Automatic Characterization of Exploitable Faults: A Machine Learning Approach

Sayandeep Saha[1], Dirmanto Jap[2], Sikhar Patranabis[1], Debdeep Mukhopadhyay[1],
Shivam Bhasin[2], and Pallab Dasgupta[1]

[1] Department of Computer Science and Engineering, IIT Kharagpur, India
[2] Physical Analysis & Cryptographic Engineering (PACE) Labs, Nanyang Technological University, Singapore
E-mail: {sahasayandeep, sikhar.patranabis, debdeep, pallab}@cse.iitkgp.ernet.in,{djap, sbhasin}@@ntu.edu.sg

Abstract. Characterization of the fault space of a cipher to filter out a set of faults potentially exploitable for fault attacks (FA), is a problem with immense practical value. A quantitative knowledge of the exploitable fault space is desirable in several applications, like security evaluation, cipher construction and implementation, design, and testing of countermeasures etc. In this work, we investigate this problem in the context of block ciphers. The formidable size of the fault space of a block cipher mandates the use of an automation to solve this problem, which should be able to characterize each individual fault instance quickly. On the other hand, the automation is expected to be applicable to most of the block cipher constructions. Existing techniques for automated fault attacks do not satisfy both of these goals simultaneously and hence are not directly applicable in the context of exploitable fault characterization. In this paper, we present a supervised machine learning (ML) assisted automated framework, which successfully addresses both of the criteria mentioned. The key idea is to extrapolate the knowledge of some existing FAs on a cipher to rapidly figure out new attack instances on the same. Experimental validation of the proposed framework on two state-of-the-art block ciphers – PRESENT and LED, establishes that our approach is able to provide fairly good accuracy in identifying exploitable fault instances at a reasonable cost. Finally, the effect of different S-Boxes on the fault space of a cipher is evaluated utilizing the framework.

1 Introduction

The advent of Internet of Things (IoT) and Cyber-Physical-Systems (CPS) have laid the foundations for designing smarter albeit complex applications involving embedded computing platforms. Most modern embedded devices use in-built cryptographic cores, often tailored for resource-constrained environments, as root-of-trust for authentication and information processing tasks. Block ciphers are one of the most common cryptographic primitives deployed on such devices, for being the basic constituent of virtually every symmetric key cryptosystem of

today. However, it has been shown on several occasions that albeit being mathematically secure, unless properly implemented and protected, such cryptographic cores may lead to catastrophic security vulnerabilities by leaking secrets to malicious parties. In particular, one must ensure the cryptographic security of the primitives against implementation specific attacks like passive side channel analysis and active fault attacks, which have recently gained a lot of attention from both industry and academia, due to their practicality and diversity.

Fault-based cryptanalysis or Fault Attacks (FA) is a class of active implementation based attacks, which typically exploit transient faults in the data and/or control paths of a cipher during its execution to extract the secret key. Among different sub-classes of FA, Differential Fault Analysis (DFA) attacks are particularly interesting in the context of block ciphers, due to their low data/fault complexity and easy-to-mount nature. In DFA, the adversary injects faults with certain known spatiotemporal characteristics and then analyzes the pairs of faulty and the corresponding fault-free ciphertexts to recover the secret key. It is well established that even a single properly placed malicious fault is able to compromise the security of mathematically strong block ciphers in certain cases. One prominent example of this fact is the AES [1], where a random byte fault at the 8th round of the cipher can compromise the 128-bit secret key [2].

Given a block cipher, the discovery of a DFA attack is, however, nontrivial, as not all possible faults may lead to successful attacks. Traditional approaches for DFAs are mostly manual and demand special expertise on cryptanalysis. Till date, numerous block ciphers have been designed and deployed in-field for various applications [1,3]. Moreover, there is a growing trend of designing application and platform-specific lightweight ciphers, tailored for resource-constrained environments [4, 5]. Recently, NIST has launched an international competition to standardize lightweight cryptographic primitives. It is quite apparent that thorough analysis of such a large number of ciphers is impractical with manual DFA techniques and therefore automated DFA tools must be devised for this purpose. Further, such automated tools should work with minimal manual intervention and must be applicable to a large class of block ciphers and fault models.

Recently, there have been significant advances in designing such tools [6–11]. The most prominent among these automated frameworks is the so-called Algebraic Fault Attack (AFA), which encodes a given cipher and an injected fault as a system of multivariate polynomial equations on the finite field $GF(2)$ in Algebraic Normal Form (ANF) [6–10]. The ANF system is then converted to an equivalent system in Conjunctive Normal Form (CNF) and fed to a Boolean Satisfiability (SAT) solver with the aim of extracting the key by solving the system.

In this paper, we address the problem of characterizing the fault space of a block cipher to filter out the potentially exploitable fault instances. While finding a single exploitable fault instance is sufficient from the perspective of an attacker, certifying a cipher or its implementation for fault attack resilience would not be complete without some quantitative knowledge about the space of exploitable

faults. The knowledge of the exploitable fault space might also be desired for designing ciphers with some inherent fault attack resilience, testing of countermeasures, and guided synthesis of countermeasures for resource-constrained environments. However, the formidable size of the fault spaces usually encountered in block ciphers makes the problem of exploitable fault space characterization much more difficult than finding individual DFAs. Typically, a fault instance in a cryptographic primitive depends on different aspects (e.g. the location, the width of the fault, the mathematical structure of the cipher, and the number of times a fault is injected) and consideration of each of these aspects results in a fault space of prohibitive size, which may be hard to enumerate. Even a statistical characterization of this fault space is difficult and one must obtain a sufficiently large number of samples, as the distribution of the fault space may be unknown, even for well-studied ciphers.

The very nature of the exploitable fault characterization problem demands an automated solution for this purpose. From the perspective of a cipher designer or system architect, the characterization process for each individual fault should be fast enough so that significant fraction of the fault space can be covered within a reasonable time. Another desirable requirement for such automation is that it must be generic enough for the large class of existing and future cipher designs. Unfortunately, none of the automated fault attack frameworks proposed till date satisfy both these criteria, simultaneously. In particular, the AFA, which is fairly generic in nature, involves solving a SAT problem for each individual fault instance. Although SAT solvers are remarkably good at finding solutions to a large class of NP-Complete problem instances, the time taken for solving is often prohibitively high. In fact, in the context of AFA attacks, the solver may not stop within a reasonable time for many fault instances. Although, setting a proper timeout seems to be a reasonable fix for such cases, the variation of solving times is often very high and as a result, the timeout threshold must be reasonably high as well, to guarantee the capture of every possible attack. It is thus quite evident that exhaustive analysis of the fault space by means of AFA is impractical as far as the time is concerned.

1.1 Our Contributions

The contributions of this paper are as follows:

- We present a generic, fast, and fully automated framework for exploitable fault space characterization in block ciphers. The fault attack instances are usually represented as mathematical constraints, which reduces the size of key-space by a significant extent, so that exhaustive key search becomes trivial. Based on the intuition that the constrained search spaces for different exploitable fault instances on a cipher may have certain structural similarities, we propose a machine learning (ML) framework, which, if trained with some already known exploitable fault instances on a cipher, can predict new attacks on the same. To the best of our knowledge, this is the first concrete demonstration of ML in the context of DFA.

- The proposed framework is experimentally evaluated on two state-of-the-art lightweight block ciphers – PRESENT [4] and LED [5]. In particular, we show that ML models can predict new attack instances for a block cipher with significantly high accuracy, while being trained with a reasonably small number of known attack instances on the same. **Specifically, we obtain training accuracies of 85 - 93% in our experiments.** Further, we propose a simple strategy to nullify the risk of misclassifying exploitable faults as benign ones, which is found to work fairly well for our case studies. **Our experiments establish that the false negatives can be nullified by exhaustively testing roughly 20% of the total fault samples, which is quite reasonable. Further, it is found that the ML models can predict difficult attack instances while trained on a set of relatively easier attack instances, which is indeed a remarkable result, so far the capability of the tool is concerned.**
- In order to establish the importance of exploitable fault characterization, we present an application scenario, which sheds light on a previously unexplored effect of non-linear layers on the exploitable fault space of a block cipher. We study the effect of 3 structurally similar S-Boxes in the context of DFA attacks on bit-permutation based SPN ciphers. The experiments were performed for PRESENT, which is the most prominent member of this class. It is found that the statistical characterization of exploitable fault space provides interesting information about the effect of non-linear layers in the context of DFAs. **In particular, we observe that the S-Box of SKINNY [43] is relatively more resistant to DFAs than the S-Boxes of PRESENT and SERPENT [3].**

The rest of the paper is organized as follows. In the next section, we present a brief overview of fault-based cryptanalysis with an emphasis on automated fault analysis techniques. Some necessary preliminaries are presented in Section 3. We elaborate the proposed framework in Section 4, along with supporting case studies and a potential application scenario in Section. 5. Concluding remarks are presented in Section 6.

2 Background

In this section, we outline the necessary backgrounds on fault attacks with an emphasis towards automated fault analysis. We begin with a brief introduction to fault analysis attacks on block ciphers. The AFA and some other approaches for automated fault attacks will be summarized next with emphasis on the concepts relevant for this work.

