J Cryptol (2023)36:19
https://doi.org/10.1007/s00145-023-09462-6

Journal of
CRYPTOLOGY

Research Article

Learn from Your Faults: Leakage Assessment in Fault Attacks Using Deep Learning*

Sayandeep Saha
School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore, Singapore
sayandeep.saha@ntu.edu.sg

Manaar Alam
Center for Cyber Security, New York University Abu Dhabi, Abu Dhabi, United Arab Emirates
alam.manaar@nyu.edu

Arnab Bag · Debdeep Mukhopadhyay · Pallab Dasgupta
Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India
arnabbag@iitkgp.ac.in
debdeep@iitkgp.ac.in
pallab@iitkgp.ac.in

Communicated by Franşois-Xavier Standaert

Received 10 March 2022 / Revised 20 March 2023 / Accepted 17 April 2023
Online publication 9 May 2023

Abstract. Generic vulnerability assessment of cipher implementations against Fault Attacks (FA) is a largely unexplored research area. Security assessment against FA is critical for FA countermeasures. On several occasions, countermeasures fail to fulfil their sole purpose of preventing FA due to flawed design or implementation. This paper proposes a generic, simulation-based, statistical yes/no experiment for evaluating fault-assisted information leakage based on the principle of *non-interference*. It builds on an initial idea called ALAFA that utilizes *t*-test and its higher-order variants for detecting leakage at different moments of ciphertext distributions. In this paper, we improve this idea with a Deep Learning (DL)-based leakage detection test. The DL-based detection test is not specific to only moment-based leakages. It thus can expose leakages in several cases where *t*-test-based technique demands a prohibitively large number of ciphertexts. Further, we present two generalizations of the leakage assessment experiment—one for evaluating against the statistical ineffective fault model and another for assessing fault-induced leakages originating from "non-cryptographic" peripheral components of a security module. Finally, we explore techniques for efficiently covering the fault space of a block cipher by exploiting logic-level and cipher-level fault equivalences. The efficacy of our proposals has been evaluated on a rich test suite of hardened implementations, including an open-source Statistical Ineffective Fault Attack countermeasure and a hardware security module called Secured-Hardware-Extension.

Keywords. Fault attack, Leakage assessment, Deep learning, Countermeasure.

*Sayandeep Saha and Manaar Alam worked on this project while pursuing their Ph.D. at IIT Kharagpur

1. Introduction

Fault Attacks (FA) [1,2] have recently gained significant attention from industry and academia. The core idea of fault-assisted cryptanalysis is to deliberately perturb data or control-flow of a system and gain information about the secret through faulty system responses. There exist several physical means of injecting faults with malicious intentions. Popular methods for embedded platforms include clock-glitching [3], under-powering [4], electromagnetic (EM) pulses [5], and laser-based fault injection [4,6]. It is also possible to inject faults remotely on high-end processors and Graphics Processing Units (GPU) with the Rowhammer bug [7] and malicious exploitation of dynamic voltage-frequency scaling [8,9]. The nature and precision of injected faults in a system usually vary with injection mechanisms. Classical fault tolerance techniques often fall prey to precisely placed and repeatable faults.

Fault injection is, however, only one aspect of FAs. The key extraction process also depends critically 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 physical faults known as a *fault model*. Classically, data-corruption faults affecting a few bits or bytes in the state of a cipher are exploited in FAs. Such faults can be uniformly random, or may have some statistically biased distribution (including constant-valued faults). However, one cannot rule out faults in the control-flow nor faults at the instruction-level, which have also been shown as fatal for cryptographic implementations on several occasions [10,11]. Recently, attacks have been developed using ineffective faults by exploiting the dependence of such absence of faults on the underlying data. Such attacks have been used to break most of the existing hardened implementations [12,13].

1.1. *FA Countermeasures*

This paper focuses on FAs in the context of block ciphers. Existing block ciphers alone cannot prevent FAs, and suitable countermeasures are required. FA countermeasures are incorporated at the algorithm-level [14,15] or at a lower level of abstraction, such as in the assembly instructions [10,11] or hardware circuits [16–18]. Most of these countermeasures utilize some form of redundancy (time, hardware, or information redundancy) to detect/correct the presence of a fault in the computation. *Detection countermeasures* are the most widely deployed FA countermeasures. Such countermeasures perform an explicit check to detect the faulty computation and then react by either muting or randomizing the output [15,19,20]. *Infective countermeasures* avoid this explicit check. This class of countermeasures introduces a randomized infection function in the cipher computation that masks a faulty ciphertext making it useless for attack [14,21]. *Instruction-level countermeasures* add redundant instructions in the assembly code with the assumption that an attacker may not be able to bypass all of them at once [10,11]. With the recent advent of Statistical Ineffective Fault Attack (SIFA), a new class of countermeasures has been proposed. Such *SIFA countermeasures* incorporate redundancy checks in a per-bit manner to detect/correct every fault (whether effective and ineffective) [16–18] and thereby, destroy the data-dependent statistical bias causing key leakage. Table 1 presents a summary of countermeasures and fault models.

Table 1. Different countermeasure classes and fault models.

Type	Description	Fault model	Description
Time/Space Redundancy [15]	Performs two computations on the same data and compares the result. No output if fault is found.	Bit stuck-at/flip [6,22]	Corrupts a bit intermediate state
Code-based Redundancy [20]	Redundancy using error-detection codes	Nibble/byte [2,23]	Corrupts multiple bits within a byte/nibble
Infective [14,21,24,25]	Same as time/space redundancy, but no explicit comparison. Randomizes the outcome upon fault detection	Biased bit-flips [12,22]	Data-dependent bit-flips, useful for biased FA or SIFA
Instruction Level [10,11]	Uses redundant instructions.	Bit-flips in masked S-Boxes [13,22]	Bit-flips in S-Box intermediate computation. Useful for SIFA on masked S-Boxes
Combined [13,19,21,26]	Combined SCA-FA countermeasures, CAPA, M&M	Single-/Multi-Instruction Skip [10,11]	Instruction-Skip in microprocessors
SIFA countermeasures [16,17,26]	Bit-level error-detection/correction to counter ineffective faults	Control fault [11,20,27]	Loop abort/changing outcome of if/else block

1.2. *Issues with FA Countermeasure Evaluation*

Unfortunately, many of the existing FA countermeasures [10,14,28,29] have been found insecure even (sometimes) against the fault models they are designed to protect for. A fundamental cause behind such design failures is that there exists no general mechanism for security assessment in the context of FAs. Unlike block ciphers, countermeasures are often engineered in-house, considering several other aspects like resource/performance constraints and time to market. The design team or security certification facilities often analyze them as an end product, which may leave critical loopholes unobserved. Devising a generic methodology for evaluating FA is, therefore, an open scope of research.

1.3. *Our Contributions*

Deep Learning-Based Leakage Detection for FA. In this paper, we introduce a Deep Learning (DL)-assisted and automated yes/no testing methodology for assessing the security provided by an FA countermeasure called Deep Learning Fault

Attack Leakage Assessment Test (DL-FALAT). DL-FALAT extends our previous proposal of leakage detection using *t*-test, referred to as ALAFA [30], in terms of detection ability. In short, DL-FALAT detects potential information leakage in ciphertext (also called *traces*) distributions of a block cipher under the influence of faults. The root of this approach lies in the theory of *non-interference* [31]. Informally, achieving non-interference implies that any change in the secret data processed by a program cannot be sensed by observing the public outcome. This, in turn, implies zero mutual information between the secret and the public outcome. In the context of FAs, checking non-interference results in comparing two ciphertext distributions [30]. The main utility of DL here is to realize a *detection test* for checking if two distributions are the same or different. We propose simple DL models which work well irrespective of the design-under-test and fault model (over our test suite), enabling the leakage assessment with a low ciphertext count. For FA leakage assessment, low ciphertext count is critical as one has to perform the test on several fault locations in a design for ensuring security [32–35]. Another advantage of DL for leakage assessment is that it can detect the leakage-order automatically, unlike ALAFA [30]. The statistical order of the leakage is not known *a priori* for FA countermeasures (unlike Side-Channel Attack aka. SCA countermeasures, such as masking). Finally, we present a systematic flow to interpret the outcomes of the DL-based detection test. Major strengths of DL-FALAT lie in its simplicity and the feature of not depending on any non-trivial information regarding a hardened algorithm. It is supposed to be applied at a pre-deployment stage, where an evaluator is allowed to simulate faults at different points within the implementation code and can change the keys. We also assume that the evaluator may have access to the unprotected cipher algorithm for analysis. However, it does not need to know details of the protected implementation beyond fault simulation capability.

Enhancing the Leakage Assessment Experiment. The second contribution of this work is to enhance the non-interference experiment. By non-interference experiment (referred to as *leakage assessment experiment* in this paper), we mean the process of simulating two different ciphertext distributions corresponding to two different fault/key values. As the first enhancement to this experiment, we tailor it for detecting the so-called Statistical Ineffective Fault Analysis (SIFA). As a second enhancement, we propose a *compare-with-uniform* variant of the basic experiment, which can be utilized for testing so-called non-cipher components of a security module against FAs. Most of the time, cryptographic primitives are associated with other peripheral components, such as mask generation logic or input delivery logic, which can also be targeted by an attacker leading to an exploitable leakage. Leakage of such kind can be successfully detected by the *compare-with-uniform* experiment.

Covering the Fault Space. The leakage test has to be performed for several fault locations in an implementation. However, simulating faults for every location might lead to a longer test time, as the fault space of a block cipher is quite large [32,33]. In order to efficiently handle fault spaces, we exploit different types of equivalences present in the fault space of a block cipher. More precisely, fault equivalences at gate-level circuits and the cipher/algorithm-level are exploited to provide reasonable coverage over the fault space without exhaustively testing every fault location. Such equivalences partition the fault space into several equivalence classes, and testing

each class member is sufficient. The gate-level equivalences are explored with the TetraMax tool from Synopsys, and the algorithm-level fault equivalences are found using an automated fault analysis tool called ExpFault [33].

The Test Suite. We test DL-FALAT over a representative set of FA countermeasures, including detection, infective, instruction-level, SIFA, and combined countermeasures. Both (protected) software and hardware implementations are evaluated. To evaluate the holistic leakage assessment capability of DL-FALAT on "non-cipher" components, we test a hardware-software co-design of an automotive security standard called Secured-Hardware-Extension (SHE). We also detect some non-trivial implementation vulnerabilities for this. Furthermore, DL-FALAT analyzes CAPA [19] and the vulnerability of M&M [21], which are untold in the literature yet. We also evaluate the security claims of two recent proposals called Friet [26] and DEFAULT [36]. Overall, we observe that our framework detects published attacks with no false negatives.

1.4. *Related Work*

VerFI and FIVER. A parallel line of work in this direction is due to [35,37], which applies fault diagnosis approaches specific to hardware implementations for evaluating FA countermeasures. The approach in [37], called VerFI, is based on fault simulation for a set of test vectors and faults. VerFI monitors internal signals and the ciphertexts for detecting faults and expects the implementation details. The tool proposed in [35] (FIVER) does the same, using formal verification for all possible test vectors. Both approaches check if a fault is detected at some predefined observation points, including ciphertexts. However, vulnerabilities in countermeasures are typically not limited to their fault detection modules but also depend on the recovery modules. We practically establish this fact for infective countermeasures, where we show that a faulty outcome does not always imply an attack. Also, as established by SIFA and some recent attacks [22,38], a fault-free output does not necessarily imply security. Therefore, DL-FALAT checks the information leakage due to faults rather than simply detecting the faults. Also, checking the leakage at the ciphertext seems a reasonable idea as it represents the actual exploit of an FA. Finally, both [35,37] try to speed up the fault simulation through customized tools. In DL-FALAT, we aim to improve the leakage assessment, a complementary requirement to efficient fault simulation. We, therefore, use commercially available fault simulation tools.

DL in SCA and FA. Recent years have also seen several applications of DL in the context of SCAs, including leakage detection [39–42]. However, the leakage in SCA [43] is different from that of FA.[1] One of the major issues in FA leakage assessment is that one has to test several fault locations [32–35,37]. Hence, the statistical test at each location must operate with reasonable data complexity. The

[1] The leakage function in FA varies between attack strategies, fault models, 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.

DL-based flow presented in this paper is specifically tailored for that purpose, which was not required for the DL-based SCA leakage detection approach [40]. Such test tailoring is non-trivial, as it involves careful selection of the DL models and constructing the iterative approach proposed in this work.

Recently, FAs on DL models have gained attention from the research community [44–46]. The goal of these attacks is either misclassification or information leakage from the DL models. However, this line of work differs significantly from what we explore in this paper. Further, Reinforcement Learning has been utilized to find catastrophic faults in safety-critical systems [47]. However, [47] does not deal with the security aspects of fault injection. A more relevant work is due to [48], which identifies fault locations corresponding to some output differentials in a stream cipher using machine learning. Machine learning replaces contemporary correlation-based distinguishers in this regard. Finally, in [34], authors combine Boolean Satisfiability (SAT) solving with Random Forest to explore the exploitable fault space of a block cipher. The work in [34] aims to replace complex SAT solving in many cases with machine learning for identifying exploitable faults. However, the analysis is limited to unprotected implementations. *To the best of our knowledge, DL-FALAT is the first work which evaluates the contemporary fault models and countermeasures using a DL-based framework for leakage assessment.*

1.5. *Organization of the Paper*

The paper is organized as follows. In Sect. 2, we present the concept of leakage in FA and its connection to the theory of non-interference. This is followed by the basic descriptions of the leakage assessment experiments and the t-test based detection test. Section 3 introduces the DL-based leakage detection test in detail. Section 4 outlines two enhancements to the leakage assessment experiment. The fault space exploration strategies using fault equivalence are presented in Sect. 5. Case studies on FA countermeasures are described in Sect. 6. We conclude in Sect. 7. A discussion on instruction-level fault simulation methods using GNU Debugger (GDB) is presented in "Appendix A".

2. Fault Attack and Leakage Assessment

In this section, we elaborate on the concept of information leakage for FA and relate it with *non-interference*. Subsequently, we present two basic experiments for examining leakage.

2.1. *Information Leakage in Fault Attacks*

Leakage in fault attacks is manifested as ciphertexts (or differentials of correct and faulty ciphertexts). Formally, it is described as:

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

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*), $\mathcal{P}$ denoting

the plaintext variable and $\mathcal{K}$ denoting the secret key variable. The parameter f takes value according to some fault model F. The function $\mathcal{F}$ 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). The observables can be extended to certain variables other than ciphertexts. Later in this paper, we shall use a more general form of the observables (Sect. 4.2).

According to Eq. (1), the leakage in FA depends upon three quantities. The plaintext $\mathcal{P}$ is public and can be controlled by the adversary. The key $\mathcal{K}$ is supposed to remain secret for obvious reasons. Further, as shown in [30], *the fault value f at some intermediate state of the cipher computation should also be treated as a secret for protected ciphers.* This follows directly from the arguments in [49]. In [49], it is shown that the number of leaked bits l_k due to a fault, injected before at least the last nonlinear operation in a block cipher, can be represented as $l_k = H(\mathcal{K}) - H(f) - const$. Here, $H(\cdot)$ denote the Shannon Entropy. The (small) constant factor $const$ comes due to the differential properties of the S-Boxes. If f is known, $H(f) = 0$, implying $l_k = (H(k) - const) \approx H(k)$. In other words, the entire key is leaked if the fault value is known.

We note that the proof in [30] regarding the secrecy of f considers the fault to be injected before at least the last nonlinear operation of the cipher, which is a requirement for most FAs [2,12]. Faults injected after the last nonlinear (or before the first nonlinear operation) can only be exploited if it is data-dependent (e.g., stuck-at faults). No statistical bias or differential relation can be formed without a nonlinear operation in the fault propagation path. Data-dependent faults are the only way of leaking information in such cases. Even for such cases, knowing f reveals the secret. For stuck-at-0 faults, the fault mask (value) is $f = 0$ only if the state value at injection point is 0 ($f = 1$, otherwise). The same logic applies to stuck-at-1 faults. The knowledge of the intermediate state exposes the key. To summarize, the observation regarding the exposure of fault mask in [30] is consistent even for those locations which are not followed by a nonlinear layer.

Condition for No Leakage In the case of unprotected implementations, both key and f leak via the faulty ciphertexts, and the abovementioned arguments for leakage apply directly. *Therefore, the only way of preventing FA is to prevent the information flow from both $\mathcal{K}$ 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 secure if it satisfies the two following equations:

$$\mathcal{I}(C, \mathcal{K}|\mathcal{P}) = 0 \quad \text{and} \quad \mathcal{I}(C, f|\mathcal{P}) = 0. \tag{2}$$

$\mathcal{I}(X, Y|Z)$ is the conditional mutual information between random variables X and Y given Z. These two definitions can be used interchangeably for leakage assessment. However, we aim to evaluate the hardened implementations without utilizing algorithmic details. The lack of algorithmic details refrains the analytical estimation of mutual information and leaves data-based statistical estimation as the only option. Although such data-based estimation of mutual information is possible, it is challenging and still an active area of research [50,51].

Leakage Assessment with Non-Interference The technical difficulty in estimating mutual information can be circumvented by an alternative interpretation of the leakage with

the theory of *non-interference*. *The non-interference property guarantees the absence of sensitive information flow from the input to any observable point of a system.* For FA-induced leakage, non-interference between the key or the fault value with the ciphertext or ciphertext differential implies that the attacker cannot exploit the ciphertext to extract the secret. Assessment of non-interference in programs is performed by assigning program variables with different security levels. In particular, some of the variables are secret (marked as '*high*'), and the rest of them are public ('*low*'). If the underlying program is probabilistic, then the program variables can be treated as random variables.[2] In this setting, *non-interference implies that the mutual information between the 'high' input variables and the 'low' output variables is zero*. This condition is the same as the definition of security provided in Eq. (2), considering ciphertexts as 'low' variables and key and faults as 'high' variables. However, an equivalent [31], easy-to-use formulation of non-interference exists, which does not require estimating the mutual information. *If the low outputs differ in two independent runs of a program having the same low inputs but different high inputs (h and h'), then the program leaks about its high inputs. Otherwise, the program achieves non-interference.* For probabilistic programs, the difference in low outputs is manifested as the difference between two distributions generated with the same low inputs. We utilize this notion to assess security. Comparing two ciphertext distributions suffice in this case.

