ExpFault: An Automated Framework for Exploitable Fault Characterization in Block Ciphers

Sayandeep Saha[1], Debdeep Mukhopadhyay[1] and Pallab Dasgupta[1]

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

Abstract. Malicious exploitation of faults for extracting secrets is one of the most practical and potent threats to modern cryptographic primitives. Interestingly, not every possible fault for a cryptosystem is maliciously exploitable, and evaluation of the exploitability of a fault is nontrivial. In order to devise precise defense mechanisms against such rogue faults, a comprehensive knowledge is required about the exploitable part of the fault space of a cryptosystem. Unfortunately, the fault space is diversified and of formidable size even while a single crypto-primitive is considered and traditional manual fault analysis techniques may often fall short to practically cover such a fault space within reasonable time. An automation for analyzing individual fault instances for their exploitability is thus inevitable. Such an automation is supposed to work as the core engine for analyzing the fault spaces of cryptographic primitives. In this paper, we propose an automation for evaluating the exploitability status of fault instances from block ciphers, mainly in the context of Differential Fault Analysis (DFA) attacks. The proposed framework is generic and scalable, which are perhaps the two most important features for covering diversified fault spaces of formidable size originating from different ciphers. As a proof-of-concept, we reconstruct some known attack examples on AES and PRESENT using the framework and finally analyze a recently proposed cipher GIFT [21] for the first time. It is found that the secret key of GIFT can be uniquely determined with 1 nibble fault instance injected at the beginning of the 25th round with a reasonable computational complexity of 2^{14}.

Keywords: Fault attack · Block cipher · Automation

Introduction

Almost every modern computing device provides support for cryptographic computation – both in the form of hardware extensions and software libraries. Block ciphers, being one of the most prominent constituents of modern cryptographic protocols, are deployed with most of the computing platforms. However, resource constraint of a platform is one of the determining factors for the nature of cryptographic supports provided. For example, the pervasive use of low-resource embedded electronic systems has led to the development of several lightweight block cipher algorithms like PRESENT [8], LED [9], SKINNY [10] etc. Concurrently, there is an increasing trend of developing new ciphers engineered for specific applications, so that optimal performance-resource trade-offs can be achieved. As a result, we have numerous block ciphers available today and their numbers are still increasing.

The common trend in cipher design is to evaluate the security of the cipher against classical attacks like differential, and linear cryptanalysis before it is deployed. However, security evaluation against implementation based side channel and fault attacks has also become essential, given the practicality and potency of such attacks. Usually, cipher-specific countermeasures are designed to defend against such implementation based attacks. However, countermeasures do incur overheads which have to be optimised carefully in order to provide proper security bounds within specified resource-constraints. Such precisely engineered countermeasures can be devised only if the complete attack space of a given cipher is well-understood.

In this paper, we address the problem of attack space exploration in the context of fault-based cryptanalysis of block ciphers – more precisely, for Differential Fault Analysis (DFA) attacks [1, 17, 4]. DFA is the most widely explored and complex class of fault attacks so far and is particularly interesting given their (relatively) low data/fault complexity and easy-to-mount nature. It is well-established that even a single properly placed malicious fault is able to compromise the security of mathematically strong crypto-primitives in certain cases. However, not every possible fault in a cipher is exploitable by an attacker to cause a practical attack, and determining the exploitability of a fault instance is nontrivial.[1] In this work, we refer to such malicious faults as *exploitable faults*. While finding a single exploitable fault instance for a system is sufficient from the perspective of an attacker, certifying a system for fault attack resilience demands the characterization of the complete space of exploitable faults. This is, however, a notoriously difficult task given the formidable size and diversity of the fault space even for a single block cipher. The situation clearly indicates that an automation is inevitable in this context and the very first step towards building such an automation is to devise a framework which can determine the exploitability status of an individual fault instance on a given block cipher. The main target of this paper is to propose a framework which can automate DFA attacks.

Potential Challenges in Fault Attack Automation: Typically, faults in a cipher (let us focus on the block ciphers only) are specified by multiple attributes (e.g. the location, width of the fault, fault model and the mathematical structure of the cipher), which eventually lead to a fault space of formidable size. Any automation, which targets individual fault instances for exploitability evaluation should be sufficiently fast and scalable in order to practically explore the fault space. On the other hand, the automation should be applicable to most of the available block ciphers and fault models. Another point of concern is the interpretability of an attack instance. Interpretability of an attack is extremely important to get necessary insights which may eventually lead to improved cipher and countermeasure designs.

Our Contributions: In this paper, we initiate an automated framework *ExpFault* for DFA, simultaneously meeting the goals specified in the last paragraph. In order to achieve these goals a simple strategy has been adopted, which just estimates the attack complexity instead of doing the attack explicitly to recover the secret. In the light of this simple strategy, and a rigorous formalization of the cipher description and DFA, ExpFault evaluates fault exploitability in three steps, the first among which is the identification of a set of potential wrong key distinguishers. The generic distinguisher identification step is realized by analyzing fault simulation data with assistance from standard data-mining strategies. The goodness of each DFA distinguisher is also evaluated by means of a metric based on Shannon Entropy. The next step to distinguisher identification is the evaluation of attack complexity. We propose a graph based abstraction of the cipher to realize this step, which works by automatically identifying a divide-and-conquer strategy for evaluating distinguishers on different key guesses. The choice of the divide-and-conquer strategy has a major role in determining the attack complexity. Finally, we figure out the overall attack complexity by calculating the size of the keyspace after a single fault injection and estimate the number of fault injections required to reasonably figure out the key.

In order to explain the main concepts, three attack examples corresponding to AES [7] and PRESENT [8] block ciphers have been used in this paper. However, the second major contribution of this work is to analyze a recently proposed cipher GIFT [21]. A thorough analysis of GIFT with different fault models has been performed with the help of the proposed framework. Interestingly, we found that complete 128 bit key of GIFT can be extracted with a computational complexity of 2^{14} by means of a single nibble fault injected at the 25th round of the cipher in the best case. Moreover, the attack is found to be optimal from an information theoretic perspective.[2].

[1]This claim is certainly not restricted to DFAs only and is valid for other classes of fault attacks as well.
[2]The proof of optimality is provided in the Appendix B

Related Work: Automation of fault attacks has been addressed in recent past via the Algebraic Fault Analysis (AFA) framework [18]. The main idea of AFA is to encode the cipher and a fault instance to a low-degree system of multivariate polynomial equations, which is then solved with SAT solvers by converting it to a Boolean formula in Conjunctive Normal Form (CNF). Analyzing a single fault instance in AFA thus involves solving a SAT problem, which often requires prohibitively long time, making it a bad choice in the exploitable fault space characterization context. Moreover, the attacks reported by AFA are often difficult to interpret. As a result, they do not provide any clue by which one may improve the design and implementation of the cipher. Recently, Barthe et.al. [19] have proposed a framework for synthesizing fault attacks automatically given a software implementation using concepts of program synthesis. However, their framework mainly targets for public key cryptosystems.

The most relevant work in the present context is due to Khanna et. al. [14], who proposed a framework called XFC based on principles somewhat similar to that of ExpFault. The key component of XFC is the characterization of the fault propagation path by means of coloring, where each color represents a variable. The coloring based static analysis eventually provides a scalable way for the calculation of the attack complexity as well. Albeit being scalable, the usability of the XFC scheme is found to be limited to a specific class of DFAs. More specifically, it fails to detect distinguishers, which typically exploit the constraints on the values that certain fault difference variables may assume. Impossible Differential Fault Analysis (IDFA) attacks are prominent examples of such cases. Further, XFC scheme lacks proper automation in its attack complexity analysis algorithm and makes certain simplifying assumptions, which fails to capture the most generic scenario. The potential drawbacks of XFC have been elaborated by means of examples in the Appendix C of this paper, which also establishes the relevance of different strategies used in ExpFault. A prior approach of using data mining for DFA identification was proposed in [20]. However, the work there does not provide any instance of novel fault analysis, neither discusses their optimality. Moreover, [20] does not shed any light on the exact implementation details of the framework, which the present work attempts. The appendix here presents extensive descriptions of the tool with actual outputs, illustrating the working principles of the automation with exact cipher instances.

Scopes of a Data-based Approach: The ExpFault framework mines distinguishers from fault simulation data. This data analysis approach of distinguisher identification shows enough potential to be extended for other genres of fault attacks viz. Integral Fault Attacks [15] and Differential Fault Intensity Analysis attacks (DFIA) [5]. A unified framework for automated fault analysis will be the ultimate goal which is initiated in this work by means of ExpFault. One should notice that it is not straightforward to extend equation based approaches like AFA for attacks like DFIA which are mainly statistical in nature. Data analysis thus seems to be a better alternative for such cases.

The rest of the paper is organized as follows. We start by describing two well-known ciphers AES and PRESENT, for which we provide 3 attack examples using the proposed framework (Sec. 1). In Sec. 2, the cipher, and the fault models are formalized. The ExpFault framework is described next in Sec. 3. Proof-of-Concept evaluations of some known attacks on AES and PRESENT are elaborated as examples while describing the scheme. Sec. 4 presents DFA results on the GIFT block cipher. Finally, we conclude in Sec. 5.

1 Preliminaries

In this section, we introduce some basic terminology encountered frequently in this paper. Some of them will be formally defined according to our cipher model in the following section. We also provide a brief description of the two ciphers AES and PRESENT in this section which are to be used as examples throughout this paper.

1.1 Basic Terminology

Block ciphers are the realizations of Pseudo-Random Permutations (PRP). In general, block ciphers are constructed by repeating a *round* multiple times (perhaps with slight modifications in some iterations). Each round is a sequence of *sub-operations*. In this paper, the input of each sub-operation is called an *intermediate state* (also known simply as a *state*). With the injection of a fault, states assume faulty values which differ from the correct values assumed by them in the absence of the fault. We use the term *state differential* to represent the XOR-difference between the correct and faulty computation of a state. Each state differential consists of word variables known as *state differential variables*, where the word size typically depends on the cipher under consideration.

1.2 AES

The AES block cipher is the current worldwide standard for symmetric key cryptography. The widely used version AES-128 uses a block size of 128 bits and a master key of the same size, all of which are processed as 1-byte chunks. The encryption is realized by iterating a round function 10 times. The round function of AES consists of 4 sub-operations namely, `SubBytes`, `ShiftRows`, `Mixcolumns` and `AddRoundKey`. The `SubBytes` consists of 16 identical 8×8 nonlinear S-Boxes. The `ShiftRows` sub-operation is a permutation realized at the byte level, whereas the `Mixcolumns` is a linear transformation by means of a Maximum Distance Separable (MDS) matrix. The last sub-operation in a round is the `AddRoundKey`, which performs a bitwise XOR operations between the state and 128-bit round keys generated by means of a key schedule for each round, from the master key. It is worth mentioning that the round function in the last round of AES does not include the `Mixcolumns` sub-operation.

AES is the most widely analyzed cipher in the context of fault attacks, especially for DFA [1]. Most of the DFA attempts on AES till date target the last three rounds of the cipher. The most optimal attack on AES is due to Tunstall et. al. [17], which requires only a single byte fault injection at the beginning of the 8-th round of the cipher resulting in a keyspace of size 2^8. The computational complexity of this attack is 2^{32}. In [3], Saha et. al. have shown that the same attack can still be realized even if multiple bytes at the beginning of 8-th round gets faulty. The only constraint is that the faulty bytes must remain within the same diagonal in the matrix representation of the AES state. Further, in [2], Derbez et. al. proposed Impossible Differential Fault Attack (IDFA) and Meet-in-the-Middle (MitM) attack, both of which target the beginning of the 7th round of AES. Finally, Kim et. al. proposed Integral Fault Attacks on AES [15]. Being well explored, AES is a major mean for experimentally validating our framework in this paper. More specifically, we shall show that our framework, in its current state can detect the standard DFA attempts on AES including the IDFA attacks.

1.3 PRESENT

The PRESENT [8] is a widely known block cipher of the lightweight genre. The PRESENT–80 version of the cipher utilizes an 80 bit master key with a 64 bit block size. Round keys of 64 bits are generated from the 80 bit key state for 31 iterations having the same round structure. The constituent sub-operations for the round function are `AddKey`, `sBoxLayer`, and `pLayer`, of which the `sBoxLayer` is a nonlinear layer consisting 16 identical 4×4 bijective S-Boxes. The linear diffusion layer of PRESENT is constructed with a simple bit-permutation operation which is significantly different and simpler than that of the MDS based diffusion functions of AES.

Just like AES, PRESENT has gone through several fault analysis attempts mostly targeting the 28, 29th rounds of the cipher as well as the key schedule. [12, 11, 22, 18]. In this paper, we shall use the attack proposed by Jeong et. al. [13] on the 28-th round of the cipher to explain various parts of the framework. In particular, this attack requires 2 instances of 16 bit faults injected at the beginning of the 28th round. The computational complexity of the attack is $O(2^{32})$.

2 A Formalization of the Differential Fault Analysis

In this section, we construct a formal notion of the cipher representation as well as the differential fault analysis, which perfectly suits our purpose in this paper. We begin with a general view of the DFA attacks and eventually present the formal framework.

2.1 DFA on Block Ciphers: A Generic View

The general concept of DFA remains the same for most of the ciphers, except some manual cipher-specific tricks, which make the automation a challenging task. DFAs broadly follow three major steps:

1. **Distinguisher Identification:** The key step of DFA is to identify wrong key distinguishers, which are typically relations on the state differential variables. According to the well-known wrong key assumption, a distinguisher attains a uniform distribution with a wrong key guess and a non-uniform one with a correct key guess, which eventually helps to reduce the candidate key space. In the context of DFA, however, distinguishers are mostly described as mathematical expressions rather than statistical distributions.
2. **Divide-and-Conquer:** The step following the distinguisher identification is the evaluation of the same with different key guesses to filter out the wrong keys in a computationally efficient manner. Not every distinguisher is efficiently computable and the computational efficiency lies in two facts: 1) whether it can be partitioned into independent subparts; and, 2) whether each subpart is efficiently computable, that is with a reasonable number of exhaustive key guesses.
3. **Estimating the Number of Possible Key Candidates:** The sole idea of DFA is to reduce the complexity of the exhaustive key search by means of the distinguisher. However, the reduction of the search space typically depends upon the distinguisher used. If the distinguisher is unable to sufficiently reduce the search space complexity, more faults should be injected. Thus, the quality of a distinguisher must be quantified to achieve successful and practical attacks.

Automation of the above-mentioned steps demands a mathematical specification of the cipher and the faults, to begin with. The following subsections present a formalization of the cipher and the differential fault attacks in this context. To maintain clarity, a list of notations used is provided in Table 1.