2.1 Fault Attacks on Block Ciphers: A Brief Survey

Exploitation of faults to attack cryptographic devices dates back to 1997 by Boneh et.al., who demonstrated the attack on RSA public key cryptosystem [13].

The concept of Differential Fault Analysis (DFA) was introduced by Biham et. al. [14] on the Data Encryption Standard (DES), and was readily adapted for other ciphers like AES [1], PRESENT [4], LED [5] etc. In particular, AES is the most extensively studied cipher in the context of fault attacks [2,15–19]. The basic principle of any fault attack is to cause a malicious aberration in the normal execution of the target cryptographic algorithm and to exploit the corresponding leakage to try and recover the secret key within reasonable computational complexity. Till date, DFAs are the most widely studied class of fault attacks. DFAs, in general, exploit computationally efficient key distinguishers resulting from the fault injection and recover the secret key by solving a system of equations constructed with the distinguishers. The existence of practically achievable fault models such as bit faults, nibble faults, byte faults, and diagonal faults makes DFA a potent threat for modern block ciphers.

Although DFA constitutes the most prominent class of fault attacks, there exist other variants which have recently gained significant attention mainly due to their simplicity. Differential Fault Intensity Analysis (DFIA) is a non-DFA technique [20, 21]. DFIA combines the concept of side-channel analysis with fault attack to recover the secret key by exploiting the faulty ciphertexts only. However, the number of required ciphertexts are significantly higher than in DFA. The Safe-Error Attacks (SEA), Differential Behaviour Analysis (DBA) and Fault Sensitivity Analysis (FSA) constitute the other major categories of fault attacks. The main crux of these attacks lies in the fact that depending on a particular sub-part of the secret key (such as a bit or a byte), a fault may or may not lead to a faulty computation. The very presence of a fault in a computation thus leaks critical information which leads to the extraction of the key [19, 22]. It should be noted that all these attacks including DFIA are statistical in nature and their success critically depends on the physical characteristics of the target implementation.

2.2 Automated Fault Analysis

In 2010, Courtois et. al. presented the concept of Algebraic Fault Analysis (AFA) combining the concepts of algebraic cryptanalysis and DFA [6]. Just like DFA, AFA also exploits the difference in values generated from the fault-free calculation and faulty calculation (due to the injection of fault at any intermediate round) to recover the secret. However, the representation and the analysis of AFA is significantly different from that of classical DFA. AFA is the reminiscent of algebraic cryptanalysis which represents a cipher as a large system of multivariate polynomial equations of low degree and high sparsity. Such polynomial systems, which are usually defined on the finite field $GF(2)$ in ANF form, are then fed to generic algebraic solvers with the aim of extracting the secret keys by solving [23]. Although algebraic cryptanalysis alone, so far, have not been able to break state-of-the-art block ciphers, with the addition of extra equations due to fault injection, they are found to break most of the well-known ciphers of present days [6–10]. The most popular method for solving AFA instances is to convert the equations to an equivalent representation in CNF form and then use

off-the-shelf SAT solvers for solving. Perhaps the most attractive feature of AFA is its genericness. In [10], Zhang et. al. have shown that the algebraic framework can encode most of the DFA like attacks including the attacks on key schedule and round counters. Moreover, it has been anticipated that AFA are powerful than conventional DFA in certain situations as they can exploit many equations which are otherwise beyond the perception of a human attacker [10]. All such features of AFA make it the perfect candidate for automated fault analysis.

Recently, Khanna et. al. [11] has proposed a novel approach for automated fault analysis called XFC, which is significantly different from the AFA approach. Through a coloring based abstraction, XFC first enumerates the fault propagation path of a cipher and then calculates the complexity of the key space which is expected to be reduced significantly, if a fault is exploitable. The computation procedure of XFC is fairly simple and fast compared to the AFA approaches, which makes it a potential candidate for exploitable fault characterization. However, the framework is found to be restricted to a very specific class of DFA and also lacks proper automation. Barthe et. al. [12] has proposed another alternative approach for public key algorithms where, the vulnerable locations in a software implementation are identified for a given attack condition using program synthesis techniques. However, it requires prior knowledge of the fault conditions. Also the program synthesis is not known to be a very fast approach in general.

3 Preliminaries

3.1 General Model for Block Cipher and Faults

Table 1: List of Notations

Symbol	Definition
"+"	Bitwise XOR
$\mathcal{E}_k$	A block cipher
o_j^i	ith *sub-operation* in the jth round
F_h	A fault instance
R	Number of cipher rounds
l	Number of sub-operation in each round
r	round of fault injection
N	fault multiplicity
w	fault width
T	fault position
λ	bit-width of a sub-operation
$\{f\}_{n=1}^{N}$	set of fault values for a fault injection
$\{p\}_{n=1}^{N}$	set of plaintext values for a fault injection
$M_{\mathcal{F}}$	set of exploitable faults
τ	timeout for SAT solvers
$\mathcal{S}$	Sensitivity threshold

Let $\mathcal{E}_k$ be a block cipher defined as a tuple $\mathcal{E}_k = \langle Enc, Dec \rangle$, where Enc and Dec denote the encryption and decryption functions, respectively. Further, the Enc function (and similarly the Dec function) is defined as $Enc(p) = \mathcal{A}_R \circ \mathcal{A}_{R-1} \circ ... \circ \mathcal{A}_1(p) = c$, for a plaintext $p \in \mathcal{P}$, ciphertext $c \in \mathcal{C}$, and a key $k \in \mathcal{K}$. Each $\mathcal{A}_j$ denotes a round function in a R round cipher. Further, each $\mathcal{A}_j = o_j^l \circ o_j^{l-1} \circ ... \circ o_j^1$ is a composition of certain functions of the form o_j^i, generally denoted as *sub-operations* in this work. Each $\mathcal{A}_j$ is thus assumed to have l sub-operations. A sub-operation may belong to the key schedule or the datapath of the cipher. In other words, o_j^i denotes either a key schedule sub-operation or a datapath sub-operation at any round j. We shall use the term o_j^i throughout this work to denote a sub-operation, without mentioning whether it belongs to key schedule or datapath. Table. 1 lists the notations used throughout this paper.

An injected fault in DFA usually corrupts the input of some specific sub-operation during the encryption or decryption operation of the cipher. Given the cipher model, we denote the set of faults as $\mathcal{F} = \{F_1, F_2, ..., F_H\}$, where each individual fault $F_h \in \mathcal{F}$ is specified as follows:

$$F_h = \langle o_r^i, \lambda, w, T, N, \{f\}_{n=1}^N, \{p\}_{n=1}^N \rangle \tag{1}$$

Here $r < R$ is the round of injection, and o_r^i denotes the sub-operation, input of which is altered with the fault. The parameter λ denotes the data-width of the sub-operation (more specifically, the bit-length of the input of the sub-operation). The parameter w is the width of the fault which quantifies the maximum number of bits affected by a fault. In general, *bit-based*, *nibble-based*, and *byte-based* fault models are considered which corresponds to w =1, 4, and 8, respectively. The *position* of the fault at the input of a sub-operation is denoted by T with $t \in \{0, 1, ... \frac{\lambda}{w}\}$. For practical reasons, w and T are usually defined in a way so that the injected faults always remain localized within some pre-specified block-operations of the corresponding sub-operation o_r^i. The parameter N represents the number of times, a fault is injected at a specific location to obtain a successful attack within a reasonable time. N is called the *Fault Multiplicity*.

The sets $\{f\}_{n=1}^N$, and $\{p\}_{n=1}^N$ denote the values of the injected faults and the plaintexts processed during each fault injection, respectively. In the most general case, the diffusion characteristics (and thus the exploitability) of an injected fault critically depends upon the value of the fault and the corresponding plaintext on which the fault is injected. A typical example is PRESENT cipher, where, the number of active S-Boxes due to the fault diffusion depends on the plaintext and the fault value and as a result, many faults injected at a specific position with the same multiplicity may become exploitable, whereas some of them at the same position may become unexploitable. According to the fault model in Equation 1, the total number of possible faults for a specific position T in sub-operation o_r^i is $2^{N(w+\lambda)}$, for a given fault width w. The total number of possible faults for a sub-operation o_r^i is $\left(2^{N(w+\lambda)} \times \frac{\lambda}{w}\right)$, and that for the whole cipher is o_r^i is $\left(2^{N(w+\lambda)} \times \frac{\lambda}{w} \times R \times l\right)$.

In certain cases the fault space can be pruned significantly utilizing the fact that a large number of faults may be actually equivalent. A prominent example is the AES where every byte fault at some specific position is equivalent irrespective of its value. However, there exists no automatic procedure to figure out such equivalences, till date, and the only way is to manually analyze the cipher. As a result, it is reasonable to adapt the above calculation of the size of the fault space while analyzing a general construction.