2.2. *Basic Experiments in Leakage Assessment*

This subsection presents two variants of the leakage assessment experiment based on the notion of non-interference presented before. Both fault and key are treated as secrets ('high' inputs). For simplicity, we keep the value of one of the secret inputs fixed during our testing, which results in two experiments.

The interference experiment with fixed key and varying fault value is presented in Algorithm 1. The algorithm takes a protected cipher $\mathcal{C}$, and two fault values f_1 and f_2, and a key k as inputs. Algorithm 1 runs two independent simulations of $\mathcal{C}$ for f_1 and f_2 with fixed plaintext p and key k. One should note that $\mathcal{C}$ may internally generate random numbers to randomize the outcome in each run. The *simulation traces* (the ciphertexts), denoted as $\mathcal{T}_{f_1}$ and $\mathcal{T}_{f_2}$, are then subjected to a statistical test $TEST()$. The $TEST()$ checks the equality of the distributions resulting from the two simulation traces and returns $TRUE$ if the distributions are unequal. If $TEST()$ returns $TRUE$, Algorithm 1 returns YES, indicating a violation of non-interference. The second interference experiment (ref. Algorithm 2) is realized similarly, but by varying the keys and keeping the fault value fixed. The test in Algorithm 2 runs on the ciphertext differentials. This is to handle the cases when the fault has an incomplete diffusion to the ciphertext. Considering ciphertexts rather than the differentials would leave a constant difference between the instances of two classes $\mathcal{T}_{k_1}$ and $\mathcal{T}_{k_2}$, which may result in false positives in $TEST()$.

The choice between Algorithm 1 and 2 depends on the type of application being tested and the fault model. Keeping the key fixed is found to be the most convenient

[2] A *probabilistic program* PP is a routine, which contains both probabilistic and deterministic assignments and variables, when represented in Single-Static-Assignment (SSA) form. A PP takes a joint distribution of input variables and outputs a joint distribution of output variables.

option for cases where fault values vary within some finite range (for example, in the case of byte faults, the range is $\{1, 2, \cdots, 255\}$). This is because the size of the keyspace is much larger than the size of the fault space, and this size would matter in certain situations. For example, in the case of code-based detection countermeasures, not every fault value or key value (considering a fixed plaintext and fault value) is leaky, as a faulty state might get detected by the error-detection module, causing zero leakage. In such cases, one must exhaustively search the fault/keyspace to identify potential leaky faults (resp. keys). While this search is relatively easy for a fault space of size 255, it becomes computationally intensive for the keyspace, which is much larger. Most other countermeasures, such as infective/instruction-level, behave uniformly on any fault value. Therefore, testing on arbitrary fault values suffices in practice for them. Varying the key is convenient for control-flow faults, bit-flip/stuck-at faults, or instruction-skip faults. In such cases, the fault can 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 only faulty value is "no").

The non-interference experiments can be optimized or generalized for certain countermeasure classes, fault models, or observables. One such optimization, specific to detection countermeasures has been presented in [30] (as a preprocessing step for selecting fault value pairs $((f_1, f_2))$ causing leakage). We propose two other optimizations in Sect. 4. We focus on the detection test $TEST()$ in the next subsection and the subsequent section.

2.3. t-*Test for Leakage Detection*

One way of implementing $TEST()$ is to apply Welch's t-test [30]. 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. Large absolute values of the t-test statistic (denoted as t) indicate that the datasets have different distributions. A threshold of $|t| > 4.5$ indicates that the confidence of the test is > 0.99999.

In modern block ciphers, ciphertexts are of 64, or 128 bits, and treating them as a single random variable during the t-test is impractical. One solution is to treat them as multivariate quantities. Each bit, nibble or byte of ciphertexts can be treated as a variable. We propose considering both bit and byte-level divisions separately. The t-test applies to discrete-valued variables if the sample size is reasonably large [52–54] due to the Central Limit Theorem (CLT) [55]. In all our experiments, we keep the sample size more than 500 to ensure statistical significance of the t-statistic. Being univariate, the t-test applies separately to each variable. However, information leakage may not be manifested in this univariate setting. To see this, let us consider two variables V_1 and V_2 such that $V_1 = X \oplus r$ 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, considering the joint distribution of V_1 and V_2 makes the leakage visible, as the effect of the mask r gets nullified. To capture such leakage, the t-test must be performed in a multivariate setting. One approach for extending t-test to

the multivariate setting is to consider the *centered product* (i.e., higher-order statistical moments) of different variables [30,56]. The centered product approach has been utilized successfully in ALAFA.

In ALAFA, the $TEST()$ function begins with performing a univariate test (bit/byte-level) and continues with d-th order testing, for $d = 1, 2, \ldots, G$, until a leakage is observed. G is to be specified by the user. We note that G should vary from 1 to 16 if the ciphertext is treated byte-wise and 1 to 32 or 1 to 128 if treated nibble/bitwise. The simulation time (S in Algorithm 1, 2) increases for higher G values and hence, decides the test complexity. However, higher G values ensure a stronger security guarantee.

Algorithm 1 *TEST-INTERF-FAULT*

Input: Protected Cipher $\mathcal{C}$, Fault value f_1, f_2, Key k, Simulation counter S
Output: Yes/No
$\mathcal{T}_{f_1} := \emptyset$; $\mathcal{T}_{f_2} := \emptyset$
$p := GEN_{PT}()$
for $i \leq S$ **do**
  $\mathcal{T}_{f_1} := \mathcal{T}_{f_1} \cup \mathcal{C}(p, k, f_1)$
  $\mathcal{T}_{f_2} := \mathcal{T}_{f_2} \cup \mathcal{C}(p, k, f_2)$
end for
if $(TEST(\mathcal{T}_{f_1}, \mathcal{T}_{f_2}))$ **then**
  Return Yes
else
  Return No
end if

Algorithm 2 *TEST-INTERF-KEY*

Input: Protected Cipher $\mathcal{C}$, Fault value f, Key k_1, k_2, Simulation counter S
Output: Yes/No
$\mathcal{T}_{k_1} := \emptyset$; $\mathcal{T}_{k_2} := \emptyset$
$p := GEN_{PT}()$
$corr_1 := \mathcal{C}(p, k_1)$
$corr_2 := \mathcal{C}(p, k_2)$
for $i \leq S$ **do**
  $\mathcal{T}_{k_1} := \mathcal{T}_{k_1} \cup \{corr_1 \oplus \mathcal{C}(p, k_1, f)\}$
  $\mathcal{T}_{k_2} := \mathcal{T}_{k_2} \cup \{corr_2 \oplus \mathcal{C}(p, k_2, f)\}$
end for
if $(TEST(\mathcal{T}_{k_1}, \mathcal{T}_{k_2}))$ **then**
  Return Yes
else
  Return No
end if

3. DL-FALAT: Deep Learning-based Detection Test

The t-test and its higher-order variants indeed work for realizing $TEST()$, but with some critical theoretical and practical limitations. Higher-order t-test can only capture different statistical moments, which has been shown to be sub-optimal in the context of SCA leakages [57,58], even resulting in false negatives. Nevertheless, t-test for FA leakage assessment can also be problematic from a usability perspective. The leakage-order in FA does not formally relate to the countermeasure construction. This is in contrast to SCA countermeasures such as masking, where the maximum possible leakage-order directly relates to the masking order. The multivariate and higher-order leakages in FA are usually formed due to the fault propagation and improper construction of the countermeasures (e.g., for certain infection functions in infective countermeasures). Consequently, no information regarding the maximum order of such leakages is available *a priori* to the designer or the evaluator. The security guarantee depends upon the evaluator's choice of the maximum test order G for t-test.

DL methods are renowned for learning in highly multivariate scenarios and can take several complex interrelations among different features (beyond moments) into consideration [40,59,60]. Further, DL does not require any order-related information to be given from the evaluator side, as it can automatically discover the dependencies between different input features. This motivates us to propose DL-FALAT, a DL-based fault analysis tool. During our experimentation, it is found that DL performs significantly better in

noisy scenarios and for very high leakage-orders compared to the t-test-based approach. We refer to [61] for background on DL.

3.1. *DL-based Leakage Testing: Main Idea and Challenges*

The idea behind DL-FALAT is to *train* a Neural Network (NN) as a binary classifier with two sets of ciphertexts resulting from two different secret values. Afterward, 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 learned by the network. A *better-than-random* guess over the validation set indicates the existence of leakages from the countermeasure. On contrary, if the validation accuracy is random, it implies the absence of leakage.

Although the approach stated above is simple, it poses several caveats and challenges during implementation. We list them as follows:

Decision Making. One fundamental challenge in DL is to quantify the *decision threshold* based on which one can distinguish between a leaky and a non-leaky implementation.

Sample Size. It is always desirable that the detection test returns a consistent decision with the lowest possible number of samples. The sample size becomes critical as one needs to test multiple fault locations [32,33,35], requiring several fault simulations for each of them.

Model Selection. In an ideal world, one specific DL model should work for a large class of test scenarios. The critical question is whether or not there exists one such single model. According to the "No-Free-Lunch-theorem" [62], such a universally best model cannot exist. However, our problem space is limited—we only work on ciphertext distributions in the FA context. Moreover, as we point out later, we do not need the model to fit optimally for each dataset. Instead, a sign of learning is sufficient. Therefore, it is reasonable to believe that a small set of such models might exist and works well across a large set of benchmarks considered in this work. However, constructing such models is an important problem. Model selection becomes more challenging when the number of data samples is less, as there may be a tendency to *overfit*. Overfitting [61] is a phenomenon in ML where the model starts memorizing the training data and, as a result, fails to generalize (i.e., fails to provide consistent prediction on new data). A sign for overfitting is a low training error but high validation error.

Interpretation. How to obtain meaningful insights (such as univariate or multivariate leakage, the position of leaky bytes/bits in the ciphertext, etc.) from the DL results?

We begin by addressing the first two issues simultaneously in the next subsection, as there are some interrelations between them.

3.2. *Iterative Training and Decision Making*

If better-than-random learning occurs for a DL model, it implies the existence of leakage. One key insight, in this case, is that *The learning does not require to be the "best". Even a*

Algorithm 3 *DL-TEST-INTERF-FAULT*

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

$\mathcal{T}_{f_1} := \emptyset$; $\mathcal{T}_{f_2} := \emptyset$
$p := GEN_{PT}()$
$S_t := S_{init}$
$leak := Null$
while $S_t \leq S$ **do**
  for $i \leq S_{init}/2$ **do**
    $\mathcal{T}_{f_1} := \mathcal{T}_{f_1} \cup \langle \mathcal{C}(p, k, f_1), 0 \rangle$ ▷ Add labels to the data as "0" or "1"
    $\mathcal{T}_{f_2} := \mathcal{T}_{f_2} \cup \langle \mathcal{C}(p, k, f_2), 1 \rangle$
  end for
  $\mathcal{D}_t := \mathcal{T}_{f_1} \cup \mathcal{T}_{f_2}$
  $\langle \mathcal{D}_t^1, \mathcal{D}_t^2, \cdots, \mathcal{D}_t^K \rangle :=$ `GEN-CROSS-VALID-SET`$(\mathcal{D}_t)$ ▷ Generate K subsets for cross-validation
  $A_t := \emptyset$
  for $i \leq K$ **do**
    $Tr_t^i := \bigcup_{\substack{j=0 \\ j \neq i}}^{K} \mathcal{D}_t^j$
    $Vl_t^i := \mathcal{D}_t^i$
    $a_t^i :=$ `Train-and-Validate`$(\mathcal{M}, Tr_t^i, Vl_t^i)$ ▷ Get the validation accuracy
    $A_t := A_t \cup \{a_t^i\}$
  end for
  if (`t_Test`$(A_t) \implies \mathcal{H}_0$) **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

small indication of learning is sufficient to decide leakage. However, this indication must come with high (preferably quantifiable) statistical confidence. This insight is valuable for keeping the sample size for training and validation relatively small and for selecting models.

Overall Flow. There is no clear thumb rule to determine the proper amount of data required for training in DL. Hence, we begin the training with small training and validation sets and iteratively increase their size by taking feedback from a decision-making operation, indicating whether there is any leakage. The training and validation iteration continues until leakage is detected or a user-defined dataset size limit has been reached. This iterative process helps us to test with the minimum possible number of samples.

The DL-based leakage assessment experiment is outlined in Algorithm 3. The basic experiment is the same as the one described in Algorithm 1. However, the $TEST()$ is replaced with the iterative DL-based test. A straightforward extension for Algorithm 2 is also possible. The dataset under consideration is denoted as

$\mathcal{D} = \mathcal{T}_{f_1} \cup \mathcal{T}_{f_2}$ (resp. $\mathcal{T}_{k_1} \cup \mathcal{T}_{k_2}$). The instances from the set $\mathcal{T}_{f_1}$ (resp. $\mathcal{T}_{k_1}$) are labeled as "0", and the instances from the set $\mathcal{T}_{f_2}$ (resp. $\mathcal{T}_{k_2}$) are labeled as "1". The training and validation begin with a small dataset size S_{init}. The size of the set $\mathcal{D}$ is increased adaptively in each iteration by adding an equal number of samples from both of its constituent sets. To represent the varying size of $\mathcal{D}$, from now onward, we use the notation $\mathcal{D}_t$ denoting the dataset at t-th iteration. The entire set $\mathcal{D}_t$ is divided into training and validation sets Tr_t and Vl_t, respectively. At the t-th iteration, the model is trained with Tr_t and validated over Vl_t. The test continues until a maximum dataset size S is reached or some leakage is detected. Table 2 presents the parameter settings for Algorithm 3 decided experimentally based on our test suite. We did not observe any change in the leakage trend beyond these ciphertext counts for different countermeasure classes from our test suite. The S_{init} is chosen empirically. In the case of infective countermeasures, we mostly observed multivariate leakage. For such countermeasures, we apply an optimization for saving the total learning time for multiple iterations of Algorithm 3 while keeping the test still reliable. *If leakage is not observed within* $S = 10,000$*, we perform another single learning iteration with a large sample count as a final confirmation test.* In our experiments, 20,0000 samples gave reliable results in such cases.

K-fold Cross-Validation. 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 [63] (line 11 to line 18 in Algorithm 3). The K-fold cross-validation can be explained as follows. The entire dataset $\mathcal{D}_t$ is randomly partitioned into K equal-sized subsets $\mathcal{D}_t^1, \mathcal{D}_t^2, \cdots, \mathcal{D}_t^K$ ($|\mathcal{D}_t^j| = \frac{|\mathcal{D}_t|}{K}, \forall j$). The *stratified* feature ensures that for each $\mathcal{D}_t^j$, an equal number of samples are there from both of the classes (label-0 and label-1). Next, $K-1$ of these subsets are used for training the model $\mathcal{M}$, and one subset is used as a validation set. This process is repeated K times, giving each subset one chance to be used as a validation set. The main idea is to check if the model $\mathcal{M}$ is capable of generalizing its knowledge for unseen datasets or not.

One-Sided *t*-test for Decision Making. In our testing methodology, we accumulate the validation 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 validation accuracy while validating on $\mathcal{D}_t^j$)). To check leakage, we test the following hypothesis:

$$\mathcal{H}_0 : \mu_{A_t} = 0.5, \text{ and } \quad \mathcal{H}_1 : \mu_{A_t} > 0.5. \tag{3}$$

Here, μ_{A_t} denote the mean over set A_t. In case of leakage, the alternative hypothesis $\mathcal{H}_1$ is accepted. We apply one-sided t-test with significance level $\alpha = 0.0001\%$, and degrees of freedom $K-1$. *The t-value threshold is* $t = 4.5$ *(i.e.,* $t \geq 4.5$ *implies leakage). Acceptance of the alternative hypothesis indicates that the average validation accuracy is better than random guess (i.e., 0.5), which indicates that the DL model is learning and there is leakage.* The choice of K plays a critical role in making this t-test statistically significant. A value larger than 30 is commonly recommended to make the CLT hold [64] (p. 157). We choose $K = 50$. A larger

Table 2. Parameters for DL-based Leakage Detection Flow.

S_{init}	S	K
500	10000 (for infective countermeasures) 5000 (for time/space/information/instruction redundancy, SIFA faults))	50

value increases the runtime [65] (p. 70) of the test without changing the outcomes, as checked by us.

3.3. *Selection of the DL Model and Parameter Choices*

One of the major challenges of the leakage assessment problem is to select a generalizable model ($\mathcal{M}$), which should not depend upon the design under test or the nature of leakage. As already pointed out, one advantage we have in leakage assessment is that the learning need not be the best. Any better-than-random validation accuracy is acceptable. This fact allows some flexibility for model selection and also helps to find a few models to scale for a large test suite.

Bit and Byte Models We use two models shown in Listing 1.1 and 1.2. The manifestation of leakage in the ciphertext structures is interpreted at the bit-level or byte-level. This choice is motivated by the structures of existing ciphers and countermeasures, which mostly follow bit/byte-level structures. Hence, we use two separate models for bit (Listing 1.1) and byte-level (Listing 1.2) analysis. Both models can detect leakages due to fault attacks, irrespective of the fault model or the leakage detection experiment. We make this claim based on the fact that both models compare two distributions. The only difference between them is the granularity considered for the input ciphertexts. However, the number of ciphertexts required for detecting leakage varies between the two models depending on the underlying cipher and countermeasure structure (bit/byte). Since one of the main motivations of this work is to reduce data complexity, we propose using both models simultaneously on the data for practical purposes.

