakage function is decided by the fault propagation path, which varies with the cipher, the fault location, and the countermeasure structure.

[3]Fault simulations can indeed be performed without complete knowledge of an algorithm. The evaluator can utilize the input specifications of the countermeasure algorithms (e.g., the existence of dummy rounds, the total number of rounds, etc.) without going into exact algorithmic details, and may target specific internal registers, instructions, or the control-flow.

Limitations of moment-based leakage detection have been pointed out by several researchers, mainly in the context of SCAs [Sta18, MRSS18, WMM19]. At this point, it is important to note that unlike masking schemes for SCA, FA countermeasures typically do not specify any security order in terms of the number of shares. The higher-order leakages caused in the context of FA is mostly accidental, and result from complex algorithmic manipulations, especially in the case of infective countermeasures. As a matter of fact, during a semi-black box analysis, where the mathematical structure of ciphertext outputs are invisible to the evaluator, it is extremely difficult to determine up to which statistical order the t-test is to be performed. The DL method becomes even more relevant in this context as it does not require any input regarding the order of the leakage from the user. *To the best of our knowledge this is the first application of DL in the context of FA*, and our experimental validation over a large class of representative countermeasure examples establishes that DL provides a significant improvement in terms of the number of ciphertexts required for leakage assessment. In fact, we observed pathological cases where t-test-based method becomes infeasible in terms of data complexity due to extremely high order of leakage, but DL detects leakage within a reasonable number of traces.

Straightforward application of DL for leakage assessment has several caveats in the context of FA. Intuitively, there exists information leakage, if a DL model is able to learn the difference between two distributions. This is posed as a binary classification problem. One major question is how to set a threshold which decides if a given implementation is leaky or not. While well-defined thresholds can be set for this decision in the case of t-test, there exists no standard methodology so far, in the context of DL. The most intuitive technique would be to use the validation accuracy. Another crucial question in this context is that what should be the minimum and maximum number of ciphertexts that one should use in order to detect the leakage accurately. While there is no specific answer for the upper bound, we show that for most of the test cases a reasonably small set of ciphertexts can detect leakage. More precisely, we adopt an incremental approach which begins with a sufficiently small dataset and gradually increases the size, by taking feedback from the test methodology. In order to reduce the chances of overfitting with a small dataset, we use *stratified K-fold cross validation*. The leakage is decided based on a one-sample t-test on the accuracy scores of the cross-validation experiment. Such an incremental testing methodology is crucial for deciding the minimum size of the dataset required and hence saves in terms of fault simulation time.

Leakage Interpretation: Interpretability of the leakage is another desired property for a leakage assessment test. Although DL-FALAT considers the entire ciphertext as a whole unlike the t-test (which performs a point-wise analysis), we show that the leakage points can still be detected efficiently by means of a sensitivity analysis. However, often it is found that the classification happens only based on certain leakage points instead of all the leakage points in the ciphertexts. One reason for such behaviour is the existence of leakages of different statistical orders at different parts of a trace in certain examples. However, it also partially depends on the value of the secret intermediate under process. In order to have a clear understanding of the leakage profile, it is important to expose all the leakage points. In order to enable complete leakage exposure, we propose an iterative approach where the most important features are eliminated at each iteration to give other features an opportunity for meaningfully taking part in decision making. Using this approach, we were able to meaningfully interpret all the leakage points on a trace for several complex countermeasure examples.

Evaluation for SIFA: In the second vertical of this work, we show how to enhance the leakage testing methodology for detecting the so-called Statistical Ineffective Fault Analysis (SIFA). The approach of ours can inherently detect the data dependency of the correct and faulted encryptions just by analyzing the ciphertexts, and thus the chances of SIFA can be meaningfully interpreted for different implementations. We show the efficacy of our approach by means of several examples and different kinds of SIFA faults. We also evaluate two recently proposed SIFA countermeasures [SJR+19, SRM19], one of which has a publicly available hardware implementation [SRM19].

Generalized Leakage Assessment: The third vertical of this work mainly concerns about the applicability of the leakage assessment test in broader contexts beyond hardened cipher implementations. We propose a *fixed-vs-random* variant of the basic test which can be utilized for testing so-called "non-cryptographic" components of a security module against FAs. Most of the time, the cryptographic primitives are associated with several other peripheral components such as the mask generation logic or the input delivery logic, which can also be targeted by an attacker leading to exploitable leakage. It is thus desirable to test the overall implementation of a security block as a whole, instead of testing the cipher as an isolated component. We show that leakages of the aforementioned kind, which often do not directly associate with the leakage of a key, can be successfully detected by the *fixed-vs-random* enhancement of the leakage assessment test.

Validation over Diverse Implementation Classes: The efficacy of DL-FALAT is established by evaluating a representative set of implementations containing detection, infection, instruction-level, and hardware-level countermeasures. Both data-flow and control-faults have been tested. One interesting exercise is to evaluate the countermeasures against instruction-level faults, for which, in this work, we developed an instruction-level fault simulator based on the GNU Debugger (GDB) software. We also outline a simple and customizable strategy for fault simulation in hardware designs, which has been utilized to evaluate one of our examples. In order to evaluate the holistic leakage assessment capability of DL-FALAT on “non-cipher” components, we test a recently proposed automotive security standard called Secure Hardware Extension (SHE). Interestingly, it was observed that certain implementation aspects of the SHE framework might lead to key leakage even from the "non-cryptographic" part of the implementation. DL-FALAT successfully detects the chances of such leakages, and eventually, we develop a key-recovery attack based on one such leakage instance establishing its exploitability.

Related Work:

It is worth mentioning that a prior approach for leakage assessment in the context of fault attacks was proposed in [SKP+19b] which used *t*-test for leakage assessment. DL-FALAT significantly enhances the power of the test, thanks to the power of DL, and we also present pathological cases where DL succeeds while *t*-test fails within reasonable data complexity. Furthermore, the work in [SKP+19b] does not shed light on SIFA attacks, and the general applicability of the test system-level vulnerability assessment tool beyond cipher implementations. Another recent contribution in this direction is due to [AWMN20], which presents a fault diagnosis approach for evaluating countermeasures against FAs. The approach in [AWMN20] is based on monitoring certain internal signals for detecting faults and not the ciphertexts only, and hence it expects significant knowledge from the side of the user. Moreover, vulnerabilities in countermeasures are typically not limited to their fault detection modules but also depends on the recovery modules as we practically show in the case of several infective and SIFA countermeasures. Hence, checking the leakage at the ciphertext seems to be a better idea as it represents the true exploit of a fault attack[1]. Furthermore, the approach in [AWMN20] is specific to hardware designs, while our approach holistically covers hardware and software designs.

Recent years have seen several applications of DL in the context of SCA attacks including leakage detection [YLMZ18, HGG18, Per19, Tim19, WMM19, KPH+19, PHJ+18, MDP20]. However, no work, to the best of our knowledge, has utilized the power of DL in the context of FAs. From this perspective, our work is the first application of DL in FA evaluation, which may open up several new directions for FA research in future.

The paper is organized as follows. In Sec. 1, we present the formalization of the leakage in FA and its connection to the theory of non-interference, which is followed by the basic descriptions of the leakage assessment tests in Sec. 2 with *t*-test. Sec. 3 introduces the DL-based leakage assessment methodology in detail, along with an approach for leakage interpretation. In both

[1]Note that in this work we do not consider combined SCA and FA. However, given the testing methodology in both cases bears certain similarities, we strongly believe that our approach can be scaled for such combined attacks.

of these sections, we use running examples of state-of-the-art countermeasures to explain our claims. Sec 4 outlines several enhancements of the basic leakage assessment test including a test methodology for SIFA countermeasures, and the *fixed-vs-random* generalization of the test for evaluating leakages from"non-cipher" components. Sec. 5 briefly describes the ways of simulating faults on low-level software and hardware implementations (details of which can be found in the Appendix A and B, respectively). Case studies on several representative FA countermeasures including the SHE hardware security module (HSM) are presented in Sec. 6. We specifically stress on that fact that both algorithm-level, as well as implementation-specific countermeasures, are included in our testbench consisting a total of 21 different case-studies (including an open-source hardware design). Sec. 7 presents a discussion from the perspective of applicability and further generalizations. We conclude in Sec. 8.

1 Fault Attacks and Information Leakage

State-of-the-art FA countermeasures largely vary in their internal computation (detection, correction, or infection based) or by their way of implementation (algorithm-level, instruction-level or hardware-level). However, irrespective of these intrinsic details, they are deemed to be vulnerable if they somehow start causing information leakage under the influence of faults. FAs are widely regarded as a special case of SCAs. In [SLIO11], an information-theoretic definition for FA-induced leakage was provided, mainly for unprotected implementations. However, it is difficult to build an explicit correspondence between the leakage definitions in [SLIO11] to the SCA leakage, which is important in the context of this work. The information leakage in SCA is defined as: $\mathscr{L}_{SCA} = \delta + L(X) + \alpha$, where X denotes an intermediate key-dependent variable, δ denotes a constant offset, $L(\cdot)$ stands for Hamming weight/distance leakage model, and α denotes a normally distributed random variable representing the noise components. $\mathscr{L}_{SCA}$ typically represents the power/electromagnetic signals which form the set of observables for the attacker.

In contrast, the leakage in fault attacks is manifested as ciphertexts (or differentials of correct and faulty ciphertexts)[1]. Formally, it can be described as:

$$\mathscr{L}_{FA} = C = \mathscr{F}(f, \mathscr{P}, \mathscr{K}) \tag{1}$$

with f denoting the value of the intermediate state differential at the point of fault injection (also denoted as the *value of the fault mask* or simply *fault value*), $\mathscr{P}$ denoting the plaintext variable and $\mathscr{K}$ denoting the secret key variable. The parameter f takes value according to some fault model F (e.g., byte fault model, nibble fault model, etc.). Furthermore, the function $\mathscr{F}$ represents the fault propagation path through the cipher computation. The observable for the adversary in FAs is the *ciphertext under the influence of faults* (C), (resp. the differential between the correct and the faulty ciphertext denoted as ΔC)[2].

According to Eq. (1), the leakage in FA depends upon three quantities, one of which (the plaintext $\mathscr{P}$) can be controlled by the adversary. For a properly protected cipher, the key is supposed to remain secret, and no information regarding this should be leaked. Further, we claim that the fault value f should also be treated as a secret for protected ciphers. In other words, the fault value should not visibly influence the observable ciphertexts (resp. ciphertext differentials), and the secrecy of f is equivalent to the secrecy of the key and vice versa. This claim requires a proof, and we present a sketch of it in the following theorem:

Theorem 1. *Leaking the exact value of the fault has the same effect as leaking the secret key.*

Proof. Let us consider the information-theoretic quantification of fault attack leakage from [SLIO11]. The leakage can be expressed as $lk \approx \mathbb{H}(\mathscr{K}) - \mathbb{H}(f)$, where $\mathbb{H}(\cdot)$ denotes the Shannon entropy.

[1]The leakage assessment test we are going to present mostly analyzes the ciphertexts, except in some special cases (such as for detecting SIFA), where the ciphertext differentials are analyzed.

[2]The observable definitions in fault attacks may go beyond the ciphertexts or ciphertext differentials. Later in this paper, we shall use a more general definition of the observables (Sec. 4.3).

With the knowledge of the exact fault value $\mathbb{H}(f) = 0$. Consequently, the leakage becomes $lk \approx \mathbb{H}(\mathscr{K})$ that is the key is revealed [SLIO11]. So it can be concluded that the leakage of the exact fault value implies the leaking of the secret key. [1] □

In the case of unprotected implementations, both the key and f leak via the faulty ciphertexts (or ciphertext differentials). *Therefore, the only way of preventing FA is to prevent the information flow from both $\mathscr{K}$ and f to the ciphertexts obtained during a fault injection event*. In practice, all the existing fault attack countermeasures try to achieve this. Accordingly, a countermeasure is considered to be secure if and only if it satisfies the following equation:

$$\mathscr{I}(C, \mathscr{K} | \mathscr{P}) = 0 \tag{2}$$

where $\mathscr{I}(X, Y|Z)$ denote the conditional mutual information between two random variables X and Y given Z. Equivalently, we can also write:

$$\mathscr{I}(C, f | \mathscr{P}) = 0 \tag{3}$$

It is quite evident that these two definitions can be used interchangeably for leakage assessment in the context of FAs. Both of these definitions include mutual information. In this context, one should note that our goal is to evaluate the hardened implementations without utilizing any algorithmic details of them. Lack of algorithmic details refrain the analytical estimation of MI and leaves data-based statistical estimation as the only option. However, the statistical measurement of MI is challenging in general. As we show in the next subsection, this technical difficulty can be circumvented by an alternative interpretation of the leakage with the theory of *non-interference*.

1.1 Non-Interference

Non-interference is a well-explored security paradigm having wide applicability in the domain of program analysis. In simple words, *the non-interference property guarantees the absence of sensitive information flow from the input to any observable point of a system.* In [BBD+15], the notion of non-interference has been utilized to give an alternative formulation for the SCA leakage. In the context of FA-induced leakage, non-interference between the key or the fault value with the ciphertext or ciphertext differential (or the observable, in general) implies that the attacker cannot exploit the ciphertext to extract the secret.

Due to its vast applicability in the domain of program analysis, non-interference is often defined in terms of program variables. This abstraction fits well in the present context, as any hardened cipher implementation can be formally treated as a program. From now onward we shall treat a protected cipher algorithm as a *probabilistic program*[2]. Formally, a *probabilistic program PP can be described as a routine, which contains both probabilistic and deterministic assignments, when represented in Single-Static-Assignment (SSA) form.* The input, output, and intermediate program variables in *PP* can be either deterministic or probabilistic. Further, the program variables may have different security levels. In particular, some of the variables are secret (marked as 'high'), and the rest of them are public ('low'). All the program variables can be treated as random variables. In the most general case, *PP* takes a joint distribution of input variables and outputs a joint distribution of output variables. The non-interference definition now can be stated as follows:

Definition 1. Let H, L and L' denote the random variables associated with the secret (high) inputs, public (low) inputs and public outputs, respectively. Then a probabilistic program *PP* is said to be *non-interfering* if and only if $\mathscr{I}(L', H|L) = 0$.

[1] One should note that the formulation provided at [SLIO11] claims that at least one bit of ambiguity in the key remains which arises mainly due to that fact that the number of solutions in the associated difference equations is more than one in some cases (specifically for attacks like Differential Fault Analysis (DFA)). However, even in the worst case, the number of ambiguous bits remains as a small constant. In other words, the approximation factor in the above leakage equations are small constants, and one can trivially assume that almost the entire key leaks if the fault value gets exposed.

[2] A probabilistic program is the most natural model in this context as fault attack countermeasures often utilize random numbers.

One should note that the definition above is equivalent to that of Equation (2) and (3). However, this definition too expects the quantification of mutual information which has already been pointed out as difficult. Fortunately, there exists a more intuitive and easy-to-compute equivalent definition of non-interference based on equivalence relations. The intuition comes from the fact that *if the low outputs differ in two independent runs of a program having the same low inputs but different high inputs, then the program leaks about its high inputs.* The high inputs which cause indistinguishable low outputs can be treated as the member of the same equivalence class of high inputs (as they cannot be distinguished by an adversary). In the context of probabilistic programs, this alternative definition of non-interference is stated as follows:

Definition 2. A probabilistic program *PP* is said to be non-interfering if and only if the marginal distribution of the program variables with respect to the "low" output variables are equal, for all inputs which differ in their "high" variables but are equivalent in terms of their "low" variables.

The above definition is said to be violated if there exist two high inputs h and h' for which the low equivalence in the input does not imply low equivalence in outputs. Further, it can be shown that the two definitions provided for non-interference are actually equivalent. However, detailed proof of this is out of the scope of this paper, and a proof sketch can be found in [CHM04]. In the rest of the paper, we utilize the second definition to assess the security of the protected implementations. Next, we describe a simple test to determine whether there is any interference between the secret variables and the ciphertexts for a protected cipher during a fault injection event.

Algorithm 1 *TEST-INTERFERENCE-FAULT*

Input: Protected Cipher $\mathscr{C}$, Fault value f_1, f_2, Simulation counter S
Output: Yes/No
1: $\mathscr{T}_{f_1} := \emptyset$; $\mathscr{T}_{f_2} := \emptyset$
2: $p := GEN_{PT}()$
3: $k := GEN_{KEY}()$
4: **for** $i \leq S$ **do**
5: $\quad \mathscr{T}_{f_1} := \mathscr{T}_{f_1} \cup \mathscr{C}(p,k,f_1)$
6: $\quad \mathscr{T}_{f_2} := \mathscr{T}_{f_2} \cup \mathscr{C}(p,k,f_2)$
7: **end for**
8: **if** $(TEST(\mathscr{T}_{f_1}, \mathscr{T}_{f_2}))$ **then**
9: $\quad$ **Return** Yes
10: **else**
11: $\quad$ **Return** No
12: **end if**

Algorithm 2 *TEST-INTERFERENCE-KEY*

Input: Protected Cipher $\mathscr{C}$, Fault value f, Key k_1, k_2, Simulation counter S
Output: Yes/No
1: $\mathscr{T}_{k_1} := \emptyset$; $\mathscr{T}_{k_2} := \emptyset$
2: $p := GEN_{PT}()$
3: **for** $i \leq S$ **do**
4: $\quad \mathscr{T}_{k_1} := \mathscr{T}_{k_1} \cup \mathscr{C}(p,k_1,f)$
5: $\quad \mathscr{T}_{k_2} := \mathscr{T}_{k_2} \cup \mathscr{C}(p,k_2,f)$
6: **end for**
7: **if** $(TEST(\mathscr{T}_{k_1}, \mathscr{T}_{k_2}))$ **then**
8: $\quad$ **Return** Yes
9: **else**
10: $\quad$ **Return** No
11: **end if**

2 Leakage Assessment Test

In this section, we present the basic forms of our leakage assessment test for evaluating protected block ciphers. Although throughout this paper, we focus on block ciphers, the concept of FA-assisted leakage is fundamental and is equally applicable for public-key implementations. The main target of the tests we propose is to figure out the existence of interference (if any) between the key and the observables so that an erroneous design can be quickly ruled out.

As already pointed out in Sec. 1, both the fault and the key are the secrets ('high' inputs) in the present scenario. On the other hand, the plaintext is treated as the low input. For the sake of simplicity, we keep one of the secret inputs fixed during our testing. The choice of the "fixed" secret is somewhat driven by the type of application being tested, and also by the nature of the fault considered. Keeping the key fixed is found to be the most convenient option for the cases where fault values vary within some finite range (for example, in the case of random byte faults the range is $\{1,2,\cdots,255\}$). This is due to the fact that the size of the keyspace is much larger than the size of the fault space, and this size would matter for some of the situations as we shall elaborate later. However, varying the key is a viable option too, and convenient in situations like control-flow faults, stuck-at faults, or instruction-skip faults, where the fault can typically take a single value

(e.g., a control fault may change the execution flow of a program by altering a decision from "yes" to "no". The faulty value here is "no", and no other valuation is possible at a specific instant). For completeness, here we shall outline both of the models and later use them as per requirements.