3.2 Algebraic Representation of Ciphers

Multivariate polynomial representation, which is quite well-known in the context of AFA [10], is considered one of the most generic and informative representations for block ciphers. In this work, we utilize the polynomial representations to encode both the ciphers and the faults. The usual way of representing block ciphers algebraically is to assign a set of symbolic variables for each iterative round, where each variable represents a bit from some intermediate state of the cipher. Each cipher sub-operation is then represented as a set of multivariate polynomial equations over the polynomial ring constructed on these variables, with $GF(2)$ being the base ring. The equation system should be sparse and low-degree in addition, to make the cipher representation easy to solve.

In order to elaborate the process of polynomial encoding, we consider the example of the PRESENT block cipher. PRESENT is a lightweight block cipher proposed by Bogdanov et. al. in CHES 2007 [4]. It has a Substitution-Permutation Network (SPN) based round function which is iterated 31 times to generate the ciphertext. The basic version PRESENT-80 has a block size of 64-bits and a master key of size 80 bits, which is utilized to generate 64-bit round keys for each round function by means of an iterated key-schedule. Each round of PRESENT consists of three sub-operations, namely, *addRoundKey*, *sBoxlayer*, and *pLayer*. The *addRoundKey* sub-operation, computing bitwise XOR between the state bits and round key bits is represented as:

$$y_i = x_i + k_i, \text{ for } 1 \leq i \leq 64 \tag{2}$$

where, x_i, k_i represents the input state bits and round key bits, respectively, and y_i represents the output bits of the *addRoundKey* sub-operation. Similarly, the *pLayer* operation, which is a 64-bit permutation can be expressed as:

$$y_{\pi(i)} = x_i, \text{ for } 1 \leq i \leq 64 \tag{3}$$

where $\pi(i)$ is the permutation table. The non-linear substitution operation *sBoxlayer* of PRESENT consists of 16 identical 4×4 bijective S-Boxes, each of which can be represented by a system of non-linear polynomials. The solvability of a typical cipher polynomial system critically depends on the S-Box representation, which is expected to be sufficiently sparse and consisting of low-degree polynomials. One way of representing the PRESENT S-Boxes is the following:

$$\begin{aligned} y_1 = {} & x_1x_2x_4 + x_1x_3x_4 \\ & + x_1 + x_2x_3x_4 + x_2x_3 + x_3 + x_4 + 1 \end{aligned}$$

$$\begin{aligned} y_2 &= x_1x_2x_4 + x_1x_3x_4 + x_1x_3 + x_1x_4 + \\ &\quad x_1 + x_2 + x_3x_4 + 1 \\ y_3 &= x_1x_2x_4 + x_1x_2 + x_1x_3x_4 + x_1x_3 + \\ &\quad x_1 + x_2x_3x_4 + x_3 \\ y_4 &= x_1 + x_2x_3 + x_2 + x_4 \end{aligned} \tag{4}$$

Here x_is ($1 \leq i \leq 4$) and y_is ($1 \leq i \leq 4$) represent the input and output bits of a 4×4 S-Box, respectively.

Each injected fault instance can be added in the cipher equation system in terms of new equations. Let us assume that the fault is injected at the input state of the ith sub-operation o_r^i at the rth round of the cipher. For convenience, we denote the input of o_r^i as $X^i = x_1||x_2||...||x_\lambda$, where λ is the bit-length of X^i. In the case of PRESENT $\lambda = 64$. Let, after the injection of the fault, the input state changes to $Y^i = y_1||y_2||...||y_\lambda$. Then the state differential can be represented as $D^i = d_1||d_2...||d_\lambda$, where $d_z = x_z + y_z$ with $1 \leq z < \lambda$. Further, depending on the width of the fault w, there can be $m = \frac{\lambda}{w}$ possible locations in X^i, which might have got altered. Let us partition the state differential D^i in m, w-bit chunks as $D^i = D_1^i||D_2^i||...||D_m^i$, where $D_t^i = d_{w\times(t-1)+1}||d_{w\times(t-1)+2}||...||d_{w\times t}$ for $1 \leq t \leq m$. Assuming T be the location of the fault, the fault effect can be modelled with the following equations:

$$D_t^i = 0, \text{ for } 1 \leq t \leq m, t \neq T \tag{5}$$

$$(1 + d_{w\times(t-1)+1})(1 + d_{w\times(t-1)+2})...(1+d_{w\times t}) = 0, \text{ for } t = T \tag{6}$$

It is notable that the location T of a fault can be unknown in certain cases, and this can also be modeled with equations of slightly complex form [10]. However, for exploitable fault characterization, it is reasonable to assume that the locations are known as we are working in the evaluator mode.

4 Proposed Methodology

4.1 Motivation

The goal of the present work is to efficiently filter out the exploitable faults for a given cryptosystem. It is apparent that the ANF polynomials provide a reasonable way for modeling the ciphers and the faults [10]. Although, the ANF description and its corresponding CNF is easy to construct, solving them is non-trivial as the decision problem associated with the solvability of an ANF system is NP-Complete. In practice, SAT solvers are used for solving the associated CNF systems and it is observed that the solving times vary significantly depending on the instance.

One key observation regarding the cipher equation systems is that they are never unsatisfiable, which is due to the fact that for a given plaintext-ciphertext

pair there always exists a key. However, it is not practically feasible to figure out the key without fault injections, as the size of the key search space is prohibitively large. The search space complexity reduces with the injection of faults. The size of the search space is expected to reach below some certain limit which is possible to search exhaustively with modern SAT solvers within reasonable time, if a sufficient number of faults are injected at proper locations.

The above-mentioned observation clearly specifies the condition for distinguishing the exploitable faults from the non-exploitable ones. To be precise, if a SAT solver terminates with the solution within a prespecified time limit, the fault instance is considered to be exploitable. Otherwise, the fault is considered non-malicious. Setting a proper time-limit for the SAT solver is, however, a critical task. A relatively low time-limit is unreliable as it may fail to capture some potential attack instances. As an example, for the PRESENT cipher we observed that most of the 1-bit fault instances with fault multiplicity 2, injected at the inputs of 28-th round S-Box operation, are solvable within 3 minutes. This observation is similar to that mentioned in [10]. However, we observed that when nibble faults are considered at the 28th round, the variation of solving time is significantly high; in fact, there are cases with solving times around $16 - 24$ hours. These performance figures are obtained with Intel Core i5 machines running CryptominiSAT-5 [33] as the SAT solver in a single threaded manner. Moreover, such cases comprise nearly 12% of the total number of samples considered. This is not insignificant in a statistical sense, where failure in detecting some attack instances cannot be tolerated. Such instances do not follow any specific pattern through which one can visually characterize them without solving them. This observation necessarily implies that one has to be more careful while setting solver timeouts and a high value of timeout is preferable. However, setting high timeout limits the number of instances one can acquire through exhaustive SAT solving within a practically feasible time span.

According to the fault model described at Section 3.1, the size of the fault space in a cipher is prohibitively large. As a concrete example, there are total $2^{(64+4)} = 2^{68}$ possible nibble fault instances, with *fault multiplicity* $N = 1$, for any specific *position* T, on any sub-operation o_r^i in the PRESENT cipher. The number is even larger if one considers other positions, sub-operations, fault-multiplicity, and fault-models. Moreover, the ratio of exploitable faults to the total number of faults is unknown apriori. The whole situation suggests that in order to obtain a reliable understanding of the exploitable fault space even in a statistical sense, one must test a significantly large sample from fault space. Also, to obtain a sufficiently large set of exploitable faults for testing purpose, a large number of fault instances must be examined. With a high timeout required for SAT solvers, exhaustive SAT solving is clearly impractical for fault space characterization and a fast mechanism is required. **Our aim in this paper is to prepare an efficient alternative to the exhaustive enumeration of the fault space via SAT solving.**

4.2 Empirical Hardness Prediction of Satisfiability Problems

NP-Complete problems are ubiquitous in computer science, especially in AI. While they are hard-to-solve on worst case inputs, there exists numerous "easy" instances which are of great practical value. In general, the algorithms for solving NP-Complete problems exhibit extreme runtime variations even across the solvable instances and there is no describable relationship between the instance size and the algorithm runtime as such. Over the past decade, a considerable body of work has shown how to use supervised ML models to answer questions regarding solvability or runtime using features of the problem instances and algorithm performance data [24–28]. Such ML models are popularly known as Empirical Hardness Models (EHM). Some applications of EHMs include proper algorithm portfolio selection for a problem instance [28], algorithm parameter tuning [25], hard benchmark construction [29], and analysis of algorithm performance and instance hardness [29].

In the context of the present work, we are interested in EHMs which predict the hardness of SAT instances. The most prominent result in the context of empirical runtime estimation of SAT problems is due to Xu et. al., who constructed a portfolio based SAT solver SATzilla [28] based on EHMs. The aim of SATzilla was to select the best solver for a given SAT instance, depending upon the runtime predictions of different EHMs constructed for a set of representative SAT solvers. The SATzilla project also provided a large set of 138 features for the model construction depending on various structural properties of the CNF descriptions of the problem instance as well as some typical features obtained from runtime probing of some basic SAT solvers. In this work, we utilize some of these features for constructing EHMs which will predict the exploitability of a given fault instance without solving it explicitly. Brief description of our feature set will be provided later in this section.