The Network Architecture The models have been developed using the Python-based Keras library [66], which uses TensorFlow [67] in the backend. Both networks have 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 neurons, respectively. In both models, the output layer contains 2 neurons. The hidden layers use *Rectified Linear Unit* (`ReLu`) activation function, whereas the output layers use `Softmax` activation function. Also, `Batch Normalization` is applied between the dense layers.[3] As the loss function, we use *categorical cross-entropy*. The *Adam* optimizer [69] is chosen for the learning process (mostly with default parameter settings, as per Keras). The number of *training epochs* is fixed to 50. However, we apply early-stopping technique [61] to stop the learning process whenever we observe leakage in cross-validation. This stopping epoch varies for different dataset due to their respective leakage patterns.

[3] `Batch Normalization` speeds up the learning process [68].

Listing 1.1. Bit-Model.

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

Listing 1.2. Byte-Model.

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

It can be observed that the proposed models are simple. An advantage of simple models is the reduced risk of overfitting, especially while we try to use as little data as possible. We verified that none of our examples leads to overfitting even while trained with the minimum number of samples required. The K-fold cross-validation reduces the chances of overfitting, in general [61]. We further investigate the difference between the training and validation loss for each of the K folds to ensure no overfitting. If the two losses are similar, then the model is not overfitting, and this is the case for our models on the test suite.

3.4. *Leakage Interpretation Techniques*

Motivation There exist multiple approaches in the literature to interpret the decisions made by a DL model, and they have also been used previously in the context of SCA security [39,40]. However, some issues have not been addressed clearly in the SCA/FA literature. Firstly, it may happen that the model only takes certain leaky features (i.e., ciphertext bits/bytes) into consideration while ignoring others. Such a situation is natural as the desired classification may be easily achieved by considering a subset of features only. However, exposing all leakage points is a vital issue as it can provide valuable information on how to attack. Secondly, in the DL-based method, it is difficult to understand whether the leakage is univariate or multivariate, especially when both kinds of leakage points are present in one trace (this is the case in some of our examples). Note that t-test-based method addresses this issue by gradually increasing the analysis order d. The motivation behind leakage interpretation is to extract such information from a DL model.

Sensitivity Analysis We use the trained network model $\mathcal{M}$ for leakage interpretation and adopt an iterative approach. The very first step we perform is a *Sensitivity Analysis* (SA) [39], which returns the contribution of each feature in learning the leakage. Mathematically, the *Sensitivity* (Im_i) for each feature is computed as $Im_i = \left|\sum_j \frac{\partial y_0}{\partial x_i} \cdot X_i^j\right|$. Here, x_i denotes the i-th input of the model $\mathcal{M}$, y_0 is the first output of $\mathcal{M}$, and X_i^j is the value of the i-th input in the j-th ciphertext from the validation set. The partial derivative computes how much the output y_0 changes with respect to an input x_i. The sensitivity is an aggregate of the changes over the entire validation set for each input. For SA, we consider a fresh and sufficiently large validation set while computing the feature importance values. Although the overall ciphertext count increases, we suggest performing leakage interpretation only for fault locations showing some sign of leakage. Generating extra traces for a few leaky fault locations seems reasonable, rather than doing this for all probable fault locations in an implementation [32–34].

Finding Leakage Points The SA step assigns real values to individual features (i.e., ciphertext bit/bytes) by which they can be ranked according to their contributions to the decision making. In our analysis, we first begin with the subset of *most important features*. The most important subset of features (MI) is determined by a threshold Th_{MI}. If the Im_i value of a feature is $\geq Th_{MI}$, we consider the feature as important. We found that the average of all Im_is works well as Th_{MI} over our entire test suite, giving meaningful results. While such a choice for Th_{MI} is empirical and based on experimental observations, we note that choosing the average as Th_{MI} has some clear advantages. It does not require us to specify another extra parameter. It gives consistent results when most features have similar Im_i values, as the average will be close to that value. Also, if some of the Im_is are significantly higher than the rest, there is no risk of missing these high Im_i points. Finally, since our leakage interpretation is an iterative process, there is not much risk of missing an important feature due to Th_{MI}. If some feature is important for classification but still not designated as important in one iteration, it will become so in the next iteration.

Once the MI has been determined, the analysis follows two separate paths. In the first path, we eliminate all the features in MI from the actual trace by assigning them to 0. We repeat the learning on the modified trace and check if the model still learns the leakage. If the model does not learn, the dataset size is gradually increased to some predefined count. This count is kept higher than the standard leakage detection to gradually expose even the most difficult-to-detect leakage points (we choose this count as 20,0000 based on all our experiments). The feature elimination and training iterate until all feature points are exhausted or the model fails to learn. In the second path, the MI set obtained in an iteration is tested to check whether the leakage is univariate or multivariate. We apply the same trick of eliminating feature points in this case. However, only one point from MI is eliminated at each step, and the training is repeated with the truncated MI set. *If the leakage is univariate, even a single point in MI would be able to classify. In case of multivariate leakage, the classification would 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 subset of MI. While this is feasible if MI is small, it would be costly to perform for larger MI sizes. Experimental validation of leakage interpretation is presented in Sect. 6, where we show that it can identify some previously proposed attacks in the literature only from the ciphertexts.

3.5. *Discussion*

We further investigate the model selection issue by considering other relatively complex models, such as Convolutional Neural Nets (CNNs). It is found that the data complexities for leakage detection in CNNs are very similar to those with our models. One reason behind such observation is that we do not require the best possible learning to happen. We believe that simple networks are still better than complex ones, as they are less prone to overfitting. We also check another model type having a single neuron output and `sigmoid` activation. The motivation behind such a construction is that we target a binary classification problem. Binary classification can also be handled with a single-output network rather than a 2-output one, as we used in this work. It is found that

the results are slightly worse than the 2-output (one-hot) encoding. More precisely, we require roughly $200 - 500$ extra ciphertexts for different test cases with `sigmoid` activation. Another relevant question is whether other statistical tests, which are not moment-dependent, work better in this context than our DL-based method. We consider the χ^2-test, which has been used before for leakage detection in SCA [58]. In terms of data complexity, the χ^2-test performs similar to the DL-based test in many cases. However, there are pathological cases where the performance of the χ^2-test is inferior to the DL-based test. One typical example is an infective countermeasure called RIMBEN [25], for which DL requires 20,000 traces and χ^2-test requires roughly 80,000). Most importantly, the test-order has to be specified even for the χ^2-test, which is not required for the DL.

4. Proposed Generalizations of Leakage Assessment Tests

This section proposes two generalizations of the leakage assessment experiment itself. The first extends the experiments for SIFA faults, while the second enhances the leakage and observable definitions for "non-cipher" components.

4.1. *Handling SIFA Faults*

SIFA utilizes the fact that the *activation* (generation of a faulty value at the location of injection) and *propagation* (propagation of the fault through the circuit) of a fault depend on secret intermediate values. As a result, an injected fault may remain "ineffective" for specific intermediate values and eventually result in correct ciphertexts. As a simple example of how ineffective faults happen, 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 occur, and a correct ciphertext can be observed. In contrast, if the actual value is 1, it will result in a faulty execution. Typically, SIFA exploits 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 [12,13].

The goal of this section is to tailor the test methodology in a way that can meaningfully capture SIFA. One straightforward approach (adopted in [37]) is to declare a countermeasure as secure if every fault propagates to the output or every fault gets corrected (so that the ineffectivity of faults does not depend upon secrets). However, this is conservative and will lead to false positives in several cases. For example, masking prevents SIFA [17] for certain restricted fault models, even if there is a mix of correct and faulty ciphertexts. To defeat masking with SIFA, one would require to fault certain specific points inside S-Boxes [13], which may not be feasible for every implementation. Hence, a mix of correct and faulty executions does not necessarily mean that SIFA would occur.

SIFA Fault Models. SIFA faults are modeled in two ways. We model biased faults as the probability of a bit b remaining unchanged during fault injection ($pr_{0\to0}$ if $b = 0$ and $pr_{1\to1}$ if $b = 1$). For biased faults, this probability is not equal for $b = 0$ and $b = 1$ (i.e., $pr_{0\to0} \neq pr_{1\to1}$). An example of such faults is the stuck-at-0 (resp. stuck-at-1) fault where the probability of b remaining unchanged

is 1 for $b = 0$, and 0 for $b = 1$. Such faults create correct ciphertexts dependent on intermediate state bits [12]. The second fault model is required for performing SIFA on masked implementations. Here, we perform bit faults within the masked nonlinear operations inside S-Boxes so that the correct output becomes dependent on some unmasked intermediate value [13].

Modifications to the Basic Experiment. We now describe the modifications to the basic leakage assessment experiment. Most 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 faulty value is 1 and vice versa). Our approach is to vary the key instead of the fault values (k_1 and k_2). Only constraint over k_1 and k_2 is that if the encryption of p with k_1 results in bit value 0 (or 1) in the fault injection point, then the encryption of p with k_2 must result in bit value 1 (or 0) in the injection point. For masked implementations, if some shares of an intermediate bit b are targeted with a fault, it is required that $b = 0$ (resp. $b = 1$) for k_1 and $b = 1$ (resp. $b = 0$) k_2. Finally, we apply a simple trick which exposes the bias in fault injection (if any) at the ciphertext. *For detection countermeasures, the faulted output (usually represented as a fixed string) is replaced with random strings of the same length as correct ciphertexts.* This replacement eliminates the unwanted constant differences between the two ciphertext distributions to be tested due to fixed strings. No replacement is required for infective countermeasures as they already output randomized ciphertexts in case a fault is detected. The leakage test is performed on the differentials between the correct and faulty ciphertexts.

Why SIFA Leakage is Exposed? It is a tempting question how SIFA leakage gets exposed through the modifications proposed in the last paragraph. The differentials corresponding to the correct ciphertexts obtained in the fault injection campaign are equal to zero. The differentials corresponding to the faulty ciphertexts are random. Each of the datasets corresponding to keys k_1 and k_2 (denoted as $\mathcal{T}_{k_1}$ and $\mathcal{T}_{k_2}$) thus contains zero-valued bit/byte strings along with some random strings. Let us denote the count of zero-valued strings as Cnt_0 and random strings as Cnt_1 in one of the datasets (say in $\mathcal{T}_{k_1}$). *The ratio* $R_{=0} = \frac{Cnt_0}{|\mathcal{T}_{k_1}|}$ *nearly equals to either* $pr_{0\to 0}$ *or* $pr_{1\to 1}$ *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 $pr_{0\to 0}$ or $pr_{1\to 1}$, which eventually results in a correct ciphertext. Next, let us consider the two datasets $\mathcal{T}_{k_1}$ and $\mathcal{T}_{k_2}$. As already mentioned, b assumes different values for k_1 and k_2. One may observe that *the ratios* $R_{=0}$ *for these two datasets become different.* This is because in one of the cases (say for k_1), $R_{=0}$ equals to $pr_{0\to 0}$, while in the other case it equals to $pr_{1\to 1}$. The difference in the ratios establishes the fact that the two underlying distributions in $\mathcal{T}_{k_1}$ and $\mathcal{T}_{k_2}$ are also different, which indicates leakage. Otherwise, there is no leakage and a SIFA cannot be performed. Similar arguments can be given for the other fault model for masking implementations.

Algorithm 4 *TEST-INTERF-GENERALIZED*

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

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

4.2. *Assessing "Non-Cipher" Leakages—Compare-with-Uniform*

There are situations where a fault in some key-independent component may indirectly cause key leakage. For example, the security of a masked implementation strongly depends on the availability of uniformly random bit sequences. Any deviation from uniform randomness may enable an SCA. An adversary may de-randomize masks using faults. One concrete realization of such derandomization (for hardware) has been presented in [70]. In [70], the fault corrupts a random number generator (RNG) using Hardware Trojan Horses (HTH). Corrupting the input logic for key/nonce/mask is a general use-case for such exploits. Algorithm 2 is not applicable in such contexts as many such cases do not directly associate with key (such as the mask or nonce). Algorithm 1 will also not work because leakage of fault values does not lead to any meaningful information unless faults are injected inside the cipher computation.

Compare-with-Uniform Experiment. In order to generalize the leakage assessment for the situations mentioned above, we first extend the notion of the observables beyond ciphertexts. *An observable $\mathcal{O}$ is a set of variables either input or output to a cryptographic module.* Apart from the ciphertexts, examples of observables include the key, mask and nonce inputs to a crypto-core. The proposed enhancement to the non-interference test is based on a simple principle—*if the distribution assumed by an observable changes (to a non-uniform distribution) due to a fault injection, then such a fault can be exploited by an adversary.* To test

this, we compare any observable distribution resulting from a fault injection with a uniformly random distribution using $TEST()$. The fault here is simulated several times for a single fault value. We call this as a *compare-with-uniform* experiment. The intuition behind this test is that *if the fault event results in randomizing the outcome of the target observable $\mathcal{O}$, then no information can be extracted from it even by the attacker. Deviation from randomness may directly indicate chances of potential attacks caused due to randomness loss (e.g., nonce repetition or a non-uniform mask for SCA resistance).*

Integrated Test Flow. An integrated test flow considering all observable definitions is presented in Algorithm 4. For every fault injection point, we first check if the fault influences the observable or not (line. 6) by changing its value. Next, fault simulation is performed for a single effective fault value, and the simulation data are subject to the compare-with-uniform test. In case the test indicates no distinction from uniform random, we may safely terminate the experiment for the fault location indicating no leakage. In the other case, it suspects leakage. Further, if the observable is found key-dependent, we run one of Algorithm 1 or 2 (whichever is suitable) and establish the existence of key-dependent leakage.

5. Handling the Fault Space

Ideally, the fault simulation and the leakage detection test should be performed for each fault location and fault model. However, the number of testable locations can be reduced by exploiting the equivalences present in the fault space of a block cipher. In this section, we describe two types of equivalence relations—equivalence at the gate-level and equivalence at the block cipher-level. The gate-level fault equivalences are useful for hardware and bitsliced software implementations, whereas the equivalence due to structural redundancies of ciphers is utilized for any software/hardware implementation.

5.1. *Fault-Equivalence at Gate-Level*

Testing for stuck-at faults (bit-level) is well-studied in the domain of digital testing [71]. Generating test vectors for a given combinational circuit with W number of nets/wires (input, internal or output) requires considering total $2 \times W$ faults (both stuck-at-0 and stuck-at-1 fault for each wire). Test generation for each of these faults needs solving an NP-Complete problem [71]. While it seems challenging for large-scale circuits with millions of gates, it is practical and implemented in several commercial tools. One way of handling such a huge fault space is to reduce/*collapse* the total fault set using equivalence relations among the faults. Tests generated for such a collapsed set of faults guarantee good fault coverage over the entire circuit. This is referred to as *fault-collapsing* [71].

Fault Equivalence and Dominance Fault collapsing utilizes two fundamental properties called *fault-equivalence* and *fault-dominance* to generate a reduced fault set which covers all possible single stuck-at-fault scenarios. The fault equivalence is defined as follows:

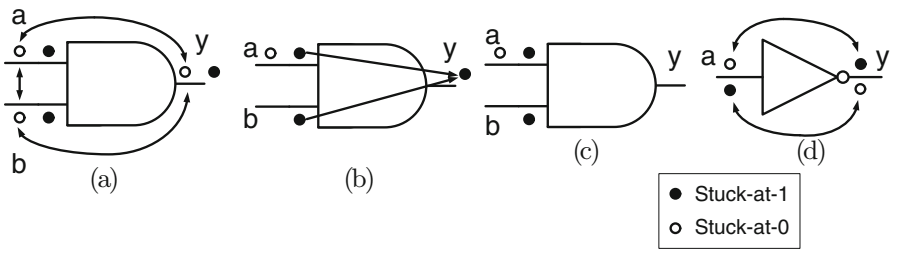

Fig. 1. Fault collapsing for AND: **a** Equivalent stuck-at-0 faults; **b** stuck-at-1 fault at y dominates the stuck-at-1 faults at the input nets; **c** Collapsed fault set; **d** Fault equivalence for NOT gate .

Definition 1. (*Fault equivalence*) Let Z_{fl} denotes the input–output mapping realized by a circuit Z with a fault fl induced in it (at some specific net). Two faults fl_1 and fl_2 are considered *equivalent* if $Z_{fl_1}(x) = Z_{fl_2}(x)$, for $x \in I$ (I is the set of all possible inputs to the circuit).

Fault equivalence (ref. Definition 1) can be tracked structurally from the circuit netlist. For example, we refer to the AND gate shown in Fig. 1a. In this case, the stuck-at-0 faults at the inputs and the output are equivalent. It can be observed that the test pattern $a = 1, b = 1$ detects the stuck-at-0 faults at a or b. The same pattern detects the stuck-at-0 fault at the output y. Hence, *stuck-at-0 faults at a, b and y are equivalent*. From another viewpoint, a stuck-at-0 fault at any of a, b or y sets the output value y to 0. Hence, the corresponding mappings $Z_{a,st0}$ $Z_{b,st0}$ and $Z_{y,st0}$ are equivalent. Simulating any one of these three faults will have the same impact on the output.

The fault dominance is defined as follows:

Definition 2. (*Fault dominance*) Let T_{fl_1} be the set of all tests that detect a fault fl_1. A fault fl_2 dominates fl_1 if and only if fl_1 and fl_2 are equivalent under T_{fl_1}.

The idea of fault dominance (ref. Definition 2) is illustrated in Fig. 1b where the stuck-at-1 fault at y dominates the stuck-at-faults at a and b. The test vectors $a = 0, b = 1$ and $a = 1, b = 0$ detects the stuck-at-1 faults at a and b, respectively. The same test vectors can also detect stuck-at-1 fault at y. The reduced fault set after collapsing is shown in Fig. 1c. One may observe that instead of a total of 6 faults, one needs to test only 3. A similar example of collapsing based on equivalence is shown in Fig. 1d for a NOT gate.