The interference test with fixed key and varying fault value is presented in Algorithm. 1. The algorithm takes a protected cipher $\mathscr{C}$, and two fault values f_1 and f_2 as inputs. It outputs "Yes" if there exists interference and "No", otherwise. Internally, Algorithm. 1 runs two independent simulations of $\mathscr{C}$ for f_1 and f_2 with fixed plaintext p and key k. One should note that $\mathscr{C}$ is a probabilistic algorithm which may internally generate random numbers to randomize the outcome in each run. The *simulation traces* (the ciphertexts), denoted as $\mathscr{T}_{f_1}$ and $\mathscr{T}_{f_2}$ are then subjected to a statistical test $TEST$. The $TEST$ reasons about the equality of the distributions of the two simulation traces and returns $TRUE$ if the distributions are unequal. If the distributions are unequal the Algorithm. 1 returns YES indicating interference.

The second interference test can be realized in a very similar manner by utilizing the leakage definition corresponding to Eq. (2) instead of Eq. (3). In other words, we keep the fault value fixed and take two different instances of the key (say k_1 and k_2) for testing the probable interference. The resulting distributions are to be compared with $TEST$. The alternative algorithm for interference testing is presented in Algorithm. 2.

One crucial factor, in this case, is the fixed values of the faults we exploit for the testing. Although it may seem a little artificial at the beginning, it is an entirely practical choice in our simulation-based setting for validating implementations at early design stages. From a theoretical perspective, this is the strongest form of fault model possible from an attacker's point of view. Ideally, a countermeasure should be able to throttle attacks against even the strongest possible adversary and hence testing against the strongest fault assumptions seems to be a natural choice.

2.1 *t*-test for the Detection of Leakage

To realize the $TEST$ function, the proposal in [SKP+19a] utilized Welch's t-test, which is one of the most common approaches for understanding if two sets of data are significantly different from each other. A t-test gives a probability to examine the validity of the null hypothesis as the samples in both sets were drawn from the same population, i.e., the two sets are not distinguishable. Mathematically, if two samples M_0 and M_1 are under test, and M_0 (resp. M_1) have a sample mean μ_0 (resp. μ_1) and sample variance s_0^2 (resp. s_1^2), then the t-test statistic t and degrees of freedom v is calculated as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \quad \text{and} \quad v = \frac{\left(\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}\right)^2}{\frac{\left(\frac{s_0^2}{n_0}\right)^2}{n_0 - 1} + \frac{\left(\frac{s_0^2}{n_0}\right)^2}{n_1 - 1}} \tag{4}$$

where $n_0 = |M_0|$ and $n_1 = |M_1|$. Large absolute values of the t-test statistic (denoted as t) indicate that the data-sets are statistically different. Usually, a threshold of $|t| > 4.5$ indicates that the confidence of the test is > 0.99999. So, 4.5 is used as a threshold to determine whether two data-sets are equal or not.

2.2 *t*-test in a Multivariate Setting

Referring to Algorithm. 1 (resp. Algorithm. 2), the two sets $\mathscr{T}_{f_1}$ and $\mathscr{T}_{f_2}$ (resp. $\mathscr{T}_{k_1}$ and $\mathscr{T}_{k_2}$) contain ciphertexts under the influence of faults. In most of the modern block ciphers, ciphertexts are of 64, or 128 bits, and treating them as a single entity during the t-test is impractical[1]. One reasonable solution is to treat them as multivariate quantities. Each bit, nibble or byte of a ciphertext can be treated as a variable. The proposal is to consider both bit and byte (or nibble if required)-level divisions separately. In the simplest case, the t-test can be applied to each variable of the observable in a univariate manner. However, information leakage may not always be manifested in this univariate setting. To see this, let us consider two variables V_1 and V_2 such that $V_1 = X \oplus r$

[1]Treating them as a single monolithic variable necessitates dealing with distributions having a support of 2^{128}.

and $V_2 = r$. Here X is a leakage component depending on the key and the fault value, and r is a random variable. In a univariate setting, if we run the t-test on two different instances of V_1 caused by two different fault values (to be precise, $X = X_{f_1}$ in the first distribution and $X = X_{f_2}$ in the second one), the t-test concludes that these two distributions are equal. This is due to the presence of the random mask r. However, if one considers the joint distribution of V_1 and V_2, the leakage should be visible as the effect of the mask r gets nullified. However, to capture this leakage, the t-test must be performed in a multivariate setting [1].

One approach for extending t-test to the multivariate setting is to consider the *centered product* of different variables from the observable. The centered product, which has already been utilized for higher order leakage assessment in TVLA testing can be defined as follows:

Definition 3. Given a set of observables $M = \{V^{i\in\{1,2,\cdots n\}}\}$ (with each $V^i = \langle V^i_{(1)}, V^i_{(2)}, \cdots, V^i_{(l)}\rangle$), and a subset of indices $\mathscr{J} = \{j_1, j_2, \ldots, j_d\}$ (with $d \leq l$), a centered product of the i-th ciphertext can be defined as $\prod_{j\in\mathscr{J}} \left(V^i_{(j)} - \mu^M_{(j)}\right)$. Here $\mu^M_{(j)}$ denote the mean over the set M at the variable location $j \in \mathscr{J}$.

A ciphertext with l variables converts to a tuple of $\binom{l}{d}$ elements for a d-th order testing. The t-test can then be applied in a univariate manner over these tuples.

The $TEST$ function proceeds as follows. First, it performs a univariate test (bit/byte level) over two given set of observables (from now onward, we shall use the words *traces* and observables, interchangeably). Next, it goes for d-th order testing for $d = 1, 2, \ldots, G$, where G is a parameter specified by the user. The parameter G decides the test complexity as the simulation time (S in Algorithm. 1) also increases for higher values of G. Higher values of G ensure stronger security guarantees from the test. However, the test terminates once leakage is observed for a d value.

3 DL-FALAT: Deep Learning based Leakage Assessment

Simply speaking, the main idea of the FA leakage assessment test is to statistically distinguish two distributions resulting from two different valuations of the secret. The effectiveness of this test, however, strongly depends upon the function $TEST$ mentioned in Algorithm. 1 and 2. Application of t-test and its higher-order variants (as proposed in [SKP+19a]) indeed works, but only with some critical theoretical and practical shortcomings. First of all, t-test is inherently univariate. The higher-order extension described previously only considers different statistical moments. In the context of SCA it has been shown in several occasions that such moment-based approach may even lead to false-negatives [Sta18, MRSS18] in certain situations, especially while leakages are higher-order and multivariate in nature. In contrast, DL methods are renowned for learning in highly multivariate scenarios and can take into consideration several complex interrelations among different features. This, by itself, is a clear motivation for adopting DL in the present context. Nevertheless, the motivation behind using DL becomes even stronger for FA leakage assessment while we consider the fact that the true order of leakage is completely unknown for FA countermeasures. One should note that typical FA countermeasures are not designed keeping the order of leakage in mind, as it is done for SCA countermeasures, such as masking. Whatever leakage is caused, is either due to design flaw or is determined by the fault model. As a result, an evaluator will have no clue about up to which order the test has to be performed to ensure sound security. As we shall show later, we have identified three infective countermeasures showing leakages in third-order, second-order and 16-th order (or 128-th order while considering bit-level leakage) leakages, respectively. Clearly, it is difficult to know the correct testing order apriori (the value of G from the last section). The DL, in this context, becomes the appropriate tool as it does not require any order-related information to be given from the evaluator side. Furthermore, it is found to perform significantly better in noisy scenarios, and for very high leakage orders compared to the t-test-based approach.

[1] The arguments are the same for Algorithm. 2

3.1 The Main Idea

The main idea behind DL-FALAT is to train a Neural Network (NN) with two different sets of ciphertexts resulting from computations based on two different secret valuations. Afterwards, the classification capability of the trained model is evaluated on a validation set. The accuracy result obtained over the validation set signifies the amount of information learnt by the network. A *better-than-random* guess over the validation set indicates leakage happening from the countermeasure.

Although the approach stated above is fairly simple, it poses several caveats and challenges during implementation. The first and foremost challenge is how to arrive at a sufficiently confident and meaningful decision in a non-parametric heuristic-based approach, such as DL. It is well-known that DL methods are highly data-dependent, and there is no clear thumb-rule for determining the proper amount of data required for training. *One advantage specifically in the context of leakage detection is that the learning need not require to be the "best". Instead, even a small sign of learning will indicate leakage.* This observation is tempting as it allows some flexibility in terms of choosing the model as well as the minimum amount of data required.

Based on the above observation, we present an iterative flow for DL-based leakage assessment. For simplicity and genericness, we assume having a fixed model $\mathscr{M}$, which we use for all of our test cases. The description of the $\mathscr{M}$ is deferred till Sec. 3.3. Before describing the model, we elaborate on the training and validation flow.

3.2 Iterative Training and Validation

We begin the training with reasonably small training and validation sets and gradually increase their size by taking feedback from the leakage detection step. The dataset under consideration is denoted as $\mathscr{D} = \mathscr{T}_{f_1} \cup \mathscr{T}_{f_2}$ (resp. $\mathscr{T}_{k_1} \cup \mathscr{T}_{k_2}$). The instances from the set $\mathscr{T}_{f_1}$ (resp. $\mathscr{T}_{k_1}$) are labeled as 0, and the instances from the set $\mathscr{T}_{f_2}$ (resp. $\mathscr{T}_{k_2}$) are labeled as 1. In our iterative testing, the size of the set $\mathscr{D}$ is increased adaptively in each iteration by adding equal number of samples from both of its constituent sets. To represent the varying size of $\mathscr{D}$, from now onward, we use the notation $\mathscr{D}_t$ denoting the dataset at tth iteration. The entire set $\mathscr{D}_t$ is divided into training and validation sets Tr_t and Vl_t, respectively. For training and validation to be robust even over small datasets, we adopt the *stratified K-fold cross-validation* approach, which is well-known for preventing overfitting.

The K-fold cross-validation can be explained as follows. The entire dataset $\mathscr{D}_t$ is randomly partitioned into K equal-sized subsets $\mathscr{D}_t^1, \mathscr{D}_t^2, \cdots, \mathscr{D}_t^K$ ($|\mathscr{D}_t^j| = \frac{|\mathscr{D}_t|}{K}$, $\forall j$). The *stratified* feature ensures that for each $\mathscr{D}_t^j$, $Pr[label(x) = 1 | x \in \mathscr{D}_t^j)] = Pr[label(x) = 0 | x \in \mathscr{D}_t^j)] = 0.5$. Next, $K-1$ of these subsets are used for training the model $\mathscr{M}$, and one subset is used as validation set. Furthermore, this process is repeated K times within one iteration t giving each subset one chance to be used as validation set. The main idea is to check if the model $\mathscr{M}$ is capable of generalizing its knowledge for different validation sets or not.

In our testing methodology, we accumulate the testing accuracy (as fraction of correctly classified examples) for all the K validation sets in a specific iteration t (the corresponding set is denoted as $A_t = \langle a_t^1, a_t^2, \cdots a_t^K \rangle$, where each a_t^j denote the test accuracy while validating on $\mathscr{D}_t^j$)). In order to check the leakage, we next perform the following hypothesis test.

$$\begin{aligned} &\mathscr{H}_0 : \mu_{A_t} = 0.5, \text{ and} \\ &\mathscr{H}_1 : \mu_{A_t} > 0.5. \end{aligned} \tag{5}$$

Here μ_{A_t} denote the mean over the set A_t. In the case of leakage, the alternative hypothesis $\mathscr{H}_1$ is accepted. We apply one-sided t-test with significance level $\alpha = 0.0001$, and degrees of freedom $K-1$. The value of K is set as 50 in all of our tests to ensure a sufficiently confident result.

The setting of the constant mean value (i.e. the RHS in Eq. (5)) as 0.5 in the above-mentioned t-test is quite intuitive. Acceptance of the alternative hypothesis indicates that the average validation accuracy is better than random guess in most of the cases, which indicates some sort of learning to be happening. Next, we outline how the size of the dataset is adaptively increased in our test methodology.

Algorithm 3 *DL-TEST-INTERFERENCE-FAULT*

Input: Protected cipher $\mathscr{C}$, Fault value f_1, f_2,
Simulation counter S, Initial simulation counter S_{init}, Model $\mathscr{M}$
Output: Yes/No

$\mathscr{T}_{f_1} := \emptyset$; $\mathscr{T}_{f_2} := \emptyset$
$p := GEN_{PT}()$
$k := GEN_{KEY}()$
$S_t := S_{init}$
$leak := Null$
while $S_t \leq S$ **do**
  for $i \leq S_{init}/2$ **do**
    $\mathscr{T}_{f_1} := \mathscr{T}_{f_1} \cup \langle \mathscr{C}(p,k,f_1), 0 \rangle$ ▷ Add labels to the data as 0 or 1
    $\mathscr{T}_{f_2} := \mathscr{T}_{f_2} \cup \langle \mathscr{C}(p,k,f_2), 1 \rangle$
  end for
  $\mathscr{D}_t := \mathscr{T}_{f_1} \cup \mathscr{T}_{f_2}$
  $\langle \mathscr{D}_t^1, \mathscr{D}_t^2, \cdots, \mathscr{D}_t^K \rangle :=$ `GEN-CROSS-VALID-SET`$(\mathscr{D}_t)$ ▷ Generate K subsets for cross validation
  $A_t := \emptyset$
  for $i \leq K$ **do**
    $Tr_t^i := \bigcup_{\substack{j=0 \\ j \neq i}}^{K} \mathscr{D}_t^j$
    $Vl_t^i := \mathscr{D}_t^i$
    $a_t^i :=$ `Train-and-Validate`$(\mathscr{M}, Tr_t^i, Vl_t^i)$ ▷ Get the validation accuracy
    $A_t := A_t \cup \{a_t^i\}$
  end for
  if (`t_Test`$(A_t) \implies \mathscr{H}_0$) **then** ▷ Perform one-tailed t-test
    $leak = False$
  else
    $leak = True$
  end if
  if $leak$ **then**
    Return Yes
  else
    if $S_t \leq S$ **then**
      $S_t = S_t + S_{init}$
    else
      break
    end if
  end if
end while
Return No

Algorithm 3 outlines our idea of adaptively increasing the dataset size. The training and testing begin with a reasonably small dataset of size S_{init}. In all of our experiments, S_{init} was set as 500 (by inspection). The maximum limit of simulation is set as S. A full K-fold cross-validation is performed at each iteration, and the associated hypothesis testing is performed. In case, some leakage is detected the test terminates. If no leakage is found, the dataset is increased further in the next iteration by adding new simulation data. If no leakage is observed even after S simulations, the test decides no leakage. The choice of S is to be decided by the user just like the t-tests. However, we found that for most of the leaky examples of ours, the number of traces required for leakage detection is significantly less than the pre-decided value of S. As a general suggestion, (based on our experience) one should set a high value of S (20000 to 50000) while testing infective countermeasures as most of them show higher-order leakages. A moderate value of S (5000 to 20000) is desired for testing SIFA and other biased fault attacks. While there is substantial noise in fault injections (e.g. for countermeasures having dummy rounds), we suggest a very high S value of 2000000 to 500000, and for the rest of the cases, one can go with a relatively low value. Note that these suggestions are purely based on what we observed experimentally, and we do not claim to have any theoretical proof of these values. While Algorithm 3 presents the leakage assessment strategy with respect to two different fault values, it is worth mentioning that an equivalent version exists for varying keys and a fixed fault value (similar to Algorithm 2). However, we do not repeat it here.

One of the greatest advantages of the proposed test is that it can save simulation time by deciding when to terminate. Here we would like to point out that even though the fault simulation process can be accelerated in several ways (such as parallelization), in a typical low-level tool-based simulation scenario such as instruction-skip or HDL simulation (detailed in the appendix), simulation demands the most significant computational effort. Moreover, one should keep in mind

that, simulating faults for a single location or model is not sufficient and in practice, one has to perform such simulations for several different locations[1]

The next subsection will present the DL models we used for leakage detection. A thorough discussion on leakage interpretation will be presented in the subsequent subsection.

3.3 The Deep Learning Models

Choice of the model $\mathscr{M}$ is crucial for every machine learning/deep learning problem. The quality of learning strongly depends on the chosen model. One of the major challenges of the leakage assessment problem is to select a generalizable model for this purpose, which should not depend upon the design under test, or the nature of the leakage. As already pointed out, one advantage that we have in leakage assessment is the learning need not be the best. Even a reasonable validation accuracy is acceptable. However, one must be careful about the overfitting of a model.

Keeping all above-mentioned factors in mind, we selected the two following model shown in Listing 1, and 2. It is observed that the manifestation of leakages in the ciphertext structures can be at bit-level or byte-level. The pattern usually depends upon the structure of the countermeasure under consideration. Hence, in order to analyze the leakage, we consider two DL model architectures - one for bit-level analysis (Listing 1), and the other for byte-level analysis (Listing 2). The models have been developed using a Python-based Keras library, which uses TensorFlow in the backend. Both the networks consist of one input layer, two fully connected (or `Dense`) hidden layers, and one output layer. The hidden layers in the bit-model contain 8 and 4 neurons, whereas the hidden layers in the byte-model contain 32 and 16 layers, respectively. In both models, the output layer contains two neurons. The hidden layers in both the models use *Rectified Linear Unit* (`ReLu`) activation function, whereas the output layers use `Softmax` activation function. A `BatchNormalization`[2] layer separates each of the hidden layers from each other.

Listing 1: Bit-Model

```
model = Sequential([
  Dense(8, input_dim=128,
      activation='relu')
  BatchNormalization()
  Dense(4, activation='relu')
  BatchNormalization()
  Dense(2, activation='softmax')])
```

Listing 2: Byte-Model

```
model = Sequential([
  Dense(32, input_dim=16,
      activation='relu')
  BatchNormalization()
  Dense(16, activation='relu')
  BatchNormalization()
  Dense(2, activation='softmax')])
```

As the loss function, we used *categorical cross-entropy*. The *Adam* optimizer is chosen (with most of its parameter values set to default, as specified in the Keras documentation) for the learning process. Specifically, we used *Glorot weight initialization*[3] for the output layer with `Softmax` activation function and *He weight initialization*[4] for the hidden layers with `ReLu` activation functions.

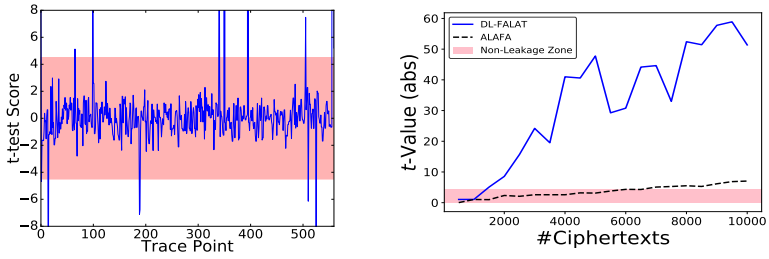
Interestingly, we observed that the byte-level model itself is fairly generic for learning even those cases where the countermeasure structure is inherently bit-level. However, the learning is

[1] While listing out different fault locations for fault simulation, one may typically take advantage of the fact the cipher structures are highly regular in nature. Thus, simulating faults only for a couple of locations in final 2-3 rounds would suffice. However, the number would still be high, given different fault models. Similarly, for fault simulations at lower levels of abstractions, one may have to consider several instructions (for software) or gate inputs/registers (for hardware) even if a few high-level locations are identified based on the structural regularity. In a nutshell, it is always desirable to keep the number of fault simulations as low as possible while evaluating leakage.