2.2 Representing A Block Cipher

A block cipher is a mapping $\mathscr{F}_k : \mathscr{P} \rightarrow \mathscr{C}$, where, $\mathscr{P}$ and $\mathscr{C}$ denote the plaintext and ciphertext space, respectively. The mapping is typically specified by a key $k \in \mathscr{K}$. Structurally, they can be represented as a tuple of invertible functions as:

$$\mathscr{F}_k = \langle o_1^1, o_1^2, ..., o_1^d, o_2^1, o_2^2, ..., o_2^d, ...o_R^1, o_R^2, ..., o_R^d \rangle \tag{1}$$

Typically, for a given $p \in \mathscr{P}$ and a fixed $k \in \mathscr{K}$, there exists a unique $c \in \mathscr{C}$ such that, $c = o_R^d(o_R^{d-1}(...(o_1^2(o_1^1(p))...)$. Here, each o_j^i represents the i-th sub-operation in the j-th round of a R round cipher. Further, each o_j^i can be represented as:

$$o_j^i(x_1, x_2, ...x_l) = \bigoplus_{h_1=1}^{h_1=l} a_{h_1} \cdot x_{h_1}, \quad \text{if } o_j^i \text{ is linear} \tag{2}$$

$$o_j^i(x_1, x_2, ...x_l) = \bigoplus_{h_1=1}^{h_1=2^l} a_{h_1} \cdot \prod_{h_2 \in I} x_{h_2}, \quad \text{if } o_j^i \text{ is nonlinear} \tag{3}$$

Table 1: List of Notations used

Notation	Meaning
$\lvert\cdot\rvert$	Size of a set
$\mathscr{F}_k$	Block Cipher
$\mathscr{P}, \mathscr{C}, \mathscr{K}$	Plaintext, Ciphertext and Key space
R	Total number of iterative rounds in the block cipher.
d	Total number of sub-operations in each round.
o^i_j	The i-th sub-operation in the j-th round.
$\mathscr{E}_k$	Data-centric view of the cipher.
s^i_j	The state at the input of the i-th sub-operation in the j-th round.
λ, m	Block size; word-size (size of each word in the cipher state in bits)
$l = \frac{\lambda}{m}$	word count
F	A fault instance.
X	Fault affected register.
r, wd, t, f	Fault round, width, location and value.
δ^i_j	state differential at the input of the i-th sub-operation in the j-th round.
w^{ij}_z	a state differential variable (discrete random variable) corresponding to a m-bit word of the state differential δ^i_j
Δ_k	Set of state differentials of the cipher
$p_z^{w^{ij}}$	probability distribution of w^{ij}_z
$H(\cdot)$	Entropy
$T(.)$	Dataset for state differentials for each w^{ij}_z.
$\{\mathscr{D}^i_j\}$	Set of distinguishers formed with state differentials
$\mathscr{T}$	the enumeration algorithm for the key set using distinguisher
$Comp(\mathscr{T})$	the complexity of the distinguisher enumeration algorithm
$\mathscr{R}$	Remaining key space.
IS, VS	Itemset and Variable set.
MKS, VG	Maximal independent key set, and Variable group.

Here, $I \subseteq \{1,2,....l\}$, and each a_{h_1} is a constant. The data width of the function inputs is a notable factor in this description. Given the block width of a cipher is λ bits, it is processed as m-bit words, where $m = \frac{\lambda}{l}$. We call m as the *word size* of the cipher. It is worth mentioning that the data width of each sub-operation might not be the same for a given cipher. In such cases, we assume the data width of the input of nonlinear sub-operations as the *word size*.

In an alternative data-centric view, the cipher $\mathscr{F}_k$ is represented as a sequence of states as follows:

$$\mathscr{E}_k = \langle s^1_1, s^2_1, ..., s^d_1, s^1_2, s^2_2, ..., s^d_2, ...s^1_R, s^2_R, ..., s^d_R, c\rangle \tag{4}$$

where each s^i_j represents the input of the i-th sub-operation in the j-th round of a R round cipher. Intuitively this representation presents an execution trace of the cipher on a plaintext p and a key k. Each s^i_j actually refers to an *internal state* (or simply *state*) of the cipher. The $\mathscr{E}_k$ is also referred to as the *execution trace* of the cipher. The state sequence begins with the plaintext $p = s^1_1$. Each state s^i_j is a vector of length l of m-bit words. The values assumed by the state vectors are subject to change with the variation of the plaintext and the key. Intuitively, the data-centric specification formally represents the simulation data from the cipher.

2.3 Formalization of the DFA

The formalization of DFA requires a precise specification of the injected faults. In general, it is assumed that injected faults are localized and transient so that they can affect at least one bit from a chunk of contiguous bits within a state, at some specific round. If a fault affects some part of the input state of the sub-function o^i_j, the output of o^i_j will differ from its expected value. We provide a formal representation of a fault as a tuple $F = \langle s^{i_1}_r, \lambda, wd, t\rangle$, which is similar to that of [18]. Here, $s^{i_1}_r$ represents the state, where the fault is injected. It is apparent that $r < R$. The λ parameter denotes the data width of the state, wd is the width of the fault, and t is the fault location within the state. Let us denote any $s^i_j = \langle V_1, V_2, ...V_l\rangle$, where each V_z ($z \in \{1,2,...,l\}$) is an m-bit variable. The localized fault, depending on the scenario, will affect one or more of these variables. In general, this is determined by the width of the fault wd. To simplify the matter we assume that wd is either $\leq m$ or it is a multiple of m. As a result, one or more of the V_zs can be affected by the fault. For simplicity, it is further assumed that only consecutive V_zs can be affected by the fault and the location of that is indicated by the fault location parameter t, in the fault model. The width of the fault wd is often used to represent the fault models. In this work, we only consider standard fault models (the bit ($wd = 1$), nibble ($wd = 4$), and byte ($wd = 8$) fault models), although the framework is not limited to them.

Once the cipher and the fault model are determined, we can now formally describe the DFA attack on a cipher. In order to construct a general model for DFA, we first need to formally define the state differentials and the state differential variables, already introduced in Sec. 1.1. Let us consider the execution trace $\mathscr{E}_k$ of a cipher, as described in (4). In order to capture the effect of an injected fault, we define a *faulty execution trace* $\mathscr{E}_k^{'} = \langle s_1^1, s_1^2, ..., s_1^d, ..., s_r^{'i_1}, s_r^{'i_1+1}, ..., s_r^{'d}, ..., s_R^{'d}, c' \rangle$. Here each $s_j^{'i}$ denotes the faulty input of the i-th sub-operation in the j-th round ($r \leq j \leq R$) starting from the injection point of the fault at round r. Before the r-th round the states remain the same. A *state differential* is defined as $\delta_j^i = s_j^i \bigoplus s'^i_j = \langle V_1^{ij} \oplus V_1^{'ij}, V_2^{ij} \oplus V_2^{'ij}, ..., V_l^{ij} \oplus V_l^{'ij} \rangle = \langle w_1^{ij}, w_2^{ij}, ..$ $.., w_l^{ij} \rangle, \;\; r \leq j < R$, where, V_z^{ij} denotes the z-th m-bit correct state variable, and $V_z^{'ij}$ denotes the corresponding faulty state variable. For each such word, $\oplus$ denote the bitwise XOR operation. Each w_z^{ij} denotes a *state differential variable*. Finally, we define another formal structure called *differential execution trace* Δ_k as, $\Delta_k = \langle \delta_r^{i_1}, \delta_r^{i_1+1}, ..., \delta_r^d, ..., \delta_R^1, \delta_R^2, ..., \delta_R^d \rangle$. Each of the state differentials δ_j^i in Δ_k may potentially form a distinguisher.

Given a cipher $\mathscr{F}_k$ and a fault F in it, the DFA can be formally described as:

$$\mathscr{A} = \langle \{\mathscr{D}_j^i\}, \mathscr{T}, \mathscr{R} \rangle \tag{5}$$

where:

- $\{\mathscr{D}_j^i\}$ denotes a set of distinguishers, each of which could be a non uniform distribution or mathematical relation, over the state differential variables of some state differential δ_j^i.

- $\mathscr{T}$ is the exhaustive enumeration algorithm for the key set $\mathscr{K}$ via distinguisher evaluation. A proper divide-and-conquer strategy is essential for this enumeration algorithm, which enables the evaluation of the distinguishers in parts. The time complexity of the enumeration algorithm is one of the determining factors of the overall DFA complexity, which is $O(2^n)$, with $n \leq \log_2(|\mathscr{K}|)$. For practical cases $n \ll \log_2(|\mathscr{K}|)$, whereas $n = \log_2(|\mathscr{K}|)$ implies no gain from the perspective of an attacker.

- $\mathscr{R}$ is the remaining key search space after the injection of a single instance of the fault F. The evaluation of the distinguishers over the complete key set $\mathscr{K}$ partitions the set into two non-overlapping subsets $\mathscr{K}_w$ and $\mathscr{K}_{cr}$; the first one being the set of wrong keys and the second one being the set of candidate keys one of which is the correct key. Evidently, $\mathscr{R} = \mathscr{K}_{cr}$ and $|\mathscr{R}| \ll |\mathscr{K}|$ for an efficient fault attack. One should note that, it is sufficient to consider the search space reduction for one single fault instance, as the reduction for multiple fault instances can be easily calculated from that. $\mathscr{R}$ is often represented as the solution set of a system of equations or inequations, involving the keys and distinguisher variables.

3 A Framework for Exploitable Fault Characterization

In this section, we describe the proposed automated framework in detail. The following subsections, will provide generic algorithms for computing each of the components described in (5). The input to the framework is a mathematical description (linear layers as matrices and the S-Boxes as tables) and an executable model (software/hardware implementation) of the target block cipher along with an enumeration of the fault space under consideration. The output is the exploitable fault space.

3.1 Automatic Identification of Distinguishers

Algorithm 1 Procedure *RngChk*

```
Input: The dataset for a state $\delta^i_j$ as $T_{\delta^i_j} = \langle T_{w^{ij}_1}, T_{w^{ij}_2}, ..., T_{w^{ij}_l} \rangle$
Output: $\langle \{Rng_{w^{ij}_z}\}^l_{z=1}, H_{Ind}(\delta^i_j) \rangle$
1: $H_{Ind}(\delta^i_j) := 0$
2: for each $T_{w^{ij}_z} \in T_{\delta^i_j}$ do    ▷ [1]
3:     Store all distinct values assumed by $w^{ij}_z$ in $Rng_{w^{ij}_z}$    ▷ [2]
4:     Calculate the probability distribution of $w^{ij}_z$ as $p'^{w^{ij}_z}$    ▷ [3]
5:     Calculate the Entropy of $w^{ij}_z$ as $H_{Ind}(w^{ij}_z)$ using $p'^{w^{ij}_z}$
6: end for
7: Return $\langle \{Rng_{w^{ij}_z}\}^l_{z=1}, H_{Ind}(\delta^i_j) \rangle$
```

[1] $z = 1, 2, ..., l$
[2] Values of w^{ij}_z belongs to the set $\{0, 1, ...2^m - 1\}$
[3] $p'^{w^{ij}_z}_q := \frac{\#q}{|T_{w^{ij}_z}|}$, where $\#q$ denote the frequency of $q \in \{0, 1, ...2^m - 1\}$ in $T_{w^{ij}_z}$

Algorithm 2 Procedure *Miner*

```
Input: $T_{\delta^i_j} = \langle T_{w^{ij}_1}, T_{w^{ij}_2}, ..., T_{w^{ij}_l} \rangle$
Output: $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}^{|VS_{\delta^i_j}|}_{v=1}, H_{Assn}(\delta^i_j) \rangle$
1: $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}^{|VS_{\delta^i_j}|}_{v=1} \rangle := \texttt{Apriori}(T_{\delta^i_j})$
2: $H_{Assn}(\delta^i_j) := 0$
3: for each $v \in VS_{\delta^i_j}$ do
4:     $tot := \texttt{VarCount}(v) \times m$    ▷ [4]
5:     $p'^v_q := \frac{1}{|IS^v_{\delta^i_j}|}, \forall q \in IS^v_{\delta^i_j}$    ▷ [5]
6:     $p'^v_q := 0, \forall q \notin IS^v_{\delta^i_j}$
7:     $H_{Assn}(v) := -\sum_{q=0}^{2^{tot}-1} p'^v_q \log_2(p'^v_q)$    ▷ [6]
8:     $H_{Assn}(\delta^i_j) := H_{Assn}(\delta^i_j) + H_{Assn}(v)$
9: end for
10: Return $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}^{|VS_{\delta^i_j}|}_{v=1}, H_{Assn}(\delta^i_j) \rangle$
```

[4] `VarCount` returns the number of variables in a variable set
[5] Calculate the probability distribution of each variable set
[6] Calculate the entropy of variable sets

In the last section, we have abstractly defined distinguishers as state differentials having certain mathematical or statistical properties. However, a metric is required which can identify the state differentials having such special properties and also quantify the goodness of distinguishers. We define such a metric based on the *entropy of state differentials*. Here the state differential variables are considered as random variables.

Definition 1. [Entropy of a State Differential] *The entropy of a state differential $\delta^i_j = \langle w^{ij}_1, w^{ij}_2, ..., w^{ij}_l \rangle$, where each w^{ij}_z is a discrete random variable with probability distribution $p^{w^{ij}_z}$, is defined as $H(\delta^i_j) = H(w^{ij}_1, w^{ij}_2, ..., w^{ij}_l)$, that is the joint entropy of the random variables in the state differential.*

Definition 2. [Maximum Entropy of a State Differential] *The maximum entropy of a state differential $\delta^i_j = \langle w^{ij}_1, w^{ij}_2, ..., w^{ij}_l \rangle$, is defined as $H_{max}(\delta^i_j) = \sum_{z=1}^{l} H_{max}(w^{ij}_z) = \sum_{z=1}^{l} (-\sum_{q=0}^{2^m-1} p^{w^{ij}_z}_q \log_2(p^{w^{ij}_z}_q))$, where each w^{ij}_z is independent and uniformly distributed within the range $[0, 2^m - 1]$, given m is the bit width of variable w^{ij}_z.*

Note that, the maximum entropy defined here assumes the uniformity and independence of the associated random variables within a specific range $[0, 2^m - 1]$, where m is the bit length of each variable. In case, the variable is not uniform within this complete range the entropy will be less than the maximum entropy. Correlations among the variables will also cause entropy reduction. Next, we define the *distinguishing criteria* – the decision criterion for determining the distinguishing capability of state differentials.

Definition 3. [Distinguisher Criteria] *A state differential δ^i_j is called a distinguisher if the entropy $H(\delta^i_j)$ is less than the maximum entropy of the state differential.*

The main idea of our dynamic distinguisher identification scheme is to learn the distinguishers from the fault simulation data, acquired from the executable cipher model by varying the plaintexts, keys, and the fault values. Let us consider the differential execution trace Δ_k corresponding to a fault F. The values assumed by the variables associated with Δ_k vary with the change of plaintext, key and the fault value. Such variations result in a fault simulation dataset which is analyzed to identify distinguishers. Let us denote the datasets corresponding to each state differential δ^i_j as $T_{\delta^i_j}$.