4.3 ML Model for Exploitable Fault Identifier

In this subsection, we shall describe the ML-based framework in detail. In nutshell, our aim is to construct a binary classifier, which, if trained with certain number of exploitable and unexploitable fault instances, can predict the exploitability of any fault instance queried to it. Before going to further details, we formally describe the exploitable fault space for a given block cipher and an exploitable fault in the context of SAT solvability.

Definition 1 (Exploitable Fault Space) *Given a cipher $\mathcal{E}_k$ and a corresponding fault space $\mathcal{F}$, the exploitable fault space $M_{\mathcal{F}} \subset \mathcal{F}$ for $\mathcal{E}_k$ is defined as a set of faults such that $\forall F_h \in M_{\mathcal{F}}$, it is possible to extract n_e bits of the secret key k, where $0 < n_e \leq |k|$.*

In other words, exploitable fault space denotes the set of faults for which the combination of the injected fault and a plaintext results in the extraction of n_e bits of the secret key. From the perspective of a cipher evaluator, two distinct scenarios can be considered at this point. In the first one, it is assumed that none

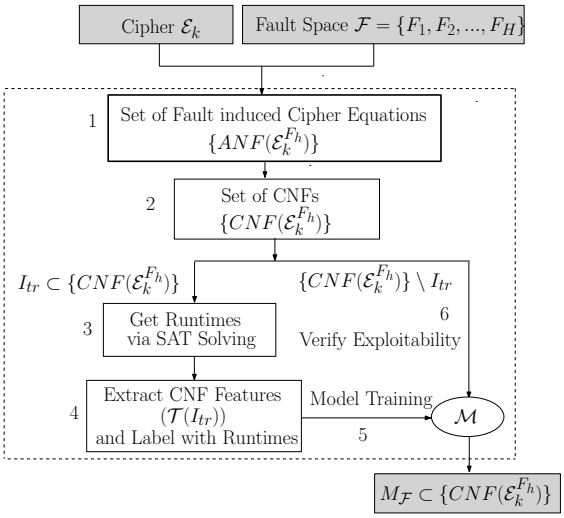


Fig. 1: The Exploitable Fault Characterization Framework: Basic Idea

of the key bits are known apriori and faults are inserted to extract the complete master key of the cipher. Indeed, one may increase the number of injections to reduce the complexity of the search space in this scenario. However, it is practically reasonable to assume some upper bound on the number of injections. In other words, the fault multiplicity N in the fault model is always $\leq$ some pre-specified threshold. The second scenario in this context occurs when some specific key bits are assumed to be known. This model is extremely useful when only a subset of the key can be extracted by the fault injection due to incomplete diffusion of the faults. In a typical AFA framework, it is not possible to obtain a unique solution for the incompletely defused faults unless some of the key bits are known. However, in this work, we mainly elaborate the first scenario. It is worth mentioning that, the second scenario can be dealt with the framework we are going to propose, without any significant changes.

The framework for exploitable fault space characterization is depicted in Figure. 1. Referring to the figure, let $\mathcal{E}_k^{F_h}$ indicate the cipher $\mathcal{E}_k$, with a fault F_h from its fault space $\mathcal{F}$ injected in it. This can be easily modelled as an ANF equation system denoted as $ANF(\mathcal{E}_k^{F_h})$. The very next step is to convert $ANF(\mathcal{E}_k^{F_h})$ to the corresponding CNF model denoted by $CNF(\mathcal{E}_k^{F_h})$. At this point, we specify the exploitable faults in terms of solvability of SAT problems, with the following definition:

Definition 2 (Exploitable Fault) *A fault $F_h \in \mathcal{F}$ for the cipher $\mathcal{E}_k$ is called exploitable if the $CNF(\mathcal{E}_k^{F_h})$ is solvable by a SAT solver within a pre-specified time bound τ.*

Given fault instances from the fault space of a cipher, we construct CNF encoding for each of them. A small fraction I_{tr} of these CNFs are solved ex-

haustively with SAT solver and labeled accordingly depending on whether they are solvable or not within the threshold τ. Next, a binary classifier $\mathcal{M}$ is trained with these labeled instances, which is the EHM in this case. The ML model is defined as:

$$\mathcal{M} : \mathcal{T}(CNF(\mathcal{E}_k^{F_h})) \mapsto \{0, 1\} \tag{7}$$

Here, $\mathcal{T}$ is an abstract function which represents the features extracted from the CNFs. In the present context, $\mathcal{T}$ outputs the feature vectors from the SATzilla feature set [28]. For convenience, we use the following nomenclature:

Class 0 : Denotes the class of exploitable faults.
Class 1 : Denotes the class of benign/unexploitable faults.

One important difference of our EHM model with the conventional EHM models is that we do not predict the runtime of an instance but use the labels 0 and 1 to classify the faults into two classes. In other words, we solve a classification rather than a regression problem solved in conventional EHMs [27]. The reason is that we just do not exploit the runtime information in our framework. The main motive of ours is to distinguish instances whose search space size is within the practical search capability of a solver, from those instances which are beyond the practical limit. It is apparent that our classifier based construction is sufficient for this purpose. In the next section, we describe the feature set utilized for the classification.

4.4 Feature Set Description

In this work, we use the features suggested by the SATzilla – a portfolio based SAT solving tool [28]. The SATzilla project proposed a rich set of 138 features to be extracted from the CNF description of a SAT instances for the construction of runtime predicting EHMs. The feature set of SATzilla is a compilation of several algorithm-independent properties of SAT instances made by the Artificial Intelligence (AI) community on various occations [29]. A widely known example of such algorithm-independent properties is the so-called *phase-transition* of random 3-SAT instances. In short, SAT instances, generated randomly on a fixed number of variables, and containing only 3-variable clauses, tend to become unsatisfiable as the clause-to-variable ratio crosses a specific value of 4.26 [24]. Intuitively, the reason for such a behavior is that instances with fewer clauses are underconstrained and thus almost always satisfiable, while those with many clauses are overconstrained and unsatisfiable for most of the cases. The SATzilla feature set is divided into 12 groups. Some of the feature groups consist of structural features like the one described in the example, whereas the others include features extracted from runtime behaviors of the SAT instances on solvers from different genre – like Davis-Putnam-Logemann-Loveland (DPLL) solvers, or local search solvers [27, 28].

The structural features of SATzilla are divided into five feature groups, which include simple features like various variable-clause statistics as well as

features based on complex clause-variable interactions in the CNF formula obtained through different graph-based abstractions. Intuitively, these statistical measurements somewhat quantify the difficulty of an instance. The so-called runtime features in SATzilla, also known as "probing" features, are computed with short-time runs of the instances on different genres of candidate solvers. The computation times for these 12 groups of features are not uniform and there exist both structural and runtime features which are computationally expensive. The computation time of the features also provide significant information regarding the instance hardness and as a result, they are included as the final feature group. Further details on the feature set can be found in [26, 27].

4.5 Handling the False Negatives:

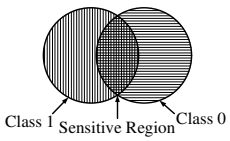


Fig. 2: Sensitive Region: Conceptual Illustration

The ML model proposed in this work provides quick answers regarding the exploitability of the fault instances queried to it. Such a quick answering system has an enormous impact on the exploitable characterization problem as it makes the problem tractable from a practical sense. However, the efficiency comes at the cost of accuracy. Being a ML-based approach, there will always be some *false positives* (a benign fault instance classified as exploitable) and *false negatives* (an exploitable instance classified as benign). While a small number of false positives can still be tolerated, false negatives can be crucial for some applications, for example, generating a test set of exploitable faults for testing countermeasures. If some typical exploitable faults are missed, they may lead to successful attacks on the countermeasure.

In this work, we provide a potential solution for the misclassification issue. More precisely, we try to statistically eliminate the chances of false negative cases – that is the chances of an attack getting misclassified. The main idea is to first determine the cases for which the classification confidence of the classifier is not very high. We denote such cases as *sensitive instances*. Note that, sensitive instances are determined on the validation data-set once the classifier is trained and deployed for use. Intuitively, such sensitive instances are prone to misclassification (we have also validated this claim experimentally.). Each sensitive instance is exhaustively tested with SAT solver. Fig. 2 presents a conceptual schematic of what we mean by sensitive instances. Typically, we assume that the two classes defined in terms of the feature vectors can be overlapping, and the region of overlap constitute the set of sensitive instances.