[2] `BatchNormalization` is useful to regularize the learning. Furthermore, it decouples the learning process of the hidden layers from each other. It also speeds up the learning process [IS15].

[3] Helps to avoid weights getting diminished or exploded during training [GB10]. It draws the initial weights of NNs from either a normal or uniform distribution with a centred mean and standard deviation scaled to the layer's number of input and output neurons.

[4] It has been shown that the *He weight initialization* [HZRS15] is effective for optimizing weights associated with `ReLu` activation functions.



(a) (b)

Figure 1: Infection countermeasure [GST12] single-byte fault: a) t-test scores ($d = 3$) from ALAFA; b) Variation of absolute t-test scores for DL-FALAT and ALAFA with respect to ciphertext count.

found to be very slow and requires a lot more samples compared to the bit-level model for such cases. Our general suggestion, thus, is to apply both the models for leakage assessment and use the one who performs the best in a typical scenario. In all of our experiments, we observed that one of these two models is able to learn the target leakage with a reasonable number of ciphertexts provided as samples. We elaborate this experimentally in the next subsection, where we show that DL-FALAT outperforms the t-test-based strategy (ALAFA) for all the countermeasure examples under consideration.

3.4 Illustrative Examples

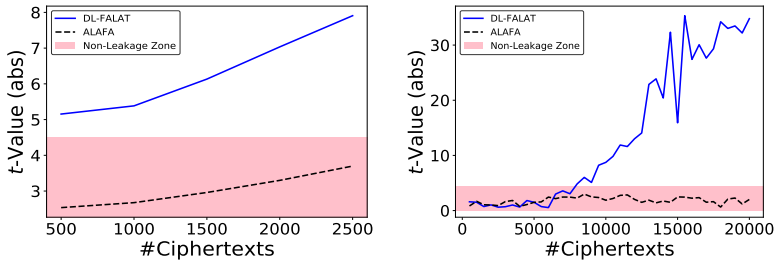
In order to establish the efficacy of the DL-based approach over t-test based ALAFA[1], here we present three illustrative examples. Two of our examples are based on the infective countermeasure proposed in [GST12]. The third one is based on the countermeasure from [WLD+16].

Example 1. The infective countermeasure proposed in [GST12] tries to randomize the outcome upon the detection of a fault. The target is to make the faulty ciphertext completely unexploitable for the attacker. The main proposal is based on the AES block cipher. The protected implementation executes each round of AES two times – the first one contributes in actual encryption, and the second one is redundant. Furthermore, there are random "dummy" rounds (round computations over a random string changing at each encryption). An arbitrary number of dummy round computations take place between each actual and redundant round computation. The entire execution sequence of dummy rounds is controlled by a random string which changes for each encryption. The detection of a faulty execution takes place by taking the XOR difference between each actual and redundant round and passing the differential through a non-linear transform. The transformed differential is next combined with the cipher, redundant, and dummy rounds to spread the "infection". The final step executes a compulsory dummy round computation to spread the randomized fault-induced infection further through the entire state and eventually, a randomized state is outputted.

One should note that the purpose of dummy rounds in [GST12] is to confuse the attacker regarding the actual location of fault injection. In the presence of randomly executing dummy rounds between each cipher and redundant round computation, the attacker cannot detect at which iteration the fault has to be induced in order to corrupt a cipher (or redundant) round and not a dummy round. The induction of fault in a dummy round does not cause any information leakage through the ciphertext. Overall, the impact of fault induction in dummy rounds can be considered to be an addition of noise in the faulty ciphertext distribution by the countermeasure.

As we show next, this infective countermeasure fails to prevent the FAs from happening. Without loss of generality, we consider fault injection targeting the 9th round of AES in this case. In the first case study, the dummy rounds are considered to be absent, while they will be present in the second case.

[1] We refer to the t-test based leakage assessment as "ALAFA" in the rest of the paper.



(a) (b)

Figure 2: Comparative analysis of DL-FALAT with ALAFA: a) Infection countermeasure [GST12] with dummy rounds and a single-byte fault. The absolute values of t-statistic have been plotted for different count of dummy rounds $\#dum$. The amount of noise increases with the increase in $\#dum$ according to the Eq. (6). b) Variation of absolute t-test scores for DL-FALAT and ALAFA in the case of RIMBEN countermeasure with the count of ciphertexts.

Fig. 1(a) illustrates the leakage profile from this countermeasure for ALAFA. One may observe, that the leakage is higher-order and happens for $d = 2$, and even for $d = 3$ (shown in the plot). In fact, there are leakage components of order 4 (will be explained, eventually in the next section). Fig. 1(b) illustrates the outcome from the DL-FALAT in terms of absolute t-values from its decision making step. No information has been supplied to DL-FALAT regarding the leakage order d. However, the leakage has been detected roughly with 1400 ciphertexts. For the sake of comparison, the ciphertext count from ALAFA is also shown in Fig. 1(b). The leakage detection here requires almost 7000 ciphertexts which is almost 5 times more than that of DL-FALAT.

Example 2. In the second illustration of ours, the leakage detection is performed on [GST12] with the dummy rounds included. Fig. 2(a) presents the leakage profiles with respect to the number of dummy rounds for both ALAFA and DL-FALAT [1]. The dummy round counts are varied to control the amount of noise added to the ciphertext distribution. Let the total count of dummy rounds be $\#dum$. The total count of cipher and redundant rounds is 22 in the case of AES. Also, each round (cipher, redundant, or dummy) takes one loop iteration to complete. Given that the fault induction targets the input of the 9th round of AES, and the target loop iteration for injection is set to $(22 + \#dum - 2)$, the signal probability (that is while the fault actually corrupts the AES 9th round, rather than a dummy round or any other AES round computation) is given by the following formula:

$$Pr_{sig} = \frac{(19 + \#dum)!/(19)!(\#dum)!}{(21 + \#dum)!/(21)!(\#dum)!} \tag{6}$$

For reasonable values of $\#dum$ (i.e. $\#dum = 20, 25, 30, 35, 40$) the signal probabilities are 0.256, 0.202, 0.164, 0.136 and 0.114, respectively. As it can be observed, DL-FALAT outperforms ALAFA by a very large margin for all the noisy cases. Even for a sufficiently large count of ciphertexts (1000000), ALAFA fails to detect the leakage while DL-FALAT succeeds.

Example 3. This example presents another case study, where ALAFA fails to detect any leakage with a reasonable number of traces, while DL-FALAT indicates leakage even with a moderate trace count. The countermeasure we analyze is due to [WLD+16], also called RIMBEN (Random Infection based on Modified Benes Network). RIMBEN detects the presence of a fault during execution by taking the differential of a cipher and a redundant computation state at each intermediate round during encryption. The fault is then propagated through the computation, and at the end, the faulty ciphertext (C) is XOR-ed (masked) with a random bit string and returned as output. The random bit string is also generated from the fault differential ΔC, by means of a preprocessing logic and two

[1] Only for this case study, we present the leakage result by varying the count of dummy rounds. For each valuation of dummy round count, same (1000000) number of ciphertexts are considered.

consecutive Benes network. The random bitstrings used by this construction for masking faulty ciphertexts always have a Hamming Weight (HW) of $\frac{N}{2}$, where N denote the block size of the cipher, as well as the size of the $N \times N$ Benes network. Standard values of N are 128 or 64. In the present context, we consider a protected AES implementation for which $N = 128$.

The analysis results on RIMBEN have been illustrated in Fig. 2(b). Once again, in this case, we consider a fault injection at the 9th round of AES state. The analysis has been performed for both bit and byte-level abstractions of the ciphertexts, with the bit-level results being more prominent. As it can be observed, the leakage is observed by the DL-FALAT within 8000 ciphertexts. In contrast, ALAFA cannot detect any leakage even while higher orders up to 128 is considered[1].

It is tempting to discuss how leakage happens in the case of RIMBEN. The key observation here is that the HW of the random bitstring to be XOR-ed with the faulty ciphertext C is constant and known to an attacker who knows the construction. Furthermore, if the fault diffusion is incomplete (e.g. for fault induction in 9th round of AES, the fault is defused only in 32 bits of the 128-bit state), several bits of the random string are directly exposed to the attacker (as the corresponding bits in the differential ΔC of correct and faulty ciphertext are zero). As a result, the attacker can calculate the HW of the bits XOR-ed with the non-zero part of ΔC. The exposure of this HW enables guessing the non-zero part of ΔC. In case the HW associated with the non-zero part is low, the guessing would have very low entropy which makes the attack reasonably feasible. In a nutshell, the attacker can expose the ΔC in case of incomplete fault diffusion even at the presence of the infection operation. An attack of a similar kind has been presented in [FCL+19].

It is apparent from the attack described in the previous paragraph that all 128 bits of the ciphertexts are somewhat important in leaking the secret in this case, as they together decide the HW. Moreover, the leakage is typically due to the low entropy of the mask. As it was observed through our experiments, the moment-based approach of ALAFA cannot capture this HW-based leakage. However, DL, being able to learn even complex relations within patterns, is successful in this context. This example, by itself, provides a strong motivation for using DL instead of other conventional statistical approaches for FA leakage assessment.

Understanding the cause of leakages from the DL-based approach is not straightforward, unlike the point-wise approach in ALAFA. However, explaining the leakages is still feasible. In the next subsection, we present a methodology for leakage interpretation and explain the leakages obtained in this subsection to greater detail.

3.5 Leakage Interpretation

One desirable property of a good leakage detection test is its interpretability. The goal of this subsection is to investigate how the outcomes of the DL-based leakage test can be interpreted. There exist multiple approaches in the literature to interpret the classification process of a DL model, and they have also been used previously in the context of SCA security [Tim19, WMM19]. However, there still exist certain important questions which were not clearly addressed. Firstly, in a typical countermeasure assessment problem, it may happen that the model only takes certain leaky features into consideration while ignoring others. This is perfectly natural as the desired classification may be easily achieved by considering those features only. Exposing all leakage points on a ciphertext thus become a non-trivial issue[2]. The second issue takes place while a leaky ciphertext contains several leakage points (which is the case for most of our examples). From the DL-based method, it is difficult to understand whether the leakage is univariate or multivariate. Note that ALAFA answers this question in a reliable manner by gradually increasing the analysis order d. However, it has already pointed out at the beginning of this section, that such incremental

[1] Considering higher-order leakages in ALAFA requires the construction of all possible subsets up to the specific leakage order. In the present case, we need to go up to order 128. The total number of subsets to be considered up to order 128 is 2^{128}, which is clearly infeasible to cover. For the sake of experiments, we considered all possible subsets up to order 5 for ALAFA, and the single case where the order of test is 128. The result being shown in the plots are for test order 128.

[2] Such analysis can also give valuable information on how to attack.

approach with t-test is sub-optimal due to several reasons. One of the major motivation of leakage interpretation thus is to extract such information from a DL model.

We begin with the trained network model $\mathscr{M}$ during the leakage interpretation. Like the leakage detection test, once again, we adopt an iterative approach. The very first step we perform, is a *Sensitivity Analysis* (SA) [Tim19], which returns the contribution of each feature in learning the leakage. Mathematically, the *Sensitivity* (Im_i) for each feature is computed using the following equation:

$$Im_i = \left| \sum_j \frac{\partial y_0}{\partial x_i} \cdot X_i^j \right| \tag{7}$$

Here x_i denote the i-th input of the model $\mathscr{M}$, y_0 is the first output of $\mathscr{M}$, and X_i^j is the value of the i-th input in the j-th ciphertext from the validation set. One important distinction with the leakage detection test described in Algorithm 3 is that here we take the trained model and consider a fresh and sufficiently large validation set while computing the feature importance values.

The SA step assigns real values to individual features by means of which they can be ranked according to their contribution in the learning. In our analysis, we first begin with the subset of *most important features*. The determination of the most important subset, which we denote as MI, is driven by a threshold Th_{MI}. In our experiments, we found that the average over all Im_is works reasonably good as a threshold.

Once the MI has been determined, the analysis follows two separate paths. In the first path, we simply eliminate all the features in MI by assigning them with value 0. The next step is to repeat the learning once again (by calling Algorithm 3 on the modified feature vectors) and see if the model still learns. In case of a negative result, the dataset is increased gradually as described in Algorithm 3 up to some predefined threshold. Our suggestion regarding this threshold is to keep it larger than the standard leakage detection test. This is because the main aim of leakage interpretation is to expose even the most difficult-to-detect leakage points rather than just exposing if there is leakage or not. The method of feature elimination and training continues iteratively, until either all feature points are exhausted, or the model fails to learn.

The second path in our analysis tests the MI set obtained in an iteration. This operation aims to check if the leakages at different points are univariate or multivariate. We apply the same trick of eliminating feature points in this case. However, only one point is eliminated at each step, and the training is repeated. The main idea here is that *in case of a univariate leakage, even a single point in the ciphertext would be able to classify, whereas in case of multivariate leakage the classification would surely require multiple points.* Note that, this mechanism can only distinguish between univariate and multivariate leakages and would not necessarily indicate the exact leakage-order. In order to achieve the exact order, one must perform the analysis for each feasible subset of MI. While this is feasible if MI is small, it would be extremely costly to perform for larger MI sizes. In all our experiments, we performed the single-point elimination based strategy. A more detailed, albeit efficient analysis of leakage order is left as an open problem. We would also like to point out that the proposed approach of ours might be applicable for analyzing leakages in SCA traces too. However, the leakage scenario is quite different for SCA, and it is out of the scope of the current work to comment on something conclusive regarding its applicability for SCA leakage interpretation.

Example 4. In order to explain the leakage interpretation methodology, once again we refer to the countermeasure proposed in [GST12]. For a fault injection at the 9th AES round for this countermeasure, the randomized ciphertext obtained is given by the following equation:

$$C = \begin{bmatrix} m_0 \oplus 2r_1 \oplus 1r_2 & 1r_3 & 3r_4 \oplus 1r_5 \oplus 1r_6 & 3r_7 \\ 1r_1 \oplus 3r_2 & 1r_3 & 2r_4 \oplus 3r_5 \oplus 1r_6 & m_1 \oplus 2r_7 \\ 1r_1 \oplus 2r_2 & 3r_3 & m_2 \oplus 1r_4 \oplus 2r_5 \oplus 3r_6 & 1r_7 \\ 3r_1 \oplus 1r_2 & m_3 \oplus 2r_3 & 1r_4 \oplus 1r_5 \oplus 2r_6 & 1r_7 \end{bmatrix} \tag{8}$$

Here, each m_i (8-bit) refers to a component corresponding the faulty intermediate, and the r_is (8-bit) are random. Careful observation reveals that each m_i can be extracted from a given randomized

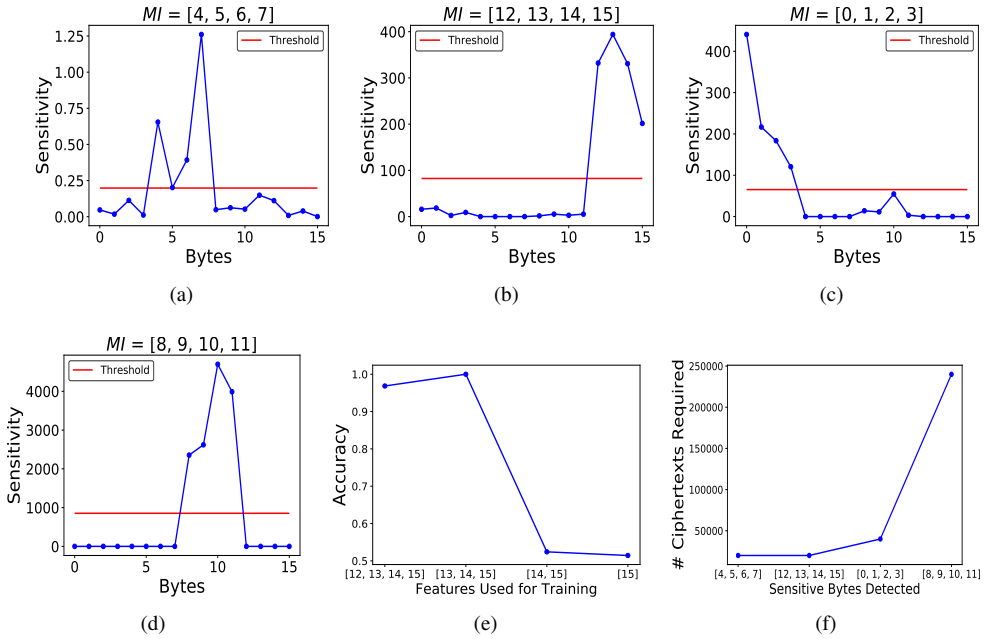


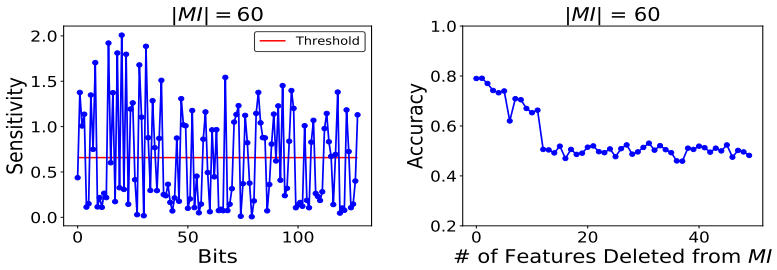
Figure 3: Leakage interpretation for infective countermeasure [GST12]; (a) SA results for the first iteration; (b) SA results for the second iteration; (c) SA results for the third iteration; (d) SA results for the fourth iteration; (e) *MI* analysis for the first iteration; (f) Variation of ciphertext count for detecting the leakage points in different SA iterations.

ciphertext as the random components cancel each other. In other words, all four columns have leakage components in this case. We shall now explain how the leakage interpretation of DL reveals valuable information regarding this ciphertext structure.

The first set of leaky points (i.e. the set *MI*) that gets exposed by the SA are the bytes $[4,5,6,7]$ in the AES state (see the second column in Eq. (8), and the Fig. 3(a)). *The reason behind this column getting exposed first is that the leakage here is of the first order, and is associated with only a single random variable* r_3. The number of ciphertexts required to expose this leakage prominently is 1400. However, in order to extract as many leakage points as possible in one iteration, we begin the analysis here with a larger dataset of size 20000. In the next step, we remove these points from the dataset in order to expose other leakage points. This elimination readily exposes leakages in points $[12,13,14,15]$, which also involve a single random variable (ref. Fig. 3(b)). Note that, we do not require to increase the dataset size for revealing these leakages. This is natural as the order of the leakages is the same as the previous set. In a very similar manner, the next set of leakage points getting exposed are $[0,1,2,3]$, for which 40000 ciphertexts are required (ref. Fig. 3(c)). Finally, the third leakage column gets exposed with ciphertext count of 240000 (ref. Fig. 3(d)). The gradual increase in the size of the dataset required for exposing the leakages (Fig. 3(f)) gives an indirect indication that the leakages of all the columns are not of the same order.