Fault Dominance and Biased Faults One may note that the dominance property only claims equivalence on a set of test vectors T_{fl_1}. In practice, there can be test vectors outside T_{fl_1}, which detects the fault fl_2. Referring to the AND gate example in Fig. 1b, the stuck-at-1 fault at y gets detected even with $a = 0, b = 0$, whereas none of the stuck-at-1 faults at a and b gets detected with this input. While this is not an issue for conventional Automatic Test Pattern Generation (ATPG), it is important to analyze if such collapsing is also appropriate in a FA context or not. More precisely, *we want to evaluate that if no fault simulation is performed at the fault location y (and a decision regarding its exploitability is made based on fault simulations at a or b), would it result in a fault negative?* As it turns out, this is not an issue for attacks based on random fault

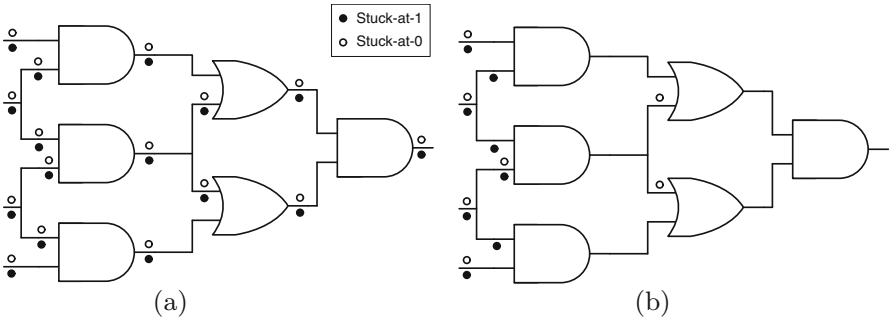

Fig. 2. Fault collapsing for a combinational circuit: **a** Uncollapsed faults (total 32); **b** Collapsed faults (total 15).

models (e.g., DFA). This is because such attacks require at most one input for a given fault location which can activate or propagate the fault. The definition of fault dominance guarantees this. For attacks considering biased and ineffective faults, however, such dominance-based collapsing may result in a slight variation in the bias. For example, the stuck-at-1 fault at the output y of the AND gate will result in correct computation for input value $(a = 1, b = 1)$, and faulty computation for $(a = 0, b = 0)$, $(a = 0, b = 1)$ and $(a = 1, b = 0)$. On the other hand, if decision regarding this fault location is made based on the stuck-at-1 fault at a, there will be faulty computation for $(a = 0, b = 1)$ and correct computation for $(a = 0, b = 0)$, $(a = 1, b = 1)$ and $(a = 1, b = 0)$. Similar observations can be made for fault simulation at b. Although it is an approximation to use the fault simulations of a or b to decide about leakage at y, the value dependency of the fault persists. Any value-dependent bias in fault is sufficient for an attack. Hence, the collapsing remains sound even for FA.

Reduction in Fault Set Fault collapsing at the gate-level provides a certain amount of reduction in the size of the fault space for single stuck-at faults. Figure 2 shows a simple illustration of this claim. Further, Table 3 provides the counts for the collapsed and uncollapsed fault lists for an unprotected AES implementation, as well as a TI implementation of PRESENT and a SIFA-protected implementation of PRESENT (ref. column 2–3). The fault lists are obtained by running a complete ATPG in full-scan mode over the circuits using Synopsys TetraMAX.[4] We have also provided the fault coverage statistics over the circuits. Fault coverage is the ratio of detected fault count and total (collapsed) fault count. Although in these cases, the fault coverage is 100%, in certain situations, fault coverage may go below 100% as some faults may remain undetectable even after an ATPG run. Such undetectable faults, however, do not influence FA testing as undetectable faults can never corrupt the ciphertexts. A full-scan ATPG converts the sequential circuit to a combinational one and labels those faults as detectable, which reaches some circuit register. In a block cipher datapath, if a fault reaches a state register, it

[4]The syntheses were performed using Synopsys Design Compiler and DFT Compiler (with STMicroelectronics CMOS65—a 65nm technology library due to STMicroelectronics). No area/timing optimization was imposed during synthesis. All Synopsys tools utilized in this work are under registered trademarks of Synopsys Inc (https://www.synopsys.com).

Table 3. Fault collapsing with gate-level and algorithm-level equivalence.

Hardware	#Uncollapsed fault-list	#Collapsed fault-list	%Fault-coverage	#Collapsed fault-list after algorithm-level equivalence
Unprotected AES	26,358	23,560	100%	660
TI-PRESENT	22,049	17,918	100%	1051
ANTISIFA	66,489	54,147	100%	3182

also reaches the ciphertext with high probability. Hence, enlisting detectable faults based on the full-scan circuit is sound.

Handling Bit-Flip Faults So far, we have discussed bit stuck-at faults on the nets of a circuit. It is also common to consider single bit-flip faults in fault attacks. The list of bit-flip faults is decided based on the list of stuck-at faults, as the fault-list contains every feasible single-bit fault location. A bit-flip fault for a net can be expressed as the conjunction of stuck-at-0 and stuck-at-1 fault.

Handling Multi-Bit Faults FAs also exploit multiple-bit fault models such as byte/nibble faults. Considering every possible multiple-bit fault would result in a fault space which is exponential over the single-bit fault space. Instead, we utilize certain features of the practical faults to restrict the fault space. *Most practical faults only corrupt certain consecutive bits in a register. Hence, we only consider faults within a byte, a nibble, or (in rare cases) within multiple consecutive bytes.* Further, *multi-bit faults are captured only at the register boundaries*. This is derived from practical observations. Even single-bit faults fan-out to multiple bits at a register [16,71]. A multiple-bit fault inside the combinational path would eventually result in a single/multiple-bit fault at some register boundary. Overall, considering all single-bit faults in the combinational path and multiple-bit faults at register boundaries should holistically cover most of the feasible faults in a target implementation.

5.2. *Fault Equivalence in Block Ciphers*

Block ciphers are constructed by repeating basic sub-blocks (such as S-Boxes and diffusion layers) several times. Such sub-blocks are equivalent in terms of fault attacks with respect to the attack complexity. For example, all 16 S-Boxes in an AES round have similar fault propagation properties and hence, similar attack complexities. Such equivalences can be exploited to drastically reduce the fault space. The idea is to deduce such equivalences from an unprotected version of the cipher under test (preferably a high-level algorithmic representation as used in automated fault attack tools such as ExpFault [72]). Such equivalence relations are used later to test protected implementations.

Definition 3. (*Fault Equivalence in Block Ciphers*) Two fault locations fl_1 and fl_2 according to a specific fault model are considered equivalent if they result in attacks with the same complexity. The attack complexity is defined as a tuple $\langle Rm, Eval \rangle$ where Rm denotes the exhaustive key search complexity after the attack and $Eval$ denotes the complexity of associated key guessing operation.

The definition of fault attack complexity above follows the one defined in ExpFault [33]. One should note that this definition mentions the fault locations and does not comment on fault values. The fault model specification abstracts the value of a fault (byte/nibble faults are usually multi-valued). For example, in a byte fault model, every fault value at a specific location is considered equivalent, and showing exploitability for one fault value pair is sufficient. For biased faults, every statistical bias at a fault location is equivalent.

Cipher-Level Equivalence: An Example We consider the AES block cipher to illustrate the utility of such equivalences. If a byte fault is injected at the input of a 9th round S-Box, it results in an attack recovering 32 key bits. We used the automated fault analysis tool ExpFault [72] for exploring all byte fault locations at the input of the 9th round S-Box operation.[5] Every byte location results in an attack that requires an exhaustive search of 2^8 (i.e., $Rm = 2^8$). For the evaluation of the keys, at most, 32 key bits have to be guessed simultaneously, making the key guessing complexity $Eval = 2^{32}$. Hence, all 16 byte locations (i.e., S-Box) inputs are considered equivalent, and testing one of them should suffice. Any byte fault between the 8th and 9th round MixColumns is equivalent to each other. We also note that FA countermeasures usually do not destroy the structural similarities within the original cipher structures. Hence, deciding the equivalence over an unprotected implementation and using those exploitable locations for testing the protected implementations works fine.

To further illustrate the concept of cipher/algorithm-level fault equivalence, we now use graphical representations generated from the ExpFault tool (called *Cipher Dependency Graph* or *CDG* in ExpFault's terminology). Although such graphs are not among the normal outputs of ExpFault, they can be generated for debugging purposes from the version of the tool we used. Figure 3 displays one such graphical representation of the last two rounds of AES. Each node here corresponds to a state bit. Each topological layer in the graph represents the input of a sub-operation (i.e., `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`). The S-Box and `MixColumns` layers are represented as complete subgraphs, and red nodes represent the key bits. The directions of the arrows are towards the ciphertext, and the last topological level represents the ciphertext.

Each topological layer (except those involving key addition) of the AES CDG contains 128 nodes. Starting from the 9th round input (as we consider the fault injection at the 9th round), the entire CDG contains four subgraphs disconnected from each other. For the sake of representation, we place these four subgraphs as Fig. 3a–d. Without loss of generality, we consider two independent fault injection scenarios at two different byte locations in subgraph Fig. 3b, c. The fault propagation path for Fig. 3b is colored blue, and the other is colored green. Both subgraphs involve the same number of key bits from the last round, which this attack can extract. Moreover, both graphs are isomorphic to

[5] There are 16 such locations.

each other if we ignore a few nodes from the first topological layer. The complexity components *Rm* and *Eval* are the same for these two fault injections due to the isomorphic graph structures. Hence, these two fault locations can be considered equivalent, and analyzing one would be sufficient. The CDG structure confirms that all 16 S-Box inputs (input nodes to the 8×8 complete subgraphs in the first two layers) are equivalent.

5.3. *Putting it All Together*

Overall, we go by the following steps:

- Perform the algorithm-level fault collapsing using the ExpFault tool. Get the list of equivalent fault locations and select only a single location from each equivalence class. Such locations are described as the inputs to some sub-operation (e.g., `SubBytes`, `ShiftRows`, `MixColumns`) by ExpFault [72].
- Select the module which implements the sub-operation specified in the previous step. For hardware/bitsliced implementation, perform gate-level fault collapsing for this module only and populate the fault list. Simulate each fault from this fault list by stitching extra gates at the fault locations. For example, a bit-flip fault can be generated by stitching a 2-input XOR gate at the fault location. One input of the XOR gate is attached to the fault location, whereas the other input is set to 1 to flip the value at the fault location. This strategy is also used for generating stuck-at and other fault models by changing the gate type (e.g., using AND/OR gates). For software implementations, target every instruction within this module using the GDB-based methodology described in "Appendix A".
- Acquire simulation data for each fault location and apply the DL-based leakage detection test.

Column 4 in Table 3 illustrates the outcome of such testing in terms of fault locations tested (for hardware implementations). Testing one S-Box per round for AES and PRESENT is sufficient, and the size of the corresponding fault set is significantly smaller than the entire fault space of the circuits.

6. Case Studies

This section presents the case studies used to evaluate the proposals made in the last few sections. Our evaluation set contains representatives from each of the countermeasure classes described in Table 1. Moreover, to establish the usefulness of the *compare-with-uniform* extension in Sect. 4.2, we present a scenario of mask-derandomization and evaluate the firmware of a Hardware Security Module (HSM) called SHE [76]. The redundancy, infective, and instruction countermeasure are implemented in software. The combined SCA-FA and SIFA countermeasures are implemented in hardware, except for CAPA [19] and M&M [21], which are simulated in Python. We implement these two countermeasures for KATAN-32 [77] block cipher and test representative fault locations at different building blocks to only verify the security claims from the papers. Additionally, we also test two recently proposed countermeasures—Friet [26], and DEFAULT [36], which are simulated in C. The SHE design is a hardware/software co-design where

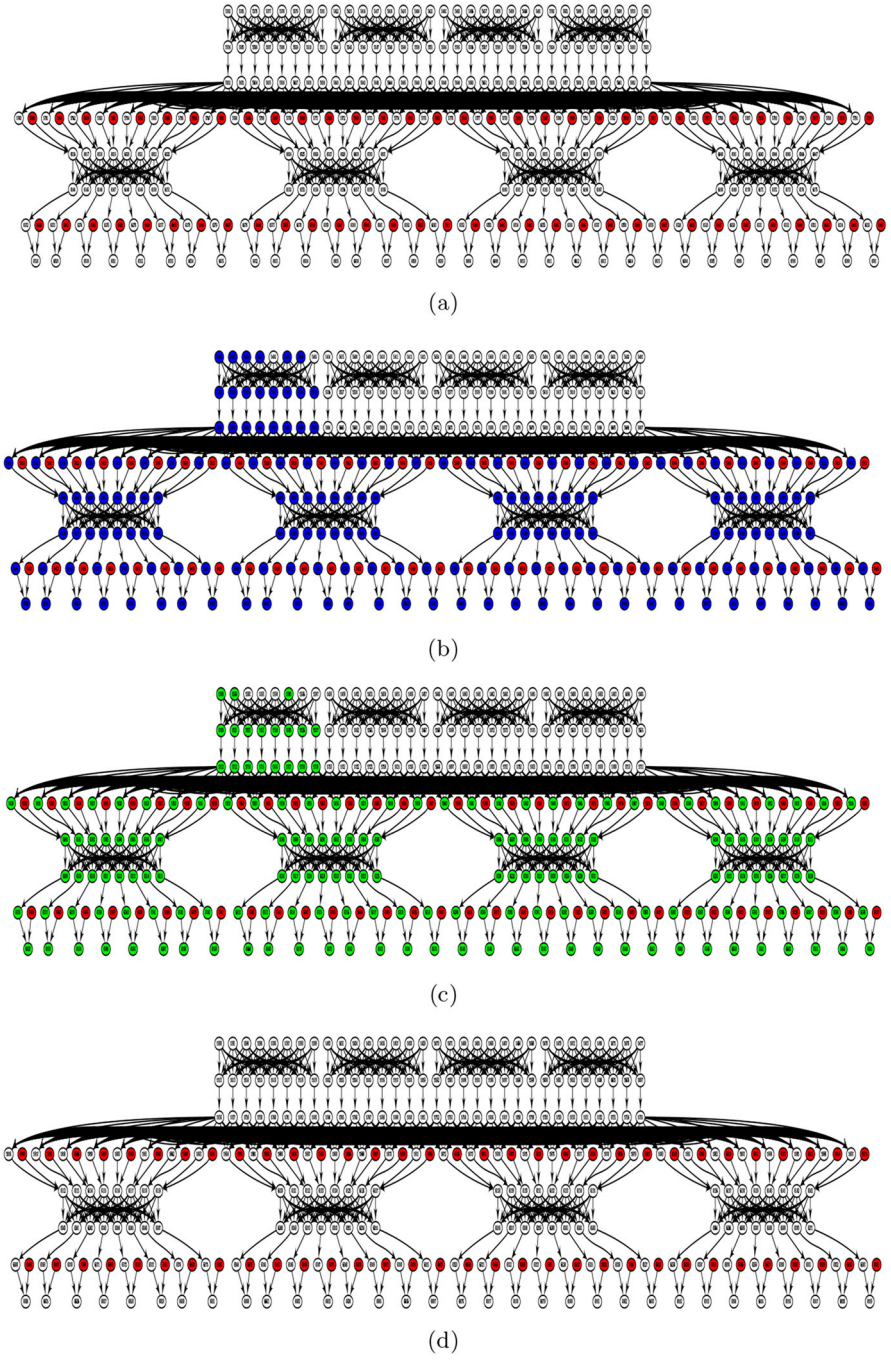

Fig. 3. Illustration: Cipher-level fault equivalence.

Table 4. Summary of results.

Countermeasures		1-byte Fault	Single Inst. Skip	Multi Inst. Skip	Skip-based Control Fault	SIFA Faults (Biased Bit-Flip Faults)*	SIFA Faults (Unbiased Bit-Flip at Gate Input)
Time/Space/ Information Redundancy; Infective	Simple time/space redundancy+	Secure	Secure	Secure	Secure	Insecure	Insecure
	1-bit parity [15] (information redundancy)+	Insecure	Insecure	Insecure	Secure	Insecure	Insecure
	Infective [14] (without noise)	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [14] (with noise)	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [25] (RIMBEN)	Insecure	Insecure	Insecure	–	Insecure	Insecure
	Infective [28]	Secure	Secure	Secure	Insecure	Insecure	Insecure
	Infective [24]	Insecure	Insecure	Insecure	–	Insecure	Insecure
Inst. Level	Idempotent Inst. [10]	Secure	Secure	Insecure	–	Insecure	Insecure
SCA+FA Combined	Masking [73]+ Classical FA Countermeasure	Secure	–	–	–	Secure	Insecure
	CAPA [19]***	Secure	Secure	Secure	–	Secure	Secure
	M&M [21]***	Secure	Secure	Secure	–	Secure	Insecure
	Friet [26]***	Secure	Secure	Secure	–	Secure	Secure

Table 4. continued.

Countermeasures		1-byte Fault	Single Inst. Skip	Multi Inst. Skip	Skip-based Control Fault	SIFA Faults (Biased Bit-Flip Faults)*	SIFA Faults (Unbiased Bit-Flip at Gate Input)
SIFA Counter-measure	AntiSIFA [74]**	Secure	–	–	–	Secure	Secure
	Impeccable Circuits II [16]**	Secure	–	–	–	Secure	Secure
Fault-Resilient Cipher	DEFAULT [36]***	Insecure	Insecure	Insecure	–	Insecure	Insecure
Security Module	SHE Firmware [75]	–	–	Insecure for faults in data transfer	–	–	–

*Bit-stuck-at faults are special cases of biased bit-flip faults
+Insecure against paired faults at the comparison and combined FA-SCA
**Secure up to a predefined security order
***Instruction-level faults are simulated at high-level by corrupting constituent bitwise-logical statements

the crypto-core is in hardware, and the rest of the computation is running as firmware in a soft-core processor (for simplicity, we check some parts of the firmware). Table 4 summarizes the outcomes of leakage assessment over the test suite for different fault models.