Determination of the sensitive instances or this region of overlap is, however, not straightforward and could be dealt in many ways. In this paper, we take a very simple albeit effective strategy. We use *Random Forest* (RF) of decision trees as our classification algorithm [38]. Random forest is constructed with several decision trees, each of which is a weak learner. Usually, such *ensemble methods* of learning performs majority voting among the decisions of the constituent weak learners (decision trees in the present context) to determine the class of the instance. Here, we propose a simple methodology for eliminating the false negatives using the properties of the RF algorithm. The proposed approach is reminiscent of classification with reject – a well-studied area in ML, where a classifier can reject some instances if the classification confidence is low for them [32]. Let Cl be the random variable denoting the predicted class of a given instance x in the two-class classification problem we are dealing with. For any instance x, we try to figure out the quantities $Pr[Cl = 0 \mid x]$ and $Pr[Cl = 1 \mid x]$, which are basically the probabilities of x lying in any of the two classes. Evidently, the sum of these two quantities is 1. Note that the probabilities are calculated purely based on the decisions made by the classifier. In other words, it is calculated exploiting the properties of the classification algorithm. Next, we calculate the following quantity:

$$\delta = (Pr[Cl = 0 \mid x] - Pr[Cl = 1 \mid x]) \tag{8}$$

It is easy to observe that having a large value for δ implies the classifier is reasonably confident about the class of the instance x. In that case, we consider the decision of the classifier as the correct decision. For the other case, where δ is less than some predefined threshold $\mathcal{S}$, we invoke the SAT solver to determine the actual class of the instance. The overall flow for false negative removal is summarized in Algorithm 1.

Algorithm 1 Procedure $CLASSIFY\_FAULTS$

Input: A random fault instance F_h
Output: Exploitability status of F_h
Construct $ANF(\mathcal{E}_k^{F_h})$ and then $CNF(\mathcal{E}_k^{F_h})$
Compute $x = \mathcal{T}(CNF(\mathcal{E}_k^{F_h}))$
Compute $\langle Pr[Cl = 0 \mid x], Pr[Cl = 1 \mid x] \rangle = \mathcal{M}(x)$
Compute δ using Equation (8)
if ($|\delta| < \mathcal{S}$) **then** ▷ $\mathcal{S}$ is a predefined threshold
    Query the SAT engine with $CNF(\mathcal{E}_k^{F_h})$
    if ($CNF(\mathcal{E}_k^{F_h})$ is solvable within τ) **then**
        Return $F_h \in M_{\mathcal{F}}$
    else
        Return $F_h \notin M_{\mathcal{F}}$
    end if
else
    if ($\delta > 0$) **then**
        Return 0 ▷ $F_h \in M_{\mathcal{F}}$
    else
        Return 1 ▷ $F_h \notin M_{\mathcal{F}}$
    end if
end if

Each tree in an RF returns the class probabilities for a given instance. The class probability of a single tree is the fraction of samples of the same class in a leaf node of the tree. Let the total number of trees in the forest be t_r. The class probability a random instance x is defined as:

$$Pr[Cl = c \mid x] = \frac{1}{t_r} \sum_{h=1}^{t_r} Pr_h[Cl = c \mid x] \tag{9}$$

where, $Pr_h[Cl = c \mid x]$ denotes the probability of x being a member of a class c according to the tree h in the forest.

The success of this mechanism, however, critically depends on the threshold $\mathcal{S}$, which is somewhat specific to the cipher under consideration, and is determined experimentally utilizing the validation data. Ideally, one would expect to nullify the false negatives without doing too many exhaustive validations. Although no theoretical guarantee can be provided by our mechanism for this, experimentally we found that for typical block ciphers, such as PRESENT and LED, one can reasonably fulfill this criterion. Detailed results supporting this claim will be provided in Sec. 5.

5 Case Studies

This section presents the experimental validation of the proposed framework by means of case studies. Two state-of-the-art block ciphers – PRESENT and LED are selected for this purpose. The motivation behind selecting these two specific ciphers is that they utilize the same non-linear, but significantly distinct linear layers. One main application of the proposed framework is to quantitatively examine the effect of different cipher sub-operations in the context of fault attacks, and in this paper, we mainly elaborate this application. The structural features of PRESENT and LED allow us to make a fair comparison between their diffusion layers. In order to evaluate the effect of the nonlinear S-Box layer, we further perform a series of experiments on the PRESENT block cipher by replacing its S-Box with three alternative S-Boxes of similar mathematical properties. In the following two subsections, we present the detailed study of the PRESENT and LED ciphers with the proposed framework. The study involving the S-Box replacement will be presented after that.

5.1 Learning Exploitable Faults for PRESENT

Table 2: Setup for the ML on PRESENT and LED

Cipher	PRESENT	LED
Target Rounds	$27 - 30$	$29 - 32$
Maximum number of times a fault is injected (N)	2	2
Timeout for the SAT solver (τ)	24 hrs	48 hrs

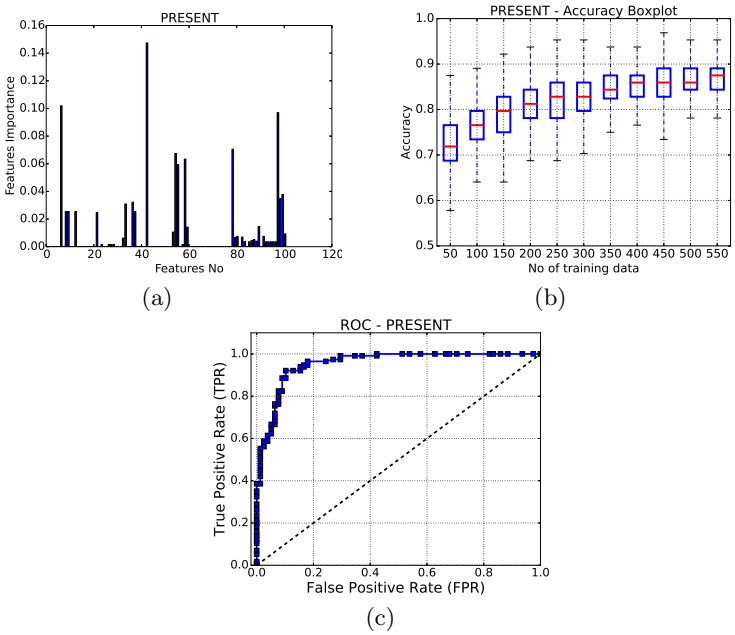


Fig. 3: Machine learning results for PRESENT. (a) Feature importance; (b) ROC curve; and (b) Variation of accuracy with the size of training set.

The basics of PRESENT block cipher has already been described in Sec. 3.2. Several fault attack examples have been proposed on PRESENT, mostly targeting the 29-th and 28-th round of the cipher as well as the key schedule of PRESENT [10, 21, 34–37]. Zhang et.al. [10] presented an AFA on PRESENT, requiring 2 bit-fault instances on average, at the 28-th round of the cipher in the best case. The solving times of the corresponding CNFs are mostly around 3 minutes.

Experimental Setup In order to validate the proposed framework, we create random AFA instances following different fault models. In order to make the ML classifier generic, we decided to train it on instances from different fault models. Two competitive fault models for PRESENT are the bit and nibble fault models, both of which can generate plenty of exploitable fault instances. In any case, we end up getting a CNF, the solvability of which determines the exploitability of an instance. So the ML classifier is supposed to learn to estimate the search complexity of an instance in some way. Hence, there is no harm in combining instances from two fault models as such. Table 2, presents the basic setup we used for the experiments on PRESENT and LED. Experiments on any given cipher begins with an initial *profiling phase*, where the parameters mentioned in Table 2 are determined and attack samples for training are gathered. For PRESENT, we mainly targeted the rounds $27 - 30$ in our experiments as one can hardly find any exploitable fault beyond these rounds. Further, the fault multiplicity

(N) was restricted to 2, (that is, N can assume values 1 and 2) considering low fault-complexities of DFAs. Interestingly, it was observed that the nibble fault instances (injected 2 times in succession) at 28-th round do not result in successful attacks, even after 2 days. Further, many of these instances (almost 12%) take $16-24$ hrs of solving time. No, successful attack instances were found taking time beyond 24 hours in our experiments, which were conducted on a machine with Intel Core i5 running CryptominiSAT-5 [33] as the SAT solver in a single threaded manner. We thus set the SAT timeout $\tau = 24$ hours for PRESENT. For the sake of experimentation, we exhaustively characterized a set of 1000 samples from the fault space of PRESENT and LED, individually. However, one should note that such exhaustive characterization was only required to prove the applicability of the proposed methodology, and in practice, a much smaller number of instances are required for training the ML classifier, as well shall show later. For every new cipher, such *profiling* should be performed only once just to build the ML classifier.

Feature Selection The first step in our experiments is to evaluate the feature set. Although we started with a well-accepted feature set, it is always interesting to know how these features impact the learning process and which are the most important features in the present context. Identification of the main contributing features for a given problem may also lead to significant reduction of the feature space dimensionality by the selection of actually useful features, which also reduces the chances of overfitting. We therefore perform a quantitative assessment of the importance of various features using the RF algorithm. Before proceeding further, it is worth mentioning that some of the SATzilla features might be computationally expensive depending on problem instances. It was found that the unit propagation features (which belong to the group of DPLL probing features) and the linear programming features in our case takes even more than 15 minutes of computing time for certain instances. As a result, we did not consider them in our experiments which left us with 123 features in total.