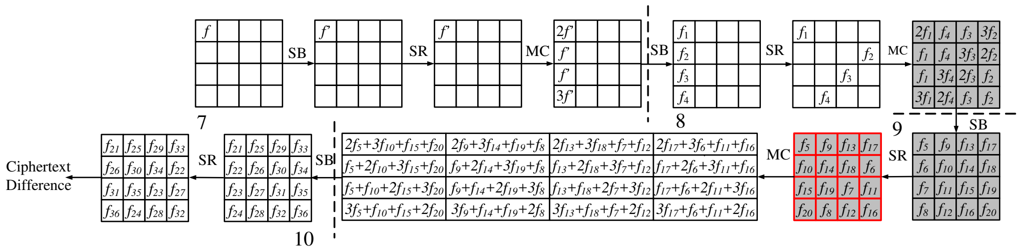


Figure 1: Fault propagation in impossible differential fault attack on AES and formation of the IDFA property (marked in red). None of the variables in this state differential can assume the value 0 for the correct key guess.

Each $T_{\delta^i_j}$ is a table, containing l, m-bit variables w^{ij}_z ($1 \leq z \leq l$) and data values, corresponding to each of them. For convenience, we further denote each column of a $T_{\delta^i_j}$ as $T_{w^{ij}_z}$. Corresponding to each fault according to our formalization, we have many such tables corresponding to each state differential in Δ_k. Typically, a subset of the possible state differentials actually qualifies as potential distinguishers. We denote $T_{\Delta_k} = \langle T_{\delta^1_r}, T_{\delta^2_r}, ..., T_{\delta^d_r}, ..., T_{\delta^1_R}, T_{\delta^2_R}, ..., T_{\delta^d_R} \rangle$ as the set of the tables for the state differentials. Our data-based framework tests each δ^i_j separately and decides whether it constructs a distinguisher. Two distinct cases can be identified in the course of the distinguisher identification which we outline next.

3.1.1 Case 1. The Variables are Independent, but not Uniform within the Complete Range:

In this typical case, the probability distributions of individual state differential variables change, while they still remain independent. Decrease in individual entropies of the variables due to their non-uniformity over the complete range $[0, 2^m - 1]$ (note that uniformity may still hold over some sub-range of $[0, 2^m - 1]$), causes a drop in the total state differential entropy. The situation is described in Algorithm 1, where the changed probability distributions are denoted as $p'^{w^{ij}}_z$ ($z = 1, 2, ..., l$), and the joint state differential entropy as $H_{Ind}(\delta^i_j) = \sum_{z=1}^{l} (-\sum_{q=0}^{2^m-1} p'^{w^{ij}_q}_z \log_2(p'^{w^{ij}_q}_z))$. Each column of the table $T_{\delta^i_j}$ (denoted as $T_{w^{ij}_z}$), corresponding to each variable w^{ij}_z is treated separately for missing values (if any) within the range of $[0, 2^m - 1]$. As a concrete example, if a state differential pose an impossible differential property, none of the w^{ij}_zs can assume value 0, and as a result, the value 0 will be missing in the table $T_{w^{ij}_z}$ for any z. Information regarding the values which are not missing are important in the context of the distinguisher and hence preserved for each w^{ij}_z in the set $Rng_{w^{ij}_z}$. Typical examples of Case. 1 include the IDFA attack on AES and the attack on PRESENT described in [13].

Example 1: IDFA Attack distinguisher on AES: IDFA attacks exploit a typical cipher property that, depending on the fault, the variables of a state differential corresponding to some internal state of a cipher may not attain certain values within their domains. Such a property is used in IDFA attacks to distinguish correct key guesses from wrong ones. For the IDFA attack on AES, a byte fault is injected at the beginning of the 7-th round of the cipher resulting in some state differentials none of whose variables can assume the value 0, with the correct key guess. The situation is elaborated in Fig. 1, where each large square represents an intermediate state differential of AES, with the fault injected at the beginning of the 7-th round in the 0-th byte location. Each small square in the figure represents a state differential variable of size one byte. The shaded states in Fig. 1 denote the existence of an impossible differential, with all bytes being active (fault difference cannot be 0). It is convenient to use the last among them as a distinguisher due to its proximity to the ciphertext (marked red in Fig. 1).

Table 2: Frequent Itemset Mining: Toy Example

TID	v_1	v_2	v_3	v_4	v_5
1	1	5	7	8	11
2	2	4	6	9	13
3	1	5	7	10	2
4	2	4	6	11	4
5	3	9	8	6	5
6	1	10	11	9	8

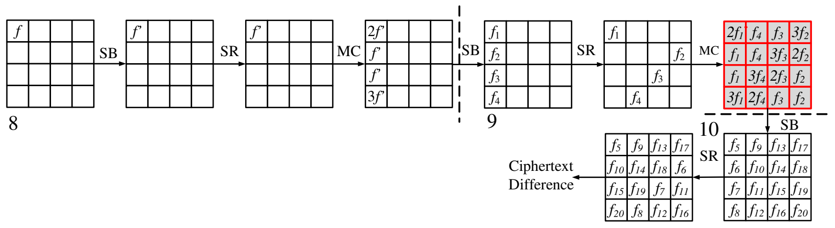


Figure 2: Fault propagation in AES with the fault injected at the beginning of 8-th round. Distinguisher is formed at the input of the 10-th round S-Box (marked in red)

It is apparent that the distinguisher identification framework of ours identify this impossible differential property as an instance of case 1. The `RngChk` function detects the absence of 0 in each of the variables, and as a result, the entropy becomes $H_{Ind}(\delta_9^3) = \sum_{z=1}^{16}(-\sum_{q=0}^{2^8-1} p_z'^{w_q^{ij}} \log_2(p_z'^{w_q^{ij}})) = \sum_{z=1}^{16}(-0 - \sum_{q=1}^{2^8-1} \frac{1}{255}\log_2(\frac{1}{255})) = \sum_{z=1}^{16}(-255 \times \frac{1}{255}\log_2(\frac{1}{255})) = 127.90$, which makes the state differential qualify as a distinguisher. One should note that, the state differentials δ_9^1 and δ_9^2 also possess the impossible differential property and are detected by the `RngChk` routine.

Example 2: A Distinguisher on PRESENT In this example, a fault is injected at the beginning of the 28-th round of the PRESENT cipher. The width of the fault is of 16 bits. The distinguisher identification algorithm, in this case, identify the input state of the S-Box of the $30-th$ round (δ_{30}^1) as the best distinguisher. The `RngChk` function identifies that each of the 4 bit variables in this state differential can assume only 2 values among 2^4 possible values (although the 2 values assumed may change depending on the fault locations), and as a result, the entropy becomes $H_{Ind}(\delta_{30}^2) = \sum_{z=1}^{16}(-2 \times \frac{1}{2}\log_2(\frac{1}{2})) = 16$.

3.1.2 Case 2. The Variables are not Independent

The second case of the distinguisher identification problem deals with the scenarios where correlations exist between some of the variables within a state differential, which eventually cause the reduction of state differential entropy. Typical examples exist for the ciphers with MDS matrices. Detection of the associations/correlations among the variables is crucial for calculating the entropy $H_{Assn}(\delta_j^i) = H(w_1^{ij}, w_2^{ij}, ..., w_l^{ij})$ in this case. We utilize well-known association rule mining (itemset mining) strategies for this purpose.

Frequent Itemset and Association Rule Mining:
Association rule/itemset mining is a widely explored, classical problem in the domain of data mining, which refers to the discovery of association relationships or correlations among a set of items. Formally, given a large number of variables (attributes) ($var_1, var_2, ..., var_n$), and a table/database of values they assume within their respective domains, an *item* is defined as $var_q = val$, where *val* lies in the domain of var_q. The simplest case occurs while dealing with discrete-valued variables having small ranges, where each item can be defined precisely. If $I = \{i_1, i_2, ..., i_a\}$ is a set of all items constructed from a table of discrete valued variables, then any $I_s \subset I$ is called an *itemset*. The prime task of an association rule mining algorithm is to figure out associations (if any) of the form $A \Rightarrow B$, where both A and B are propositional logic formulae over the items.

In the present context, we are mainly interested in itemsets and the variables associated with them. The number of all possible itemsets are exponential with the size of I, and most of them are not interesting for practical purpose. This fact leads to the finding of itemsets occurring frequently in a table, which is known as *frequent itemset mining*. The frequent itemset mining task is governed by a statistical parameter *support*, which represents the frequency of occurrence of an itemset in the database. Formally support of an itemset I_s in a table/database DB is defined as, $supp(I_s) = |I_s(t_i)|/|DB|$, where $I_s(t_i) = \{t_i | t_i \ is \ an \ entry \ in \ DB \ and \ t_i \ contains \ I_s\}$. An itemset is called a frequent itemset if its support is greater than or equal to some predefined minimum support value. Further, an itemset is called a *maximal frequent itemset* if none of its immediate supersets is frequent.

To illustrate the above-mentioned concepts precisely, let us consider the toy database presented in Table. 2. There are 5 discrete valued variables in this table having value ranges from 1 to 13. We set the support as $\frac{2}{6} = 0.33$. It can be easily figured out from Table. 2, that there are two itemsets of size 3, beyond this support threshold – namely $(v_1 = 1, v_2 = 5, v_3 = 7)$ and $(v_1 = 2, v_2 = 4, v_3 = 6)$. It is worth to note that, no superset of these itemsets are frequent (that is, these are the maximal frequent itemsets), and all subsets of these are frequent. Further, it is interesting to note that, for variable v_4 and v_5 all the itemsets are of cardinality 1. Intuitively, this implies that the variables v_4 and v_5 are statistically uncorrelated. Note that, setting the proper support is imperative, as otherwise, one may obtain a large number of itemsets of little practical interest.

Finding Itemsets within State Differentials:

In the context of distinguisher identification, we are mainly interested in the maximal frequent itemsets within some reasonable support. The key idea is to figure out the variables within a state differential, which are strongly correlated. For this purpose, we utilize the well-known *Apriori* association rule mining framework. The complete procedure is described in Algorithm 2. The algorithm takes a $T_{\delta^i_j}$ as input, which is then fed to the `Apriori` function after some basic preprocessing. From, each of the itemsets generated by the miner, we separate out the variables and create sets called *Variable Sets*. Variables within the same variable set are dependent, whereas they are assumed to be independent across different variable sets. Multiple itemsets exist corresponding to each *Variable Set* and a table is formed which stores each *Variable Set*, along with its corresponding itemsets. This table contains complete information regarding the distinguisher of our interest and is represented here as a pair $(VS_{\delta^i_j}, \ \{IS^v_{\delta^i_j}\}_{v=1}^{|VS_{\delta^i_j}|})$, where $VS_{\delta^i_j}$ denote the set of all variable sets and $\{IS^v_{\delta^i_j}\}_{v=1}^{|VS_{\delta^i_j}|}$ denote the set of itemsets corresponding to each variable set v. Next, the state differential entropy is calculated using this table, which involves the calculation of the joint distribution followed by the joint entropy of each variable set $v \in VS_{\delta^i_j}$ (line 6-8 in Algorithm 2). Using the independence assumption of the variable sets, these entropies can be summed up giving the total entropy of the state as $H_{Assn}(\delta^i_j)$.

Example 3. A Distinguisher for AES with a Byte Fault Injected at the Beginning of 8-th Round: This example elaborates the case 2 of the distinguisher identification problem. In this attack a byte fault is injected at the 0th byte of AES state at the beginning of 8th round. The propagation of the fault is illustrated in Fig. 2. Let us consider the state differential δ^1_{10}, which is the output of the 9-th round MixColumn. This state differential achieves the smallest entropy value and is eventually selected as the potential distinguisher for the attack. We now elaborate the entropy calculation for this state differential. The state differential $\delta^1_{10} = \langle w^{110}_1, w^{110}_2, ..., w^{110}_l \rangle$ contains 16 state differential variables, each with bit-width $m = 8$. The maximum entropy here is $H_{max}(\delta^1_{10}) = 128$. However, the function `Miner` reveals variable associations. More specifically, there are 4 variable sets $(w^{110}_1, w^{110}_2, w^{110}_3, w^{110}_4)$, $(w^{110}_5, w^{110}_6, w^{110}_7, w^{110}_8)$, $(w^{110}_9, w^{110}_{10}, w^{110}_{11}, w^{110}_{12})$, and $(w^{110}_{13}, w^{110}_{14}, w^{110}_{15}, w^{110}_{16})$ (variable numbering was done column-wise maintaining the convention in AES), each having 255 itemsets for them. The joint entropy of each variable set v becomes $H_{Assn}(v) = \sum_{q=1}^{255} \frac{1}{255} \log_2(255) = 7.99$, which finally results in the state differential entropy of

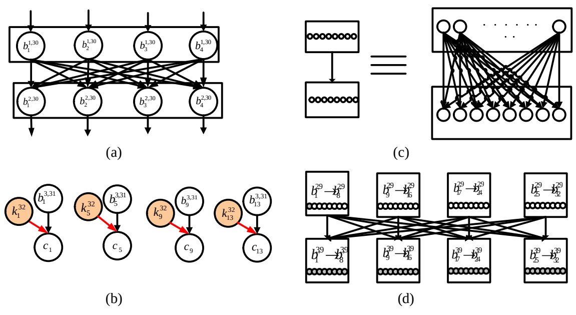


Figure 3: Example: Subgraphs corresponding to different sub-operations of a cipher

$H_{Assn}(\delta_{10}^{1}) = 4 \times 7.99 = 31.96$.

Complete Distinguisher Identification Flow:
The complete distinguisher identification algorithm takes the dataset $T_{\Delta_k} = \langle T_{\delta_r^1}, T_{\delta_r^2}, ..., T_{\delta_R^d} \rangle$ as input, and outputs a set $Dist = \{\langle \mathscr{D}_j^i,\ H_j^i \rangle\}$, where $\mathscr{D}_j^i$ is a distinguisher corresponding to the state δ_j^i (only if δ_j^i satisfies the distinguishing criterion), and H_j^i is the entropy of this distinguisher. The entropy H_j^i is typically the minimum of $H_{Ind}(\delta_j^i)$, $H_{Assn}(\delta_j^i)$ (returned by `RngChk` and `Miner`, respectively), and $H_{max}(\delta_j^i)$ (calculated according to the Definition 2). Indeed, $H_j^i < H_{max}(\delta_j^i)$ is the essential criterion for a state differential to qualify as a distinguisher. It is worth to note that, $\mathscr{D}_j^i$ contains the complete description of a distinguisher, obtained by combining the outputs of `RngChk` and `Miner`, given by, $\mathscr{D}_j^i := \langle \{w_z^{ij}\}_{z=1}^{l},\ \{Rng_{w_z^{ij}}\}_{z=1}^{l},\ VS_{\delta_j^i},\ \{IS_{\delta_j^i}^{v}\}_{v=1}^{|VS_{\delta_j^i}|} \rangle$. The pseudocode for this algorithm is rather straightforward and is thus omitted here.