Another path of leakage interpretation is to analyze whether the leakages are univariate or multivariate. Without loss of generality, we perform this analysis only for the set $MI = [12,13,14,15]$. One may clearly observe from Fig. 3(e), that the learning requires the ciphertext byte 13 to be included in the feature set. Furthermore, owing to the fact that point 13 individually does not leak any information (otherwise the model would have ignored the rest of the features), it is established that the leakage is multivariate in nature in this case. However, we would like to point out that the true order of leakage can only be exposed if all possible subsets of this *MI* are individually used for learning.

Example 5. In a similar manner to the previous example, we performed the leakage analysis



(a) (b)

Figure 4: Infection countermeasure RIMBEN [WLD+16] with single-byte fault: a) SA analysis results from the first iteration; b) *MI* analysis results from the first iteration.

for the RIMBEN countermeasure. The first step of SA reveals the set of 60 points, as shown in Fig. 4(a). *However, it can also be observed that the points which do not get included in MI also have some observable sensitivity.* The size 60 of the *MI* set can be explained by the fact that the HW of the masking string in RIMBEN is 64, and so knowledge of roughly half of the ciphertext bits somewhat sets the boundary between two classes. The analysis of *MI* shows that after removing 10 points from this set of 60, the validation accuracy becomes 0.5. Once again, the analysis reveals that the leakage is multivariate. Further steps in this analysis are similar, and we do not repeat them here.

4 Leakage Assessment for Different Implementation Classes and Fault Models

The previous section outlined the main idea of the DL-FALAT. In the rest of this paper we mainly utilize the two tests mentioned in Algorithm. 1 and 2 with the DL-based leakage assessment test replacing the $TEST$ operation. However, a couple of simple optimizations and enhancements have to be performed to handle certain countermeasure and fault classes as well as applications beyond cipher implementations. In the next subsection, we begin with one such optimization useful for detection-based countermeasures. It should be noted that while testing a countermeasure, it is not mandatory to know its respective class. Instead, one can directly begin by applying the optimization we are going to describe here. If the optimization is useful, the test inherently takes advantage of it. The test can also proceed as it is, even if the optimization does not bring any advantage.

Algorithm 4 *PREPROCESSING*

Input: Protected Cipher $\mathscr{C}$, Fault model F, Simulation counter S'
Output: Fault value pair (f_1, f_2)

1: $\mathscr{T}_{f_i} := \emptyset, \quad \forall f_i \in F$
2: Initialize dictionary $D_C = \emptyset$
3: $p := GEN_{PT}()$
4: $k := GEN_{KEY}()$
5: **for** each $f_i \in F$ **do**
6: **for** $i_1 \leq S'$ **do**
7: $\mathscr{T}_{f_i} := \mathscr{T}_{f_i} \cup \mathscr{C}(p,k,f_i)$
8: **end for**
9: **end for**
10: **for** $c \in \cup_{f_i} T_{f_i}$ **do**
11: Populate dictionary $D_C[c] := D_C[c] + 1$
12: **end for**
13: Select (c_1, c_2) such that $D_C[c_1]$ and $D_C[c_2]$ are maximum and second maximum.
14: Get the faults (f_1, f_2) corresponding to (c_1, c_2)
15: **Return** (f_1, f_2)

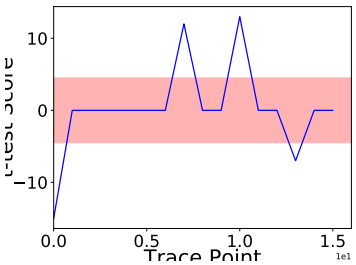


Figure 5: ALAFA results: Parity-based detection countermeasure ($d = 1$). The points crossing the red region indicates leakage.

4.1 A Prepossessing Targeting Detection Countermeasures

The leakage assessment test presented in Algorithm. 1 takes an arbitrary pair of (f_1, f_2) for testing. Strictly speaking, the test framework proposed here is a rejection test which flags a problem if there exists at least one pair of (f_1, f_2) for which leakage can be identified. *Although theoretically, it indicates that in the worst case, all pairs should be tested, this is not the case in practice. Most of the countermeasures behave uniformly on any fault once it is detected, and this fact makes our test to succeed with fairly good probability for any arbitrary* (f_1, f_2). The same claim is also true for the case while the fault value is kept fixed, and the key is varied.

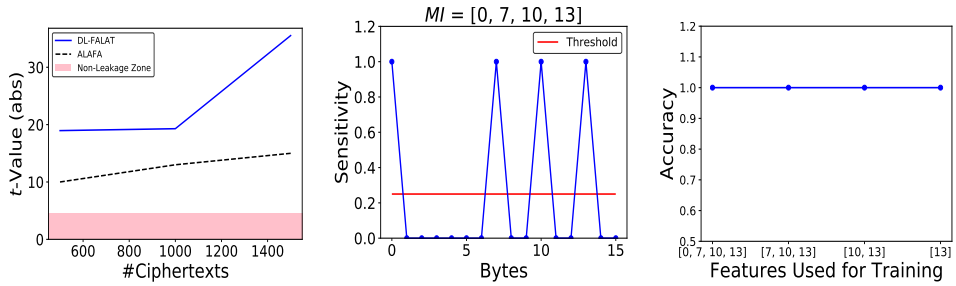
However, there exist certain exceptions to this, mainly due to some of the detection based countermeasures. For detection-based countermeasures using error-correcting codes, leakage only happens for specific fault values. Therefore, to identify the existence of leakage, at least one of the faults in (f_1, f_2) pair must be a leaky fault value.[1] As an example, we consider a simple parity-based countermeasure for AES having 1-bit parity for each byte. This error detection logic can only detect 50% of the fault values (for which the parity is even) and is bypassed for the rest. Upon the detection of a fault, a random string is outputted. In case both f_1 and f_2 are detected faults, their corresponding ciphertext distributions will be random and hence indistinguishable.

In our current setting, the knowledge of the countermeasure algorithm is not taken into account, and there is no direct way of determining which of the fault values leak. However, a simple preprocessing step can solve this issue. One important observation here is that *leakage can be exploited by an adversary only when the ciphertext corresponding to a leaky fault value remains the same over multiple simulations with the same fault value. Otherwise, the attacker cannot distinguish the exploitable ciphertexts from the random ones, even with the complete knowledge of the algorithm.* Based on this observation, we propose a simple preprocessing strategy to identify a potential set of leaky fault values. This algorithm executes over the fault simulation data before Algorithm. 1 and isolates a set of leaky fault values if the leakage is conditional upon the fault values. In case the leakage is uniform over all fault values, the algorithm returns a random (f_1, f_2).

The preprocessing algorithm is described in Algorithm. 4. The algorithm first simulates each fault value under a given fault model a sufficient number of times (S'). The next step is to create a dictionary D_C, which is keyed by a ciphertext, and stores the frequency of occurrence of that ciphertext. For certain countermeasures, a leaky fault always maps to the same ciphertext. In such cases, the frequency distribution of the dictionary will be non-uniform. The dictionary entries with high frequency indicate the repetition of the same ciphertext which in turn indicates a leaky fault.

Note that, in the case of randomized countermeasures, the preprocessing step will always return a random pair of (f_1, f_2) as ciphertext repetition occurs only with negligible probability. However, the leaky fault usually occurs when a countermeasure gets bypassed by the fault value. Hence, there is a little chance of a leaky fault getting randomized. In summary, the algorithms described so far do not have any false negatives unless the leakage testing strategy fails. It is worth noting that, the leakage test with fixed fault and the varying key is less suitable in the context of this optimization. This is because the above-mentioned preprocessing step demands a sweep through all possible fault values (or rather all possible internal state configurations) to detect a so-called leaky fault while it exists. There exist code based countermeasures for which the percentage of leaky faults is substantially low. If instead of sweeping through the fault values one chooses to change the keys, the number of iterations required may be significantly high (as one need to continue until at least one leaky fault is obtained, and that solely depends on the configuration of the faulty internal state). Varying fault values, however, seems to be a rather easier choice here as the fault value space is usually small. In fact, this is one of the potential examples where varying fault values seems to be a natural choice. However, it is important to understand that an equivalent version of Algorithm. 4 do exist for varying keys. However, we do not outline it here for the sake of simplicity.

[1] For the other case, where the key is varying, at least one of the key should lead to an intermediate configuration for which the fixed fault value becomes leaky.



(a) (b) (c)

Figure 6: Leakage analysis and leakage interpretation results for the parity example: a) Comparative leakage analysis of DL-FALAT and ALAFA with varying number of ciphertexts; b) SA of leakage; c) Analysis of MI set indicating univariate leakage.

Example 6. In order to explain the working principle of the preprocessing along with the non-interference test here, we utilize the 1-bit parity-based error detection on the block cipher AES. To detect a proper fault pair (f_1, f_2), we apply Algorithm. 4 for all possible byte fault values at the 9-th round input of the AES cipher. Each fault value is simulated several times ($S' = 10000$). The dictionary $D_C[c]$ here contains equally populated bins for 50% of the fault values which bypass the countermeasure and hence generate a constant faulty ciphertext in each case. Selection of the first two highest entries of $D_C[c]$ thus provides two leaky faults f_1, f_2. The univariate ($d = 1$) t-test (i.e. with ALAFA, where the t-test is performed independently for each byte location) profile for these two faults has been shown in Fig. 5. It can be observed that there are 4 leaky locations in the ciphertext. This is due to the incomplete diffusion of the fault injected at the 9-th round. The comparative analysis between DL-FALAT and ALAFA (Fig. 6(a)) in terms of ciphertext count shows that DL-FALAT attains much higher t-scores for any specific count of ciphertexts. Higher t-scores indeed indicate stronger statistical confidence. Hence, even though both ALAFA and DL-FALAT detects leakage with few ciphertexts, the confidence of DL-FALAT is higher than that of ALAFA.

Another important exercise in this context is to perform the SA on the results of DL-FALAT. As it can be observed from Fig. 6(b), the SA reveals the same leakage points revealed by ALAFA in Fig. 5. The analysis of the MI set further proves that the leakage is univariate in this case (Fig. 6(c)). This is because the learning can be performed with high accuracy even with a single feature point in this case.

4.2 Detecting SIFA attacks

In this subsection, we adopt the proposed test methodology for detecting a specific attack class called SIFA. SIFA is a recently proposed fault attack technique which utilizes the fact that the *activation* and *propagation* of a fault depends on secret intermediates. As a result, in many cases, one can observe correct ciphertexts even while faults are being injected. As a simple example of how this happens, consider that an attacker injects a stuck-at-0 fault to some intermediate bit of the cipher. If the actual value of the bit is 0, no alteration will take place, and a correct ciphertext can be observed. In contrast, if the actual bit value is 1, it will result in a faulty execution. Typically, SIFA attacks exploit the correct ciphertexts for key recovery instead of faulted ones, and this feature is crucial for bypassing most of the existing state-of-the-art FA countermeasures [DEK+18, DEG+18].

Although many (if not all) of the state-of-the-art FA countermeasures fall prey under SIFA, the success of the attacks strongly depends on the fault models. Typically, it is found that biased bit-faults cause SIFA. However, as shown in [SJR+19], SIFA based on biased register-level faults can be successfully prevented using masking schemes. However, the masking schemes themselves may fall prey against certain typical SIFA faults, while the propagation effect of the faults through the masked circuit are taken into account [DEG+18]. Most of the recently proposed

SIFA countermeasures thus suggest the use of error-correction [SRM19, SJR+19, BKHL19] to prevent such attacks from happening.

Algorithm 5 *TEST-INTERFERENCE-SIFA*

Input: Protected cipher $\mathscr{C}$, Fault value f, Key k_1, k_2,
Simulation counter S, Initial simulation counter S_{init}, Model $\mathscr{M}$
Output: Yes/No

$\mathscr{T}_{k_1} := \emptyset$; $\mathscr{T}_{k_2} := \emptyset$
$p := GEN_{PT}()$
$c_{corr_1} := \mathscr{C}(p, k_1)$
$c_{corr_2} := \mathscr{C}(p, k_2)$
$S_t := S_{init}$
$leak := Null$
while $S_t \leq S$ **do**
  for $i \leq S_{init}/2$ **do**
    $\mathscr{T}_{k_1} := \mathscr{T}_{k_1} \cup \langle c_{corr_1} \oplus \mathscr{C}(p, k_1, f), 0 \rangle$
    $\mathscr{T}_{k_2} := \mathscr{T}_{k_2} \cup \langle c_{corr_2} \oplus \mathscr{C}(p, k_2, f), 1 \rangle$
  end for
  $\mathscr{D}_t := \mathscr{T}_{k_1} \cup \mathscr{T}_{k_2}$
  $\langle \mathscr{D}_t^1, \mathscr{D}_t^2, \cdots, \mathscr{D}_t^K \rangle :=$ `GEN-CROSS-VALID-SET`$(\mathscr{D}_t)$
  $A_t := \emptyset$
  for $i \leq K$ **do**
    $Tr_t := \bigcup_{\substack{j=0 \\ j \neq i}}^{K} \mathscr{D}_t^j$
    $Vl_t := \mathscr{D}_t^i$
    $a_t^i :=$ `Train-and-Validate`$(\mathscr{M}, Tr_t, Vl_t)$
    $A_t := A_t \cup \{a_t^i\}$
  end for
  if (`t_Test`$(A_t) \implies \mathscr{H}_0$) **then**
    $leak = False$
  else
    $leak = True$
  end if
  if $leak$ **then**
    Return Yes
  else
    if $S_t \leq S$ **then**
      $S_t = S_t + S_{init}$
    else
      break
    end if
  end if
end while
Return No

The goal of this section is to tailor the test methodology in a way so that it can capture the intricate fault model dependencies of SIFA attacks, and based on that can certify the countermeasures. One straightforward approach is to declare a countermeasure to be vulnerable even if a single faulty ciphertext is obtained during fault simulation (that is to say 100% correctness ensures SIFA security). This simple approach has been adopted in [AWMN20]. However, we show via several examples that this is very conservative and will lead to false positives by undermining the fault models. In a practical case, it is indeed feasible that complete SIFA protection is not targeted due to resource constraints (most of the existing SIFA countermeasures are costly in terms of resources), but security up to some extent can still be achieved for several different classes of SIFA fault models. A conservative approach would not be able to verify such situations, and as a result, a more fine-grained alternative is desired.

While testing for SIFA, we typically exploit the fundamental fact that fault activation and propagation depend on secret faulted intermediates. Note that, most of the SIFA fault models use bit-level faults for which only one possible value of fault exists (i.e. if the bit is originally 0, the faulted value is 1 and vice versa). Our approach, in this case, is to vary the key instead of the fault values (ref. Algorithm 2). The fault is to be simulated with different probabilities and level-of-abstractions. At this point, we need to formalize the fault models in the context of SIFA (which we have avoided, so far, for other fault models as it was not required) as there is a probability associated with the faults in this case. Faults are typically modelled as transition probabilities of bits. Each bit b within a state under the influence of fault injection may get flipped with some

probability depending on its value. Mathematically, for each bit, there can be 4 associated transition probabilities given as:

$$\begin{aligned} pr_{0\to0} &= Pr[b' = 0 | b = 0] \\ pr_{0\to1} &= Pr[b' = 1 | b = 0] \\ pr_{1\to0} &= Pr[b' = 0 | b = 1] \\ pr_{1\to1} &= Pr[b' = 1 | b = 1] \end{aligned} \tag{9}$$

Here b' represents the altered valuation of b. Typically, a bit fault is considered *biased* if $pr_{0\to1} \neq pr_{1\to0}$ (resp. $pr_{0\to0} \neq pr_{1\to1}$, as $pr_{0\to0} + pr_{0\to1} = 1$ and $pr_{1\to0} + pr_{1\to1} = 1$)[1]. Note that, faults can also affect multiple bits, and there can be different transition probabilities for each of the affected bits. For simplicity, here we keep the analysis restricted to single-bit faults. However, it can be extended for multi-bit biased faults in a straightforward manner as outlined in [SJR+19].

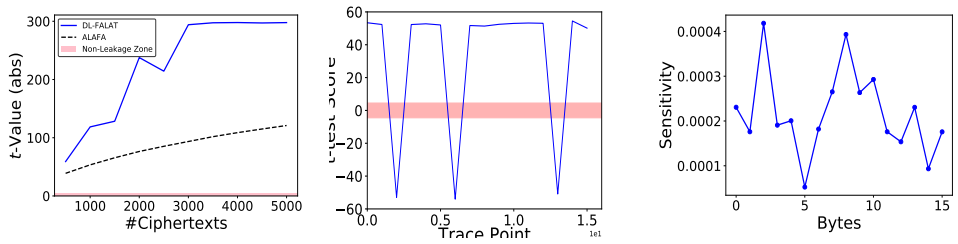
The SIFA attacks exploit this bias in transition probability for detecting the correct keys. The idea is to partially decrypt the correct ciphertexts obtained during a fault injection campaign up to the injection location (or maybe to some location in the cipher state which follows the injection location during encryption) by making key guesses. The aforementioned bias due to the fault injection becomes visible with very high probability for the correct key. In contrast, for the wrong key guesses, the resulting distribution from partial decryption remains very close to random.

Going back to the testing methodology, here we use a basic trick which exposes the bias in fault injection (if any) at the ciphertext level. The test happens with a fixed plaintext p and two chosen keys k_1 and k_2. Naturally, we have access to the correct ciphertexts corresponding to both k_1 and k_2. *For all the obtained ciphertexts during the fault injection campaign, we consider the XOR differential with the correct ciphertexts.* Our test is next performed as it is on this differential data rather than the ciphertexts. The steps are summarized in Algorithm 5.

Why SIFA Leakage gets Exposed It is indeed a tempting question that how SIFA leakage gets exposed through the aforementioned modification of DL-FALAT. Considering the differential makes the correct ciphertexts obtained in the injection campaign equal to zero. The differentails corresponding to the obtained faulty ciphertexts are either random (in case the countermeasure randomizes the faulty output), or some non-zero constant value (in case, the cipher outputs some pre-defined constant value upon the detection of a fault). Each dataset $\mathscr{T}_{k_1}$ or $\mathscr{T}_{k_2}$, thus contains a lot of zero-valued bit/byte strings along with some random or non-zero constant strings. Let us denote the count of zero-valued strings as Cnt_0 and non-zero strings as Cnt_1 in one of the datasets (say in $\mathscr{T}_{k_1}$). *The ratio* $R_{=0} = \frac{Cnt_0}{|\mathscr{T}_{k_1}|}$ *roughly equals to either* $pr_{0\to0}$ *or* $pr_{1\to1}$ *depending on the value of the faulted intermediate bit* b *while the plaintext* p *is encrypted with* k_1. This is because b remains unaltered either with probability $pr_{0\to0}$ or $pr_{1\to1}$ which eventually results in a correct ciphertext. Likewise, the ratio $R_{\neq0} = \frac{Cnt_1}{|\mathscr{T}_{k_1}|}$ roughly equals to either $pr_{0\to1}$ or $pr_{1\to0}$. Next, let us consider the two datasets $\mathscr{T}_{k_1}$ and $\mathscr{T}_{k_2}$. With little inspection, we can find out two keys k_1 and k_2, such that for one of them the intermediate bit b under the influence of fault assumes a value 1. For the other key, it assumes a value 0. One may observe that, *the ratios* $R_{=0}$ *(resp.* $R_{\neq0}$*) for these two datasets become different under the condition imposed previously on the transition probabilities.* This is because in one of the cases (say for k_1), $R_{=0}$ equals to $pr_{0\to0}$, while in the other case it equals to $pr_{1\to1}$. Difference in the ratios establish the fact that the two underlying distributions in $\mathscr{T}_{k_1}$ and $\mathscr{T}_{k_2}$ are also different, which indicates leakage.