In this work, we evaluate the feature importance based on the *mean decrease of gini-impurity* of each feature during the construction of the decision trees [38]. Every node ζ, in a given decision tree γ, of an RF Γ imposes a partition on the dataset by putting some threshold condition on a single feature, so that similar samples end up in the same partition. The optimal split at a node is calculated based on a statistical measure which quantifies how well a potential split is separating the samples of different classes at this particular node. The *gini-impurity* is one of the most popular measure for such purposes and is actually used in random forests [38]. Let us assume that a node ζ in some decision tree $\gamma \in \Gamma$ has total $|\zeta|$ samples among which the subset ζ^q consists of samples from class $q \in \{0, 1\}$. Then the *gini-impurity* of ζ is calculated as:

$$\mathcal{G}(\zeta) = 1 - (p_\zeta^0)^2 - (p_\zeta^1)^2 \tag{10}$$

where, $p_\zeta^q = \frac{|\zeta^q|}{|\zeta|}$ for $q \in \{0, 1\}$. Let the node ζ partitions the dataset into two nodes (subsets) ζ_{left} and ζ_{right}, using some threshold condition t_θ on some

feature θ, and the gini-impurity of these two nodes are $\mathcal{G}(\zeta_{left})$ and $\mathcal{G}(\zeta_{right})$, respectively. Then the decrease in impurity at the node ζ, due to this specific split is calculated as:

$$\Delta\mathcal{G}(\zeta) = \mathcal{G}(\zeta) - p_{left}\mathcal{G}(\zeta_{left}) - p_{right}\mathcal{G}(\zeta_{right}) \tag{11}$$

where, $p_{left} = \frac{|\zeta_{left}|}{|\zeta|}$, and $p_{right} = \frac{|\zeta_{right}|}{|\zeta|}$. In an exhaustive search over all variables θ available at ζ and the space of corresponding t_θs, the optimal split at ζ, for a particular tree γ can be determined which is quantified as $\Delta^\theta\mathcal{G}(\zeta, \gamma)$. In the calculation of feature importance, decrease in gini-impurities are accumulated in a per-variable basis and the importance value of the feature θ, is calculated as follows:

$$\mathcal{I}_\mathcal{G}(\theta) = \sum_{\gamma\in\Gamma}\sum_{\zeta\in\gamma}\Delta^\theta\mathcal{G}(\zeta, \gamma) \tag{12}$$

The result of the feature importance assessment experiment is presented in Figure 3a, where the X-axis represents the index of a feature and the Y-axis represents its importance scaled within an interval of $[0, 1]$. It is interesting to observe that, there are almost 66 features, for which the importance value is 0. Further investigation reveals that these features obtain constant values for all the instances. As a result, they can be safely ignored for the further experiments.

It can be observed from Figure 3a, that the feature no. 42 is the most important one for our experiments. This feature corresponds to the aggregated computation time for the Variable-Clause Graph (VCG) and Variable-Graph (VG) graph-based features. A VCG is a bipartite graph, with nodes corresponding to each variable and clause. The edges in this graph represent the occurrence of a variable in a clause. The VG has a node for each variable and an edge between variables that occur together in at least one clause. Intuitively, the computation time is a crude representative for the dense-nature of these graphs, which is usually high if the search space is very large and complex. However, it is difficult to directly relate this feature with quick solvability of an instance as other selected features also play a significant role. In fact, it was observed that every structural feature group have some contribution in the classification, which is somewhat expected (feature no. $0-59$ in Figure 3a). In contrast, the contributions from the runtime features were not so regular. In particular, only the survey propagation features (based on estimates of variable bias in a SAT formula obtained using probabilistic inference [30].) were found to play some role in the classification (feature no. 79-96 in Figure 3a). Interestingly, the features $98-100$, which corresponds to the approximate search-space size (estimated with the average depth of contradictions in DPLL search trees [31]), were found to play some role in the classification. This is indeed expected, as the classification margin in this work is defined based on the search space size.

Classification We next measured the classification accuracy of the RF classifier with the reduced set of features. In order to check the robustness of the learning, we ran each of our experiments several times. For each repetition, new

Table 3: Misclassification Handling for PRESENT

$\mathcal{S}$ Value	0.10	0.12	0.14	0.16	0.18	0.20	0.22	0.24	0.26	0.28	0.30
% Sensitive Instances	6.0	6.0	10.0	13.2	17.2	20.4	20.4	22.0	22.8	24.8	27.6
% False Negatives beyond $\mathcal{S}$	4.1	4.1	3.0	1.8	0.6	0.2	0.0	0.0	0.0	0.0	0.0

training and validation sets were chosen from a set of 640 labeled samples (the remaining 360 samples collected at the profiling phase were utilized for further validation and false negative removal experiments), where the sizes of them are in the ratio $7:3$. The sample set consists of 320 exploitable and 320 unexploitable fault instances in order to achieve an unbiased training. The average accuracy obtained in our experiment was 85%. We also provide the Receiver Operating Characteristics (ROC) curve for the RF classifier, which is considered to be a good representative for the quality of a classifier. The Area Under Curve (AUC) represents the goodness of a classifier, which ranges between 0 to 1 with higher values representing a better classifier. The ROC curve for the PRESENT example is provided in Fig. 3c, which shows that the classifier performs reasonably well in this case. Fig. 3b presents the variation of accuracy with the size of training dataset as a box plot. It can be observed that reasonable accuracy can be reached within 450 training instances (which is around 70% of our dataset size) and accuracy does not improve much after that.

Handling False Negatives Although our classifier reaches a reasonable good accuracy of 85%, there are almost 15% instances which get misclassified in this process, which contains both false positives and false negatives. As pointed out in Sec. 4.5, false negatives are not acceptable in certain scenarios. The approach presented in Sec. 4.5 critically depends on the threshold parameter $\mathcal{S}$, which must be set in a way so that the percentage of false negatives become 0 or at least negligibly small. If the percentage of instances below $\mathcal{S}$ is too high, it would be costly to estimate all of them via exhaustive SAT solving. However, the reasonably good accuracy of our classifier suggests that the percentage of such sensitive instances may not be very high. We tested our proposed fix from Section. 4.5 on a new set of 250 test instances with different $\mathcal{S}$ values. Table 3 presents the outcome of the experiment. The percentage of instances to be justified via SAT solving and the percentage of false negatives beyond $\mathcal{S}$ is presented in the table for each choice of $\mathcal{S}$. It can be observed from Table 3 that a threshold of 0.22 nullifies the number of false negatives and keeps the percentage of sensitive instances (see Section. 4.5) to 20%, which is indeed reasonable.

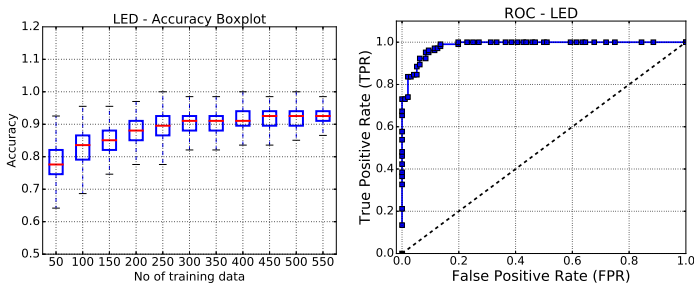
Gain over Exhaustive SAT Solving It would be interesting to estimate the overall gain of our ML assisted methodology compared to exhaustive characterization via SAT solving. For the sake of elaboration, let us consider a scenario where only nibble faults are injected at the 28th round of PRESENT. Further, each fault is assumed to be of multiplicity 2. The size of the resulting fault space is $2^{2\times(64+4)} = 2^{136}$, which is impossible to enumerate. Even if one considers a

reasonable-sized sample of 10000 fault instances, the exhaustive characterization with SAT solving only would be impractical. Considering a timeout threshold of 24 hrs ($\tau = 24$ hrs), characterization of these many instances even with a parallel machine with a reasonable number of cores would take an impractical amount of time. For example, if one considers a 24 core system, the characterization would require 416 days, in the worst case. Even with an optimistic consideration of roughly 50% of the instances hitting the timeout threshold, the time requirement is still high. In contrast, the proposed framework can provide a fairly reasonable solution. Firstly, the size of the training set is extremely small, and also saturates after reaching a reasonable accuracy. One can rapidly characterize any number of fault instances after training with a reasonable error probability and the time requirement for that is insignificant. For a statistical understanding of the exploitable fault space, such error bounds can be reasonably tolerated. For even more security-critical applications, like evaluating a countermeasure or quantification of security bounds, the misclassified attack instances can be crucial. The proposed method works fine even in those cases with a reasonable overhead of characterizing 22% of the instances exhaustively. For a set of 10000 fault instances, it would require 83 days, even in the worst case which is much better than the figures obtained with exhaustive characterization.

Discussion One of the goals of the ML framework is to discover new attacks while trained on a set of known attack instances. It was found that the proposed framework is able to do that with reasonably high accuracy. More specifically, we found that if the training set contains only of fault instances injected at even-numbered nibbles at the 28th round, it can successfully predict all attacks from odd-numbered nibbles. This clearly indicates the capability of discovering new attacks. The proposed framework also successfully validated the claim that with the bit permutation based linear layer of PRESENT, the fault diffusion (and thus the attack) strongly depends on the plaintext and the value of the injected fault. Although this might not be a totally new observation, our framework figures it out, automatically, and can quantify this claim statistically within reasonable amount of time.