Determining the proper distinguisher:
The distinguisher identification step usually returns a set of potential distinguishers with their respective entropies specifying their qualities. In general, the distinguisher having the lowest entropy is the best for obvious reasons. However, the evaluation complexity of a given distinguisher plays a crucial role in its selection for a practical attack, as will be shown in the next subsection. After the completion of the first phase of the algorithm, we simply retain all the discovered distinguishers. This is because their usefulness is difficult to decide at this point. However, some of the distinguishers can be instantly eliminated based on some simple rules. It is mandatory to have an S-Box between any distinguisher and the ciphertext. In an even strict sense, if one intends to extract round keys from a specific round with a given distinguisher, he/she must have an S-Box between the distinguisher and the key addition step. Otherwise, the difference equations for key extraction cannot be constructed. Based on this rule, one can clearly eliminate some of the distinguishers, if possible. A concrete example of such a situation is discussed in the next section in the context of IDFA attack on AES.

3.2 Enabling Divide-and-Conquer in Distinguisher Enumeration algorithm $\mathscr{T}$

Injection of a fault results in a set of distinguishers with different entropy values, as shown in the previous subsection. However, only a few of them are practically utilizable for attack, as the usability of a distinguisher depends on the complexity of evaluating it exhaustively. In DFA, the distinguishers are evaluated in the form of a system of difference equations (or inequations) and the solution space of the system results in a reduced set of candidate keys containing the correct key. Given this system, the practicality of a DFA attack depends on two factors:

1. *Distinguisher Evaluation Complexity:* The complexity of exhaustively enumerating the system for all possible key guesses.

2. *Offline Complexity:* Size of the remaining keyspace after distinguisher evaluation, which has to be searched exhaustively.

Following the notation described in Eq. 5, we denote the *Distinguisher Evaluation Complexity* with $Comp(\mathscr{T})$ where $\mathscr{T}$ is the distinguisher enumeration algorithm and $Comp(\cdot)$ denotes the complexity of an algorithm. The *Offline Complexity*, on the other hand, is denoted with $|\mathscr{R}|$ where $\mathscr{R}$ is the remaining key space after distinguisher evaluation. The $Comp(\mathscr{T})$ and the $|\mathscr{R}|$, can be estimated once the systems of equations for the distinguishers are in place. The *Attack Complexity* of a DFA can be determined as:

$$Comp(\mathscr{A}) = max(Comp(\mathscr{T}), |\mathscr{R}|) \tag{6}$$

One should note that, the definition of attack complexity at this point assumes that only one fault instance has been injected. The attack complexity indeed depends on the number of faults injected. However, for most of the cases the required number of injections for making an attack practical can be determined from the value of $Comp(\mathscr{A})$ with a single fault instance and so we define it in terms of a single injection.

Knowing the systems of difference equations apriori, is not a very practical assumption for an automated tool, as it depends upon the distinguisher(s) chosen. The most critical factor associated with distinguisher evaluation is to choose a proper divide-and-conquer strategy for enumerating the solution space of the difference equation system. Instead of guessing the complete key at once (which has a prohibitively large complexity), such a strategy allows guessing small key parts exhaustively and as a result the correct key can be recovered in parts with a practical time complexity. In this work, we construct such equation systems automatically in an abstract form, which is suitable for the purpose of attack complexity evaluation. Further, this abstract description can be extended to concrete fault difference equations, if required. To automatically determine the divide-and-conquer strategy we propose a graph based abstraction of the cipher called *Cipher Dependency Graph* (CDG). Let us represent each state s^i_j as $s^i_j = \langle b^{ij}_1, b^{ij}_2, ..., b^{ij}_\lambda \rangle$, where each b^{ij}_z corresponds to a *bit variable*[1]. Given this representation of the states, we define the CDG for a block cipher as follows:

Definition 4. [Cipher Dependency Graph] *A Cipher Dependency Graph (CDG) for a block cipher is a directed acyclic graph (DAG) $\mathscr{G}\langle \mathbb{V}, \mathbb{E} \rangle$, where every node $v \in \mathbb{V}$ corresponds to a bit variable b^{ij}_z ($1 \leq z \leq \lambda$) at the input of round j and sub-operation i of the cipher. A directed edge $e \in \mathbb{E}$ represents the dependency between two bit variables belonging to two consecutive states s^i_j and s^{i+1}_j (or s^1_{j+1}) imposed by the sub-operation o^{i+1}_j at the abstraction level of bits, considering the bit variable of s^i_j as input, and that of s^{i+1}_j as the output, respectively.*

Certain simplifying assumptions were made, while constructing the CDGs. Some basic CDG building blocks are illustrated in Fig. 3. For the S-Boxes, we assume that each output variable is dependent on all the S-Box inputs (Fig. 3 (a)). The key addition operations are represented by structures shown in Fig. 3 (b). Permutation layers are often straightforward and thus not shown here. However, some linear operations like MDS matrices need special care (more specifically the linear layers which involves XOR operations). Fig. 3 (d) represents one such scenario for 8-bit variables, which are shown in groups for convenience. The MDS structures are also complete graphs (of 32 vertices in this example). It is worth to mention that, the graph $\mathscr{G}$ is completely cipher-specific, and thus one needs to construct it only once while doing the exploitable fault analysis for a specific cipher. A CDG, corresponding to a fault attack test case on PRESENT is illustrated in Fig. 4. For ease of understanding, only the sub-graph relevant to an attack example is shown[2]. Interestingly, the CDG is already divided into clearly identifiable levels.

[1]This is in contrary to the last subsection, where they (the states) were represented as vectors of variables of size m bits.
[2]We have described the distinguisher corresponding to this attack in Example 2.

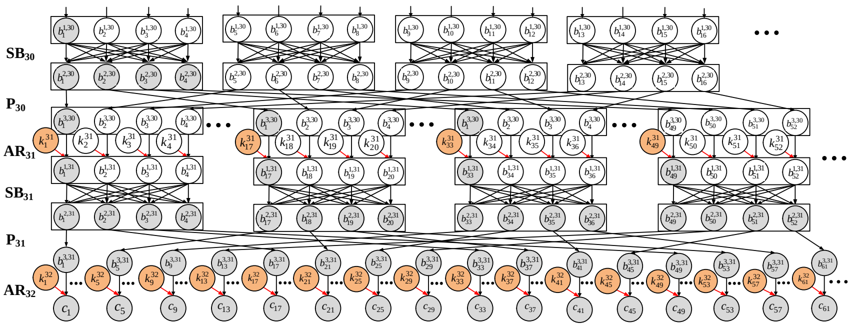


Figure 4: Example: Finding Key Parts for the Distinguisher Evaluations in PRESENT

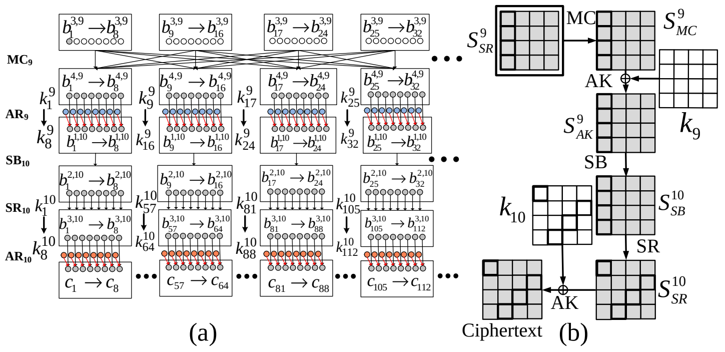


Figure 5: Example: Finding Key Parts for the Distinguisher Evaluations in AES:

Construction of a Divide and Conquer Strategy:
The next step to the CDG construction is the identification of independent key parts to be guessed. For a given distinguisher, we initiate a series of breadth first searches (BFS) up to the ciphertexts nodes of the CDG. Each BFS search begins with a bit variable at the state, where the distinguisher has been constructed. The search typically figures out all the mutually dependent bit variables starting from the start node, in the form of the BFS tree (refer to Fig. 4 for example). Once the BFS tree is obtained, one can figure out the key nodes attached to it in $O(1)$ complexity.

Example 4: For the sake of illustration, let us refer to Fig. 4 once again. The distinguisher under consideration is the one described in Example 2, which is being constructed at the input of the 30th round S-Box operation (the first layer of nodes shown in Fig. 4.) In the figure, the colored circles represent the associated state bits one must compute to calculate the first bit in the distinguisher. The key bits one need to guess to calculate the shaded state bits are shown in red, while the associated state bits are represented in grey. All the colored variables here are the part of a BFS tree. Further, from the BFS tree of Fig. 4, the key variables to be guessed can be extracted which are 20 in number for the first bit. In summary, to calculate the first bit of the distinguisher, it is sufficient to guess these 20 bits together and no other key bit is required to be guessed. This provides the divide-and-conquer we require.

Optimizations: Certain intricacies are there to be taken care of while collecting the independent key parts for a distinguisher. Interestingly, not all key variables obtained by the BFS search are necessary. To illustrate this, we refer to the Fig. 5(a), which corresponds to the partial CDG for the IDFA attack on AES. [1] For convenience, the word level representation of the CDG is also provided along with (Fig. 5(b)). It is easy to observe from the word level representation that the key bits

[1]The IDFA distinguisher was described in Example 1.

corresponding to k_9 are not required to be guessed for distinguisher evaluation. The reason behind this fact is that there is no nonlinear layer between the key variables in k_9 and the distinguisher in S^9_{SR}. As a result, these key variables get cancelled out with the calculation of the differential. However, these key variables will still be detected by the BFS based search. Fortunately, we can easily enhance the proposed mechanism to encompass such scenarios. The idea is to keep the track of the non-linear layers (S-Boxes) encountered, at each level of the CDG during the BFS traversal. This can be easily done by maintaining counters within the nodes of the CDG. While collecting the key variables, if it is found that the level corresponding to the key variables is not preceded by any S-Box level, the keys can be discarded. Referring to Fig. 5(a), the key nodes in blue color thus can be discarded. The first bit of this distinguisher can be evaluated by guessing just 32 key bits of k_{10}.

Calculation of the Distinguisher Evaluation Complexity: The BFS based key part finding algorithm actually returns sets of key bits, corresponding to each bit of the distinguisher state. However, in order to calculate the quantities $Comp(\mathscr{T})$ and $|\mathscr{R}|$ we need to exploit some more structural properties of the cipher, already present in the CDG. As for most of the time, we are dealing with m bit distinguisher variables, it is trivial to combine the key bit sets corresponding to each m bit variable. One should also consider combining the key bit sets corresponding to the variable sets (if any). While evaluating any of these variables/variable sets, the corresponding keys must be guessed simultaneously. At this point, certain other things are to be taken care of. Let us consider a distinguisher $\delta^i_j = \langle w^{ij}_1, w^{ij}_2, ..., w^{ij}_l \rangle$. Corresponding to each w^{ij}_z, there exists a set of key bit variables. An obvious way is to view the relationships as a bipartite graph, as shown in Fig 6. Without loss of generality, we just consider variables and not the variable sets in this discussion, although the same logic applies to the later one. Let us denote the key set corresponding to each variable w^{ij}_z as *Key Set* (KS_z). The key sets, however, may have overlaps. As a concrete example, one may consider the PRESENT case study depicted in Fig. 4. All 4 consecutive nibbles in the distinguisher at round 30 (shown in the diagram as layer SB_{30}), depends upon the same 16 round key bits from the last round. Such overlaps are extremely important from the divide-and-conquer point of view. This is because, the overlaps indicate that all the difference equations, that can be constructed involving these key bits and the associated variables w^{ij}_z will share the key variables. As a result they must be evaluated simultaneously. Putting it in a more simplified manner, if there are overlaps, computation related to all the overlapping variables must be performed simultaneously. To deal with such cases, we define *Maximum Independent Key Sets* (MKS), which are non-overlapping subsets of key variables, constructed by taking the union of overlapping KS_zs. Each MKS_h also impose a grouping on the corresponding w^{ij}_zs attached to its component KSs. We call such groupings as *Variable Groups* (VG).[1] Intuitively, each $\langle VG_h, MKS_h \rangle$ tuple refers to a set of independent equations to be solved for the key extraction. In our graph based representation, we informally refer them as *independently computable chunks/subparts*.

Calculation of $Comp(\mathscr{T})$ becomes trivial after the above-mentioned grouping. Let us consider, an MKS as MKS_h and the corresponding variable group as VG_h (note that variable groups may also include variable sets as its elements.). Each VG_h can be evaluated independently. Let us assume that we have M such VG_hs along with their corresponding MKS_hs. The time complexity of computing each of them is given as $Comp(\mathscr{T}_h) = 2^{|MKS_h|},\ 1 \leq h \leq M$. It is quite obvious that such a search can be performed (and should be) in a parallel manner. As a result, the overall complexity of the distinguisher enumeration algorithm $\mathscr{T}$ becomes:

$$Comp(\mathscr{T}) = max_h(Comp(\mathscr{T}_1), Comp(\mathscr{T}_2), ..., Comp(\mathscr{T}_M)) \qquad (7)$$

Example 7. IDFA attack on AES: In this example, we figure out the evaluation complexity of the IDFA diatinguisher. It is observed that each byte of the distinguisher depends on 32 key bits from the 10-th round. Further, 4 consecutive state differential variables are found to depend on the same 32 key bits. Following our notation size of each VG_h here is 4 state differential variables

[1]Note that we have used the term "group" to differentiate it from the variable sets. From this point onwards, we shall use *variable set* and *variable group* to identify these two separate entities. Variable sets can be members of variable groups.

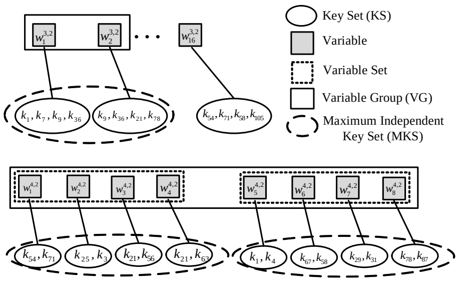


Figure 6: Illustration of the Relationships between Key and Distinguisher Variables

and the associated MKS_h contains 32 bits of key. Clearly, the distinguisher evaluation complexity $Comp(\mathscr{T})$ becomes 2^{32} in this case.

Example 8. AES with 8th round fault injection: In this case, the distinguisher consists of 4 variable sets, containing 4 variables each. Each 8-bit variable in the distinguisher state depends on 8 consecutive key bits (that is key bytes), and with the existence of variable sets having cardinality 4, one must consider $8 \times 4 = 32$ key bits simultaneously, for guessing (i.e. $|MKS_h| = 32$). Further, the key bytes associated with each variable set are independent, and hence each VG_h will contain only a single variable set. Overall, $Comp(\mathscr{T}) = 2^{32}$.

Example 9. PRESENT with 28th round fault injection: The distinguisher here is formed at the input of the 30-th round S-Box. As it can be seen from Fig. 4, each distinguisher bit (actually each nibble) here depends on 20 key bits. However, due to the overlappings present in different nibble-wise key sets (KSs), the distinguisher evaluation process can eventually be partitioned into 4 independent (MKS, VG) pairs, each having 32 key bits involved – 16 from the last round and rest from the penultimate round. The size of corresponding VG_hs become 4 state differential variables each. As a result, $Comp(\mathscr{T})$ becomes 2^{32}.