Due to the presence of the gate and algorithm-level fault equivalences, we only need to simulate faults for one S-Box per round at its inputs, outputs, or intermediates points for most of our test cases involving AES and PRESENT. To further (reasonably) reduce the number of locations to be tested, we target only the last three rounds (six rounds for infective countermeasures [14,28] as dummy rounds are present) of the ciphers in most of our test cases. This is because most fault attacks target only the last few rounds of block ciphers. Although attacks such as Fault Template Attacks (FTA) [22] exist, which are also applicable for middle rounds, their working principle is the same in every round. Therefore, checking the last few rounds suffice.

A summary of implementations tested, along with timing results, is given in Table 5. The experiments are performed on three systems with Intel Xeon processors, each having 64 processing elements. The code length here presents the total number of instructions executed (for X86-64 architecture). For instruction-skip experiments, we model up to 3 consecutive skips. The instruction-level countermeasure tested implements up to 2 redundant instructions and requires two consecutive skips to expose leakage. An attack discovered for SHE requires 3 consecutive skips. Therefore, we need not go beyond 3 skips in this work. However, this choice is empirical and can be increased to any value until the skips are consecutive. We present the average leakage detection time (fault

Table 5. Leakage detection time of the examples tested with Algorithm 3.

Software Examples	Code Size (# inst.)	#Checked Insts.	Avg. Leakage Detection Time/fault location (in seconds)	Hardware Examples	Total Fault Count (in full-scan) (Collapsed)	#Location Checked × #rounds	Avg Leakage Detection Time/fault location (in seconds)
Time Redundancy	83,270	532	564.8	Combined SCA + FA	17,918	1051 × 3	304.45
Parity [15]	50,544	597	425.94	AntiSIFA [17]	54,147	3182 × 3	302.63
Infective [14]	20,8380	1170	620.41	Impeccable Circuits II [16] (3-way red)	17,731	3744 × 1	300.38
Infective [25]	97,519	532	682.65	Impeccable Circuits II [16] (7-way red)	88,721	13,895 × 1	301.76
Infective [28]	16,6520	1052	868.82				
Infective [24]	90,491	532	573.26				
Idempotent Inst. [10]	96,593	1127	426.38				
SHE Firmware [76]	205	205	436.16				

simulation + learning) for each fault location. In the following subsections, we compare the results of the DL-based leakage detection with t-test. The t-test results are denoted as ALAFA.

6.1. *Infective Countermeasures*

We consider four infective countermeasures in our test suite.

Example 1. (**Infective Countermeasure** [14]) This infective countermeasure randomizes the outcome with an infection function upon detecting a fault. The protected implementation executes each round of AES two times—the first one contributes to actual encryption, and the second one is redundant. Furthermore, there are (optional) random "dummy" rounds (round computations over a random state changing at each encryption). Dummy round computations randomly occur between each actual and redundant round to confuse the attacker regarding the correct fault injection location. The nonzero XOR differential between actual and redundant computation is used to "infect" the state

during fault injection, which is further combined with the actual, redundant and dummy round computations. This results in a randomized ciphertext.

Our first experiment considers the countermeasure without the dummy rounds. Without loss of generality, we describe fault injections at the 9th round of AES state. Leakage is observed in this case. Figure 4a compares the outcome from DL-FALAT to that of ALAFA [30] in terms of absolute t-values. The byte-wise testing performs better for both ALAFA and DL-FALAT in this case. The leakage has been detected with 1400 ciphertexts (when the line crosses the red region at $t = 4.5$) for DL-FALAT, while ALAFA requires 7000 ciphertexts.

Next, we figure out the leakage-orders for the DL-FALAT, for which we perform the SA (ref. Section 3.4). The SA is reported here for the 9th round exploitable fault location. The first set of leaky points (i.e., the MI) are the bytes [4, 5, 6, 7] from the 16-byte ciphertext (Fig. 4b). MI sets are constructed using the average sensitivity of all points in the trace as threshold Th_{MI} (red lines in Fig. 4b–e), as described is Sect. 3.4. The ciphertext count required to expose this leakage is 1400. An analysis of the MI set reveals this leakage to be multivariate (Fig. 4f) as at least 3 bytes in MI are required for learning the leakage. We iteratively continue by removing the features in the MI set and increasing the ciphertext count to expose other leakage points. The second set of leakage points is ([12, 13, 14, 15]). Exposing these points need no further increase in the ciphertext count, giving some hint that the leakage-order of the first two sets might be equal (Fig. 4c). The next set of leakage points getting exposed are [0, 1, 2, 3], for which 40,000 ciphertexts are required (ref. Figure 4d). Finally, the third leakage column gets exposed with ciphertext count of 20,0000 (ref. Figure 4e). Leakage is multivariate for all these MI sets. The variation in ciphertext counts for different leakage sets indicates that the statistical order may not be the same for all of them, which is supported by the actual attack presented in [28]. Precisely, column [4, 5, 6, 7] and [12, 13, 14, 15] have (bivariate) leakage-order 1, column [0, 1, 2, 3] has an leakage-order 2, and the third column has order 3 (ref. Equation 5 in [28]). This experiment justifies the importance and validity of our leakage interpretation step.

Example 2. (**Countermeasure** [14] **with Dummy Rounds**) In this example, the leakage detection is performed on [14] with the dummy rounds included. Dummy rounds induce noise in fault injection as the attacker cannot determine the exact round of injection. The amount of noise depends on the dummy round count ($\#dum$). For reasonable dummy round counts 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, if we target AES 9th round. Figure 5a presents the leakage profiles for the number of dummy rounds for both ALAFA and DL-FALAT. DL-FALAT outperforms ALAFA by a huge margin for all noisy cases. Even for a large ciphertext count of 20,0000, ALAFA fails to detect the leakage while DL-FALAT succeeds. The leakage interpretation results are very similar to that of the previous example.

Example 3. (**RIMBEN** [25]) RIMBEN (Random Infection based on Modified Benes Network) detects the presence of a fault by taking the differential of a cipher and a

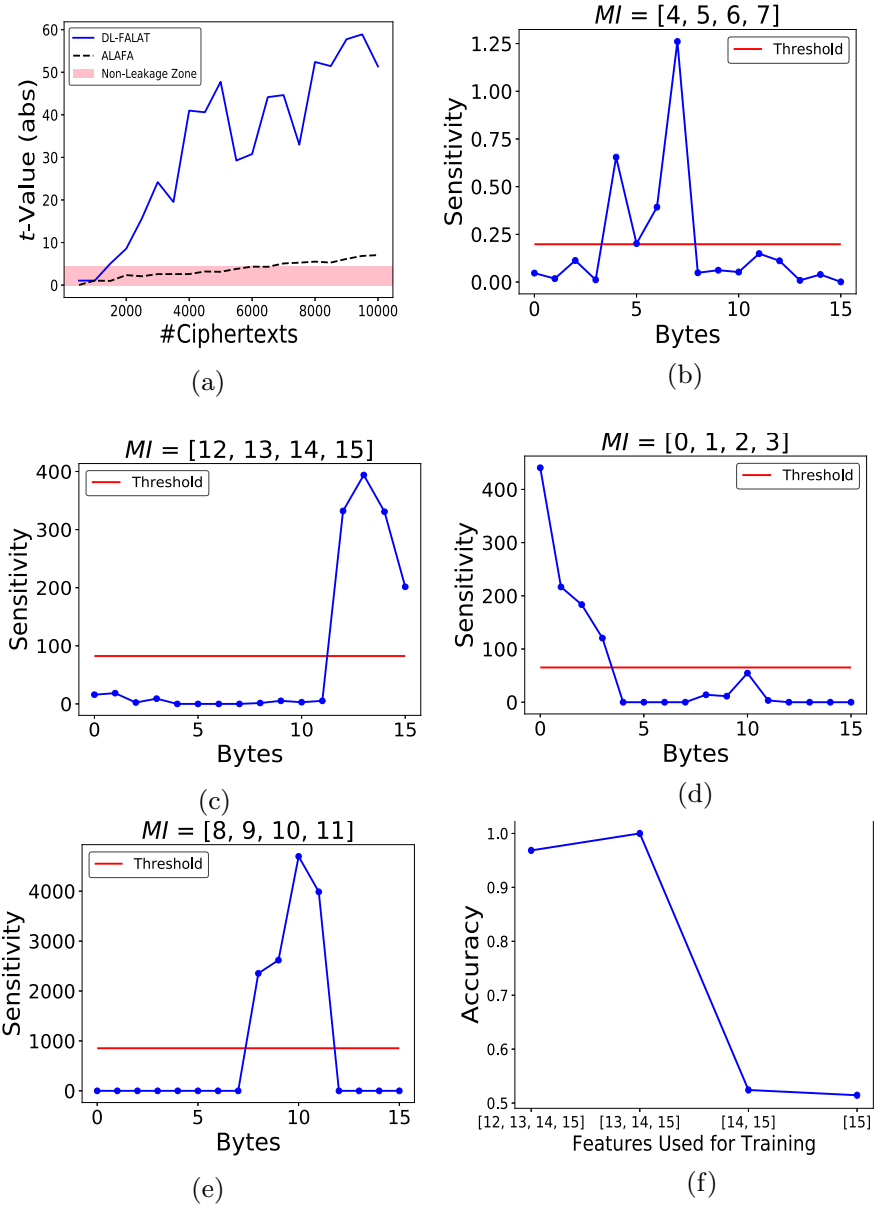

Fig. 4. Leakage of Infective countermeasure [14] with single-byte fault: **a** Variation of absolute t-test scores for DL-FALAT and ALAFA with respect to ciphertext count; **b** SA results for the first iteration; **c** SA results for the second iteration; **d** SA results for the third iteration; **e** SA results for the fourth iteration; **f** MI analysis for the first iteration .

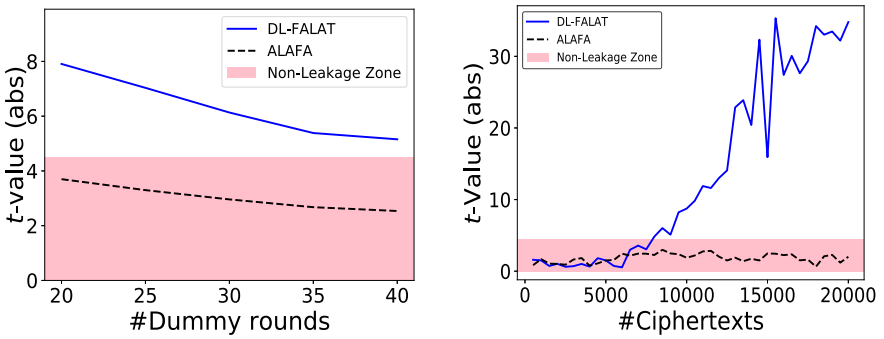

Fig. 5. Comparative analysis of DL-FALAT with ALAFA: **a** Infective countermeasure [14] with dummy rounds and a single-byte fault. The absolute values of t-statistic have been plotted for different count of dummy rounds $\#dum$. The amount of noise increases with the increase in $\#dum$; **b** Variation of absolute t-test scores for DL-FALAT and ALAFA for RIMBEN countermeasure .

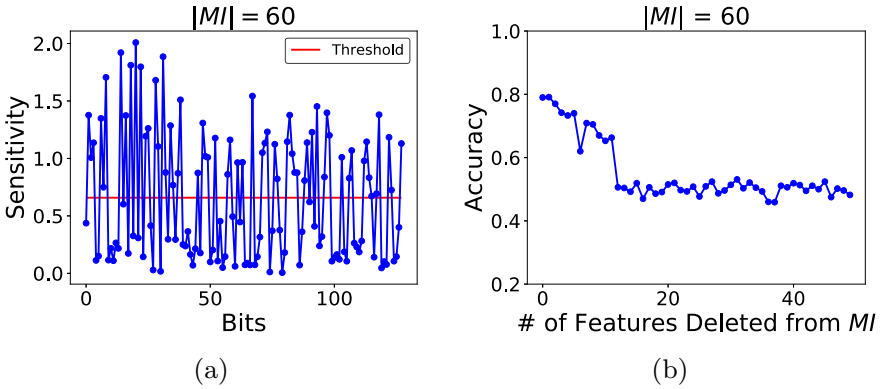

Fig. 6. Infective countermeasure RIMBEN [25] with single-byte fault: **a** SA analysis results from first iteration; **b** MI analysis results from first iteration.

redundant computation state at each round during encryption. The fault is 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 generated from the fault differential ΔC, utilizing a preprocessing logic and two consecutive Benes networks. The random bitstring outputted by this construction has 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. We consider a protected AES implementation for which $N = 128$.

Figure 5b illustrates the analysis on RIMBEN. We report results for 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. The bit-level results (presented here) require lower ciphertext count. The DL-FALAT observes the leakage within 8000 ciphertexts.

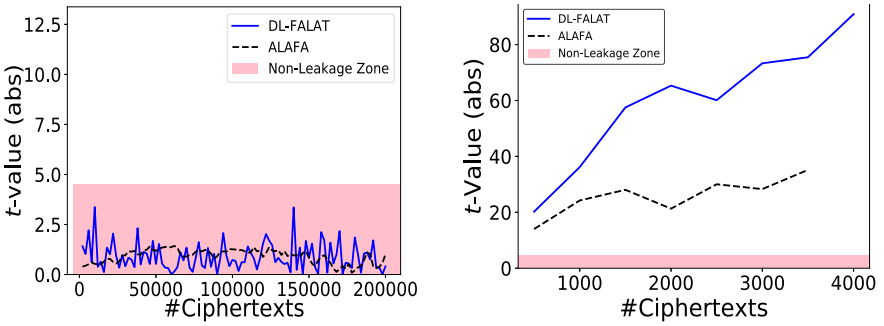

Fig. 7. a Infective countermeasure [28] with single-byte fault: DL-FALAT and ALAFA leakage profile; **b** Infective countermeasure [28] with instruction-skip-based loop-abort: DL-FALAT and ALAFA leakage profile .

In contrast, ALAFA cannot detect any leakage.[6] ALAFA fails because all 128 bits participate in decision-making. Leakage detection with 128-th-order analysis would require an impractically large trace count to converge.

While performing the leakage interpretation, the first step of SA reveals the set of 60 points, as shown in Fig. 6a. *However, the points not included in this MI also have some observable sensitivity, which is exposed in the next iteration of analysis.* The size of the MI set is 60. This is because the HW of the masking string in RIMBEN is 64. Therefore, knowledge of roughly half (60) of the ciphertext bits is sufficient for the DL model to decide the boundary between two classes with better-than-random accuracy. The analysis of MI shows that after removing 10 points the validation accuracy becomes 0.5 (Fig. 6b). This implies that the leakage is highly multivariate, as considering even 50 points keeps the entropy of the mask sufficiently high, refraining from classification. Attacks reported in [78] also exploit this low HW feature of RIMBEN.

Example 4. (**Infective countermeasure** [28]) This example considers the infective countermeasure proposed at CHES 2014 [28] as an improvement over [14]. Suppose a single/multi-byte data-corruption happens in any cipher, redundant or dummy round. In that case, the protected cipher outputs a fresh random string instead of a randomized infected intermediate state as in [14]. The countermeasure is first tested for a single-byte fault model. As can be seen in Fig. 7a, no leakage is observed in this case, both by DL-FALAT and ALAFA.

An instruction-skip in this countermeasure corrupting the loop counter variable (during the last 2 rounds) creates a univariate information leakage, as shown in Fig. 7b. The cipher outputs the input of the 10-th round instead of a random string during such loop abort faults. Faults at other control variables, such as the "if" condition, which checks

[6] 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 number of subsets up to order 128 is 2^{128}, which is infeasible to cover. So we consider one case where the order of test is 128. The result shown in the plots is for test order 128.

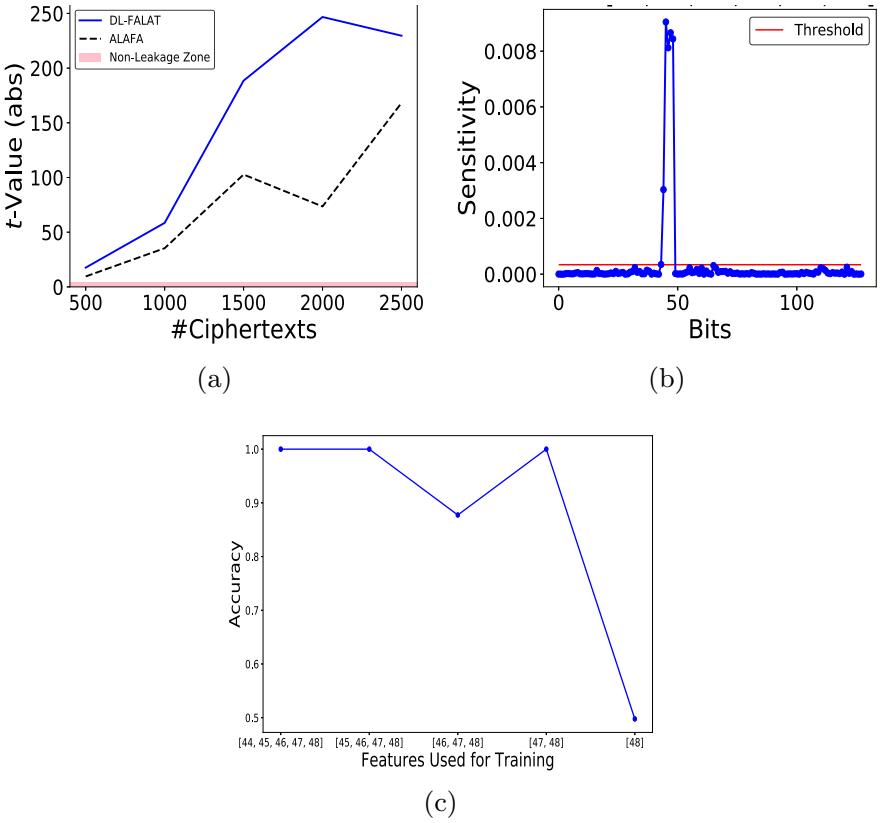

Fig. 8. Infective countermeasure [24] with single-byte fault: **a** Variation of leakage with ciphertext count for DL-FALAT and ALAFA; **b** SA for the DL-FALAT leakage for one iteration of iterative leakage interpretation; **c** Analysis of the MI set in one iteration of leakage interpretation.

if the state differential is nonzero, also lead to information leakage. Due to its structure, such control variables are predominant in this countermeasure.