5.2 Exploitable Fault Space Characterization for LED

LED is a 64-bit block cipher proposed in CHES 2011 [5]. LED utilizes a round function which is similar to that of AES; more specifically it has the following sub-operations in sequence – *SubByte ShiftRow*, *MixColumn* and *addRoundKey*. In contrast to AES, the 64-bit key is added once in each 4 rounds. All the diffusion layer operations have identifiable nibble-wise structures. The 4×4 S-Box of PRESENT is used as the confusion layer. Interestingly LED has no key schedule and the same key is used in all rounds. Like PRESENT, LED has also been subjected to DFA and DFIA [21, 39, 40]. Most of the DFA attempts on LED targeted the last 3 rounds of LED [7, 9, 39, 40]. Recently, Li et. al. [41] have proposed an Impossible Differential Fault Analysis attack on the 29-th round of



(a) (b)

Fig. 4: Machine learning results for LED. (a) ROC Curve; and (b) Variation of accuracy with the size of training set.

the cipher which requires 43 nibble faults to be injected at a particular nibble. Jovanovic et. al. [7] and Zhao et. al. [9] independently presented AFA attacks on LED, where they show that it is possible to attack the cipher at 30th round with a single fault instance.

ML Experiments In this work, we mainly focus on the last 5 rounds of the LED cipher. However, unlike the previous experiment on PRESENT, a slightly different strategy was adopted. In order to examine the proper potential of the ML model in discovering newer attack instances across different rounds, we intentionally trained it with samples from the 30 and the 31st rounds and tested it on instances from rounds 29 and 32. The RF model is trained with a total of 450 instances from the 30 and the 31st rounds and tested on 190 instances from rounds 29 and 32. The setup for the data acquisition is given in Table 2. The accuracy box plot and ROC curve for the classifier are provided in Fig. 4a and 4b, respectively. It can be observed that the accuracy is almost 93%. The features used were similar to the PRESENT experiments. Handling of misclassification was also performed and the result is presented in Table. 4.

Discovery of New Attacks We observed a quite interesting phenomenon in this experiment which clearly establishes the capability of the ML tool in discovering newer attack instances. More specifically, we found that the ML tool can identify attacks on 29th round of the cipher, even if it not trained with any instances from the 29th round. The attack instances observed at the 29th round of LED are mainly bit-fault instances with 2 fault injections. **Attacks on the 29th round of LED are usually difficult than the 30th round attacks [41, 42], and it is quite remarkable that the ML model can figure out difficult attacks just by learning easier attack instances.**

Discussion So far we have discussed two ciphers with same S-Box and different diffusion layers. A comparative study of these two experiments establishes that compared to PRESENT, the fault space of LED is quite regular in nature. For

example, almost all of the 30 round nibble faults in LED resulted in a successful attack, whereas for PRESENT there was a significant number of unexploitable instances at 28th round. Form the perspective of an adversary, targeting a cipher having bit-permutation based diffusion layers thus become a little more challenging as he/she must attack it with more number of fault injections in order to obtain a successful attack.

5.3 Analyzing the Effect of S-Boxes on Fault Attacks

Table 4: Misclassification Handling for LED

$\mathcal{S}$ Value	0.10	0.12	0.14	0.16	0.18	0.20	0.22	0.24	0.26	0.28	0.30
% Sensitive Instances	4.2	6.0	6.0	11.6	13.2	15.2	17.6	17.6	21.2	23.8	23.8
% False Negatives beyond $\mathcal{S}$	2.4	1.6	0.9	0.3	0.18	0.0	0.0	0.0	0.0	0.0	0.0

Table 5: Mathematical Properties of PRESENT, SERPENT, and SKINNY S-Boxes

Property	PRESENT	SERPENT	SKINNY
Size	4×4	4×4	4×4
Differential Branch Number	3	3	2
Differential Uniformity	4	4	4
Max. Degree of Component Functions	3	3	3
Min. Degree of Component Functions	2	2	2
Linearity	8	8	8
Nonlinearity	4	4	4
Max. Differential Probability	0.25	0.25	0.25
Max. Degree of Polynomial Representation (with Lexicographic Variable Ordering)	3	3	3

The S-Boxes are one of the most important resources in a block cipher construction. However, till date, no quantitative analysis was performed to evaluate the effect of S-Boxes on the fault attacks as such. In classical DFA, the attack complexity is related with the average number of solutions of the S-Box difference equations having the form $S(x) \oplus S(x \oplus \alpha) = \beta$. However, S-Boxes were never characterized in the context of fault attacks considering the cipher as a whole. The characterization of the exploitable fault space in this work gives us the opportunity to perform such analysis.

In this experiment, we study the effect of 3 different S-Boxes on the PRESENT cipher, with respect to fault attack. More specifically, we replace the original S-Box of PRESENT with the S-Box of SKINNY [43], and the S_0 S-Box of the SERPENT [3], and study their effect on the exploitable fault space. The algebraic characteristics of these 3 S-Boxes are almost identical and presented in Table. 5. The exploitable fault space in each case was characterized with our

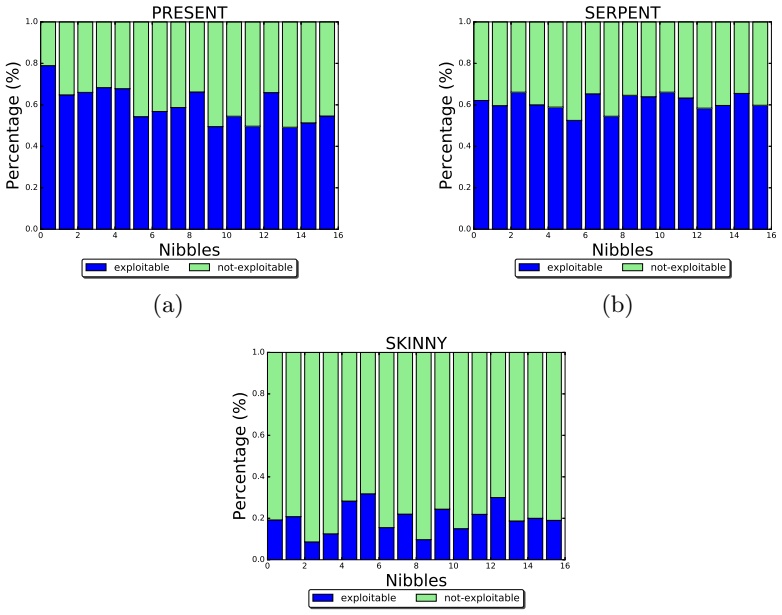


Fig. 5: Exploitable Fault Spaces with. (a) PRESENT S-Box; (b) SERPENT S-Box; and, (c) SKINNY S-Box.

ML-based framework. For the sake of simplicity, we only tested with nibble faults injected with $N = 2$ (that is two times for each fault instances) at the 28th round. The obtained test accuracies were similar to that of the PRESENT experiment and so we do not repeat them here. Further, for each of the S-Box case, we consider 1000 fault instances for each nibble location (there are total 16 nibble locations.). The characterized fault spaces of the three S-Box test cases are depicted in Fig. 5a, 5b and 5c , respectively. It is interesting to observe that although the PRESENT and SERPENT S-Box results in almost similar behavior, the SKINNY S-Box results in a significantly different fault distribution. More specifically, whereas most of the fault instances for the PRESENT

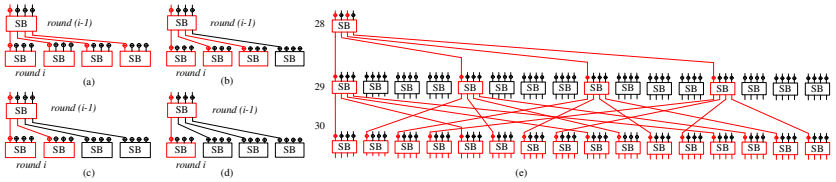


Fig. 6: Relation between the HW of SBox output differential and Fault Diffusion in PRESENT. (a) 4 S-boxes activated (b) 3 S-boxes activated (c) 2 S-boxes activated and (d) 1 S-box activated at the ith level. (e) Fault propagation up to 30th round.

and SERPENT are exploitable (60% exploitable faults on average), the situation is reverse in the case of SKINNY (23% exploitable faults on average).