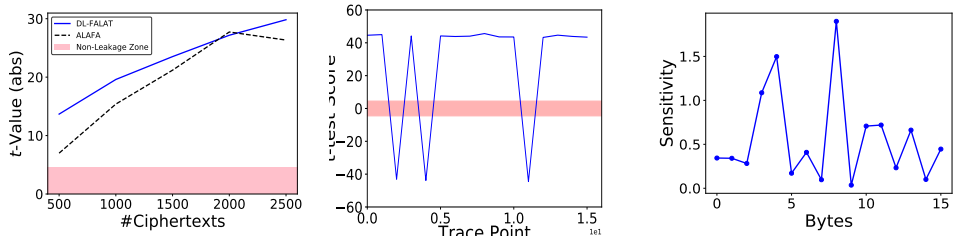
The most important feature of the aforementioned leakage is that it depends on the actual value of a secret intermediate bit. There exist SIFA classes which do not typically require a biased fault injection to be made to enable a data-dependent leakage. Such attacks, as outlined in [DEG+18] exploit the underlying circuit structures as well as the fact that fault propagation to the output of

[1] SIFA attacks based on biased faults are the dual of conventional *biased statistical fault attacks (SFA)*, where the faulty ciphertexts are used for key recovery. Our test of SIFA attacks ,thus, also covers the cases of SFA.



(a) (b) (c)

Figure 7: Leakage analysis for SIFA (on AES with redundancy) with fault transition probabilities $pr_{0\to1} = 0$, $pr_{1\to0} = 1$, $pr_{0\to0} = 1$, $pr_{1\to1} = 0$: a) Variation of leakage with ciphertext count; b) Point-wise (univariate) leakage profile for ALAFA; c) SA analysis result with DL-FALAT.



(a) (b) (c)

Figure 8: Leakage analysis for SIFA (on AES with redundancy) with fault transition probabilities $pr_{0\to1} = 0.75$, $pr_{1\to0} = 0.25$, $pr_{0\to0} = 0.25$, $pr_{1\to1} = 0.75$: a) Variation of leakage with ciphertext count; b) Point-wise (univariate) leakage profile for ALAFA; c) SA analysis result with DL-FALAT.

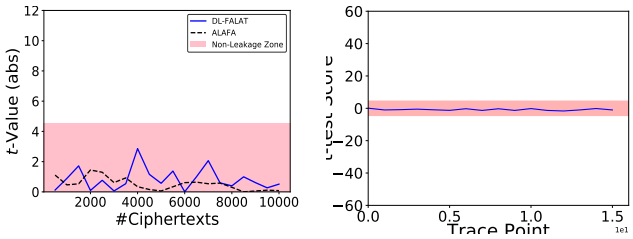
certain non-linear functions is data-dependent. Our proposed test can detect even such complex leakages without any alteration.

Example 7. In order to further elaborate the SIFA detection methodology here we present three test cases, where an FA-protected implementation of AES is targeted with SIFA faults having different transition probabilities. As FA protection, we use simple redundancy-based countermeasure, i.e. the computation is repeated twice on the same data. Upon detection of a fault, a constant is outputted. In the first example we consider a stuck-at-0 fault ($pr_{0\to1} = 0$, $pr_{1\to0} = 1$, $pr_{0\to0} = 1$, $pr_{1\to1} = 0$). Clearly, this is a case where SIFA attack is feasible. The leakage profile for this attack is shown in Fig. 7(a), which presents the variation of leakage with respect to ciphertext count for both ALAFA and DL-FALAT. While both of them detects leakage quite efficiently, the t values obtained in DL-FALAT are much higher than that of ALAFA. Higher t values indeed indicate stronger evidence in support of leakage. The point-wise leakage profile from ALAFA, and the SA results of DL-FALAT are shown in Fig. 7(b) and Fig. 7(c), respectively. It is worth mentioning that the leakage is univariate in this case.

The next example injects faults with $pr_{0\to1} = 0.75$, $pr_{1\to0} = 0.25$, $pr_{0\to0} = 0.25$, $pr_{1\to1} = 0.75$. Once again the leakage is observed by both ALAFA and DL-FALAT (see Fig. 8(a-c)) having a similar trend like the previous case.

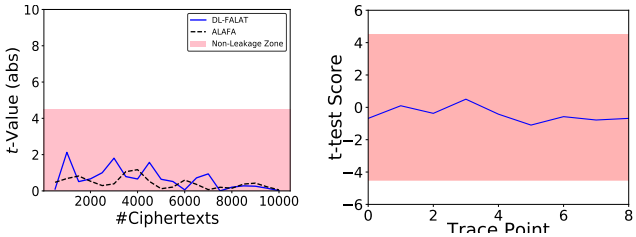
In the third case we simulate faults with $pr_{0\to1} = 0.5$, $pr_{1\to0} = 0.5$, $pr_{0\to0} = 0.5$, $pr_{1\to1} = 0.5$. Note that such transition probabilities violate the basic condition for SIFA. As a matter of fact, the leakage detection tests do not indicate any leakage in this case, as indicated in Fig. 9(a),(b). This case is also interesting due to the fact that *SIFA attack does not happen here even though faulty ciphertexts are obtained*. Since there is no leakage, we do not perform the SA for DL-FALAT. This is normal as the model, in this case, learns nothing and SA does not provide any meaningful result for such cases.

Example 8. The previous examples only considered cases, where the underlying cipher does not



(a) (b)

Figure 9: Leakage analysis for SIFA (on AES with redundancy) with fault transition probabilities $pr_{0\to1} = 0.5$, $pr_{1\to0} = 0.5$, $pr_{0\to0} = 0.5$, $pr_{1\to1} = 0.5$: a) Variation of leakage with ciphertext count; b) Point-wise (univariate) leakage profile for ALAFA; c) SA analysis result with DL-FALAT.



(a) (b)

Figure 10: Leakage analysis for SIFA (on TI-PRESENT with redundancy) with fault transition probabilities $pr_{0\to1} = 0$, $pr_{1\to0} = 1$, $pr_{0\to0} = 1$, $pr_{1\to1} = 0$: a) Variation of leakage with ciphertext count; b) Point-wise (univariate) leakage profile for ALAFA.

include any SCA countermeasure such as masking. As shown in [SJR+19], masking provides security against some classes of SIFA faults, especially while faults do not start corrupting intermediate computations of S-Boxes [SJR+19]. The proposed testing methodology is able to evaluate such intricate defence mechanisms. As evidence to this, we inject stuck-at-0 faults at one of the shares of a 3-share threshold implementation of PRESENT (integrated with the redundancy-based fault detection as the previous example). As indicated in Fig. 10, an attack does not take place in this case. Note that, the fault is biased in this case, and one can also obtain both correct and faulty ciphertexts. However, the attack is still prevented. This example clearly shows why just considering implementations having 100% error-correction coverage as SIFA secure is a conservative rule, and an assessment test like ours is essential. Later in the case studies, we shall show further test cases of SIFA, including test cases on two recently proposed SIFA countermeasures.

4.3 Generalized Interference Test – Compare with Uniformly Random

While the interference tests along with the preprocessing step are capable of handling the typical FA countermeasures on ciphers, there exist other situations which are not explicitly related with leaking the key. In other words, information leakage may not always involve the secret key. For example, there exist cryptographic primitives and protocols whose security strongly depends on the availability of uniformly random bit sequences. Any deviation from uniform randomness may lead to catastrophic consequences. An adversary may try to violate this condition by means of faults. As a concrete instance of such a fault, one may consider the instruction-skip fault model on software implementations. Instruction-skip faults, due to their high repeatability are ideal candidates for causing such randomness violations. For hardware implementations, such a fault may target the random number generator architecture. One such example is described in [JPCM17] using Hardware Trojan Horses (HTH). One should note that such randomness violations cannot be captured directly by the proposed interference test as the random strings are independent of the key.

In this subsection, we propose a simple solution which enhances the power of the interference

Algorithm 6 *TEST-INTERFERENCE-GENERALIZED*

Input: Protected Cipher $\mathscr{C}$, Fault value f_1, f_2 Target Observable $\mathscr{O}$, Simulation counter S
Output: Yes/No

$\mathscr{T}_{fm} := \emptyset$;
$p := GEN_{PT}()$
$k := GEN_{KEY}()$
$\mathscr{O}^c :=$ `Simulate`$(\mathscr{C}, p, k,$ NULL$)$
$\mathscr{O}^{fm} :=$ `Simulate`$(\mathscr{C}, p, k, f_m)$ **for** $m \in \{1,2\}$
if $(\mathscr{O}^c != \mathscr{O}^{fm})$ **then**
  for $i \leq S$ **do**
    $\mathscr{T}_{fm} := \mathscr{T}_{fm} \cup$ `Simulate`$(\mathscr{C}, p, k, f_m)$
  end for
  for $i \leq S$ **do**
    $\mathscr{U} := \mathscr{U} \cup GEN_{UNIFORM}()$
  end for
  if $(TEST(\mathscr{T}_{fm}, \mathscr{U}))$ **then** ▷ $TEST$ can be performed with the DL-based approach.
    if $(\mathscr{O} = g(\mathscr{K}))$ **then** ▷ If $\mathscr{O}$ is a function (g) of key.
      if $(f_1 != f_2)$ **then**
        Return `TEST-INTERFERENCE-FAULT`$(\mathscr{C}, p, f_1, f_2, S)$ ▷ Alternatively, `DL-TEST-INTERFERENCE-FAULT`.
      else
        $k_1 := GEN_{KEY}()$
        $k_2 := GEN_{KEY}()$
        Return `TEST-INTERFERENCE-KEY`$(\mathscr{C}, p, f, k_1, k_2, S)$ ▷ $f_1 = f_2 = f$ ▷ Alternatively, `DL-TEST-INTERFERENCE-KEY`.
      end if
    else
      Return Yes
    end if
  else
    Return No
  end if
else
  Return No
end if

test and makes it applicable for the situations where leakage does not directly involve the secret key. Before stating the test explicitly, we first enhance our definition of observables. *An observable $\mathscr{O}$ in the present context is a set of variables which are either input or output to a cryptographic module.* Note that in this definition we do not consider the trivial public inputs like plaintexts as observables, because they are usually controllable by an adversary. Examples of observables include the ciphertexts and the key and mask inputs to a crypto-core.

The enhancement to the non-interference test we are going propose now is based on a simple principle – *if the distribution assumed by an observable changes due to fault injection, then there is a chance that it may be exploitable.* Although this principle is slightly restricted in the sense that the caution raised might be a false positive, it has very low chances of resulting in a false negative. The detailed description is presented in Algorithm. 6. In order to keep the enhancement in sync with the entire test flow, we present all the logical steps to be followed by an evaluator during the evaluation. The inputs comprise two fault values f_1 and f_2 as well as the observable $\mathscr{O}$ decided by the evaluator according to the definition stated in the last paragraph. Note that the target of the fault, in this case, is not restricted within the protected cipher module but also incorporates the supporting modules such as randomness generation logic and input delivery logic. At the very first step, the algorithm simulates at least one of the fault events and checks if it alters the output. If there is no output alteration, the fault is considered as "non-leaking"[1]. In the other case, we repeat the fault simulation S times for a fixed key and plaintext. The next step is to perform the $TEST$ (or the DL extension of it) operation between this simulation trace $\mathscr{T}_{f_m}$ and a uniformly random distribution $\mathscr{U}$. We call this test as the *fixed-vs-random* test for FA leakage assessment. The intuition behind this test is that *if the fault event results in randomizing the outcome of the target observable $\mathscr{O}$, then no information can be extracted from it even by the attacker. Also, randomization of the observable will not be able to cause events which may result in attacks due to randomness loss (for example, repetition of a nonce, or a non-uniform mask for SCA resistance.).* In contrary, deviation from randomness may directly indicate chances of potential attacks.

[1]However, it may happen that a typical fault value being simulated remains “ineffective”. Hence we recommend performing this first step for both of the fault values provided.

An affirmative answer from the $TEST$ module indicates a potentially suspicious event. The next step of the attacker is to validate if the observable under consideration is dependent on the key or not. In case of a key-dependent observable, one should run either Algorithm. 1 or Algorithm. 2 (or their DL counterparts). The choice of test can be resolved by checking if the two fault values provided are equal or not. *Although in most of the situations Algorithm. 1 or Algorithm. 2 can be used interchangeably in certain cases (some of the instruction-skip based faults or for control faults) only Algorithm. 2 becomes applicable as only one fault value is possible.*[1] We utilize this fact for deciding the appropriate interference test and pinpoint the potential vulnerability within an implementation. To illustrate the efficacy of the generalized interference test, we present the following example:

Example 9. Let us consider an SCA resistant AES implementation which expects a fresh random mask of 128 bits for every execution. The SCA security strongly depends on the uniform random distribution of the mask. Without loss of generality, we assume a software implementation in this case, along with an instruction-skip fault model. For a target architecture having 32-bit bus width, the 128-bit mask is supplied to the AES module in chunks of 32-bits as shown in Listing. 3. In this pseudo code, the mask values are assumed to originate from memory locations M11, M12, M13, M14. The observable $\mathcal{O}$, in this case, is the mask register of the AES. During the execution of the masked AES, they are supposed to be copied to the internal registers, which we assume to contain some constant initial value (this is a reasonable assumption as most of the cases we may have some precharge logic implemented for embedded microprocessors). Now, an adversary may skip one or multiple of these instructions causing the mask to remain fixed for all of the executions. It is worth mentioning that there can be several other ways of affecting the randomness of the mask. For the sake of simplicity, here we mention this procedure.

The procedure described in Algorithm. 6 can identify this loss of randomness if the observable is set to the mask input of the AES module. In this case, the $TEST$ function detects a deviation from the uniform randomness. Note that, there is no point of running `TEST-INTERFERENCE-KEY` or `TEST-INTERFERENCE-FAULT` in this case as the mask does not vary with the key. However, if the target observable is key dependent, the `TEST-INTERFERENCE-KEY` or `TEST-INTERFERENCE-FAULT` should be called for assuring the potency of the leakage as well as to pinpoint if the countermeasure is erroneous or not. To be precise, in this specific example, the skip event indirectly eases the key leakage through the power side channel. The result of the leakage assessment test is presented in Fig. 11 for ALAFA, and in Fig. 12 for DL-FALAT. Without loss of generality, here we consider a software implementation targeting a 32-bit ARM-Cortex platform. The first 32-bit data-transfer is skipped. It can be seen in Fig. 11 that the leakage profile indicates a difference from uniformity for the first 4 bytes of the mask (the t-test is performed at each byte location of the observable here). Very similar results were derived for DL-FALAT (Fig. 12(a)). Interestingly, the SA plot (Fig. 12(b)) too, clearly indicates the 4 leaking bytes in the observable with relatively higher sensitivity values. The reason behind some of the unwanted points getting substantial sensitivity values can be explained by the fact that the analysis here has been performed with minimum possible amount of data, for which we have already obtained reasonably good leakage. A gradual increase in the dataset would make the sensitivity values corresponding to non-leaky bytes close to zero.

5 Fault Simulation on Hardware and Software Implementations

The proposed test framework simulates the faults to evaluate the countermeasures. Although simulation of data-flow or control faults are rather straightforward on an algorithmic or high-level

[1] Further details on instruction-skip faults are provided in the Appendix A.

Listing 3: Mask Derandomization Idea

```
mov reg1, <M11>   <= Skipped
mov reg2, <M12>
mov reg3, <M13>
mov reg4, <M14>
```

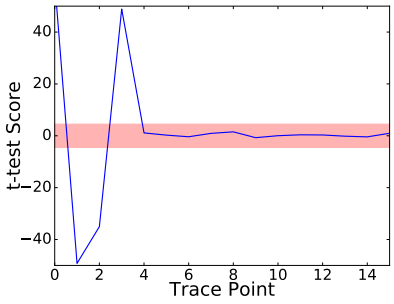
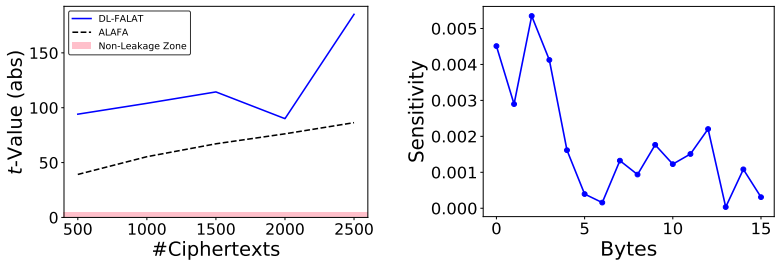


Figure 11: ALAFA results: Derandomization of mask.



(a) (b)

Figure 12: DL-FALAT result for the mask derandomization example: a) Variation of leakage with ciphertext count; b) SA plot.

code description,[1] it is a little more challenging on low-level codes or hardware modules. One of the challenging situations in this regard is the simulation of instruction-level faults. Especially, emulating skips or register/memory value corruptions at specific clock cycles or loop iterations requires dedicated tools which are not always available for most of the architectures. While one straightforward option could be to unroll loops, this is not a scalable option for every situation. Also, the simulation of certain control faults become challenging. The other interesting practical situation takes place, while one has to simulate faults on hardware design. The design can be at different levels of abstractions, such as Register Transfer Level (RTL) or gate-level netlist, or even a mix of them. Fault simulation on hardware designs is a well-studied topic, and there exists several open-source [LH96] or commercial tools dedicated for this purpose. However, fault simulation in the context of FAs may become challenging as we are not only talking about single-bit stuck-at type faults in these cases. Moreover, most of the fault models used in FA are transient in nature, whereas the conventional logic testing community models permanent faults.

Observing the challenges outlined in the previous paragraph, we decide to construct dedicated fault induction methodologies both for low-level software, as well as hardware designs. As a side contribution in this work, we present a generic and easy-to-use instruction-skip simulator based on GDB tool, and a hardware fault simulation strategy using state-of-the-art circuit design suites such as Synopsys/Xilinx-Vivado. The detailed description of both fault-simulation strategies are presented in Appendix A and Appendix. B, respectively. In the next section, we present detailed case studies establishing the usefulness of our proposed test methodology.

6 Case Studies

In this section, we present further experimental evidence to establish the efficacy of the proposed framework. As already outlined in the introduction, countermeasures can be realized either at algorithm-level or at the implementation-level, such as redundant instructions or gate-level

[1] Even if the program is obfuscated or not well-understood one can blindly select a program variable and change its value.