Example 5. (**Infective countermeasure** [24]) This countermeasure utilizes an infection function having a linear diffusion function followed by a randomized nonlinear mixing function. Both ALAFA (for $d = 2$) and DL-FALAT indicate leakage for single-byte fault model (Fig. 8a).[7] The leakage is multivariate, and DL-FALAT automatically discovers that. As shown in Fig. 8b, 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 features in one MI set exposes another set of leakage points without incrementing the dataset size. This indicates that the order of leakages might be the same throughout the ciphertext. Analysis of an MI set is presented in Fig. 8c, indicating a multivariate leakage.

[7] Note that for this countermeasure, leakage has been observed while the ciphertexts are considered bitwise.

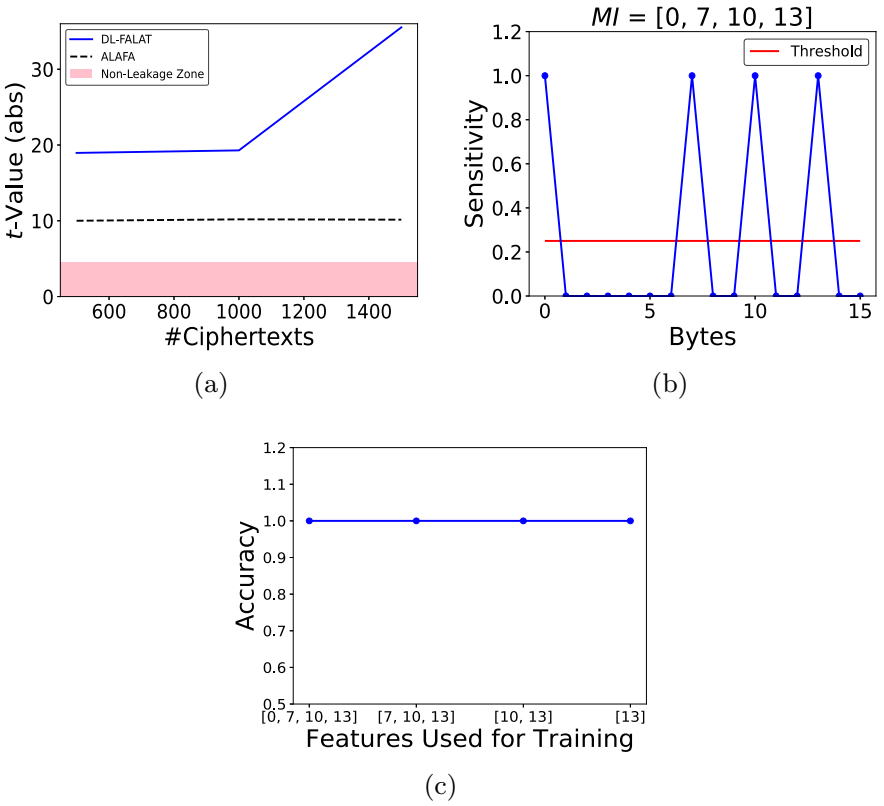

Fig. 9. Leakage analysis and leakage interpretation for the parity example: **a** Comparative leakage analysis of DL-FALAT and ALAFA; **b** SA of leakage; **c** *MI* analysis indicating univariate leakage .

6.2. *Detection Countermeasures*

We consider a simple time redundancy countermeasure and an information redundancy countermeasure using 1-bit parity.

Example 6. (**Simple two-way redundancy** [15]) Our first example utilizes simple two-way redundancy for error-detection. This countermeasure returns $\perp$ (or a random string) if there is a mismatch between two redundant branches at the ciphertext. Among different fault models, here we mention the case with single-byte fault. All faults get detected for a single-byte fault in one computation branch, and the constant output $\perp$ is indistinguishable for two different fault values (f_1, f_2). Hence, no leakage is caused for single-byte faults. 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 at the end. In this paper, we do not consider multiple-cycle (i.e., multiple non-consecutive faults) fault scenarios. However, the leakage assessment experiments remain unchanged for those cases.

Example 7. (**One-bit parity** [15]) Next, we consider 1-bit parity-based error-detection implemented on AES. The countermeasure is bypassed for 50% of the byte faults having even parity and hence insecure. To quickly discover a leaky fault pair (f_1, f_2) for applying Algorithm 1, we use the preprocessing step mentioned in [30]. Figure 9a presents the leakage profile. The analysis of the MI set indicates 4 leakage points for a 9th round fault injection. The leakage is univariate, as the learning can be performed with high accuracy even with a single feature point (Fig. 9b, c). One should note that although the leakage is observed in this case, finding a leaky fault is rare for a well-formed code-based redundancy. Hence, even if there is leakage for code-based countermeasures, the exploitability depends on the rarity of the leaky faults.

6.3. *Instruction-level Countermeasures*

Example 8. (**Idempotent instructions** [10,11]) The instruction-level countermeasures against FAs rely on the fact that an adversary can only skip a certain number of consecutive instructions simultaneously. The instruction redundancy protects against output corruption for a certain number of consecutive skips. To test the applicability of DL-FALAT for such instruction-level countermeasures, we implement the scheme proposed in [10] for AES without any algorithm-level protection. The scheme in [10] replicates some machine instructions in a code multiple times if there is no impact of replicating these instructions on the outcome of the code. Such instructions are called *idempotent instructions*. For example, a `load` instruction moving data from a memory location to a register can be treated as idempotent. Such a `load` instruction does not affect the computation even if it is repeated (consecutively) multiple times. Many arithmetic and logical instructions in X86-64/ARM architecture are idempotent. Also, many non-idempotent instructions can be implemented by combining multiple idempotent instructions [10,11]. In our example, each idempotent instruction is duplicated once. The instruction-skip experiments are performed with the GDB-based tool described in "Appendix A". While performing a single instruction-skip, we observe no leakage. However, leakage can be observed if two consecutive instructions are skipped simultaneously (ref. Figure 10). Leakage interpretation experiments confirm that the leakage is univariate.

6.4. *Leakage Assessment for SIFA*

In this subsection, we test SIFA leakage according to the enhancements proposed in Sect. 4.1. We first validate an FA-protected (with time redundancy) unmasked AES, followed by a combined SCA-FA-protected PRESENT (hardware implementation). Next, we validate hardware implementations of two recently proposed SIFA countermeasures, AntiSIFA [74] and Impeccable Circuits II [16].

Example 9. (**FA-protected AES** [15]) In our first example (FA-protected AES), we simulate a stuck-at-0 fault ($pr_{0\to0} = 1$, $pr_{1\to1} = 0$). The leakage profile for this attack is shown in Fig. 11a, which presents the (univariate) leakage profile for both ALAFA and DL-FALAT. We observe a similar leakage for $pr_{0\to0} = 0.75$, $pr_{1\to1} = 0.25$. However, an experiment with $pr_{0\to0} = 0.5$, $pr_{1\to1} = 0.5$ (Fig. 11b) does not show any leakage

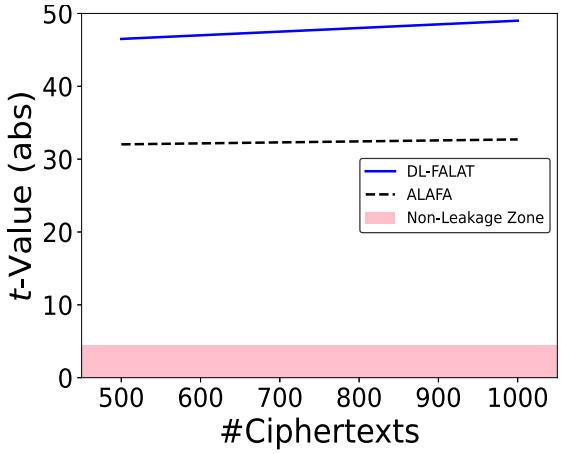

Fig. 10. Instruction-level countermeasure with duplicate idempotent instructions [10]. Two consecutive skips expose univariate leakage .

even though there is a mix of correct and faulty ciphertexts. This is expected and shows that having many ineffective faults does not always indicate the chances of SIFA. This observation also emphasizes why a dedicated test for SIFA is needed instead of only relying on the absence of ineffective faults. Such a conservative strategy underestimates the security in certain cases when only unbiased bit-flips happen on a target.

Example 10. (*Masking and FA protection* [13]) Next, we test a combined countermeasure on PRESENT [79] that uses Threshold Implementation (TI) [73] for SCA protection and simple time redundancy (two identical computations followed by check at the end) for FA protection. *We observe no leakage with single-bit stuck-at-0 faults in the S-Box input (one share is corrupted) or linear layer input. The masking here changes the impact of a stuck-at fault similar to the situation where* $pr_{0\to0} = 0.5$, $pr_{1\to1} = 0.5$. However, we observe leakage (ref. Figure 11c) while injecting inside TI equations (we injected a single bit-flip fault in a register at the middle of the S-Box).

Example 11. (**SIFA countermeasure: ANTISIFA** [17]) We next focus on two SIFA countermeasures from [16,17]. The first countermeasure, 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 at the end of each round. Moreover, the error-correction logic is instantiated multiple times to make it fault-tolerant. The original proposal presents an example implementation with single-bit error-correction. While tested with single-bit faults (even the one inside S-Boxes, as described in the last example), it is found that the countermeasure successfully prevents SIFA. This supports the claims made in the original paper (ref. Figure 12a).

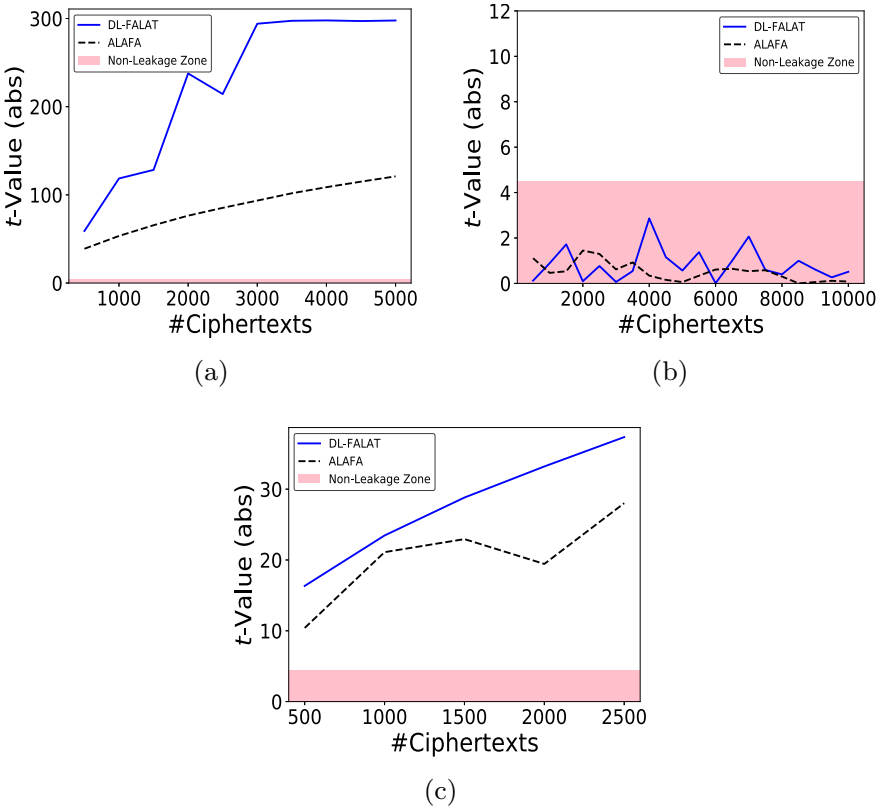

Fig. 11. **a** Variation of leakage with ciphertext count for SIFA (on AES with redundancy) with $pr_{0\to 0} = 1$, $pr_{1\to 1} = 0$; **b** Variation of leakage with ciphertext count for SIFA (on AES with redundancy) with $pr_{0\to 0} = 0.5$, $pr_{1\to 1} = 0.5$; **c** Variation of leakage with ciphertext count for SIFA on masking .

Example 12. (**SIFA countermeasure: Impeccable Circuits-II** [16]) The final experiment on SIFA is for an open-source hardware implementation of the Impeccable-Circuits II [16]. This test case is implemented on CRAFT [80], which is a tweakable block cipher engineered for incorporating code-based countermeasures. The main idea of this countermeasure is to throttle the negative impacts of fault propagation by introducing 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. We test the open-source hardware implementation for this countermeasure,[8] having single-bit error-correction (3-way redundancy), and 2-bit error-correction (7-way redundancy). In the experiments, we test for different single-bit and multi-bit faults. Here, we only mention results SIFA testing with stuck-at-0/1 faults for a single round. For 3-way (resp. 7-way) redundancy, it is found that single-bit (resp. 2-bit) faults get corrected. The results are depicted in Fig. 12b, c, respectively. Such results establish that the proposal in [16] abides by its claims.

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

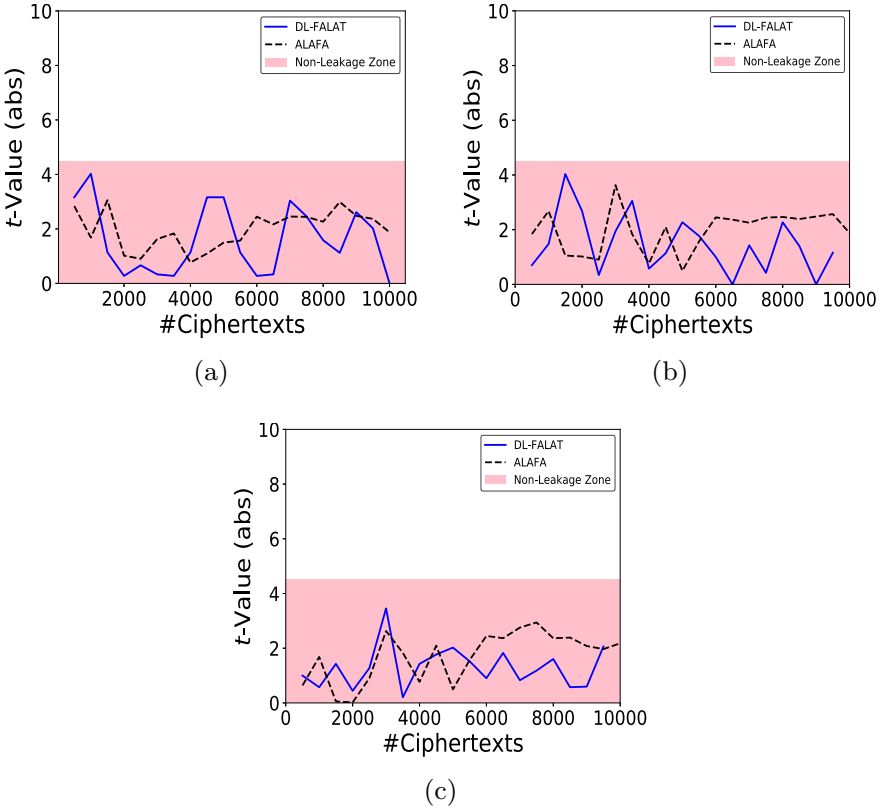

Fig. 12. Evaluating SIFA countermeasures: **a** AntiSIFA [17] with single-bit fault; **b** Impeccable Circuit II [16] with single-bit error-correction and single-bit fault; **c** Impeccable Circuit II with 2-bit error-correction and 2-bit fault .

6.5. *FA Security of Other Combined Countermeasures*

Example 13. (**CAPA** [19]) CAPA [19] and M&M [21] are two recently proposed combined countermeasures claiming security against combined SCA-FA adversaries. However, in this paper, we only evaluate their FA security. CAPA adapts multiparty computation (with active and passive security guarantees) for a block cipher. The computation is divided into *tiles*, with each tile representing one party of the computation. The communications between tiles are limited and secured using extra randomness (called *Beaver triples*). The input is first shared into d independent shares to provide SCA security. Each share is processed within a tile. The input is multiplied with a (or multiple) randomly generated, nonzero *hash key* α to generate information-theoretic hashes. The hash key is also maintained in a shared manner. CAPA computes on the shared values and their corresponding hashes for each gate. The *hash check* is performed during the computation of the nonlinear gates, and the computation aborts upon finding a mismatch. The active (i.e., FA) security stems from the fact that the hash key is changed randomly at every cipher execution. In order to bypass the hash check, an adversary must inject a

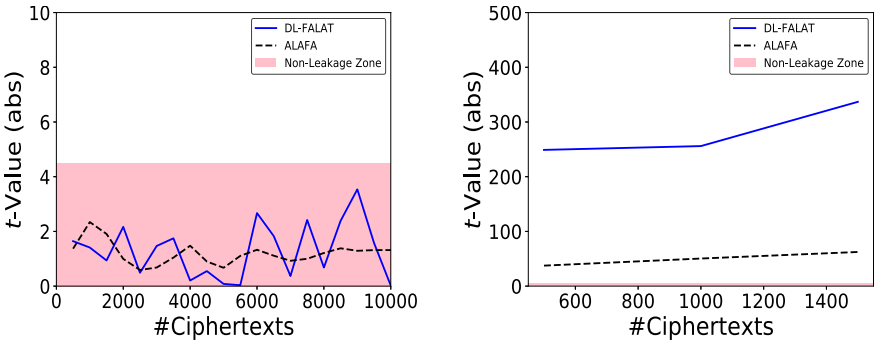

Fig. 13. **a** CAPA [19] with bit-flip SIFA fault during an AND gate computation: DL-FALAT and ALAFA leakage profile with varying ciphertext count; **b** M&M [21] with bit-flip SIFA fault during an AND gate computation: DL-FALAT and ALAFA leakage profile with varying ciphertext count.

fault such that the hash value of the correct and the faulty states are equal. This happens with probability 2^{-sm}, where m is the number of hash keys and $GF(2^s)$ is the finite field for the cipher [19].