3.3 Complexity Evaluation of the Remaining Key Space $\mathscr{R}$

The final step in finding a successful DFA is the evaluation of the remaining keyspace size ($|\mathscr{R}|$) after the fault injection. Often, the complexity remains beyond the practical exhaustive search complexity with a single fault injection and as a result, one might require multiple faulty ciphertexts. Nevertheless, the required number of faults for the successful attack can be estimated from the remaining space complexity of a single injection, and hence we specifically focus on the remaining search space with a single fault. A distinguisher $\mathscr{D}_j^i$ and the corresponding key parts obtained in the last two steps can be utilized to figure out the remaining keyspace complexity efficiently. Another important component of this computation is the *differential characteristic* of the S-Boxes. Differential characteristic (DC) of an S-Box S basically reports the average number of solutions an S-Box differential equation may have. They can trivially be calculated from the Differential Distribution Table (DDT) of an S-Box. The DC values for different ciphers can be found in [14].

The algorithm for remaining search space evaluation is presented in Algorithm 3. The main idea in this step is to figure out the probability, with which the distinguishing property occurs during distinguisher enumeration with random key guesses. This probability is then multiplied with the total key space in the corresponding MKS, giving the remaining search space complexity. Referring to the algorithm, the input consists of the corresponding distinguisher $\mathscr{D}_j^i$, and a set of tuples with cardinality M, which contains the MKSs and corresponding VGs. As an additional component, the DC characteristic of the S-Boxes $\mathscr{H}_S^h$ corresponding to each MKS_h, VG_h pair is also supplied. The $\mathscr{H}_S^h$ is the DC value corresponding to each (MKS_h, VG_h) pair. In some cases, a distinguisher may involve multiple S-Box layers and as a result $\mathscr{H}_S^h$ should be multiplied many times for each distinguisher variable (or variable set) evaluation. To keep things simple we directly provide the algorithm with properly tailored values within $\mathscr{H}_S^h$. Values of the $\mathscr{H}_S^h$ with above-mentioned tailoring can be trivially obtained from the CDGs described in the last subsection, just by keeping track of the S-Boxes encountered with the distinguisher.

Example 9. IDFA attack on AES: In this case, it turns out that $|\mathscr{R}|_{VG_1} = 2^{32} \times (\frac{255}{2^8})^4$ (roughly equal to $2^{32} - 2^{26}$). This is because each state differential variable in VG_1 assumes the distinguishing property with probability $\frac{255}{2^8}$ and there are 4 such variables in VG_1. The size of the remaining keyspaces are the same for other three $\langle MKS_h, VG_h \rangle$ pairs. The large size of the remaining keyspace indicates the need of multiple fault injection. Although, the estimation of the required number of faults here is slightly nontrivial due to the impossible differential inequalities involved, it can be estimated using the construction from [2]. Overall, the attack complexity is $O(2^{32})$ and total 2^{11} faults will be required to extract the key uniquely [2].

Example 10. AES with 8th round fault injection: The MKS and VGs, which are the inputs to the Algorithm 3 are 4 in number in this case. Further, each VG contains a single variable set and 32 key bits corresponding to that. One needs to consider the number of itemsets corresponding to each variable set (or variable group, as in this case each group contains a single variable set) in this case. For each of the 4 variable sets, the probability of occurrence of the distinguishing criterion is $\mathbb{P}[VG_h] = \frac{255}{2^{32}}$. The DC characteristic of AES S-Box is found to be 1 and the total number of key possibilities are 2^{32}. The remaining keyspace corresponding to each variable set thus becomes $2^{32} \times 2^{-24} = 2^8$, leading to a complete remaining search space complexity of $(2^8)^4 = 2^{32}$. One should exhaustively search this remaining keyspace for the correct key. The total complexity of the attack, considering both $Comp(\mathscr{T})$ and $|\mathscr{R}|$ thus remains 2^{32}.

In [17], Tunstall et. al. presented a 2 step approach for the attack described here, which eventually reduces the remaining keyspace size to 2^8. The idea is to complete the attack just described, and then to exploit another distinguisher δ_9^1 which was previously costly to evaluate on the complete keyspace. However, one should notice that the attack complexity still remains 2^{32}. The distinguisher identification framework of ExpFault detects both the distinguishers. The two-step attack requires the existence of the inverted key schedule equations. The proposed tool in its current form requires some manual intervention to handle attacks using multiple distinguishers. In future versions of the tool this dependency will be removed. However, it is worth mentioning that, the results already outputted by the tool are sufficient to easily discover such attacks.

Example 11. Attack on PRESENT: The distinguisher evaluation process, in this case, can eventually be partitioned into 4 independent $\langle MKS_h, VG_h \rangle$ pairs, each having evaluation complexity of 2^{32}. For each of the 4 (MKS_h, VG_h) pair, the probability of occurrence of the distinguishing criterion is $\mathbb{P}[VG_h] = (\frac{2}{16})^4 = 2^{-12}$, and the remaining key space size is $|\mathscr{R}|_{VG_h} = 2^{20}$. With a single fault injection, thus the keyspace reduces to 2^{80} from 2^{128} [1] in this case, and the attack demands the injection of at least another fault (complexity becomes $(2^{32} \times (2^{-12})^2)^4 = 2^{32}$, which is fairly reasonable). In summary, with 2 fault injections of 16 bit width, the 80 bit key can be figured out with $Comp(\mathscr{T}) = 2^{32}$ and $|\mathscr{R}| = 2^{32}$.

Discussion: Before going to the case studies, let us summarize the ExpFault framework in nutshell. For each fault instance, the tool analyzes all state differentials starting from the fault injection point and figures out a set of distinguishers from them. Each of these distinguishers is then analyzed with the graph-based framework and the evaluation and offline complexities are determined. The best performing distinguisher can be instantly chosen to realize the attack. However, in certain cases, multiple distinguishers can be combined to get better attacks. In the

[1]The distinguisher here simultaneously extracts round keys from last two rounds of PRESENT. Total 128 key bits are extracted which can uniquely determine the 80 bit master key by using key scheduling equations.

Algorithm 3 Procedure *EVAL_ SEARCH_ SPACE*

```
Input: $\mathscr{D}_j^i$, $\{\langle MKS_h,\ VG_h, \mathscr{H}_S^h\rangle\}_{h=1}^M$
Output: Complexity of the remaining search space $\mathscr{R}$, after one fault injection
    ($|\mathscr{R}|$).
1:  $|\mathscr{R}| := 1$
2:  for each $VG_h$ do
3:      $\mathbb{P}[VG_h] := 1$
4:      for each $g_h \in VG_h$ do
5:          if $(VS_{\delta_j^i} == \phi)$ then            ▷ $\mathscr{D}_j^i$ includes no variable sets
6:              $count := |Rng_{g_h}|$
7:              $b_c := m$
8:          else                                        ▷ $\mathscr{D}_j^i$ includes variable sets
9:              $count := |IS_{\delta_j^i}^{g_h}|$
10:             $b_c :=$ VarCount$(g_h) \times m$
11:         end if
12:         $\mathbb{P}[VG_h] := \mathbb{P}[VG_h] \times \frac{count}{2^{b_c}}$
13:     end for
14:     $k_{size} :=$ BitCount$(MKS_h)$                                       ▷ [1]
15:     $|\mathscr{R}|_{VG_h} := 2^{k_{size}} \times \mathbb{P}[VG_h] \times (\mathscr{H}_S^h)^{|VG_h|}$
16:     $|\mathscr{R}| := |\mathscr{R}| \times |\mathscr{R}|_{VG_h}$
17: end for
18: Return $|\mathscr{R}|$
```

[1] `BitCount` returns the number of bit variables in (MKS_h)

Table 3: Summary of DFA attacks on GIFT. We consider a fault injection attempt a successful attack only if both the evaluation complexity and $|\mathscr{R}|$ is less than the size of the key space.

Fault Width	Round	Attack Results				
		Evaluation Complexity	$\lvert\mathscr{R}\rvert$	No. Faults	Keys Extracted	Comments
4	24	—	—	—	—	No attack found
	25	2^{14}	2^{14}	1	128	Best attack found
	26	2^2	2^2	1	96	Cannot extract full key
	27	2^2	2^2	1	64	Cannot extract full key
8	24	—	—	—	—	No attack found
	25	2^{16}	2^{16}	1	128	Best attack found
	26	2^2	2^4	1	96	Cannot extract full key
	27	2^2	2^4	1	64	Cannot extract full key

next section, we provide a typical example of such situations. Our tool was able to figure out the optimal attacks in these cases. Further details on the framework are provided in Appendix A.

4 Case Studies

Until now we have provided proof-of-concept evaluations of the proposed framework on two well-known ciphers – AES-128 and PRESENT-80. Three known attacks have been elaborated step by step in the form of examples. However, both of the ciphers have been examined thoroughly for exploitable faults. It has been observed that exploitable faults are limited up to 7-th round in AES and 28-th round in PRESENT, which agrees with the existing literature. In this section, we evaluate the ExpFault framework on a recently proposed cipher called GIFT [21]. To the best of our knowledge, GIFT has never been considered explicitly in the context of DFA. With help of the framework, we were able to figure out several interesting attack instances, which establishes the effectiveness of the proposed framework in the context of exploitable fault characterization.

4.1 Differential Fault Attack on GIFT Block Cipher

GIFT [21] is a lightweight block cipher proposed in CHES 2017. The basic construction of the algorithm bears resemblance to the PRESENT block cipher. However, specific changes were made to make it even more lightweight while ensuring improved resistance against certain attacks like linear hulls. More specifically, GIFT utilizes a different 4×4 S-Box and an improved bit permutation layer along with a new key addition layer which uses only 32 and 64 bit round keys for

Table 4: Distinguishers of the best attack found

Distinguisher	Location	Description
$\mathscr{D}^2_{27}$	Input of `PermBits` in round 27	$w^{227}_1 \in \{0,3,7,11,15\}$, $w^{227}_2 \in \{0,3,7,11,15\}$, $w^{227}_3 \in \{0,3,7,11,15\}$, $w^{227}_4 \in \{0,3,7,11,15\}$, $w^{227}_5 \in \{0,3,5,7,9,13\}$, $w^{227}_6 \in \{0,3,5,7,9,13\}$, $w^{227}_7 \in \{0,3,5,7,9,13\}$, $w^{227}_8 \in \{0,3,5,7,9,13\}$, $w^{227}_9 \in \{0,5,6,9,10,13,14\}$, $w^{227}_{10} \in \{0,5,6,9,10,13,14\}$, $w^{227}_{11} \in \{0,5,6,9,10,13,14\}$, $w^{227}_{12} \in \{0,5,6,9,10,13,14\}$, $w^{227}_{13} \in \{0,5,6,8,9,10,11,12,15\}$, $w^{227}_{14} \in \{0,5,6,8,9,10,11,12,15\}$, $w^{227}_{15} \in \{0,5,6,8,9,10,11,12,15\}$, $w^{227}_{16} \in \{0,5,6,8,9,10,11,12,15\}$
$\mathscr{D}^1_{27}$	Input of `SubCells` in round 27	$w^{127}_1 \in \{0,8\}$, $w^{127}_2 \in \{0,8\}$, $w^{127}_3 \in \{0,8\}$, $w^{127}_4 \in \{0,8\}$, $w^{127}_5 \in \{0,4\}$, $w^{127}_6 \in \{0,4\}$, $w^{127}_7 \in \{0,4\}$, $w^{127}_8 \in \{0,4\}$, $w^{127}_9 \in \{0,2\}$, $w^{127}_{10} \in \{0,2\}$, $w^{127}_{11} \in \{0,2\}$, $w^{127}_{12} \in \{0,2\}$, $w^{127}_{13} \in \{0,1\}$, $w^{127}_{14} \in \{0,1\}$, $w^{127}_{15} \in \{0,1\}$, $w^{127}_{16} \in \{0,1\}$
$\mathscr{D}^1_{26}$	Input of `SubCells` in round 26	$w^{126}_1 \in \{0,8\}$, $w^{126}_2 \in \{0\}$, $w^{126}_3 \in \{0\}$, $w^{126}_4 \in \{0\}$, $w^{126}_5 \in \{0,4\}$, $w^{126}_6 \in \{0\}$, $w^{126}_7 \in \{0\}$, $w^{126}_8 \in \{0\}$, $w^{126}_9 \in \{0,2\}$, $w^{126}_{10} \in \{0\}$, $w^{126}_{11} \in \{0\}$, $w^{126}_{12} \in \{0\}$, $w^{126}_{13} \in \{0,1\}$, $w^{126}_{14} \in \{0\}$, $w^{126}_{15} \in \{0\}$, $w^{126}_{16} \in \{0\}$
$\mathscr{D}^1_{25}$	Input of `SubCells` in round 25	$w^{125}_1 \in \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$, $w^{125}_2 \in \{0\}$, $w^{125}_3 \in \{0\}$, $w^{125}_4 \in \{0\}$, $w^{125}_5 \in \{0\}$, $w^{125}_6 \in \{0\}$, $w^{125}_7 \in \{0\}$, $w^{125}_8 \in \{0\}$, $w^{125}_9 \in \{0\}$, $w^{125}_{10} \in \{0\}$, $w^{125}_{11} \in \{0\}$, $w^{125}_{12} \in \{0\}$, $w^{125}_{13} \in \{0\}$, $w^{125}_{14} \in \{0\}$, $w^{125}_{15} \in \{0\}$, $w^{125}_{16} \in \{0\}$

GIFT-64 and GIFT-128, respectively. The round keys are derived from a 128-bit key state utilizing a simple linear key schedule operation. It has been demonstrated in [21], that even with these simplifications, GIFT is able to provide comparable (and sometimes improved) security margins than that of PRESENT, SIMON [26], and SKINNY, while classical attacks are considered.

In this work, we focus on GIFT-64 which iterates 28 times to generate the ciphertext. The round structure consists of 3 sub-operations `SubCells`, `PermBits`, and `AddRoundKey`, where `SubCells` is the nonlinear S-Box layer, `PermBits` is the bit permutation and `AddRoundKey` is the key addition layer. In order to optimize hardware resources, which is, in fact, the main design goal of GIFT, the `AddRoundKey` layer XORs only 32 round key bits and 6 round constant bits in each round. More specifically, 2 round key bits are XORed with each nibble. The simple key schedule operation of GIFT deserves special mention in this context. It is observed that *for any* 4 *consecutive rounds, the round keys used are completely independent of each other*. This observation has a significant impact on the complexities of the fault attacks. In this subsection, we aim to examine the security of GIFT against differential fault attacks, which to the best of our knowledge has never been considered explicitly. In [22] authors proposed a side channel assisted DFA on PRESENT, which is claimed to be equally applicable for GIFT. In [23] Breier et. al. proposed a side channel assisted differential plaintext attack on PRESENT which seems to be applicable to GIFT as well. None of these attacks, so far, have tried classical DFA attacks on GIFT, which makes GIFT a perfect candidate to be evaluated with our proposed framework.