Analysis of the Observations In order to explain the observations made in this experiment, we had an in-depth look in the 3 S-Boxes as well as the diffusion layer of PRESENT. The fault diffusion in PRESENT linear layer depends on the number of active S-Boxes (S-Boxes whose inputs are affected by the faults.). *For a multi-round fault propagation, the number of active S-Boxes in the* i^{th} *round depends on the* Hamming Weights *(HW) of the output S-Box differential in the* $(i-1)^{th}$ *round.* Fig. 6 emphasizes this claim with a very simple example. The lines colored red indicate non-zero differential value and the red S-Boxes are the active S-Boxes. Now let us consider the fault diffusion tree for the 28th round nibble fault injection in the PRESENT structure, shown in Fig. 6e up to 30th round, for convenience. It can be observed that most of the S-Boxes obtain an input difference of 1 bit. In other words, the inputs of the S-Boxes will have a single bit flipped. With this observation, the investigation boils down to the following question – *If the HW of the input difference of an S-Box is* 1*, what is the HW of the output difference?* For all three S-Boxes considered, the average HW of the output difference should be 2 when the average is considered over all possible input differences (this is due to the Strict Avalanche Criteria (SAC)). However, for the typical case, where the HW of the input difference is restricted to 1, the average HW of the output differences vary significantly. More specifically, the average is quite low for the SKINNY S-Box where it attains a value of 2.2. For the PRESENT S-Box, the value is 2.45 and for SERPENT it is 2.5. This stems from the fact that, *for the SKINNY S-Box, there exists input difference values, for which the HW of the output differences become* 1*. Whereas for PRESENT and SERPENT S-Boxes, the minimum HW of the output differences is* 2 *for any given input difference. In essence,* the fault diffusion with PRESENT and SERPENT S-Box is more rapid on average, than the SKINNY S-Box, which got reflected in the profile observed for exploitable fault spaces.

The result presented in this subsection is unique from several aspects. Firstly, it shows empirically that even if the S-Boxes are mathematically equivalent, they may have different effects in the context of fault attacks. Secondly, the proposed framework of ours can identify such interesting phenomenon for different cipher sub-blocks, which are otherwise not exposed from standard characterization. This clearly establishes the efficacy of the proposed approach.

6 Conclusion

Exploitable fault space characterization is an extremely relevant but relatively less explored topic in the fault attack research. We address this problem in the context of block ciphers, in this paper, and eventually, come up with a reasonable solution. The proposed solution is able to efficiently handle the prohibitively large fault space of a cipher with reasonable computational overhead. The ML-based

framework proposed here is not limited to block ciphers only. It is quite well-known that even stream ciphers, public key algorithms [44] and hash functions can be mapped to algebraic systems [45]. From that perspective, the framework can be easily extended to handle those cases.

In this paper, we have elaborated an application of exploitable fault characterization for the evaluation of cipher sub-operations in the context of fault attacks. However, several other applications are possible as already anticipated in the introduction section. One of the potential applications could be the guided synthesis of countermeasures for resource-constrained environments. Through a statistical characterization of the exploitable fault space in a per-round manner, one can potentially identify locations which are the most sensitive to the fault attacks. One can then implement costly countermeasures for those positions only, whereas less sophisticated countermeasures would work well for the rest of the cipher. Another important application scenario could be the testing of implementations and countermeasures with a large number of exploitable fault instances, rather than with random faults. Such testings are highly desired to correctly evaluate fault attack threats, or to certify a system against such attacks and can be realized quite efficiently with the proposed framework.

References

1. V. Rijmen and J. Daemen, "Advanced encryption standard," in *Proc. of FIPS Publications, NIST*, pp. 19–22, 2001.
2. M. Tunstall, et. al., "Differential fault analysis of the advanced encryption standard using a single fault," in *WISTP*. Springer, 2011, pp. 224–233.
3. E. Biham, et. al., "SERPENT: A new block cipher proposal," in *FSE*. Springer, 1998, pp. 222–238.
4. A. Bogdanov, et. al., "PRESENT: An ultra-lightweight block cipher," in *CHES*. Springer, 2007, pp. 450–466.
5. J. Guo, et. al., "The LED Block Cipher," in *CHES*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 326–341.
6. N. T. Courtois, et. al., "Fault-algebraic attacks on inner rounds of DES," *e-Smart'10 Proceedings: The Future of Digital Security Technologies*, 2010.
7. P. Jovanovic, et. al., "An algebraic fault attack on the LED block cipher." *IACR Cryptology ePrint Archive*, vol. 2012, p. 400, 2012.
8. F. Zhang, et. al., "Improved algebraic fault analysis: A case study on PICCOLO and applications to other lightweight block ciphers," in *COSADE*. Springer, 2013, pp. 62–79.
9. X. Zhao, et. al., "Algebraic differential fault attacks on LED using a single fault injection." *IACR Cryptology ePrint Archive*, vol. 2012, p. 347, 2012.
10. F. Zhang, et. al., "A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 5, pp. 1039–1054, 2016.
11. P. Khanna, et. al., "XFC: A Framework for eXploitable Fault Characterization in Block Ciphers," in *DAC*. IEEE, 2017.
12. G. Barthe, et. al., "Synthesis of fault attacks on cryptographic implementations," in *ACMCCS*. ACM, 2014, pp. 1016–1027.

13. D. Boneh, et. al., "On the importance of checking cryptographic protocols for faults," in *EUROCRYPT*. Springer, 1997, pp. 37–51.
14. E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," *CRYPTO*, pp. 513–525, 1997.
15. G. Piret and J.-J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," in *CHES*, vol. 2779. Springer, 2003, pp. 77–88.
16. C. Giraud, "DFA on AES," in *Int. Conf. Adv. Encryption Standard*. Springer, 2004, pp. 27–41.
17. S. S. Ali and D. Mukhopadhyay, "A differential fault analysis on AES key schedule using single fault," in *FDTC*. IEEE, 2011, pp. 35–42.
18. T. Fuhr, et. al., "Fault attacks on AES with faulty ciphertexts only," in *FDTC*. IEEE, 2013, pp. 108–118.
19. B. Robisson and P. Manet, "Differential behavioral analysis," in *CHES*. Springer, 2007, pp. 413–426.
20. N. F. Ghalaty, et. al., "Differential fault intensity analysis," in *FDTC*. IEEE, 2014, pp. 49–58.
21. N. F. Ghalaty, et. al., "Differential fault intensity analysis on PRESENT and LED block ciphers," in *COSADE*. Springer, 2015, pp. 174–188.
22. J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (AES)," in *CAV*. Springer, 2003, pp. 162–181.
23. N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations," in *ASIACRYPT*. Springer, 2002, pp. 267–287.
24. D. Mitchell, et. al., "Hard and easy distributions of SAT problems," in *AAAI*, vol. 92, 1992, pp. 459–465.
25. F. Hutter, et. al., "Performance prediction and automated tuning of randomized and parametric algorithms," *CP*, vol. 4204, pp. 213–228, 2006.
26. E. Nudelman et. al., "Understanding random SAT: Beyond the clauses-to-variables ratio," in *CP*. Springer, 2004, pp. 438–452.
27. F. Hutter, et. al., "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.
28. L. Xu, et. al., "SATzilla: portfolio-based algorithm selection for SAT," *Journal of AI research*, vol. 32, pp. 565–606, 2008.
29. K. Leyton-Brown, et. al., "Empirical hardness models: Methodology and a case study on combinatorial auctions," *J. ACM*, vol. 56, no. 4, p. 22, 2009.
30. E. Hsu, et. al., "Probabilistically estimating backbones and variable bias: Experimental overview," in *CP*. Springer, 2008, pp. 613–617.
31. L. Lobjois, *et al.*, "Branch and bound algorithm selection by performance prediction," in *AAAI/IAAI*, 1998, pp. 353–358.
32. K. Varshney, et. al., "A risk bound for ensemble classification with a reject option," in *SSP* IEEE, 2011, pp. 769–772.
33. M. Soos, et. al, "Extending SAT solvers to cryptographic problems," in *SAT*. Springer, 2009, pp. 244–257.
34. G. Wang and S. Wang, "Differential fault analysis on PRESENT key schedule," in *CIS*. IEEE, 2010, pp. 362–366.
35. X. Zhao, et. al., "Fault-propagate pattern based DFA on PRESENT and PRINTcipher," *J. Wuhan Univ. Natur. Sci.*, vol. 17, no. 6, pp. 485–493, 2012.
36. N. Bagheri, et. al., "New differential fault analysis on PRESENT," *EURASIP J Adv Signal Process.*, vol. 2013, no. 1, p. 145, 2013.
37. F. De Santis, et. al., "Ciphertext-only fault attacks on PRESENT," in *LightSec*. Springer, 2014, pp. 85–108.

38. L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
39. W. Li, et. al., "Single byte differential fault analysis on the LED lightweight cipher in the wireless sensor network," *Int. J. Comput Int Sys.*, vol. 5, no. 5, pp. 896–904, 2012.
40. P. Jovanovic, et. al., "A Fault Attack on the LED Block Cipher." in *COSADE*. Springer, 2012, pp. 120–134.
41. W. Li, et. al., "Impossible differential fault analysis on the led lightweight cryptosystem in the vehicular ad-hoc networks," *IEEE Trans. IEEE Trans Dependable Secure Comput.*, vol. 13, no. 1, pp. 84–92, 2016.
42. G. Zhao, et. al., "Differential fault analysis on LED using Super-Sbox," *IET Information Security*, vol. 9, no. 4, pp. 209–218, 2014.
43. C. Beierle, et. al., "The SKINNY family of block ciphers and its low-latency variant MANTIS," in *CRYPTO*. Springer, 2016, pp. 123–153.
44. J. Faugere, et. al., "Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases.," in *CRYPTO*. Springer, 2003, pp. 44–60.
45. P. Luo, et. al., "Algebraic fault analysis of SHA-3," in *DATE*. IEEE, 2017, pp. 151–156.