In order to validate CAPA, we implement it for the KATAN-32 [77] block cipher in Python. KATAN-32 is a 32-bit block cipher with a simple round function mostly consisting of shift operations, with only 4 AND and 8 XOR operations per round. Additionally, there are 4 XOR operations in the key schedule. In our implementation of KATAN with CAPA, each basic gate is replaced with an equivalent CAPA gate. We also maintain $m = 8$ hash keys to provide a practical fault detection capability. The computations are performed over the field $GF(2)$. Faults are simulated for input, output and intermediate computation of one representative CAPA gate from each gate-type within a round. CAPA provides security against bit-stuck-at, bit-flip, and byte faults. Next, we perform a SIFA evaluation. SIFA evaluation is interesting as SIFA was not explicitly mentioned in the adversary model of CAPA. We found that CAPA provides security against SIFA faults. The result of one such experiment is depicted in Fig 13a. The fault model tested for this specific experiment is a single-bit flip at one of the input shares of a CAPA AND gate. If the fault propagates through the AND gate, it would corrupt (or not corrupt) the AND output depending on the data on the other input of the AND gate (in other words, it would result in a data-dependent ineffectivity of the fault). However, no leakage is observed in this case, even with such data-dependent ineffective faults. We investigated the reason behind this SIFA resistance of the scheme. In CAPA, the AND computation is performed with the help of random Beaver triples $\langle \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \rangle$, where $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ denote shares of bit variables a, b and c, respectively. For a valid Beaver triple $c = ab$. During the AND computation, the shares are blinded with $\boldsymbol{a}$, $\boldsymbol{b}$. These blinded shares are next broadcasted among all the tiles. The hash check is performed after this broadcast operation, and if the check passes, the remaining computations for the multiplication are performed. *Such hash check before the multiplication prevents SIFA, as no fault is allowed to pass through non-linear gates in this case, which is the*

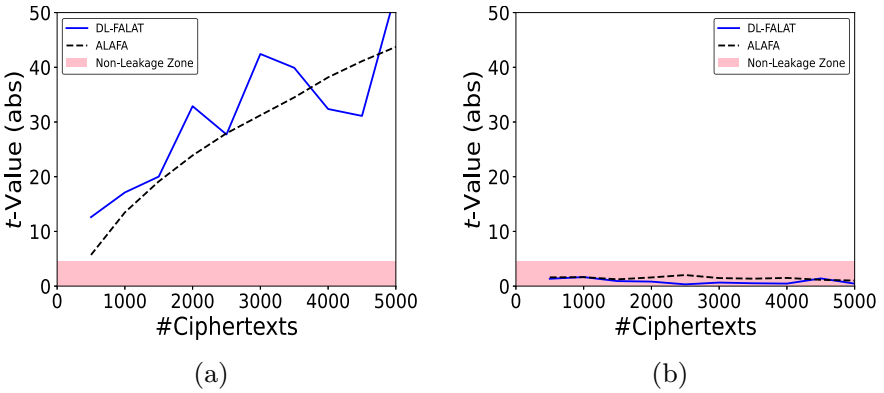

Fig. 14. **a** SIFA testing on Friet [26] with bit-flip SIFA fault during an AND gate computation: DL-FALAT and ALAFA leakage profile with varying ciphertext count; **b** SIFA testing on SIFA-protected Friet [26]: DL-FALAT and ALAFA leakage profile with varying ciphertext count .

sole cause behind attacks like SIFA and FTA [22]. *Operations until the hash check are linear. The result remains the same even for biased bit-flip faults.* In a nutshell, CAPA is found secure for the fault models and locations tested in this work.

Example 14. (**M&M** [21]) The M&M countermeasure adopts concepts similar to CAPA but is lightweight from an implementation perspective. The generation and maintenance of hashes throughout the computation are similar to that of CAPA. However, instead of checking hashes at each nonlinear gate, M&M performs an infective computation at the end. While it makes M&M lightweight compared to CAPA, it cannot provide the SIFA security anymore. To validate this, we perform the SIFA evaluation with DL-FALAT on a KATAN-32 implementation having M&M. As shown in Fig 13b, DL-FALAT indicates leakage in this case. This is, however, not surprising as M&M already excludes SIFA from its security claims. Overall, M&M respects its claimed security.

Example 15. (**Friet** [26]) Friet [26] proposes the concept of *code-abiding permutations*, where a permutation operates over a codeword and outputs a codeword from the same code. The work also presents a concrete instantiation called Friet-P, embedding a parity code. Friet-P builds on a permutation called Friet-PC having 384-bits organized into three 128-bit *limbs* (denoted as a, b and c). The operations (XOR, AND, and bitwise circular-shift) are performed between the limbs. A round of Friet-PC consists of two *limb transpositions* (linear), a *round constant addition* (linear), two *mixing* steps (linear), and a *nonlinear* step. The Friet-P permutation adds an extra limb with Friet-PC called the *parity limb* (d). After every step of the permutation round, the parity limb maintains the invariant $d = a \oplus b \oplus c$. This invariant is used for fault detection. An Authenticated Encryption (AE) scheme has been constructed based on Friet-P. The Friet-P permutation, however, does not protect against SIFA. In order to achieve SIFA protection, authors have utilized the ideas proposed in [81], requiring masking and error-detection. Every fault in this SIFA countermeasure becomes effective, eliminating SIFA.

Although Friet contains both masking and FA countermeasure, we only evaluate the claims made concerning FA (including SIFA). Friet claims security against non-SIFA attacks if the fault is limited to only one limb. It uses 1-bit parity, and there are no operations (except bit shifts) within a limb. Therefore, claiming security for faults in one limb is equivalent to claiming single-bit security corresponding to 4 bits, each belonging to a separate limb. In order to verify the security, we used the C code provided by the authors of the Friet paper.[9] The C code only provides the Friet-PC permutation. Therefore, we implemented the Friet-P and its SIFA-protected version. The code is implemented only for simulated evaluation with faults and not checked for SCA.

The leakage test is run over 128-bit tags from Friet-P, a recommended tag size in Friet [26]. We assume that the permutation is initialized with 256 bits of key, and the rest of the input bits are public [26]. To validate the security for non-SIFA faults, we inject single/multi-bit faults in one limb at a time. The key is fixed, and the fault values vary. The permutation is secure for this test as it outputs $\perp$ in all cases, irrespective of the fault value. Next, we perform SIFA testing on the permutation without SIFA protection. SIFA occurs for both stuck-at and bit-flip faults. The inputs of the AND gates are vulnerable to bit-flip faults resulting in SIFA, while the stuck-at faults work for any location. The result of the SIFA experiment with bit-flip faults is presented in Fig. 14a. Finally, we perform SIFA testing on the SIFA-protected version. As shown in Fig. 14b, this one successfully prevents SIFA, establishing the claims made in the paper. It is worth mentioning that the original proposal for SIFA protection in [81] does not provide combined security as shown in [38]. Validating combined security is currently out of the scope of DL-FALAT. We, therefore, leave the validation of the combined security of Friet as interesting future work.

6.6. *DEFAULT*

Example 16. DEFAULT is a block cipher claiming inherent resilience against DFA [36]. The proposal is distinct from all other countermeasure evaluated in this work as it does not prevent faulty ciphertexts from reaching the output. Instead, it is claimed that the attack complexity with the faulty ciphertexts will be high, making the attacks computationally impractical. In order to achieve such security, DEFAULT uses partially linear S-Boxes at the beginning and end of the cipher computation. The initial and final rounds are called *DEFAULT-Layer*, while the middle rounds are called *DEFAULT-Core*. The DEFAULT-Core is a standalone block cipher using strong S-Boxes. The DEFAULT-Layer uses partially linear S-Boxes, which are not strong enough against differential cryptanalysis but are helpful to prevent FAs. The partially linear S-Boxes in the DEFAULT-Layers make the key recovery difficult with DFA, as the DFA equations output several key suggestions instead of a unique one. DEFAULT claims to make the DFA complexity $2^{\frac{n}{2}}$ for an n-bit block cipher.

In order to evaluate the DFA security of DEFAULT, we test with fixed key and two different fault values. An implementation in C has been procured from the authors of the paper. DF-FALAT and ALAFA show leakage in this case (ref. Figure 15). This is because the output ciphertexts are neither randomized nor muted and always result in two distinct

[9] https://github.com/thisimon/Friet.git.

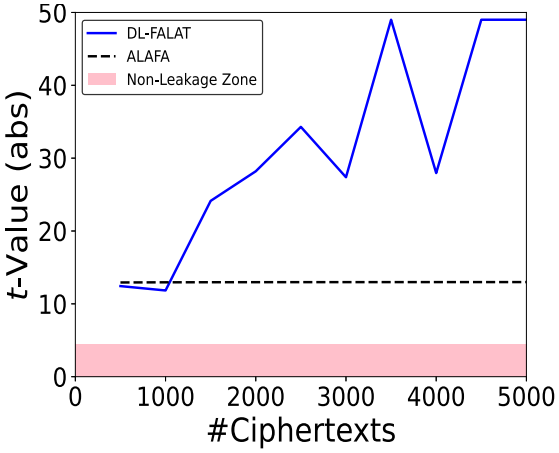

Fig. 15. Leakage assessment of DEFAULT [36].

values for two different fault values. This is not surprising as, in principle, it is possible to exploit these ciphertexts for an attack. However, the attack complexity would be higher. The task of DL-FALAT is to indicate the leakage and not to determine the exploitability of such leakage. In fact, recent DFA on DEFAULT shows that the anticipation made by DL-FALAT is indeed true. DFAs can be performed within practical complexity on any version of DEFAULT with such faulty ciphertexts [82].

Most of the countermeasures tested in this work are vulnerable to two equal faults in redundant branches. The detection countermeasures and the implicit detection mechanisms of infection countermeasures are bypassed with two equal faults. Therefore, faulty ciphertexts reach the output causing univariate leakage. Exceptions include countermeasures based on information redundancy, Friet and CAPA and M&M, where computations in redundant branches differ. Also, DEFAULT is a different variant altogether, with no redundancy. Finally, it is worth noting that the simple time/space redundancy countermeasure is vulnerable against a combined side-channel and fault attack [83]. We believe that DL-FALAT, with its observables extended with side-channel traces, will be able to detect this class of attacks. As shown in several recent combined attacks [38,84], side-channel leakage is exploited for extracting the fault differentials or ineffectivity of faults. In other words, side-channel leakage is an extended observable for such combined attacks along with faulty outcomes. In some cases, side-channel leakage also replaces faulty outcomes as observables. DL-FALAT, therefore, should extend to such leakage if the side-channel leakage is considered observable instead of (or along with) ciphertexts. This is because the type of leakage remains fundamentally the same (i.e., FA leakage) even for such combined attacks. However, a detailed investigation of this is left as future work.

6.7. *Generalized Leakage Assessment*

In order to illustrate the *compare-with-uniform* experiment presented in Sect. 4.2, we consider two use-cases. Our first use-case considers fault injection in the mask delivery logic of a masked AES implementation. De-randomization of masks, which may eventually lead to successful SCAs, is detected in this experiment. Next, we consider a use-case for the SHE HSM firmware where the key loading operation shows leakage under the influence of faults.

Example 17. (**Mask de-randomization**) In this example, we 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, and 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 1.3. In this pseudocode, the mask values originate from memory locations M11, M12, M13, and M14. The observable $\mathcal{O}$ is the mask input register of the AES. An adversary may skip one or multiple of these instructions causing the mask to remain fixed for all executions. In this case, we assume the first 32-bit data transfer is skipped resulting in a zero mask value for that 32 bits. Due to the existence of a precharge logic, the registers in a masked implementation are often set to zero before loading the mask. Skipping a load instruction, therefore, can set the respective part of mask to zero. The procedure described in Algorithm 4 can identify such loss of randomness by detecting a deviation from the uniform random. Note that there is no need for running `TEST-INTERF-KEY` or `TEST-INTERF-FAULT` as the mask does not vary with the key. The leakage assessment results are presented in Fig. 16.

Listing 1.3. Mask Derandomization.

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

Example 18. (**Non-Cipher Leakage from SHE**) This example considers an automotive security standard called SHE [75], which provides services like secure boot, encryption, key storage, and authentication for automotive electronic control units (ECU). SHE standard recommends an HSM, which includes a hardware implementation of AES for encryption and authentication support, a True Random Number Generator (TRNG), and private memory and storage with restricted access from other hardware components outside the HSM. The secure storage contains the firmware(s) and the secret keys used by the ECU. Multiple commercial microprocessors support SHE as a Cryptographic Service Engine (CSE) [75,85]. The HSM can only be accessed through specialized instructions and registers in such implementations. The CSE firmware is executed on specialized hardware called *CSE core*, often realized with a 32-bit processor.

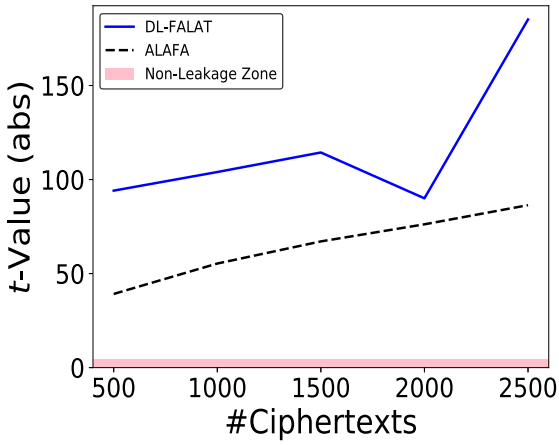

Fig. 16. Leakage assessment results .

In order to evaluate the robustness of a SHE implementation against FA, we construct a proof-of-concept implementation (ref. Figure 17a) according to the specifications given in [75,76]. The entire prototype has been implemented on the ZedBoard Zynq-7000 platform. A MicroBlaze softcore processor serves as the CSE controller (core). A 32-bit AXI data bus communicates between the CSE core and the private memory components. The firmware is written in C using the MicroBlaze C library supported by Xilinx.

We evaluate the firmware code which an instruction-skip attacker can target. Without loss of generality, we consider the basic encryption support provided by CSE. *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 1.4 (this operation, in turn, makes a call to the 32-bit data transfer instruction `swi` of MicroBlaze). In our experiments, we skip one (or more) of these data transfer operations. The observable here is the key register inside the AES core. The AES core is reset after each execution. So, the key register contains zero at the beginning. Algorithm 4 detects the presence of a key leakage in this case. The compare-with-uniform test in Fig. 17b first indicates leakage. Due to the key dependency of the observable, the next test invoked is `TEST-INTERF-KEY` (as with a skip, we had only one fault value). Figure 17c presents the result of this experiment indicating leakage.

A thorough investigation reveals that the leakage observed in this case is exploitable. The instruction-skip can result in a scenario where a 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. The rest of the key bits can also be recovered by repeating this attack three more times. The computational complexity of this attack is 2^{32}.

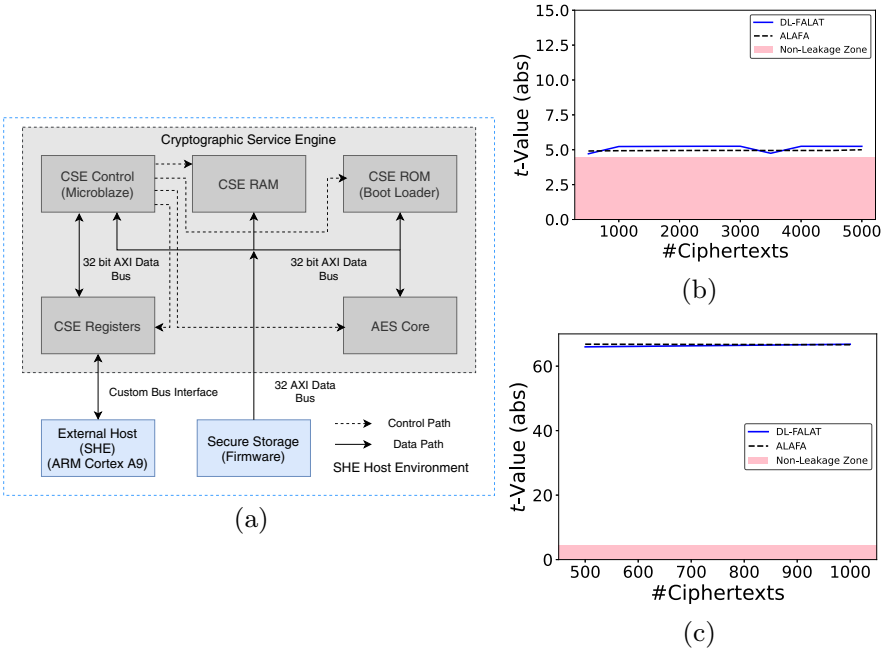

Fig. 17. **a** SHE Prototype: Basic Architecture; **b** Leakage profiles for compare-with-uniform; **c** Leakage for two different keys.