After an extensive evaluation, we figured out several interesting attacks on GIFT-64, mostly while faults were injected at 25, 26, and 27-th rounds. Evaluation of these attacks was done based on 3 parameters – the distinguisher evaluation complexity $Comp(\mathscr{T})$, size of the remaining key search space $|\mathscr{R}|$ and the number of injected faults, all of which are outputted by our tool for each fault location. A summary of these attacks is presented in Table 3. In this subsection, we shall elaborate one attack where a 4-bit fault is injected before the S-Box operation at the 25-th round. This is the most efficient attack found so far with our framework. Further details of the attacks can be found in the Appendix B of this paper, where we also provide a proof of optimality for the best attack found from an information theoretic perspective.

Injection of one 4-bit fault at the beginning of 25 round constructs several distinguishers. Examples of these distinguishers are provided in Table. 4. Here the fault is injected at nibble location 0 from left. The attack will be described based on these distinguishers, as we have observed that injections at other nibble locations result in equivalent situations. Notably, none of these distinguishers contain any variable sets (i.e. variables are independent within the state differentials). However, the values assumed by the state differential variables differ which eventually create the distinguishing properties. The very first distinguisher utilized is being constructed in round 27 at the output of the S-Boxes ($\mathscr{D}^2_{27}$). This distinguisher (Row 1 of Table. 4) results in the extraction of the round keys of the 28th round. The evaluation complexity of this distinguisher is $Comp(\mathscr{T}) = 2^8$ and the $|\mathscr{R}| = 2^{11.53} \approx 2^{12}$, with a single fault injection. In other words, one needs to guess the

32 bit round key in 8 bit chunks to reduce the key search space from 2^{32} to 2^{12}.[1] The cause of obtaining such $Comp(\mathscr{T})$ is elaborated in Appendix B with graphs outputted by the tool.

In order to obtain the master-key of GIFT, one must extract all the 128-bits. Unfortunately, the equations from the key schedule cannot be used in extracting rest of the key-bits with the knowledge of 32 bit, due to the very special nature of GIFT key schedule mentioned above. One must extract all 4 consecutive round keys in order to get a complete attack. In this scenario, our framework efficiently exploits all other distinguishers obtained. The second distinguisher that we utilize is constructed at the input of the S-Boxes at round 27 ($\mathscr{D}_{27}^{1}$). This distinguisher, which can extract the round keys of round 27, has an evaluation complexity of 2^2 provided the keys of round 28 are known. For each of the 2^{12} choices of the 28th round key, the size of the remaining key space (i.e. the keyspace of the round keys from 27-th round) is 1, which leaves us with total 2^{12} choices for the keys of last two rounds.

In the third phase of the attack, we utilize the distinguisher formed at the input of the S-Box layer of round 26 ($\mathscr{D}_{26}^{1}$). With the knowledge of the keys from round 27 and 28, keys of round 26 can be determined uniquely with an evaluation complexity of 2^2 only. As a result, the evaluation this distinguisher leaves us with total 2^{12} choices for last 3 round keys. In this final step of the attack, we exploit the state differential as distinguisher where the fault was actually injected ($\mathscr{D}_{25}^{1}$). 30 key bits of round 25 can be uniquely determined (for each of the 2^{12} choices of previous round keys), except the 2 bits associated with nibble 0, which is the fault injection point. This is because this nibble provides very little entropy reduction (only the value 0 is missing) and exhaustive search is the best option here (because we have only 2^2 choices for the key bits associated). As a result, we get total 2^2 choices for the 25-th round keys for each choice of previous round keys. Finally, the size of the total keyspace becomes $2^{12} \times 2^2 = 2^{14}$. The overall value of $Comp(\mathscr{T})$ is given by $max(2^8, 2^2 \times 2^{12}, 2^2 \times 2^{12}, 2^2 \times 2^{12}) = 2^{14}$. Each of the components of $Comp(\mathscr{T})$ corresponds to an independent computation chunk which can be executed on a single thread, in parallel to other similar chunks running in other threads. Such independent computation chucks result from the divide-and-conquer strategy automatically identified by our tool.

5 Conclusion

In this paper, we have proposed an automated framework for exploitable fault identification in modern block ciphers. The main idea is to estimate the attack complexity without doing the attack in the original sense. Moreover, the proposed framework is fairly generic to cover most of the existing block ciphers, and provides high fault coverage and degree of automation. Three step-by-step case studies on different ciphers and fault attack instances were presented to establish the claims. Further, the tool has been utilized to figure out DFA attacks on a recently proposed block cipher GIFT. Future works will target further automation and generalization of the proposed framework as well as comprehensive analysis of different existing ciphers using it. Some obvious future extensions include the attack automation on key schedule and round counters. Another extremely important goal could be the detection of integral attacks, DFIA attacks, and MitM attacks [15, 16] which also seems to be feasible in this data-analysis based framework. Further, design of countermeasures with the assistance from this tool could be another research direction.

6 Acknowledgements

We would like to acknowledge Synopsys Inc, USA for partially supporting the research through the grant entitled "Formal Methods for Physical Security Verification of Cryptographic Designs Against Fault Attacks". We are also grateful to the anonymous reviewers for their insightful comments and suggestions.

[1]Following the terminology used in this paper, here we have $|MKS_h| = 8$ and $|VG_h| = 4$

References

[1] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. *CRYPTO'97*, pages 513–525, 1997.

[2] Patrick Derbez, et. al. Meet-in-the-middle and impossible differential fault analysis on AES. In *CHES'11*, pages 274–291. Springer, 2011.

[3] D. Saha, et. al. A diagonal fault attack on the advanced encryption standard. In *IACR Cryptology ePrint Archive*, 2009:581, 2009.

[4] Pierre Dusart, et. al. Differential fault analysis on aes. In *ACNS'03*, pages 293–306. Springer, 2003.

[5] N. Ghalaty, et. al. Differential fault intensity analysis. In *FDTC 2014*, pages 49–58, IEEE 2014.

[6] Mark Hall, et. al. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[7] John Daemen, et. al. The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media, 2013.

[8] Andrey Bogdanov, et. al. PRESENT: An ultra-lightweight block cipher. In *CHES'07*, pages 450–466. Springer, 2007.

[9] J. Guo, et. al. The LED block cipher. In *CHES'11*, pages 326–341. Springer, 2011.

[10] C. Beierle, et. al. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *CRYPTO'16*, pages 123–153. Springer, 2016.

[11] G. Wang and S. Wang. Differential fault analysis on PRESENT key schedule. In *CIS*.pages 362–366, IEEE, 2010.

[12] X. Zhao, et. al., "Fault-propagate pattern based DFA on PRESENT and PRINTcipher," In *J. Wuhan Univ. Natur. Sci.*, pages 485–493, 2012.

[13] Kitae Jeong, et. al. Improved differential fault analysis on PRESENT-80/128. *International Journal of Computer Mathematics*, 90(12):2553–2563, 2013.

[14] Punit Khanna, et.al. XFC: A Framework for eXploitable Fault Characterization in Block Ciphers. In *DAC 2017*. p. 8. ACM.

[15] Chong Hee Kim. Efficient methods for exploiting faults induced at aes middle rounds. *IACR Cryptology ePrint Archive*, 2011:349, 2011.

[16] Zhiqiang Liu, et. al. Meet-in-the-middle fault analysis on word-oriented substitution-permutation network block ciphers. *Security and Communication Networks*, 8(4):672–681, 2015.

[17] Michael Tunstall, et. al. Differential fault analysis of the advanced encryption standard using a single fault. In *WISTP'11*, pages 224–233. Springer, 2011.

[18] Fan Zhang, et. al. A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. *IEEE Transactions on Information Forensics and Security*, 11(5):1039–1054, 2016.

[19] Gilles Barthe, et. al. Synthesis of fault attacks on cryptographic implementations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014*, pages 1016–1027, ACM 2014.

[20] S. Saha, et. al. An Automated Framework for Exploitable Fault Identification in Block Ciphers – A Data Mining Approach. In *PROOFS 2017*, pages 50–67, EasyChair 2017.

[21] S. Banik, et. al. GIFT: a small PRESENT. In *CHES 2017*, pages 321–345, Springer 2017.

[22] S. Patranabis, et. al. One Plus One is More than Two: A Practical Combination of Power and Fault Analysis Attacks on PRESENT and PRESENT-like Block Ciphers. In *FDTC 2017*, pages 25–32, IEEE 2017.

[23] J. Breier, et. al. SCADPA: Side-Channel Assisted Differential-Plaintext Attack on Bit Permutation Based Ciphers. In *DATE 2018* (to appear), IEEE 2018.

[24] K. Sakiyama, et. al. Information-theoretic approach to optimal differential fault analysis. *IEEE Transactions on Information Forensics and Security*, 7(1):109–120, 2012.

[25] S. Saha, et. al. Automatic Characterization of Exploitable Faults: A Machine Learning Approach. In *Cryptology ePrint Archive, Report 2017/1008*.

[26] R. Beaulieu, et. al. The SIMON and SPECK lightweight block ciphers. In *DAC 2015*, pages 1–6, IEEE 2015.

A Implementation Details of ExpFault

This section elaborates the implementation details of an initial prototype of the ExpFault framework. The tool is mostly written in Python-3, with an exception for the data mining algorithm (Apriori) for which we use the WEKA [6] toolbox implemented in Java. In the following subsections, several attributes of the tool will be elaborated. We shall also point out some limitations of our current implementation in nutshell.

A.1 Assumptions

The main reason behind the development of ExpFault is a fast and cipher-oblivious characterization of exploitable faults. As we shall show in later subsections, even in its prototype implementation, ExpFault is able to characterize individual fault instances within a very reasonable time. In this work, we have characterized each fault location of the ciphers under consideration in an exhaustive manner (only up to a reasonable number of rounds.). However, it is worth mentioning that the characterization can be made significantly faster considering the structural symmetries present in standard block ciphers. For example, it is well-known that all 16 byte locations of AES in a specific stage of computation are equivalent as fault injection points. The presence of such symmetries should extensively reduce the number of fault locations to be checked. However, a systematic analysis of such equivalences is out of scope for this paper.

Exploitable fault analysis is expected to be performed by an evaluator. During the construction of the framework, we made assumptions which are only consistent in the context of an evaluator. For example, the fault locations are assumed to be known, which is indeed a reasonable assumption in our case. However, it is worth mentioning that the attacks discovered by our tool can be extended in an unknown fault location context with a reasonable penalty incurred on the attack complexities.

A.2 Inputs and Outputs

In its current form, ExpFault takes an executable of the cipher algorithm as input, which is mainly used to generate fault simulation traces. Simulation traces are dumped in `.arff` format which is the default data format for the WEKA toolset. The framework also expects an input file describing the cipher, which is used to generate the CDG and other internal data structures. We call this input as the *Cipher Description file*. A cipher description file mainly contains abstract descriptions of

```
BEGINBLOCK SLAYER
OPTYPE NONLINEAR
OPINPUT 64
OPOUTPUT 64
% 0 = 0,1,2,3
% 1 = 0,1,2,3
% 2 = 0,1,2,3
% 3 = 0,1,2,3

% 4 = 4,5,6,7
% 5 = 4,5,6,7
% 6 = 4,5,6,7
% 7 = 4,5,6,7
:
:
:
:
ENDBLOCK SLAYER
```

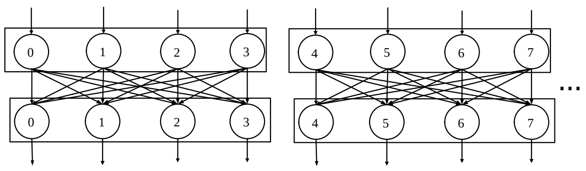


Figure 7: Code snippet from cipher description file and its corresponding graphical representation

the sub-operations as specified in Fig. 3. As a concrete example, we provide the description for two 4×4 S-Boxes in Fig. 7. It can be observed that the S-Box is specified in a bitwise manner, where the number at the left side of the "=" sign specifies the sink node of a directed edge, and the number at the right side of the "=" specifies the source nodes. The complete S-Box layer is required to be specified within a single `<BEGINBLOCK>···<ENDBLOCK>` construct. Likewise one can define other sub-operations.

The outputs corresponding to each fault injection is printed in a text file. The standard output contains the description of all the distinguishers found along with their entropy. The evaluation complexity of each distinguisher and the size of the remaining keyspace after distinguisher evaluation is also outputted along with that. Attack evaluation with multiple distinguishers and key schedule equations is not fully automated yet. However, the outputs provided are sufficient to figure out attacks using multiple distinguishers. Example of a single distinguisher is provided in Fig. 8, just to illustrate the output format and the information provided. It is also possible to extract the independent and parallely executable computation chunks (resulted from the divide-and-conquer based distinguisher evaluation) in the form of subgraphs from ExpFault. Such subgraphs are extremely useful for interpreting and implementing the attacks. Examples of such subgraphs will be provided in Appendix B.

A.3 Setup for Distinguisher Identification

Frequent itemset mining is crucially dependent on the *support* parameter of the mining algorithm. The implementation of the Apriori algorithm we used (from WEKA package [6]), iteratively decrements the support from a value of 1.0 to a predefined lower bound. To generate all desired maximal frequent itemsets, the support lower bound of Apriori was experimentally decided to be $\frac{1}{2^m}$ (m: bit length of each variable). The maximality of the itemsets was ensured experimentally by varying the support threshold as well as the data set size, which also nullifies the risk of generating an insufficient number of itemsets. For ciphers having MDS or similar operations we found that the dataset size of 12750 (that is 10 plaintexts, 5 different keys, and all 255 possible fault values) for 128-bit ciphers , and 750 for 64-bit ciphers (10 plaintexts, 5 different keys, and all 15 possible fault values), are sufficient to discover all possible itemsets. The relatively small dataset size is attributed to their deterministic fault propagation patterns (i.e. number of active S-Boxes are same for all fault values at a specific location). However, for ciphers with bit permutation operations, the

```
-----------------------------------------------------------
Distinguisher Evaluation Complexity (in log scale) 8
Remaining Key Space Complexity (in log scale) 11.53668207643374

Distinguisher Level 79
Round_no 27
Subop_no 2
Has_associations False
Entropy  43.536682076433735

V10 [0, 5, 6, 9, 10, 13, 14, ]
V12 [0, 5, 6, 8, 9, 10, 11, 12, 15, ]
V14 [0, 5, 6, 8, 9, 10, 11, 12, 15, ]
V15 [0, 5, 6, 8, 9, 10, 11, 12, 15, ]
V2 [0, 3, 7, 11, 15, ]
V6 [0, 3, 5, 7, 9, 13, ]
V8 [0, 5, 6, 9, 10, 13, 14, ]
V7 [0, 3, 5, 7, 9, 13, ]
V3 [0, 3, 7, 11, 15, ]
V1 [0, 3, 7, 11, 15, ]
V11 [0, 5, 6, 9, 10, 13, 14, ]
V9 [0, 5, 6, 9, 10, 13, 14, ]
V0 [0, 3, 7, 11, 15, ]
V13 [0, 5, 6, 8, 9, 10, 11, 12, 15, ]
V5 [0, 3, 5, 7, 9, 13, ]
V4 [0, 3, 5, 7, 9, 13, ]

No Variable sets exist..

----------------------------------
```

Figure 8: Description of a ditinguisher from the output file.