Table 1: Summary of Results

Countermeasures		1-byte fault	2-equal fault	Single Inst. Skip	Multi Inst. Skip	Skip-based Control Fault	SIFA Faults (Biased Faults)	SIFA Fault (Gate Input)
Algo. Level	Simple time/space redundancy[+]	Secure	Insecure	Secure	Secure	Secure	Insecure	Insecure
	1-bit parity [GMJK15] (information redundancy)	Insecure	–	Secure	Secure	Secure	Insecure	Insecure
	Infective [GST12] (without noise)	Insecure	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [GST12] (with noise)	Insecure	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [WLD+16] (RIMBEN)	Insecure	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [TBM14]	Secure	Insecure	Secure	Secure	Insecure	Insecure	Insecure
	Infective [GSSC15]	Insecure	Insecure	Insecure	Insecure	–	Insecure	Insecure
Inst. Level	Idempotent Inst. [MHER14]	Secure	–	Secure	Insecure	–	Insecure	Insecure
SCA+FA Combined	Masking [PMK+11]+ Classical FA Countermeasure	Secure	Insecure	Secure	Secure	–	Secure	Insecure
SIFA Counter-measure	AntiSIFA [SJR+19][**]	Secure	Secure	Secure	Secure	–	Secure	Secure
	Impeccable Circuits II [SRM19][**]	Secure	Secure	Secure	Secure	–	Secure	Secure
Security Module	SHE Hardware [she11]	–	–	–	Insecure for faults in data transfer	–	–	–

[+] Insecure against *paired* faults (one fault at datapath, and another at the comparison point of correct and faulty computation. Also vulnerable against combined FA-SCA attack.
[**] Secure upto a predefined security order

checkpoints [SRM19]. Further, algorithm-level countermeasures can be of several types, the two most prominent categories being the detection-based and infection-based countermeasures. In our test suite, we choose one representative from each countermeasure category. The algorithm-level countermeasure we evaluate are: 1) two-way time redundancy with a redundancy check at the end; 2) 1-bit parity-based countermeasure [GMJK15]; 3) infective countermeasure due to [GST12]; 4) infective countermeasure due to [TBM14]; 5) infective countermeasure due to [GSSC15]; and, 6) Benes network-based infective countermeasure RIMBEN [WLD+16]. The infective countermeasures, except the one from [GSSC15] and [TBM14], has already been illustrated in Sec 3. Also, the parity-based detection countermeasure has been used for illustrating Algorithm 4.

There exist several flavours of instruction-level countermeasures in the literature [PCM17, YGS+16, MHER14]. The main idea of this countermeasure class is to incorporate redundancies at the level of instructions so that an error can be either discovered or undone. Here we select the one proposed in [MHER14]. The main idea of this countermeasure is to replace most of the instructions in a program by equivalent sequences of so-called *idempotent* instructions. By the term *idempotent* we refer to a class of instructions which have the same effect on the program state while executed once or several times. As shown in [MHER14] and [PCM17], most of the standard ISAs contains instructions which are either idempotent or can be replaced by sequences of idempotent instructions.

One of the major contributions of this work is evaluating the state-of-the-countermeasures against SIFA attacks. We have already illustrated the vulnerabilities of some of them against SIFA and have also shown that masking provides limited protection against this class of attacks. However, masking alone is not sufficient, and in this section, we shall present an example where the leakage is identified in the case of masked countermeasures with redundancy-based fault detection. Very recently, multiple countermeasures have been proposed for countering SIFA. The common feature of them is the use of a fine-grained error-correction mechanism. In order to validate the security claims made by these countermeasure class, we evaluate the countermeasures from [SJR+19] and [SRM19]. The case study on [SRM19] is also interesting from a very different perspective. Here we evaluate an open-source hardware code of the countermeasure without any

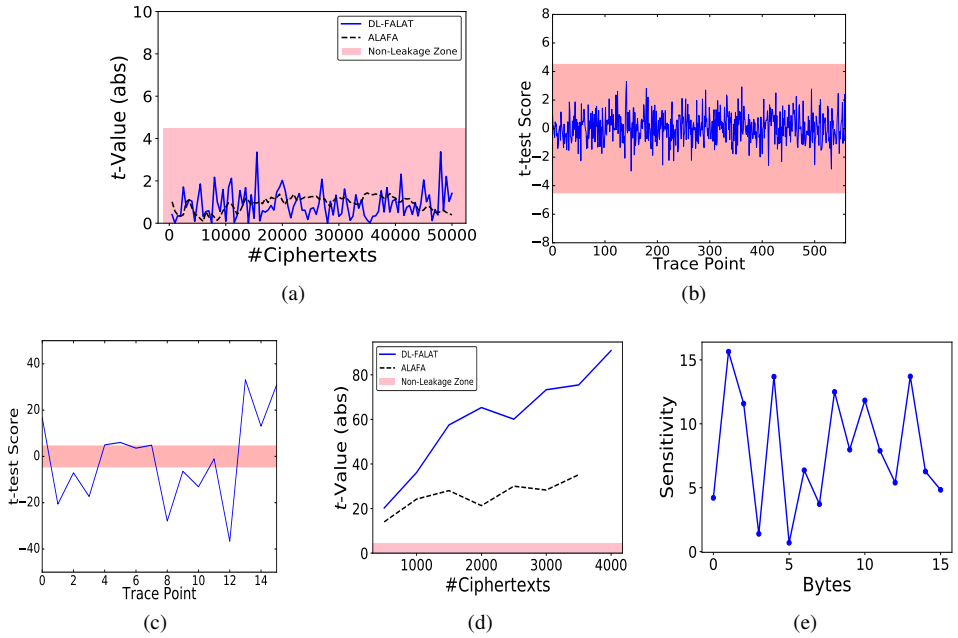


Figure 13: a) Infection countermeasure [GST12] with single-byte fault: DL-FALAT and ALAFA leakage profile with varying ciphertext count; b) Infection countermeasure [TBM14] with single-byte fault: point-wise leakage profile from ALAFA ($d = 3$); c) Infection countermeasure [TBM14] instruction-skip based loop-abort: point-wise leakage profile from ALAFA ($d = 1$); d) Infection countermeasure [TBM14] instruction-skip based loop-abort: DL-FALAT and ALAFA leakage profile with varying ciphertext count; e) Infection countermeasure [TBM14] instruction-skip based loop-abort: SA from DL-FALAT.

detailed understanding of it. As a byproduct of this analysis, we present a generic strategy for simulating faults on hardware. Our approach can be found in Appendix. B.

In the context of faults, we evaluate with single-byte-faults as well as two equal-valued single-byte faults at redundant branches wherever applicable. As instruction-level faults, we simulate single and multiple consecutive skips. Certain control faults, such as loop abort are also simulated by means of instruction-skips. The SIFA faults are also simulated at the bit-level (and multi-bit) granularity. For the sake of proper experimentation, we simulate all the faults at different locations within the protected cipher. However, the following discussion will specifically focus only on the interesting cases obtained during our experimentation. While evaluating countermeasures, the observables are the ciphertexts or the ciphertext differentials.

From a slightly different context, we also try to emphasize that implementing a reasonably secure countermeasure does not always guarantee sound security. *An attacker may exploit several simple architectural factors of an overall implementation to weaken the countermeasures, even without attacking the countermeasure itself.* One potential instance of such attacks has already been outlined in Example 9. The final example of ours will be on a more realistic scenario based on a recently proposed automotive security standard called SHE. The DL-FALAT framework, with its enhanced definition of observables, can handle such full-scale implementations.

A result summary of DL-FALAT for these case studies is provided in Table. 1. In particular, we observe that even without utilizing any significant structural details of the countermeasure algorithms, we can identify flawed schemes efficiently.

6.1 Algorithm-Level Countermeasure

Detection Countermeasures: The first example of ours utilizes simple two-way redundancy for error-detection. While considered under one-byte fault model, this countermeasure always returns

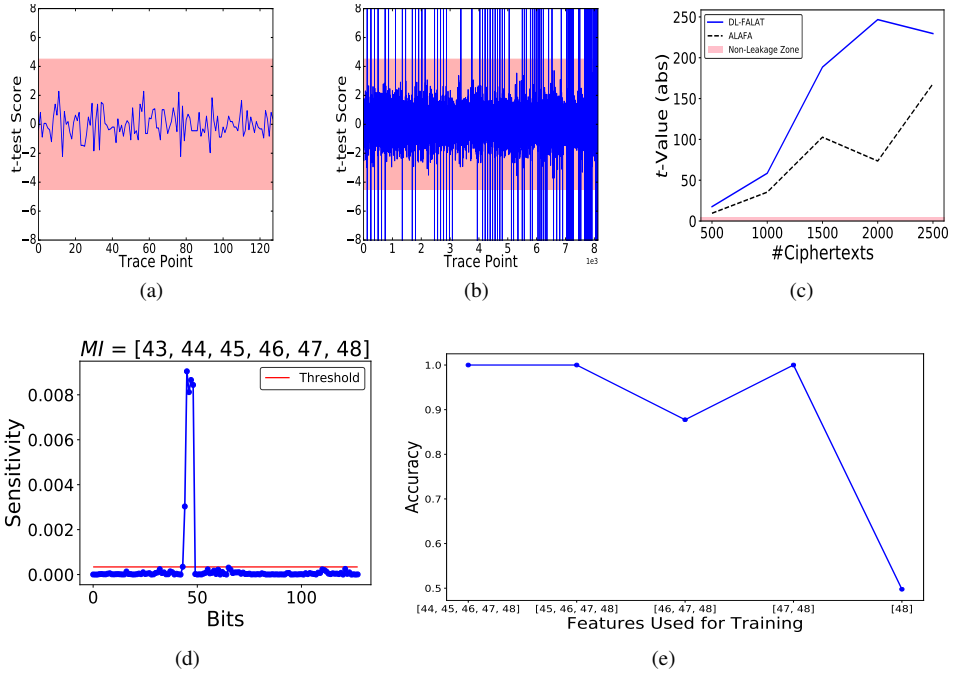


Figure 14: Infection countermeasure [GSSC15] with single-byte fault a) point-wise leakage profile from ALAFA ($d = 1$); b) point-wise leakage profile from ALAFA ($d = 2$); c) Variation of leakage with ciphertext count for DL-FALAT and ALAFA; d) SA for the DL-FALAT leakage for one iteration of iterative leakage interpretation; f) Analysis of the MI set in one iteration of leakage interpretation.

$\perp$ as every fault gets captured. The constant output $\perp$ is indistinguishable even if we consider two different fault values (f_1, f_2) and hence this countermeasure does not leak through the ciphertexts for byte faults. In contrast, for two equal valued faults at both computational branches, the faulty ciphertexts are outputted, and univariate leakage can be observed. The single-byte fault can also be generated by means of an instruction-skip. For example, in our experiments, we generated one such skip event by corrupting the key XOR-ing operation (ref. Listing. 5 in Appendix. A). However, several other instructions also result in a similar corruption as observed in our experiments[1].

Next, we consider the parity-based countermeasure already referred to in Example. 6. The countermeasure is bypassed for 50% of the byte faults having even parity and hence declared insecure. It is worth mentioning that the equal double fault attack is not directly applicable in this context as there is no explicit redundant computation. A pictorial representation of this leakage can be found in Fig. 5 and 6. However, one should note that although the leakage is observed in this case, for a well-formed code based redundancy, finding out a so-called leaky fault is significantly rare. Hence, even if there will be leakage for most of the code-based countermeasures, the exploitability depends on the rarity of the leaky faults, in general.

Infective Countermeasures: In Sec. 3, we have already presented results on the infective countermeasure proposed in [GST12]. The present example of ours considers the infective countermeasure proposed by Tupsamundre et al. at CHES 2014 [TBM14], which is an improvement over [GST12]. The main difference is that if a single/multi-byte data corruption happens in any of the cipher redundant or dummy rounds, the protected cipher is supposed to output a random string (instead of a masked intermediate state). For our experiments, in this case, we choose the number of dummy rounds to be 30 (without loss of generality). The countermeasure is first tested for single-byte fault

[1] A univariate leakage can be observed, if along with the byte fault an instruction-skip based control fault is utilized, to corrupt the outcome of the XOR operation performing the check operation at the end. However, we comment that the injection of such paired faults is challenging in most of the practical situations.

model. We simulate the single-byte faults both at a high-level and also by using instruction-skips. As it can be seen in Fig. 13 (a), no leakage is observed in this case, both by DL-FALAT and ALAFA. Fig. 13 (b) also presents the point-wise leakage from ALAFA (with $d = 3$) for completeness purpose. Quite evidently, SA for DL-FALAT cannot be performed in this case as there is no leakage. The next interesting observation is due to a control fault. It is found that faulting a variable which controls the execution sequence of redundant, dummy, and cipher rounds, has a malicious effect. An instruction-skip may corrupt the update operation of this variable during a loop iteration. DL-FALAT (and ALAFA), in this case, indicates a univariate information leakage as shown in Fig. 13(c),(d),(e). A careful investigation of this leaky event reveals that in several cases the cipher outputs the input of the 10-th round instead of a random string, thus leading to an attack. This phenomenon can be explained by the fact that *a fault in the abovementioned control variable somewhere near the last few rounds make the protected cipher misinterpret a cipher round as a redundant round. As a result, the last cipher round is aborted, and the ciphertext register is left with the output of the* 9*-th round.* In [PCM17] a similar attack was mentioned which was found by manual inspection. One should also note that the leakage, in this case, is noisy due to the presence of dummy rounds, where the noise is manifested in terms of random ciphertexts. However, ALAFA, in this case, was able to detect the leakage. The reason is that the leakage is univariate in this case, and the noise probability due to dummy rounds is significantly low (around 10%) for such loop-abort faults. As shown in Fig. 13(b), DL-FALAT still outperforms ALAFA here if compared based on the ciphertext count. Fig. 13(c) shows the results of SA with DL-FALAT. Further analysis of SA indicates a univariate leakage in this case. However, we do not present that result here as it is very similar to the parity-based countermeasure example.

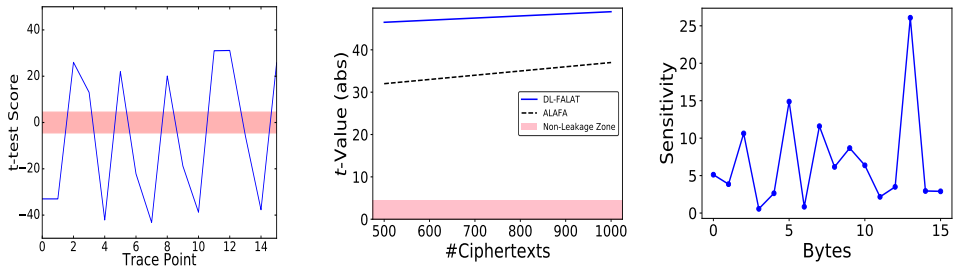
The final example of ours in the infective category is due to [GSSC15], which utilizes an infection function comprising a deterministic linear diffusion function followed by a randomized nonlinear mixing function. The multivariate case ($d = 2$) shows a significant amount of leakage even under single-byte fault model (Fig. 14(b)) with ALAFA[1]. In the case of DL-FALAT, we do not need to tell the analysis order to the tool, and the tool itself identifies a multivariate leakage in this case. Being interesting, here we show results from one iteration of the leakage interpretation experiment. The leakage interpretation step, in this case, can detect leakage point pairs by its own as expected. As it can be seen in Fig. 14(d) two consecutive points attain almost the same sensitivity values. Multiple such pairs get captured in one *MI* set during the first iteration of the interpretation experiment. We also found that removing all features included in one *MI* set readily exposes another set of leakage points without any increment in the dataset size. This fact indicates that the order of leakages might be the same throughout the ciphertext. Further, the results of an individual *MI* analysis is presented in Fig. 14(e), which clearly indicates that the leakage is multivariate. *To summarize, infective countermeasures considered in this work fails to provide security against FAs.*

It is worth mentioning that most of the algorithmic countermeasures tested in this work are vulnerable against two equal faults in redundant branches. The reason is that with two equal faults, the countermeasure mechanisms get bypassed, and actual faulty ciphertexts directly reach the output causing univariate leakage. Finally, it is worth noting that the simple time/space redundancy countermeasure is vulnerable against a combined side-channel and fault attack [LRT12]. We believe that DL-FALAT, with its observables extended with side-channel traces, will be able to detect this class of attacks.

6.2 Instruction-Level Countermeasure

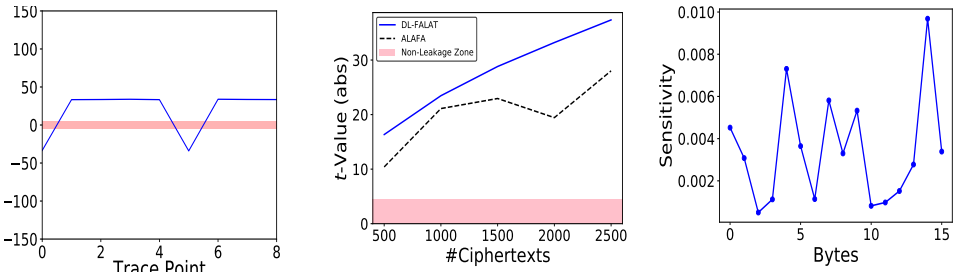
The instruction-level countermeasures against FAs mainly rely on the fact that an adversary can only skip a certain number of consecutive instructions at a time. This is a reasonable assumption for certain practical fault injection setups. However, it has been shown in [YGS+16] that for clock glitch-based injections, one single glitch may affect multiple instructions which are present in the

[1] Note that, for this countermeasure, leakage has been observed while the ciphertexts were considered bit-wise.



(a) (b) (c)

Figure 15: Instruction-level countermeasure with duplicate idempotent instructions [MHER14]. Two consecutive skips expose univariate leakage. a) Leakage profile for ALAFA ($d = 1$); b) Leakge profile of DL-FALAT and ALAFA wrt. ciphertext count. c) SA results from DL-FALAT.



(a) (b) (c)

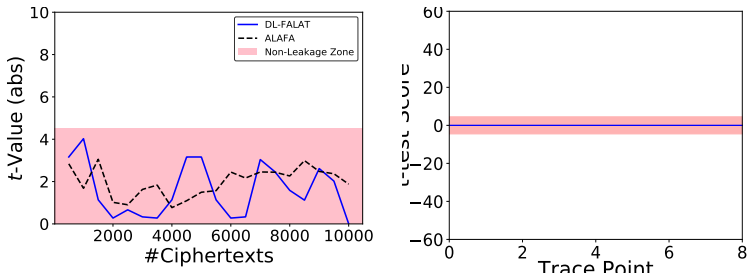
Figure 16: SIFA attack on masking: a) Leakage with ALAFA; b) Variation of t-values with ciphertext count for DL-FALAT and ALAFA; c) DL-FALAT SA annalysis.

processor pipeline during the glitch event. Such an observation necessitates testing for so-called higher-order fault injections where multiple consecutive instructions are to be skipped at the same time. Our GDB-based fault simulator easily simulates such multiple fault scenarios.

To test the applicability of DL-FALAT for instruction-level countermeasures, we implemented the scheme proposed in [MHER14] for an AES implementation without any algorithm-level protection. Each idempotent instruction is duplicated once in our implementation. While performing a first-order fault injection (that is only single instruction-skip), we observed no leakage. However, significant leakage can be observed if two consecutive instructions are skipped simultaneously, as shown in Fig. 15(a), (b), (c). *Note that while testing the evaluator need not have any idea regarding how many times an idempotent instruction has been repeated.* But he can still observe the leakage by gradually increasing the number of skipped-instructions. It establishes the fact that even without any precise knowledge regarding the countermeasure implementation, DL-FALAT can certify an implementation concerning the power of an adversary.