Listing 1.4. 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);
```

7. Conclusion

Security evaluation of an FA-protected implementation is a problem of utmost practical importance. In this paper, we have proposed a DL-assisted leakage detection test DL-FALAT, which can evaluate protected block cipher implementations as well as leakages in peripheral components of security modules. It automatically detects the order of leakage, which is not straightforward to detect from the countermeasure structure in FA. The test is not only suitable for filtering out malformed designs but can also figure out the points of vulnerabilities. We have shown how a variant of this test can be utilized to evaluate SIFA. A comprehensive guideline for evaluating the fault space of a block cipher is also presented, which utilizes the equivalences present in fault space. Experimental validation over different countermeasure classes establishes that DL-FALAT can rule out flawed designs. Although, in principle, a t-test-based leakage detection test should also detect all the leakages given enough data, DL-FALAT detects it with lower data

complexity than *t*-test in many cases. A potential future direction in this research is to adopt the test for public-key implementations and symmetric-key modes.

A. Simulation of Instruction-Level Faults using GDB

Listing 1.5. 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 1.6. X86-64 assembly (line. 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)
```

Listing 1.7 presents an example of how an instruction-skip event can be simulated using GDB.[10] We assume the availability of the high-level C code (e.g., Listing 1.5). A part of the assembly corresponding to line 5 of this code is shown in Listing 1.6. This code corresponds to the `AddRoundKey` of AES, and the line serves as a target for our injection. 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 is realized on the first instruction of Listing 1.6 by executing lines 6–10 in the GDB script of Listing 1.7. The core idea is to change the address stored in the program counter register to the following address. Any instruction can be targeted by moving the execution to the desired instruction with the `nexti` command of GDB. GDB also allows explicit modification of program counter value. Multiple consecutive instruction-skips can be implemented using this fact. The current implementation of our instruction-fault simulator expects the position of a skip in terms of the function name and target loop counter value (if required) as inputs. However, it can also simulate skips for every instruction from the beginning of a program.

Listing 1.7. GDB snippet for instruction-skip in Listing 1.5, 1.6.

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

[10] https://www.sourceware.org/gdb/.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions for improving the paper. Debdeep Mukhopadhyay would also like to thank the Department of Science and Technology (DST), Govt of India, IHUB NTIHAC Foundation, C3i Building, IIT Kanpur, and Centre on HARDWARE-SECURITY ENTREPRENEURSHIP RESEARCH & DEVELOPMENT, Meity, India, for partially funding this work.

References

[1] E. Biham, A. Shamir, Differential fault analysis of secret key cryptosystems, in *Annual International Cryptology Conference* (Springer, 1997), pp. 513–525

[2] M. Tunstall, D. Mukhopadhyay, S. Ali, Differential fault analysis of the advanced encryption standard using a single fault, in *IFIP International Workshop on Information Security Theory and Practices* (Springer, 2011), pp. 224–233

[3] M. Agoyan, J.M. Dutertre, D. Naccache, B. Robisson, A. Tria, When clocks fail: On critical paths and clock faults, in *International Conference on Smart Card Research and Advanced Applications* (Springer, 2010), pp. 182–193

[4] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, M. Renaudin, Glitch and laser fault attacks onto a secure AES implementation on a SRAM-based FPGA. *J. Cryptol.* **24**(2), 247–268 (2011)

[5] A. Dehbaoui, J.M. Dutertre, B. Robisson, A. Tria, Electromagnetic transient faults injection on a hardware and a software implementations of AES, in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography* (IEEE, 2012), pp. 7–15

[6] M. Agoyan, J.M. Dutertre, A.P. Mirbaha, D. Naccache, A.L. Ribotta, A. Tria, How to flip a bit? in *2010 IEEE 16th International On-Line Testing Symposium* (IEEE, 2010), pp. 235–239

[7] F. Zhang, X. Lou, X. Zhao, S. Bhasin, W. He, R. Ding, S. Qureshi, K. Ren, Persistent fault analysis on block ciphers, in *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 150–172

[8] K. Murdock, D. Oswald, F.D. Garcia, J. Van Bulck, D. Gruss, F. Piessens, Plundervolt: Software-based fault injection attacks against Intel SGX, in *41st IEEE Symposium on Security and Privacy* (IEEE, 2020), pp. 1466–1482

[9] M. Sabbagh, Y. Fei, D. Kaeli, A novel GPU overdrive fault attack, in *57th ACM/IEEE Design Automation Conference* (IEEE, San francisco, USA, 2020), pp. 1–6

[10] N. Moro, K. Heydemann, E. Encrenaz, B. Robisson, Formal verification of a software countermeasure against instruction skip attacks. *J. Cryptogr. Eng.* **4**(3), 145–156 (2014)

[11] S. Patranabis, A. Chakraborty, D. Mukhopadhyay, Fault tolerant infective countermeasure for AES. *J. Hardware Syst. Secur.* **1**(1), 3–17 (2017)

[12] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, R. Primas, SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Trans. Cryptographic Hardware Embedded Syst.* 547–572 (2018)

[13] C. Dobraunig, M. Eichlseder, H. Gross, S. Mangard, F. Mendel, R. Primas, Statistical ineffective fault attacks on masked AES with fault countermeasures, in *International Conference on the Theory and Application of Cryptology and Information Security* (Springer, 2018), pp. 315–342

[14] B. Gierlichs, J. Schmidt, M. Tunstall, Infective computation and dummy rounds: fault protection for block ciphers without check-before-output, in *International Conference on Cryptology and Information Security in Latin America* (Springer, 2012), pp. 305–321

[15] X. Guo, D. Mukhopadhyay, C. Jin, R. Karri, Security analysis of concurrent error detection against differential fault analysis. *J. Cryptogr. Eng.* **5**(3), 153–169 (2015)

[16] A.R. Shahmirzadi, S. Rasoolzadeh, A. Moradi, Impeccable circuits II, in *57th ACM/IEEE Design Automation Conference (DAC)* (2020), pp. 1–6

[17] S. Saha, D. Jap, D.B. Roy, A. Chakraborti, S. Bhasin, D. Mukhopadhyay, A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction. *IEEE Trans. Inf. Forens. Secur.* (2019)

[18] J. Breier, M. Khairallah, X. Hou, Y. Liu, A countermeasure against statistical ineffective fault analysis. *IACR Cryptology ePrint Archive* **2019**, 515 (2019). https://eprint.iacr.org/2019/515

[19] O. Reparaz, L. De Meyer, B. Bilgin, V. Arribas, S. Nikova, V. Nikov, N. Smart, CAPA: the spirit of beaver against physical attacks, in *Annual International Cryptology Conference* (Springer, 2018), pp. 121–151

[20] S. Patranabis, D. Mukhopadhyay, *Fault tolerant architectures for cryptography and hardware security*. (Springer, 2018)

[21] L. De Meyer, V. Arribas Abril, S. Nikova, V. Nikov, V. Rijmen, M&M: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardware Embedded Syst.* **2019**(1), 25–50 (2018)

[22] S. Saha, A. Bag, D.B. Roy, S. Patranabis, D. Mukhopadhyay, Fault template attacks on block ciphers exploiting fault propagation, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, 2020). pp. 612–643

[23] N. Bagheri, R. Ebrahimpour, N. Ghaedi, New differential fault analysis on PRESENT. *EURASIP J. Adv. Signal Process.* **2013**(1), 1–10 (2013)

[24] S. Ghosh, D. Saha, A. Sengupta, D.R. Chowdhury, Preventing fault attacks using fault randomization with a case study on AES, in *Australasian conference on information security and privacy* (Springer, 2015), pp. 343–355

[25] B. Wang, L. Liu, C. Deng, M. Zhu, S. Yin, Z. Zhou, S. Wei, Exploration of benes network in cryptographic processors: A random infection countermeasure for block ciphers against fault attacks. *IEEE Trans. Inf. Forens. Secur.* **12**(2), 309–322 (2016)

[26] T. Simon, L. Batina, J. Daemen, V. Grosso, P.M.C. Massolino, K. Papagiannopoulos, F. Regazzoni, N. Samwel, Friet: An authenticated encryption scheme with built-in fault detection, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, 2020), pp. 581–611

[27] S.D. Kumar, S. Patranabis, J. Breier, D. Mukhopadhyay, S. Bhasin, A. Chattopadhyay, A. Baksi, A practical fault attack on arx-like ciphers with a case study on ChaCha20, in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, 2017), pp. 33–40

[28] H. Tupsamudre, S. Bisht, D. Mukhopadhyay, Destroying fault invariant with randomization, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, 2014), pp. 93–111

[29] B. Yuce, N.F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, P. Schaumont, Software fault resistance is futile: Effective single-glitch attacks, in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, 2016), pp. 47–58

[30] S. Saha, S.N. Kumar, S. Patranabis, D. Mukhopadhyay, P. Dasgupta, ALAFA: Automatic leakage assessment for fault attack countermeasures, in *Proceedings of the 56th Annual Design Automation Conference 2019* (ACM, 2019), p. 136

[31] D. Clark, S. Hunt, P. Malacaria, Quantified interference: Information theory and information flow, in *Workshop on Issues in the Theory of Security* (2004)

[32] P. Khanna, C. Rebeiro, A. Hazra, XFC: A framework for exploitable fault characterization in block ciphers, in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (IEEE, (2017), pp. 1–6

[33] S. Saha, D. Mukhopadhyay, P. Dasgupta, ExpFault: an automated framework for exploitable fault characterization in block ciphers. *IACR Trans. Cryptogr. Hardware Embedded Syst.* 242–276 (2018)

[34] S. Saha, D. Jap, S. Patranabis, D. Mukhopadhyay, S. Bhasin, P. Dasgupta, Automatic characterization of exploitable faults: A machine learning approach. *IEEE Trans. Inf. Forens. Secur.* **14**(4), 954–968 (2018)

[35] J. Richter-Brockmann, A.R. Shahmirzadi, P. Sasdrich, A. Moradi, T. Güneysu, FIVER–robust verification of countermeasures against fault injections. *IACR Trans. Cryptographic Hardware Embedded Syst.* 447–473 (2021)

[36] A. Baksi, S. Bhasin, J. Breier, M. Khairallah, T. Peyrin, S. Sarkar, S.M. Sim, DEFAULT: Cipher level resistance against differential fault attack, in *27th International Conference on the Theory and Application of Cryptology and Information Security* (2021), pp. 124–156

[37] V. Arribas, F. Wegener, A. Moradi, S. Nikova, Cryptographic fault diagnosis using VerFI, in *2020 IEEE International Symposium on Hardware Oriented Security and Trust* (IEEE, 2020), pp. 229–240

[38] S. Saha, A. Bag, D. Jap, D. Mukhopadhyay, S. Bhasin, Divided we stand, united we fall: Security analysis of some SCA+SIFA countermeasures against SCA-enhanced fault template attacks, in *International*

Conference on the Theory and Application of Cryptology and Information Security (Springer, 2021), pp. 62–94
[39] B. Timon, Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Trans. Cryptographic Hardware Embedded Syst.* 107–131 (2019)
[40] F. Wegener, T. Moos, A. Moradi, DL-LA: Deep learning leakage assessment: A modern roadmap for SCA evaluations. *TCHES* **2021**, 552–598 (2021)
[41] J. Kim, S. Picek, A. Heuser, S. Bhasin, A. Hanjalic, Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Trans. Cryptographic Hardware Embedded Syst.* 148–179 (2019)
[42] L. Masure, C. Dumas, E. Prouff, A comprehensive study of deep learning for side-channel analysis. *IACR Trans. Cryptographic Hardware Embedded Syst.* 348–375 (2020)
[43] J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, P. Rohatgi, Test vector leakage assessment (TVLA) methodology in practice, in *International Cryptographic Module Conference* (2013)
[44] Y.Liu, L. Wei, B. Luo, Q. Xu, Fault injection attack on deep neural network, in *2017 IEEE/ACM International Conference on Computer-Aided Design* (2017), pp. 131–138
[45] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, T. Dumitraş, Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks, in *28th USENIX Security Symposium* (2019), pp. 497–514
[46] Rakin, A.S., Chowdhuryy, M.H.I., Yao, F., Fan, D.: Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories, in *2022 IEEE Symposium on Security and Privacy* (IEEE, 2022), pp. 1157–1174
[47] M. Moradi, B.J. Oakes, M. Saraoglu, A. Morozov, K. Janschek, J. Denil, Exploring fault parameter space using reinforcement learning-based fault injection, in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops* (IEEE, 2020), pp. 102–109
[48] A. Baksi, S. Sarkar, A. Siddhanti, R. Anand, A. Chattopadhyay, Fault location identification by machine learning. *Cryptology ePrint Archive* (2020)
[49] K. Sakiyama, Y. Li, M. Iwamoto, K. Ohta, Information-theoretic approach to optimal differential fault analysis. *IEEE Trans. Inf. Forensics Secur.* **7**(1), 109–120 (2011)
[50] C.M. Holmes, I. Nemenman, Estimation of mutual information for real-valued data with error bars and controlled bias. *Phys. Rev. E* **100**(2), 022404 (2019)
[51] L. Paninski, Estimation of entropy and mutual information. *Neural Comput.* **15**(6), 1191–1253 (2003)
[52] J.F. De Winter, D. Dodou, Five-point likert items: t-test versus Mann-Whitney-Wilcoxon (addendum added october 2012). *Pract. Assess. Res. Eval.* **15**(1), 11 (2010)
[53] S.S. Sawilowsky, R.C. Blair, A more realistic look at the robustness and type II error properties of the t-test to departures from population normality. *Psychol. Bull.* **111**(2), 352 (1992)
[54] H.M. Park, *Comparing group means: t-tests and one-way ANOVA using Stata, SAS, R, and SPSS* (2009)
[55] S.M. Ross, *Introduction to probability and statistics for engineers and scientists* (Academic press, 2020)
[56] T. Schneider, A. Moradi, Leakage assessment methodology, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, 2015), pp. 495–513
[57] F.X. Standaert, How (not) to use Welch's t-test in side-channel security evaluations, in *International Conference on Smart Card Research and Advanced Applications* (Springer, 2018), pp. 65–79
[58] A. Moradi, B. Richter, T. Schneider, F.X. Standaert, Leakage detection with the χ^2-test. *IACR Trans. Cryptographic Hardware Embedded Syst.* 209–237 (2018)
[59] W. Rawat, Z. Wang, Deep convolutional neural networks for image classification: A comprehensive review. *Neural Comput.* **29**(9), 2352–2449 (2017)
[60] A. Torfi, R.A. Shirvani, Y. Keneshloo, N. Tavaf, E.A. Fox, Natural language processing advancements by deep learning: A survey. arXiv preprint arXiv:2003.01200 (2020)
[61] I. Goodfellow, Y. Bengio, A. Courville, *Deep learning* (MIT Press, 2016)
[62] P.M. Pardalos, V. Rasskazova, M.N. Vrahatis, et al., *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems* (Springer, 2021)
[63] X. Zeng, T.R. Martinez, Distribution-balanced stratified cross-validation for accuracy estimation. *J. Exp. Theor. Artif. Intell.* **12**(1), 1–12 (2000)
[64] J.T. Roscoe, Fundamental research statistics for the behavioral sciences [by] John T. Roscoe. Holt, Rinehart and Winston (1975)
[65] M. Kuhn, K. Johnson, et al., *Applied Predictive Modeling*, vol. 26. (Springer, 2013)

[66] F. Chollet, et al., Keras documentation. keras.io (2015)
[67] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 265–283
[68] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
[69] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
[70] A.P. Johnson, S. Patranabis, R.S. Chakraborty, D. Mukhopadhyay, Remote dynamic partial reconfiguration: A threat to internet-of-things and embedded security applications. *Microprocessors Microsyst.* **52**, 131–144 (2017)
[71] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital systems testing and testable design*, vol. 2 (Computer Science Press, New York, 1990)
[72] S. Saha, D. Mukhopadhyay, P. Dasgupta, ExpFault (2018). https://cadforassurance.org/tools/sca/expfault/
[73] A. Poschmann, A. Moradi, K. Khoo, C.W. Lim, H. Wang, S. Ling, Side-channel resistant crypto for less than 2,300 GE. *J. Cryptol.* **24**(2), 322–345 (2011)
[74] S. Saha, D. Jap, D. Basu Roy, A. Chakraborty, S. Bhasin, D. 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* **2020**, 545 (2020)
[75] Using the cryptographic service engine (CSE): An introduction to the CSE module (2011). http://cache.freescale.com/files/32bit/doc/app_note/AN4234.pdf
[76] SHE—secure hardware extension functional specification version1.1 (rev 439) (2011). http://www.automotive-his.de
[77] C. De Cannière, O. Dunkelman, M. Knežević, KATAN and KTANTAN—a family of small and efficient hardware-oriented block ciphers, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, 2009), pp. 272–288
[78] J. Feng, H. Chen, Y. Li, Z.P. Jiao, W. Xi, A framework for evaluation and analysis on infection countermeasures against fault attacks. *IEEE Trans. Inf. Forensics Secur.* (2019)
[79] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J. Robshaw, Y. Seurin, C. Vikkelsoe, PRESENT: An ultra-lightweight block cipher, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, 2007), pp. 450–466
[80] C. Beierle, G. Leander, A. Moradi, S. Rasoolzadeh, CRAFT: Lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol.* **2019**(1), 5–45 (2019)
[81] J. Daemen, C. Dobraunig, M. Eichlseder, H. Groß, F. Mendel, R. Primas, Protecting against statistical ineffective fault attacks. *IACR Trans. Cryptographic Hardware Embedded Syst.* **2020**(3), 508–543 (2020)
[82] M. Nageler, C. Dobraunig, M. Eichlseder, Information-combining differential fault attacks on DEFAULT, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, 2022), pp. 168–191
[83] V. Lomne, T. Roche, A. Thillard, On the need of randomness in fault attack countermeasures-application to AES, in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography* (IEEE, 2012), pp. 85–94
[84] S. Patranabis, N. Datta, D. Jap, J. Breier, S. Bhasin, D. Mukhopadhyay, SCADFA: Combined SCA+DFA attacks on block ciphers with practical validations. *IEEE Trans. Comput.* **68**(10), 1498–1510 (2019)
[85] Fujitsu announces first single-chip solution for automotive hybrid instrument cluster with secure hardware extension (SHE). https://www.fujitsu.com/downloads/MICRO/fme/fujitsu-atlas-l-automotive-secure-hardware-extension.pdf