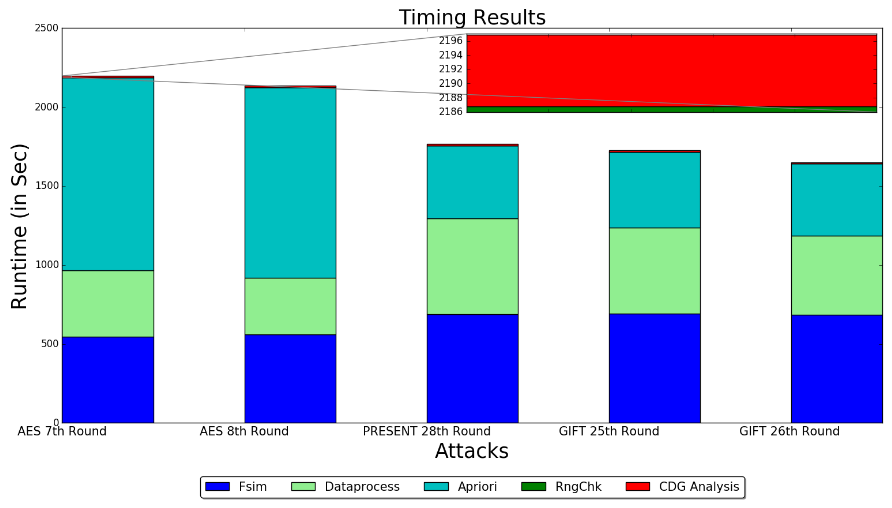


Figure 9: Runtime analysis: different components of the runtime are shown. Runtimes of `RngChk` and CDG based complexity analysis are found to be negligibly small (shown in zoom).

dataset size should be larger as the fault propagation pattern becomes probabilistic. We found that a dataset of 25500 works well for both PRESENT and GIFT. Varying the keys, plaintexts and the fault values ensure that the discovered rules/itemsets are independent of all these factors, which is essential for a DFA distinguisher. An interesting feature of the itemset generation algorithm is that it returns null when all the variables are independent. The `RngChk` function does not require any parameter setting and works fine with the dataset sizes provided above.

There always exists a risk of generating spurious itemsets in frequent itemset mining, especially with very low support values. However, detecting such spurious tuples are not very difficult as the variables from consecutive state differentials have well-defined mathematical relations. Any itemset not obeying these relations can be easily removed as spurious. In our experiments, we observed some spurious tuples for GIFT with low support values which were successfully eliminated using the structural knowledge from the cipher.[1] Such structural knowledge is already available in the form of the CDG and thus can be exploited without incurring any significant computational cost.

A.4 Analysis of Runtime

The runtime of the framework is a crucial factor for exploitable fault identification. In order to get a clear idea about different subparts of the framework, extensive runtime analysis was performed. The results are summarized in Fig. 9 for different attack examples considered in this paper. All the experiments were performed on a laptop with Intel Core i5 processor, 8 Gb RAM running Ubuntu 16.04 as the OS.

It is evident from Fig. 9 that the Apriori algorithm dominates the runtime which is about 1220 sec. for the AES 7th round attack example and moderately less in other examples. Fortunately, this step can be extensively parallelized as the analysis of state differential datasets are completely independent of each other. Although, the current prototype does not implement any such parallelization, improvement of runtime can be anticipated. Similar arguments can be made for the fault simulation which is another dominating factor in the tool runtime. As an alternative strategy for runtime improvement, knowledge about the cipher structure can be exploited. For example, it is quite well understood that ciphers with bit permutation layers cannot have variable associations. One may opt to skip the itemset mining step while analyzing such ciphers, in order to improve the runtime. The third dominating component of the runtime is the data processing operation which is an implementation specific overhead. This timing overhead is attributed to the python wrapper for reading `.arff` files. In future versions of the tool, we shall try to get rid of such unnecessary overheads. To summarize, the framework takes less than an hour to analyze each fault instance which can be improved further. The usability of ExpFault for exploitable fault space characterization is thus evident.

A.5 Current Limitations

The limitations of the initial prototype have already been elaborated in the previous subsection. The tool at its current state cannot directly handle key schedule relations and attacks on the key schedules. Handling key schedule relations during complexity calculation is not very difficult and extension of the tool can handle it. However, handling key schedule attacks will require algorithmic improvements in the complexity analysis step. It also worth to mention that the current implementation does not directly handle attacks using multiple distinguishers and minor human intervention is still required. However, the algorithm for using multiple distinguishers is straightforward and is planned to be incorporated in the next version of the tool. [2]

[1]In fact, all the itemsets found for GIFT were spurious and the cipher does not have variable associations. In such cases, the support can be increased to stop spurious itemset generation.

[2]One should note that the tool is still in its initial phase and we shall try to address all the above-mentioned issues before making it open source.

B More on the DFA of GIFT

In this section, we provide further details about the DFA attacks discovered by the ExpFault framework. Specifically, we shall elaborate the distinguisher evaluation complexity of the attack described in our case study with the help of graphs generated by our tool. This will be followed by a brief discussion on some other attack instances on the cipher. Finally, a proof of optimality will be provided for the best attack found.

B.1 Calculation of the Attack Complexity

Let us refer to the attack on GIFT described in Sec 4.1 of the paper. For each of the 4 distinguishers found we perform a series of BFS searches on the CDG of the cipher, which eventually provides independently and parallelly computable sub-parts (as described in Sec. 3.2) along with the attack complexity (described in Sec. 3.3)[1]. For each of the distinguishers, we provide the associated sub-graphs from the CDG, generated by our tool. As an example, Fig. 10(a) corresponds to the sub-graph associated with the distinguisher $\mathscr{D}_{27}^{2}$ – the first distinguisher utilized for the attack. For convenience, we refer to a magnified part of this sub-graph in Fig. 10(b).[2] The graph in Fig. 10(b) corresponds to one independently computable chunk for the $\mathscr{D}_{27}^{2}$ with 8 associated key bits, which results in an evaluation complexity of 2^8. Each numbered node corresponds to a bit variable of the cipher. The first layer of this graph (shown with its nodes colored) corresponds to state differential variables from distinguisher $\mathscr{D}_{27}^{2}$. It can be observed from Fig. 10(b) that total 4 state differential variables are associated with this independently computable chunk (their corresponding bits are shown with 4 different colors). These 4 variables construct a *variable group* VG_h according to the terminology used in the paper. The 16 key nodes in the graph are highlighted with black circles and squares. Interestingly, the square key nodes are not required to be guessed to evaluate the distinguisher, as there exists no nonlinear layer between these keys and the distinguisher. This leaves us with 8 key bits for this computing part which are the members of the MKS_h corresponding to the VG_h. Evidently, there are 4 such independent computing chunks in total, each with the same evaluation complexity of 2^8. As a result, $Comp(\mathscr{T})$ becomes 2^8.

The offline complexity $|\mathscr{R}|$ of the attack can now be computed individually for each computing subpart/chunk. Referring to the first computing subpart/chunk, there are 4 associated state differential variables w_1^{227}, w_w^{227}, w_3^{227}, w_4^{227} in VG_1, each of which can take at most 5 possible values from the set $\{0,3,7,11,15\}$. As a result, the size of the remaining key space becomes $|\mathscr{R}|_{VG_1} = \left(\frac{5}{16}\right)^4 \times 2^8 = 2.4414$. In a similar manner, remaining key space corresponding to 3 other computing chucks can be estimated as $|\mathscr{R}|_{VG_2} = \left(\frac{6}{16}\right)^4 \times 2^8 = 5.0625$, $|\mathscr{R}|_{VG_3} = \left(\frac{7}{16}\right)^4 \times 2^8 = 9.3789$ and, $|\mathscr{R}|_{VG_4} = \left(\frac{9}{16}\right)^4 \times 2^8 = 25.6289$. The offline complexity $|\mathscr{R}|$ thus becomes $2.4414 \times 5.0625 \times 9.3789 \times 25.6289 = 2^{11.53} = 2^{12}$.

Similar computations can be performed for the three other distinguishers associated with the attack. The complexity calculations are even simpler in these cases as the sub-graph from the CDG is already segregated into 16 independent components which make the complexity calculations trivial. Fig 11 presents the graph corresponding to $\mathscr{D}_{27}^{1}$ outputted by ExpFault framework. Here we assume that the 28th round keys are already set and as a result, they are not shown in the graph. It can be noticed from Fig 11(b) that the computing chuck only contains one state differential variable and 2 key bits (that is $|VG_h| = 1$ and $|MKS_h| = 2$). Each state differential variable in $\mathscr{D}_{27}^{1}$ may take 2 different values. As a result $|\mathscr{R}|_{VG_h} = \left(\frac{2}{16}\right) \times 2^2 = 0.5$, which essentially means that the key can be uniquely determined. Thus 27-th round keys can be uniquely extracted, provided some values are set for the 28-th round keys. The same calculations can be repeated for two other distinguishers and hence we omit them here.

[1]Refer to Table 4 for the description of the distinguishers.

[2]The round constant bits of GIFT cipher are not shown in the graphs as they are found to have no effect on the DFA complexity calculation.

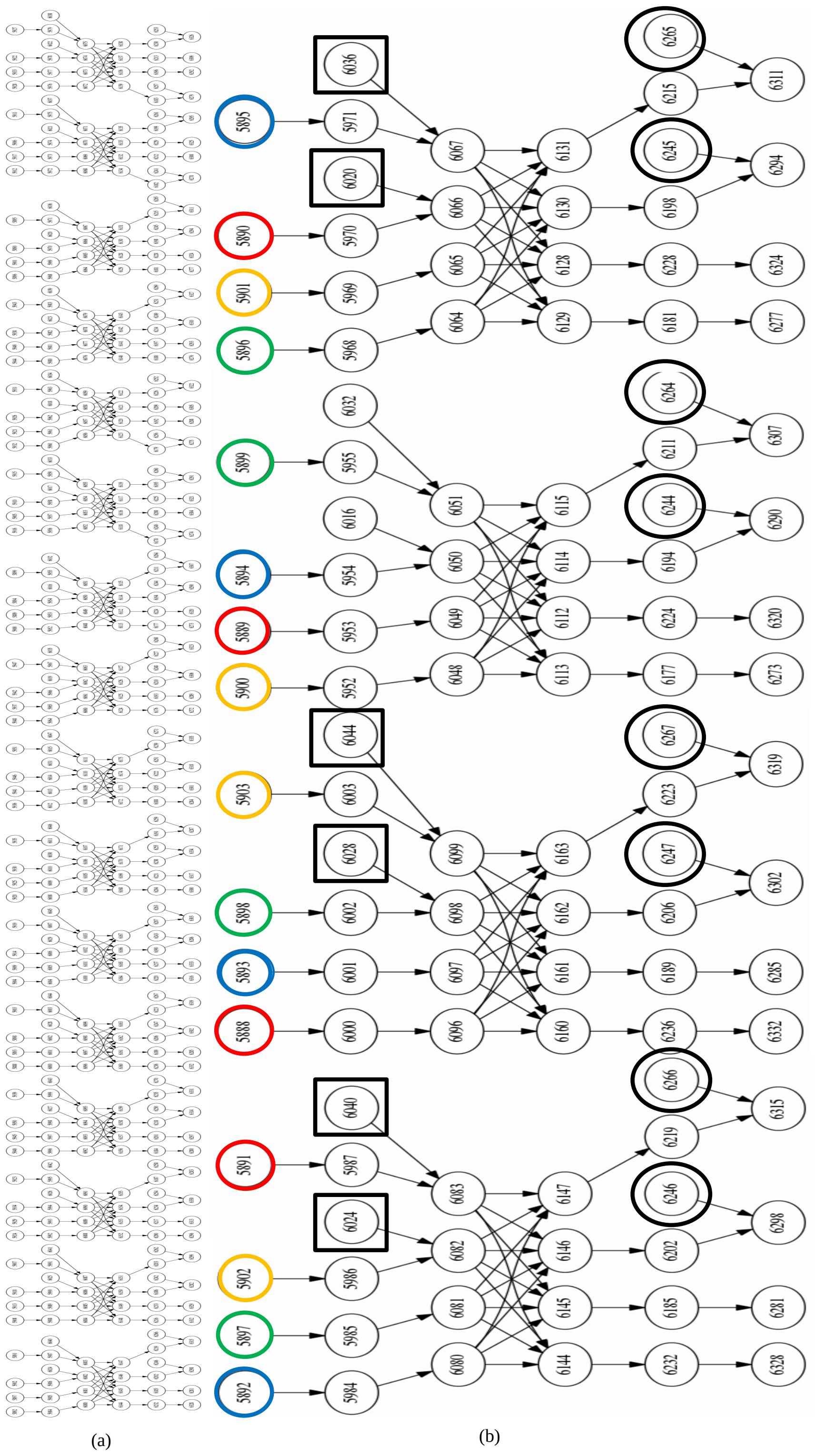
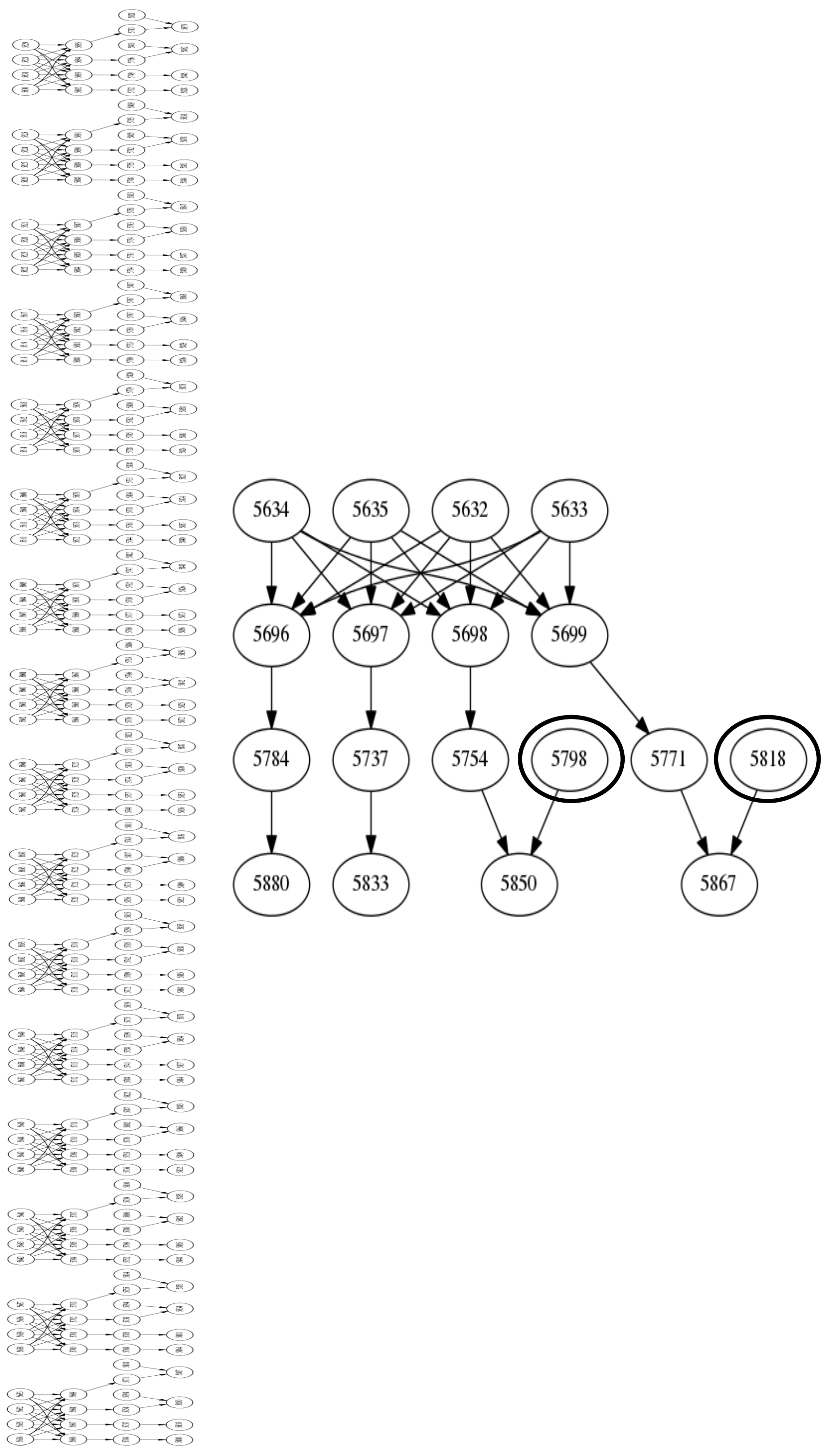