6.3 Leakage Analysis of SIFA Countermeasures

Sec. 4.2 has already presented multiple examples, where an unprotected and a masked implementation was evaluated against SIFA faults. The masked implementation was found to be able to prevent certain restricted classes of SIFA faults. However, we would like to point out that masking can still be bypassed by SIFA if all the shares corresponding to a specific bit gets corrupted and biased by faults. An easy trick for doing this is to exploit the propagation of a fault through the non-linear parts of a cipher, such as S-Boxes, for example. It is found that even if a random fault corrupts only a single input share of an S-Box (corresponding to an actual unshared bit), the fault may be propagated to all the share computations corresponding to a specific output bit. Moreover, this propagation is typically data-dependent and depends upon the actual unshared value of a bit. Such conditional fault propagation may completely nullify the protective power of masking over SIFA. However, the fault must be injected at a gate input in this case, instead of a register storing



(a) (b)

Figure 17: Evaluating SIFA countermeasures: a) AntiSIFA [SJR+19] with single-bit fault (DL-FALAT); b) Point-wise leakage profile with ALAFA

the shares. Otherwise, the fault propagation would become too rapid, and the data dependency of the leakage may not be exploitable. There exist typical hardware and software implementations for which such fault injections are feasible, and in those cases, masking is not a proper preventive measure [DEG+18].

In order to see if our test can positively identify the aforementioned leakage scenario, we simulated such gate-level faults in the case of a 3-share TI implementation of PRESENT block cipher [PMK+11]. One way of implementing the the PRESENT S-Boxes in a shared manner is to partition the S-Box function $\mathscr{S} : 2^4 \mapsto 2^4$ in two sub-functions $\mathscr{G}_1$ and $\mathscr{G}_2$ (both having lower algebraic degree than that of $\mathscr{S}$) such that $\mathscr{S}(x) = \mathscr{G}_1 \circ \mathscr{G}_2(x)$. Next, both $\mathscr{G}_1$ and $\mathscr{G}_2$ are implemented with 3-share TI [PMK+11]. During implementation, a set of registers are put in the output of $\mathscr{G}_1$ which works as the input to $\mathscr{G}_2$. *It was found that corrupting one input share of $\mathscr{G}_2$, while all the shares of a specific output bit is being computed, creates the desired impact, and the entire masking scheme can be bypassed with SIFA.* The first experiment of ours simulate this fault and performs the test described in Algorithm 5. As shown in Fig. 16, the leakages are clearly visible.

In our next two experiments, we focus on two recently proposed SIFA countermeasures in [SJR+19] and [SRM19]. The first countermeasure, also called AntiSIFA, incorporates fine-grained error correction in a per-bit manner with a masked implementation of PRESENT. The error-correction is performed with majority voting and is implemented with redundancy to make it fault-tolerant. The original proposal presents an example implementation with single-bit error correction support. While tested with single-bit faults as described in the previous paragraph, we found that the countermeasure successfully prevents the SIFA attacks by outputting only correct ciphertexts which supports the claims made in the original paper (ref. Fig. 17). The final experiment in the context of SIFA attacks is on an open-source hardware implementation of so-called Impeccable-Circuits II [SRM19]. The main idea of this countermeasure is to throttle the negative impacts of the fault propagation by the introduction of special checkpoints within the circuit, as well as forcefully making some circuit paths independent of each other. Moreover, linear code-based (resp. majority voting based) error correction is incorporated to counter SIFA attacks. In the open-source hardware implementation, the countermeasure is implemented on a recently proposed lightweight block cipher CRAFT [BLMR19] [1].

The repository for the open-source implementation contains different versions of CRAFT implementation with various types of countermeasures, including bit error correction, bit error detection, and majority voting. Among them, we tested the bit error correction and majority voting based implementations in our experiments. The faults, in these cases, were directly simulated on the hardware implementations in an automated manner. Further details on the hardware fault simulation flow can be found in Appendix B.

In the experiments, we targeted the input registers and state-registers of different design modules by introducing different single-bit and multi-bit faults in them. However, our fault simulation

[1] https://github.com/emsec/ImpeccableCircuitsII

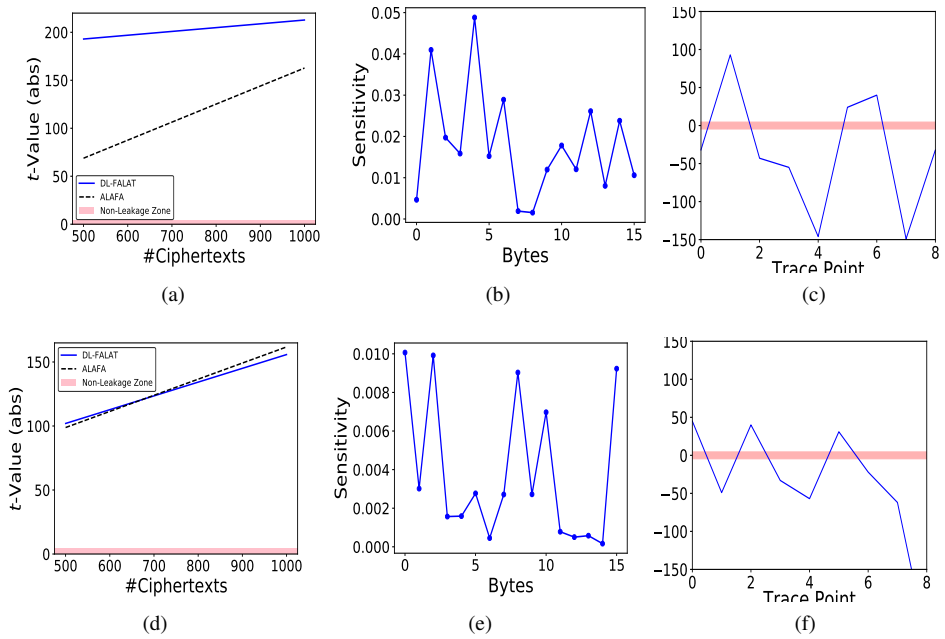


Figure 18: Evaluating SIFA countermeasures: a) Impeccable Circuit II [SRM19] with single bit error correction and 2-bit fault (DL-FALAT); b) Impeccable Circuit II with single bit error correction and 2-bit fault (SA); c) Impeccable Circuit II with single bit error correction and 2-bit fault (Point-wise leakage profile with ALAFA); d) Impeccable Circuit II with 2-bit error correction and 3-bit fault (DL-FALAT); e) Impeccable Circuit II with single bit error correction and 2-bit fault (SA); f) Impeccable Circuit II with single bit error correction and 2-bit fault (Point-wise leakage profile with ALAFA)

setup can be scaled easily for gate-level fault injections as well. In this section, we only mention the results with stuck-at-0 (resp. stuck-at-1) fault models, as they are one of the most prominent register fault models for SIFA. In case of the code based error-correcting implementations (3-way and 7-way redundancy), only one instance of the cipher was present in the implementations, along with the correction modules. However, in the majority voting- based implementations, there were three cipher instances present in the top-level implementations (1-bit error correction). We targeted one of those three instances as fault injection targets.

General observations of these experiments can be summarised as follows. In case of single-bit error correction implementation (3-way redundancy), a single-bit error was corrected in all cases (the result is very similar to the one of [SJR+19], and we do not repeat it here). However, this implementation was not successful in correcting many of the faults when two-bit stuck-at-0 faults were injected (ref. Fig. 18(a),(b),(c)). For implementation with two-bit correction (7-way redundancy), all two-bit errors were corrected successfully, but the implementation failed to correct the majority of the faults when three-bit stuck-at-0 faults were injected (ref. Fig. 18(d),(e),(f)). *Such observations clearly show that the countermeasures work fine on the fault models they were designed for.* However, it is worth mentioning that in the case of their majority voting implementation, if an attacker is also able to corrupt the voting circuit along with the normal state-fault injection, the countermeasures can still be bypassed. The fault tolerance of the voting circuit becomes really crucial in such cases. The current implementation under consideration does not take this into account (unlike the countermeasure in [SJR+19]) and, hence such multiple fault attacks are possible in these cases.

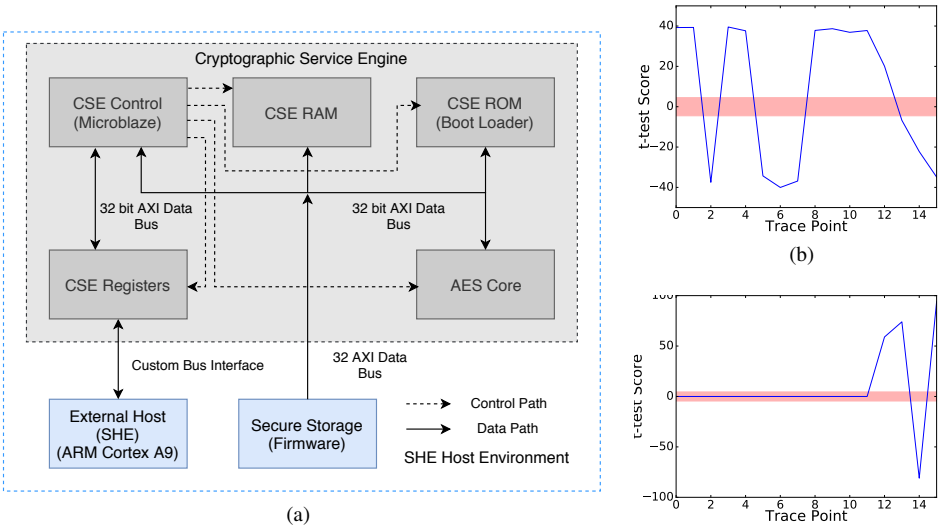


Figure 19: a) SHE Prototype: Basic Architecture; ALAFA leakage profile: SHE design; b) $TEST$ outcome while compared with uniformly random distribution; c) $TEST$ for two different keys k_1 and k_2.

6.4 Leakage Beyond the Countermeasures

This subsection aims to establish the general applicability of DL-FALAT (in the context of the Algorithm 6). A common trend in the security community is to evaluate the security of a protected block based on certain assumptions. For example, it is assumed that the mask for an SCA resistant implementation is always uniformly random. However, for an end-to-end system, the practical validity of these assumptions are often undermined. Here we show that this may have disastrous consequences.

To evaluate our claims, we consider a recently proposed standard for automotive security called SHE. SHE standard recommends a hardware security module (HSM) for automotive electronic control units (ECU), which primarily includes an AES block for encryption and authentication support, as well as a True Random Number Generator (TRNG). The services provided are the secure boot, encryption with different keys, and authentication with CMAC.

There exist commercial microprocessors from vendors like NXP, and Fujitsu, which include dedicated blocks implementing SHE. In such implementations (which are often referred to as Cryptographic Service Engine (CSE)), the HSM is kept almost isolated from the rest of the components. It is provided with Hardware AES, private ROM, RAM and configuration registers. The master processor of the host ECU can access the HSM only through the configuration registers. In addition to this, there is another external interface connected to the secure storage, which is tamper-proof and inaccessible to the user in a normal mode of operation. This secure storage contains the firmware(s) (which needs to be verified, securely booted, and updated if required), and the secret keys used by the ECU. All the functionalities of the HSM are controlled by a core engine which is referred to as CSE core. The purpose of this block is to execute the firmware for CSE which implements certain cryptographic protocols by using the hardware primitives provided. One reasonable approach for realizing this core is to utilize a small 32-bit processor.

In order to verify the robustness of this architecture, we implemented it according to the specifications given in [she11, mem11]. The basic SHE architecture is depicted in Fig. 19(a). The entire prototype has been implemented on the ZedBoard Zynq-7000 platform. A MicroBlaze softcore processor-based module serves as the CSE controller (core). CSE RAM and CSE ROM have been realized using on-chip block memory available on the Zynq-7000 FPGA device. AES core and CSE registers are purely FPGA logic-based modules. All these modules are interconnected with the CSE controller module through a 32-bit AXI data bus. For the external host, we used an

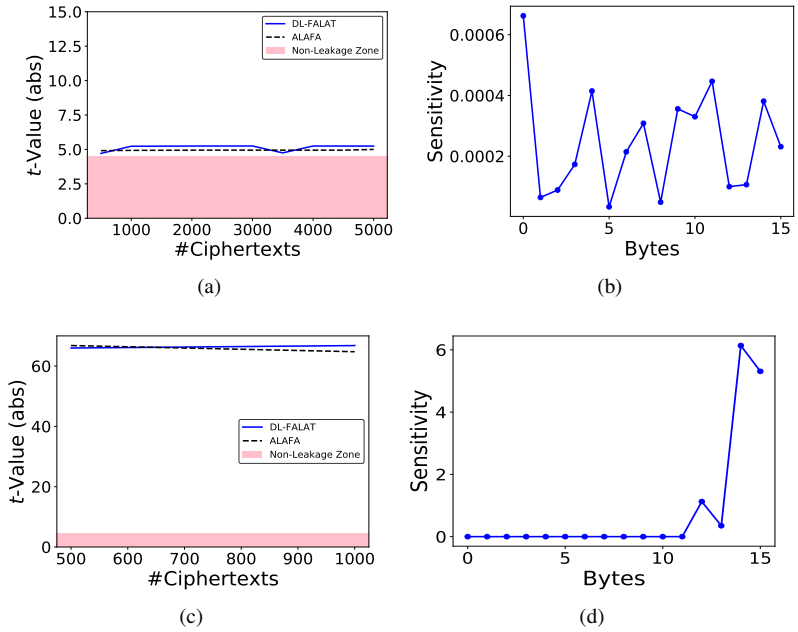


Figure 20: Evaluating SHE: a) Fixed-vs-Random with DL-FALAT; b) Fixed-vs-Random with DL-FALAT (SA); c) Fixed-vs-Fixed with DL-FALAT; d) Fixed-vs-Fixed with DL-FALAT (SA)

ARM Core available on Zynq-7000 device. Secure storage depicted in the diagram is also realized using on-chip block memory. All the data stored in these memory blocks are stored in 32-bit word aligned format to work with 32-bit AXI bus. The control logic (i.e., the firmware) of CSE controller is written in C language (with functions from Xilinx MicroBlaze C library) which executes on the embedded MicroBlaze softcore processor. All the control and data operations are performed by memory-mapped 32-bit AXI data transfer commands (`Xil_Out32()` and `Xil_In32()`). The 32-bit wide data bus is common for most applications in the automotive and embedded application area.

For the sake of simplicity, the main target of our verification here is the firmware code which can be targeted by an instruction- skip attacker[1]. The firmware implements several protocols like secure boot and CMAC. Each of their codes was evaluated. However, leakage can be observed for even simpler scenarios. As a concrete example, we consider the basic encryption support provided by CSE. The 128-bit secret key is to be copied from the RAM to the internal registers of AES in this case. The plaintext (and mask, if required) is also supplied in a similar fashion. However, an instruction-skip based leakage is observed here during the data transfer operation. More precisely, *the 32-bit bus architecture of our implementation allows the 128-bit key to be transferred to the AES core in chunks of 32-bits.* Therefore, a 128-bit transfer requires four consecutive calls to the 32-bit data transfer operation of MicroBlaze as shown in Listing. 4 (this operation, in turn, makes a call to the 32-bit data transfer instruction of MicroBlaze ISA having opcode `swi`). This part(s) of the code serves as our target spot during the transfer of the secret key. KEY_i, $i \in \{0,1,2,3\}$ represent the four 32 bit words of the key.

In our experiments, we skip one/more of these data transfer operations[2]. The observable here is the key register inside the AES core. At high-level this register serves as an input to the protected AES block and therefore qualifies as a potential observable. The enhanced DL-FALAT from Algorithm. 6 detects the presence of a key leakage in this case. Note that, since here we are evaluating an entire implementation, considering the Algorithm. 6 is the only viable option.

[1]It is worth mentioning that the AES hardware itself has also been tested. Here we have implemented simple redundancy to protect the AES core. However, in order to explain our claim here, we focus on a specific part of the firmware code.

[2]One reasonable assumption here is that the AES core is reset after each execution. So, the key register is supposed to contain an all 0 value at the beginning.

Further, the fault, in this case, can take only one value f as it is realized by skipping a data transfer operation. The leakage profiles for the first and second invocation of the $TEST$ (Algorithm. 6) are depicted in Fig. 19(b) and (c) and in Fig. 20, respectively. The randomness test in Fig. 19(b) (and Fig. 20 (a), (b)) indicates a randomness loss for the whole key as expected trivially. Due to the single valuation of the fault as well as the key dependency of the observable, the next test to be invoked is the `TEST-INTERFERENCE-KEY`. The leakage becomes more informative after this step is performed. One may observe that in Fig. 19(c) (and Fig. 20 (c), (d)) the leakage is indicated only for the last 4 bytes of the key. This is because, in our experiments, we skip first three consecutive data transfer instructions and thus set the corresponding 12 bytes to zeros. This happens for both k_1 and k_2 used in `TEST-INTERFERENCE-KEY`, and thus there is no information leakage at those 4 bytes.

Listing 4: Code Snippet for AXI Data Transfer

```
Xil_Out32(XPAR_AESCORE_0_S00_AXI_BASEADDR,KEY0);
Xil_Out32(XPAR_AESCORE_0_S00_AXI_BASEADDR+4,KEY1);
Xil_Out32(XPAR_AESCORE_0_S00_AXI_BASEADDR+8,KEY2);
Xil_Out32(XPAR_AESCORE_0_S00_AXI_BASEADDR+12,KEY3);
```

One important question here is the exploitability of the leakage. The DL-FALAT test itself cannot comment on that. However, for this specific example, we found the leakage exploitable. *Considering that fact that the AES registers are reset to zero after each execution, the aforementioned instruction-skip results in a scenario where a significant number of key bits are fixed to zero. Skipping three consecutive data transfer operations will set 96 key bits to zero, and only 32 bits of the original key will remain intact. Upon receiving the faulty ciphertexts, the adversary can run an exhaustive search of 32 bits and recover the unaltered 32 bits in the corrupted key. Repeating this attack three more times the rest of the key bits can also be recovered. The computational complexity of this attack is* 2^{32}.

One important technical point in the above experiment is to know how many instructions have to be skipped in order to omit three consecutive bus transfers. We found that the AXI bus transfer is implemented as a function call in Microblaze, which involves 3 instructions per data transfer operation. We found that corrupting any of these instructions result in a skip of data transfer operation at a high level. Corrupting three consecutive data transfers thus need 9 consecutive instructions to be corrupted. Although in our simulation platform, it is always feasible, questions may be raised regarding its practicality. We note that the advanced setups like laser-based injections are capable of skipping a large number of consecutive instructions. Moreover, given the fact that multiple instructions are usually present in the pipeline in a specific clock cycle [YGS+16], insertion of multiple glitches should be able to generate such multiple skip events. It is also worth mentioning that this observation of skipping several instructions is MicroBlaze specific and typically there exist other platforms for which 2-3 skips become sufficient. More precisely, this does not indicate any limitation of the DL-FALAT framework. One should also note that even skipping a single operation (which can be achieved by a single instruction-skip) can indicate leakage in this case. However, that leakage is not exploitable due to excessive computational complexity.

The attack outlined in the last paragraph exposes a fundamental weakness of the hardware-software co-design approach adopted for SHE implementation. The similar attack can be performed for weakening the mask bits leading to a trivial SCA attack. Fortunately, the DL-FALAT framework is able to indicate the chances of such unwanted leakages. *The presence of these vulnerabilities indicate that not only the core cryptographic primitives but the entire security firmware should take fault attacks into account.*

7 Discussion

Trivial Leakage from Unprotected Implementations: Throughout the paper, we have focused on the FA protected implementations of block ciphers. A natural question is whether the leakage assessment test is applicable for unprotected implementations as well. While the answer to this question is positive, there are some subtle issues in this context. First, let us explain why the leakage from an unprotected implementation would get detected by DL-FALAT. This is because two different fault value will always lead to two different ciphertexts in this case. Also, since there is no output randomization or muting due to the presence of faults, the faulty ciphertexts will be constants for fixed fault values. The situation clearly indicates that the leakage here is apparent, and this fact is completely in-sync with the theory developed. However, this observation is valid for any fault in the datapath, regardless of the round of injection. In other words, *although DL-FALAT can indicate leakage for an unprotected implementation, it cannot conclude whether the leakage is exploitable or not.* Given the fact that there exist automated tools for identifying exploitable faults on unprotected implementations [SMD18], this does not seem to be a serious issue. In fact, an evaluator having the knowledge of the cipher may first use such an exploitable fault identifier to figure out potentially dangerous faults, and may then use DL-FALAT on the protected implementation by simulating faults for those exploitable positions only. Such combined testing may greatly reduce the simulation and test effort.

Comments on the Generalized Test from Algorithm. 6: The main crux of the generalized interference test of Algorithm. 6 is to compare the faulty observable distribution with a uniform distribution. *However, detecting whether a distribution is indistinguishable from uniform is tricky, and* t*-test (with the multivariate extension) might not be sufficient here in all possible situations. Although DL is found to be substantially more powerful than the* t*-test in the present context, there is no concrete mathematical guarantee that it will always be able to distinguish a given string from a uniformly random one.* However, we claim that, in many situations, the aim of an attacker would be to de-randomize the observable as much as possible in order to make the attack easier. This, still, does not provide a general solution. One potential approach is to use some standard randomness test suite (such as NIST test [Rea01] or AIS test [KS11, YRM+16]) in this context. However, the case studies we have encountered here do not require any of them to detect the leakage, and hence, we keep any further study on this as a potential future work. Nevertheless, the original idea and structure of the proposed test remain the same even if we apply any such modification.

8 Conclusion

Security evaluation of a fault attack protected implementation is a problem of utmost practical importance. In this paper, we have proposed a simple Deep Learning-assisted leakage assessment test DL-FALAT, which can validate protected block cipher implementations as well as end-to-end hardware security modules. The test is not only suitable for filtering out malformed designs, but can also figure out the points of vulnerabilities. We have shown that how a variation of this test can be utilized for evaluating against a variety of SIFA faults. Furthermore, a generalization of the test to detect fault-induced leakages in so-called "non-cryptographic" components of a security module has also been presented and evaluated over an automotive security standard called SHE. As side contributions, a general strategy for instruction-level fault simulation has been proposed, which is easily extendable for most of the existing ISAs. A methodology for fault simulation in hardware designs has also been outlined and tested over an open-source implementation of a SIFA countermeasure. As concrete examples of applicability, several representative FA countermeasures, as well as a commercially used HSM, have been evaluated, for which DL-FALAT indicates non-trivial implementation issues. Evaluating DL-FALAT over a larger testbed is a potential future work. Extending the ideas for public-key implementations is another future direction of research.

References

[ADM+10] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *2010 IEEE 16th International On-Line Testing Symposium*, pages 235–239. IEEE, 2010.

[ADN+10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *International Conference on Smart Card Research and Advanced Applications*, pages 182–193. Springer, 2010.

[AWMN20] Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic fault diagnosis using verfi, 2020.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer, 2015.

[BKHL19] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A countermeasure against statistical ineffective fault analysis. *IACR Cryptology ePrint Archive*, 2019:515, 2019.

[BLMR19] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. *IACR Transactions on Symmetric Cryptology*, 2019(1):5–45, 2019.

[BM16] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *CHES*, pages 602–624. Springer, 2016.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525, Santa Barbara, USA, Aug 1997. Springer.

[CHM04] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference: Information theory and information flow. In *Workshop on Issues in the Theory of Security (WITS'04)*, 2004.

[CML+11] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and laser fault attacks onto a secure aes implementation on a sram-based fpga. *Journal of cryptology*, 24(2):247–268, 2011.

[DDRT12] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Hannes Gross, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked aes with fault countermeasures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 315–342. Springer, 2018.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. Sifa: exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 547–572, 2018.

[FCL+19] Jingyi Feng, Hua Chen, Yang Li, Zhi Peng Jiao, and Wei Xi. A framework for evaluation and analysis on infection countermeasures against fault attacks. *IEEE Transactions on Information Forensics and Security*, 2019.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GMJK15] Xiaofei Guo, Debdeep Mukhopadhyay, Chenglu Jin, and Ramesh Karri. Security analysis of concurrent error detection against differential fault analysis. *Journal of Cryptographic Engineering*, 5(3):153–169, Sep 2015.

[GSSC15] Shamit Ghosh, Dhiman Saha, Abhrajit Sengupta, and Dipanwita Roy Chowdhury. Preventing fault attacks using fault randomization with a case study on aes. In *Australasian conference on information security and privacy*, pages 343–355. Springer, 2015.

[GST12] B. Gierlichs, J. Schmidt, and M. Tunstall. Infective computation and dummy rounds: fault protection for block ciphers without check-before-output. In *LatinCrypt'12*, pages 305–321. Springer, 2012.

[HBZL19] Xiaolu Hou, Jakub Breier, Fuyuan Zhang, and Yang Liu. Fully automated differential fault analysis on software implementations of block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–29, 2019.

[HGG18] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled power analysis attacks using convolutional neural networks with domain knowledge. In *International Conference on Selected Areas in Cryptography*, pages 479–498. Springer, 2018.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JPCM17] Anju P Johnson, Sikhar Patranabis, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. Remote dynamic partial reconfiguration: A threat to internet-of-things and embedded security applications. *Microprocessors and Microsystems*, 52:131–144, 2017.

[KPH+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[KRH17] Punit Khanna, Chester Rebeiro, and Aritra Hazra. XFC: A framework for eXploitable Fault Characterization in block ciphers. In *DAC'18*, pages 8:1–8:6, Austin, USA, June 2017. IEEE.

[KS11] Wolfgang Killmann and Werner Schindler. *A proposal for: Functionality classes for random number generators, version 2.0*, 2011. [Online]. Available: `https://www.bsi.bund.de/EN/Home/homenode.htm`.

[LH96] Hyung Ki Lee and Dong Sam Ha. HOPE: An efficient parallel fault simulator for synchronous sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1048–1058, 1996.

[LRT12] Victor Lomne, Thomas Roche, and Adrian Thillard. On the need of randomness in fault attack countermeasures-application to aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 85–94. IEEE, 2012.

[MDP20] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. A comprehensive study of deep learning for side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 348–375, 2020.

[mem11] HIS memebers. SHE - secure hardware extension functional specification version1.1 (rev 439), 2011. www.automotive-his.de.

[MHER14] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[MRSS18] Amir Moradi, Bastian Richter, Tobias Schneider, and François-Xavier Standaert. Leakage detection with the x2-test. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 209–237, 2018.

[PCM17] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. Fault tolerant infective countermeasure for aes. *Journal of Hardware and Systems Security*, 1(1):3–17, 2017.

[Per19] Guilherme Perin. Deep learning model generalization in side-channel analysis. *IACR Cryptology ePrint Archive*, 2019:978, 2019.

[PHJ+18] S Picek, Annelie Heuser, A Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1), 2018.

[PMK+11] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-channel resistant crypto for less than 2,300 ge. *Journal of Cryptology*, 24(2):322–345, 2011.

[Rea01] Andrew Rukhin and et. al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-Allen and Hamilton Inc Mclean Va, 2001.

[she11] Using the cryptographic service engine (cse): An introduction to the cse module, 2011. http://cache.freescale.com/files/32bit/doc/app_note/AN4234.pdf.

[SJR+19] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborti, Shivam Bhasin, and Debdeep Mukhopadhyay. A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction. *IEEE Transactions on Information Forensics and Security*, 2019.

[SKP+19a] Sayandeep Saha, S Nishok Kumar, Sikhar Patranabis, Debdeep Mukhopadhyay, and Pallab Dasgupta. ALAFA: Automatic leakage assessment for fault attack countermeasures. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 136. ACM, 2019.

[SKP+19b] Sayandeep Saha, S Nishok Kumar, Sikhar Patranabis, Debdeep Mukhopadhyay, and Pallab Dasgupta. ALAFA: Automatic leakage assessment for fault attack countermeasures. In *DAC*, page 136. ACM, 2019.

[SLIO11] Kazuo Sakiyama, Yang Li, Mitsugu Iwamoto, and Kazuo Ohta. Information-theoretic approach to optimal differential fault analysis. *IEEE Transactions on Information Forensics and Security*, 7(1):109–120, 2011.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 495–513. Springer, 2015.

[SMD18] Sayandeep Saha, Debdeep Mukhopadhyay, and Pallab Dasgupta. ExpFault: an automated framework for exploitable fault characterization in block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 242–276, 2018.

[SRM19] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable circuits II. *IACR Cryptology ePrint Archive*, 2019:1369, 2019.

[Sta18] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In *International Conference on Smart Card Research and Advanced Applications*, pages 65–79. Springer, 2018.

[TBM14] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 93–111. Springer, 2014.

[Tim19] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 107–131, 2019.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP international workshop on information security theory and practices*, pages 224–233. Springer, 2011.

[WLD+16] Bo Wang, Leibo Liu, Chenchen Deng, Min Zhu, Shouyi Yin, Zhuoquan Zhou, and Shaojun Wei. Exploration of benes network in cryptographic processors: A random infection countermeasure for block ciphers against fault attacks. *IEEE Transactions on Information Forensics and Security*, 12(2):309–322, 2016.

[WMM19] Felix Wegener, Thorben Moos, and Amir Moradi. DL-LA: deep learning leakage assessment: A modern roadmap for SCA evaluations. *IACR Cryptology ePrint Archive*, 2019:505, 2019.

[YGS+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, 2016.

[YLMZ18] Guang Yang, Huizhong Li, Jingdian Ming, and Yongbin Zhou. Convolutional neural network based side-channel attacks in time-frequency representations. In *International Conference on Smart Card Research and Advanced Applications*, pages 1–17. Springer, 2018.

[YRM+16] Bohan Yang, Vladimir Rožić, Nele Mentens, Wim Dehaene, and Ingrid Verbauwhede. TOTAL: TRNG on-the-fly testing for attack detection using lightweight hardware. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 127–132. EDA Consortium, 2016.

[ZLZ+18] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 150–172, 2018.

A Simulation of Instruction-Skip Faults

As already pointed out in Sec 5, fault simulation for low-level software codes are fairly challenging. In this section, we elaborate a generic and easy-to-use methodology for simulating instruction-level faults. However, before describing the details of this tool-flow, let us motivate the reader why instruction level faults deserve special attention. Firstly, an instruction-level fault, such as instruction-skip is one of the most repeatable, easy-to-generate, and consistent fault models. Secondly, their occurrence at a lower level of abstraction explicitly captures certain fault cases which are difficult to simulate at a high level. For example, certain control faults such as loop-abort or change of control-flow may be realistically generated from skipping or modifying a certain set of instructions. Finally, the number of candidate fault sites in the case of instruction-skip attacks is significantly larger than that of the high-level nibble/byte (or even multiple bytes) fault sites. More specifically, each fault site at high-level may map to multiple instructions at the instruction-level. As a concrete example, let us consider the code in Listing. 5. The X86-64 assembly corresponding to line 5 of this code results in almost 30 instructions (some part provided in Listing. 6). In practice, one or more instructions from this assembly may be vulnerable. There may be cases where skipping multiple instructions simultaneously results in a desired faulty behavior at high-level. In nutshell, the attack space of instruction-level fault is more diverse than the high-level faults and has certain distinct properties. If there are q number of instructions corresponding to a high-level statement and the adversary has the power of simultaneous skipping t of them to create a desired faulty behavior, then all $\binom{q}{t}$ subsets should be considered for testing. The large attack space also provides a strong motivation for simulation as critical corner cases can be easily explored in this way.

In the context of leakage testing, one typical feature of instruction-skip faults[1] deserves special attention. Under a fixed plaintext and key the faults in certain control instructions always result in a single fault value f with instruction-skips. In other words, one cannot have two different fault values f_1 and f_2 to compare. This is a situation where an evaluator is bound to use Algorithm. 2. This fact has been addressed in Algorithm. 6 at line no. 15-20.

Listing 5: AddRoundKey of AES

```
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
state[j][i]^=
    RoundKey[round*Nb*4+i*Nb+j];
}
}
```

Listing 6: X86-64 assembly for line no. 5

```
movl  -8(%rbp), %eax
cltq
movl    -4(%rbp), %edx
movslq  %edx, %rdx
salq    $2, %rdx
addq    %rax, %rdx
leaq    state(%rip), %rax
addq    %rdx, %rax
:
:
addq    %rdx, %rax
movb    %cl, (%rax)
```

[1]For simplicity, we shall discuss about the instruction-skip faults only, which is the most prominent among instruction level fault models. However, it is worth mentioning that other situations such as register faults and instruction modifications can also be simulated easily by the simulation technique we are going to describe.

A.1 The GDB-based Fault Simulator

In this subsection, we introduce our automation for simulating instruction-skips. The simulator utilizes the GDB tool, which is one of the most common debugging support available. One great advantage of using GDB is that simulating for different platforms requires almost negligible changes to be made in the simulator. The availability of the high-level code (which is fairly reasonable in a verification environment) makes the simulation even easier as extra guidance can be provided to the GDB tool (e.g., in the form of breakpoints at high-level)[1]. Note that the availability of the high-level code does not imply that the countermeasure algorithm is well-understood for the evaluator. In the worst case, the code may be obfuscated. However, the algorithm-oblivious nature of the proposed test strategy still allows the verification to proceed.

Listing. 7 presents an example of how an instruction- skip event can be simulated using GDB. We refer to the code snippets already presented in Listing. 5 and 6 for X86-64 architecture. However, the same experiment can also be repeated for any embedded architecture like ATMega, MicroBlaze, or ARM. For the sake of explanation, in this case, we assume the availability of the high-level C code of Listing. 5. A breakpoint is set at line number 5 of this high-level code. The breakpoint is also conditioned to be encountered only when $i == 0$ and $j == 0$. Such conditional breakpoints allow us to create the instruction-skip faults at specific loop iterations. The skip itself is realized on the very first instruction of Listing. 6 by executing lines 6-10 in the GDB script of Listing. 3. The core idea here is to change the address stored in the program counter register to the next address. Note that, GDB also allows explicit modification of program counter value and even multiple instruction-skips and instruction-modifications can be implemented using this fact. Furthermore, explicit register modification feature also allows us to simulate register faults, as well as memory faults precisely. It is worth to mention that any instruction can be targeted in this way by moving the execution to the desired point with the `nexti` command provided by GDB.

The proposed instruction-skip simulator works by automatically creating such GDB scripts and executing them. It expects the position of a skip in terms of the function name and target loop counter value (if required) as inputs. However, in the general case, it can also start from the beginning of a program and simulate skips for every instruction encountered. It is worth mentioning that the power of the simulator is not limited to the instruction-skip faults only, and it can be easily extended for creating register/memory faults and even instruction modifications. One important point here is that instruction-level faults are often caused by one or more architectural features of the underlying processor. For example, in [YGS+16], it was shown that due to the presence of multiple instructions within the pipeline at a single clock cycle, one clock glitch is able to corrupt multiple instructions, which in turn results in bypassing certain instruction-level countermeasures. However, simulating such architectural events is extremely complex and often significantly slow to explore. Moreover, the aim of the present work is to evaluate implementations and algorithms and not the processor architectures. Given the fact that our GDB based fault simulator is capable of emulating the final effect of such complex micro-architectural events, it is well suited for the current purpose and should be applicable in several other such verification contexts.

Listing 7: GDB snippet for instruction-skip in Listing. 5, 6

```
break main
break 5
condition 2 (i == 0 && j == 0)
r
c
set $var1 = $instn_length($pc)
set $var2 = $pc + $v1
set $var3 = $instn_length($v2)
set $var4 = $pc + $v3 + $v1
jump *($var4)
```

[1] This is not a necessary condition, though.

B Fault Simulation on Hardware Modules

Simulating malicious faults on actual hardware designs is challenging from several aspects. One of the major issues is the coverage over a fault space of formidable size and variation. Hardware faults can affect one or multiple sequential or combinational gates within a design. The number of such components is certainly huge, and thus a complete coverage over all feasible fault models is impractical to expect.

Our take on this problem is, however, fairly simple. We keep ourselves restricted to the fact that a cipher design is under test. Conventional cipher designs use several structurally equivalent modules repeatedly, to perform the computation. For example, in the case of block ciphers, each round is almost identical. This fact provides a great advantage for rapidly shrinking the fault space. Strictly speaking, simulating fault for the last rounds of an implementation is somewhat sufficient. Nevertheless, to keep our simulation process generic, we allow our methodology to simulate any given fault simulation (not restricted to penultimate round faults only).

The second fact that we exploit is the common nature of faults in the context of FAs. Most of the fault models are transient and corrupts a few clock cycles, even in the most general scenario. We thus keep ourselves restricted in simulating those cases only, where faults are injected in at most two clock cycles. The next optimization trick is to consider the modular structure of a design, while described in some hardware description language (HDL). We strongly recommend keeping the hierarchical structure of the design intact during the synthesis. Next, faults are simulated only for the input signals of each module at one or two specified clock cycles according to a user-specified fault model. Only for some special modules, such as S-Boxes, we suggest going deeper and simulating faults for intermediate wires for fine-grained validation.

The next question is how to simulate any given fault model (staring from byte faults to stuck-at faults), over a specific set of wires. One way is to tailor the existing fault simulators for this purpose as it was done for a recently proposed tool VerFI [AWMN20]. *However, we propose an easier approach where we automatically stitch the design with a Hardware Trojan Horse (HTH). The Trojans designed for this purpose alter values of each bit to be corrupted as desired, based on a given control signal. As a concrete example, we one wants to simulate a stuck-at-0 fault to a specific bit b, a two-input AND gate is required, one of whose input would be the wire to be corrupted. If the other input is set to* 0*, it simulates a stuck-at-0 situation. Similarly, a stuck-at-1 situation can be created by means of an OR gate, and a bit-flip can be emulated with an XOR. Simulating multi-bit fault requires multiple such gates to be stitched at the desired locations within the circuit. This strategy is sufficiently generic for creating even internal wire faults within a module if required. Another great advantage is that no specific fault simulator tool is required. Instead, one can use any state-of-the-art circuit simulator for simulating the HTH-induced circuit.*

Our current realization of this methodology uses the Xilinx-Vivado simulator. The faults to be simulated are taken as user inputs. The injection clock cycle, bit-wise and module-wise location, and the fault model are required to specify the faults. As a first-cut prototype implementation, we simulate the faults one at a time. However, it is not difficult to parallelize as each fault instance results in a different circuit instance. In future, we shall focus further on speeding up the fault simulation process further using different approaches, such as emulating them on FPGAs.