Figure 10: Graph corresponding to the distinguisher $\mathscr{D}_{27}^{2}$ for the attack with a nibble fault injection at the 25-th round on GIFT; (a) The complete graph from the tool (used to extract the keys of round 28), (b) One independent computation chunk/ subpart



(a) (b)

Figure 11: Graph corresponding to the distinguisher $\mathscr{D}^1_{27}$ for the attack with a nibble fault injection at the 25-th round on GIFT; (a) The complete graph from the tool (used to extract the keys of round 27), (b) One independent computation chunk/ subpart

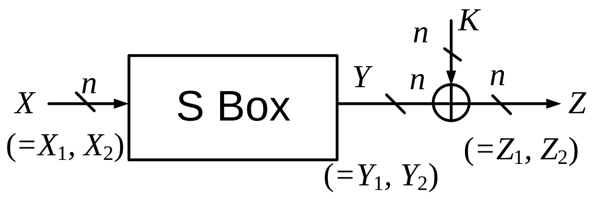


Figure 12: Cipher model used in [24]

B.2 Other Attacks On GIFT

Nibble fault injection at the 26th round of GIFT also results in a similar attack. However, in this case, 96 key bits can be extracted with a single fault injection. The attack results targeting different rounds of the cipher with varying fault widths are presented in Table. 3. It is interesting to observe that even with increased fault width of 8 bits the attacks remain almost same (except an increase in complexity by a factor of 4 due to the fault width). We found that the distinguishers corresponding to 8 bit faults and 4 bit faults are the same. This is due to the fact that, *the same set of S-Boxes become active by a nibble fault and a byte fault in GIFT*. This fact can be easily verified from the permutation table of the cipher. It is also worth mentioning that the fault propagation in GIFT (and in any bit permutation based cipher in general) is probabilistic as the number of the active S-Boxes (and thus the complexity of the resulting attack) depends on the value of the plaintext and the injected fault. The complexity figures returned by our tool are actually the best-case results and the average and worst-case attack complexity might vary up to some extent. From the perspective of a cipher evaluator, such best-case complexity figures are of utmost importance as they typically represent the highest advantage the attacker can have. However, from the perspective of an attacker, the average success rate is also very interesting. The future versions of this tool will include measures for calculating such average-case complexities. In the context of attacks on GIFT cipher, we observed that the increasing width of the fault up to 8 bits actually makes the attacks more likely to happen. This is quite obvious as a larger fault width always make the fault propagation more rapid. However, a very wide fault window may not work as the fault paths will become overlapped resulting the destruction of some distinguishing properties. Although we have not done any such analysis in this paper, it is worth mentioning that doing such analysis is quite straightforward using the ExpFault framework.

B.3 Comments on Optimality of the Attack on GIFT

In this subsection, we argue that the best attack proposed by us on GIFT (with a nibble fault injection at round 25) is optimal from information theoretic perspective. The analysis methodology is similar to that proposed by Sakiyama et.al. in [24]. However, certain modifications were made in the analysis to adapt it in the context of GIFT.

B.3.1 Main Results by Sakiyama et. al. [24]

The information theoretic optimality analysis of DFA exploits the fact that each fault model intrinsically sets an upper bound on the amount of leaked information about the secret. Any DFA attack with a given fault model must exploit the leaked information either fully or partially to extract the key. An optimal attack is one which is able to fully exploit the leaked information in order to retrieve the secret key with a practical level of complexity. The information leakage is measured in terms of Shannon Entropy of the state differentials.

Based on a simple cipher model as shown in Fig. 12, Sakiyama et.al. estimated the amount of information leakage for an $n \times n$ bit S-Box $S(\cdot)$. Let X denotes the discrete random variable associated with the input of the S-Box ($X \in \{0,1\}^n$) and Y denotes the random variable associated with the output. DFA attacks usually require two observations (correct and faulty) at the input and output of an S-Box which are denoted by random variables X_1, X_2 and Y_1, Y_2, respectively. The random variable for the key is denoted with K. Likewise, Z_1, Z_2 denote random variables after the addition of the key. Finally, ΔX, ΔY and ΔZ denote the state differential random variables.

In [24] it was proved that,

$$H(X_1X_2|\Delta Y) = H(K|Z_1Z_2) \tag{8}$$

and

$$H(K|Z_1Z_2) = H(\Delta X|\Delta Y) + H(X_1|\Delta X \Delta Y) \tag{9}$$

Further, if ΔX takes values from a set $\mathscr{X} \subset \{0,1\}^n$ then the amount of leaked information about the key can be estimated as:

$$k_{leak} = n - \log_2(|\mathscr{X}|) \tag{10}$$

As a concrete example, let us consider the 8×8 S-Box of AES. If a byte fault is injected at the 0-th byte in the beginning of the 8-th round of AES state, the leakage is of $128 - \log_2(2^8 - 1) \approx 120$ bits. This is because all the bits except the first 8 bits of the input state differential are 0, which makes $|\mathscr{X}| = 2^8 - 1$.

B.3.2 Optimality of the Attack on GIFT

The leakage characterization proposed by Sakiyama et.al. in [24] for AES measures the total leakage of the secret key at the injection point of the fault. It was argued that some of the leaked information gets missing during the propagation of the fault. In order to become optimal, an attack, however, must utilize the complete leakage measured at the injection point. Leakage measurement only at the injection point is suitable for AES, as due to its structural features the information about the complete 128 bit key state is leaked from the `AddRoundKey` layer following injection point.

The structure of GIFT differs for AES in two aspects. First, the key state of GIFT is of 128 bits, but the round keys are of size 32 bits. As a result, the information about only 32 key bits is leaked at the injection point of fault. Further, one can make the following observation from the key schedule of GIFT.

Observation: *Each of the* 4 *consecutive round keys of GIFT are independent of each other.*

It is thus evident that, in order to get information about the complete 128 key state, the leakage should be measured at multiple locations. More specifically, the leakage must be measured at the input differentials of S-Box layers for a 4 round differential propagation of the fault. Each of the S-Box layers leaks information about 32 bits of keys. The second distinct feature of GIFT is that it does not add key bits to every output bit of an S-Box. More precisely, only 2 bits of a 4 bit S-Box output is XORed with key bits. This feature has important consequences while measuring the leakage of the key.

Observing all the above mentioned facts, we propose the following strategy to measure the information leakage in case of GIFT.

1. Consider the state differentials at the input of 4 consecutive S-Box layers (in other words, the input differentials of S-Box layers).

2. Estimate the leaked information for the key bits associated, from the entropy of an input differential (using Eq. 10).

Using this strategy we now evaluate the attack described in Sec. 4.1. The following claim can be made.

Claim: *The fault attack with a nibble fault injected at the* 25th *round of GIFT is information theoretically optimal.*

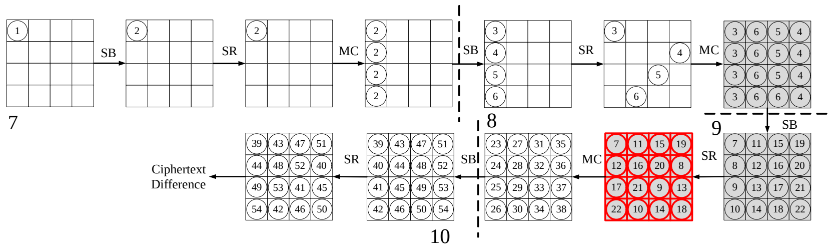


Figure 13: Motivating Example: Fault propagation in impossible differential fault attack on AES in terms of colors from XFC framework. The colors are represented by circled numbers.

Proof Sketch: The attack described in Sec. 4.1 injects a nibble fault at the input of the S-Box layer at round 25. The fault injection leaks information about the 32 bit round key of round 25. The leakage, in this case, is measured to be of 30 bits. This is because only 2 key bits are associated with the S-Box whose input was affected with the nibble fault. Rest of the 30 bits can be leaked. Next, we consider the quantity $|\mathscr{X}|$ for each of the following S-Box input differentials. They can be measured from the distinguisher descriptions provided in Table.4. As a concrete example, let us consider the input differential at the input of the 27th round S-Box layer. Leaked information here is calculated as $(64 - \log_2(2^{16})) = 48$ bits. However, associated key bits are only 32 bits, which means all the key bits can be leaked. As another example, consider the input differential before the S-Box of 28-th round (the information content of this input differential can be estimated from the entropy of the distinguisher $\mathscr{D}^2_{27}$ in Table.4. This is because all the sub-operations between $\mathscr{D}^2_{27}$ and this S-Box input are linear.) The leaked information about the 28th round key is thus $64 - 43.53 = 20.47$ bits.

Referring to the attack, we obtain the following figures for the leakage:

a Leakage for 25-th round key: 30 bits.

b Leakage for 26-th round key: 32 bits.

b Leakage for 27-th round key: 32 bits.

b Leakage for 28-th round key: 20.47 bits.

The total leakage is of 114.47 bits for the 128-bit key state. The total remaining entropy of the key, in this case, is thus $128 - 114.47 \approx 14$. It was already observed that the size of the remaining keyspace after the attack is 2^{14}. The information theoretic optimality of the attack is thus established.

C Comparison with the State-of-the-Art

Automation of fault attacks has gained significant attention from the research community in the recent past. In this section, we provide a detailed comparison of the ExpFault framework with two recently proposed automated fault attack frameworks – XFC and AFA. Brief introductions to both were provided at the beginning of this paper.

C.1 Comparison with XFC

The XFC framework has been proposed recently by Khanna et.al. [14] in DAC 2017. This framework has certain similarities with the ExpFault framework proposed by us. In particular, both of these frameworks try to figure out the attack complexities instead of doing the attack explicitly. However, we observed that XFC is somewhat oversimplistic and as a result, it fails to address certain imperative issues present in certain DFA attacks. In this subsection, we briefly discuss two attack test cases which the XFC framework fails to detect. The first among these test cases is the IDFA attack on AES [2], which targets the 7th round of the AES state. The second one is the fault attack on PRESENT at 28th round [13]. Both of these test cases were described as examples in the main text of the paper. Some of the basic aspects of these test cases will be repeated for the sake of understanding.

C.1.1 Undetected Distinguishing Properties

IDFA attacks exploit a typical cipher property that, depending on the fault, the variables of a state differential corresponding to some internal state of a cipher may not attain certain values within their domains. Such a property is used in IDFA attacks to distinguish correct key guesses from wrong ones. Perhaps, the most prominent example of an IDFA attack is on AES, where a byte fault is injected at the beginning of the 7-th round of the cipher resulting some state differentials none of whose variables can assume the value 0, with the correct key guess. The situation is elaborated in Fig. 13, where each large square represents an intermediate state differential of AES, with the fault injected at the beginning of the 7-th round in the 0-th byte location. Each small square in the figure represents a state differential variable of size one byte. The shaded states in Fig. 13 denote the existence of an impossible differential, with all bytes being active (fault difference cannot be 0) for any fault value, ciphertext and key. It is convenient to use the last among them as a distinguisher due to its proximity to the ciphertext (marked red in Fig. 13).

In order to elaborate the inability of XFC [14] in detecting the IDFA distinguisher, we mark the fault propagation path as it was done in XFC. The XFC framework represents the propagation path of a fault by means of *colors*, with each color representing a new variable, which can take all possible values within its respective domain. In Fig. 13, a color variable (corresponding to XFC) is represented with a numbered circle where the number represents a unique color. An induced fault is always assigned with a new color. The fault is then propagated through linear and nonlinear functions. The output of a nonlinear function is always assigned a new color owing to its lack of correlation with its input. On the other hand, for linear functions, a new color is assigned to the output, only if the inputs have different colors. Otherwise, the same input color is assigned to the output. This coloring continues through up to the generation of the ciphertext.[1]

The key observation here is that the coloring framework of XFC does not provide any information regarding the existence of the impossible differential property, as according to XFC, each new color variable should assume all possible values in its range (as do the corresponding state differential variables). So in this case, XFC would fail to detect the missing 0s in the state differential variables. In general, the values that a state differential variable may assume depend on several complex cipher-dependent factors and no straightforward extension of the static-analysis based coloring framework of XFC can capture this. As a concrete example of this claim, the state differential variables up to the 9-th round ShiftRow in Fig. 13, will never assume the value 0, whereas, after the 9-th round MixColumn operation, they can assume values within the full range, including 0. However, color variables cannot capture these changes in value ranges. In general, no trivial modification of XFC can capture such a variation. Another example of a similar nature can be provided for the fault attack on PRESENT-80, where a 2-byte fault is injected at the 28-th round of the cipher. The fault propagation results in a typical distinguishing property at the beginning of round 30, in which each 4-bit state differential variable can assume only 2 values from the total possible range of 16 values. While this property leads to a successful attack, the coloring scheme of XFC, for a reason similar to the one previously mentioned, fails to detect this. In a nutshell, *although the coloring framework of XFC is able to detect some distinguishing properties, it fails to detect some useful ones. Additionally, XFC does not provide any indication of the goodness of the distinguishers, so that the suitable ones can be chosen.*

[1]The color variables actually represent correlations between state differential variables. If the colors corresponding to two state differential variables are same, then they are correlated.

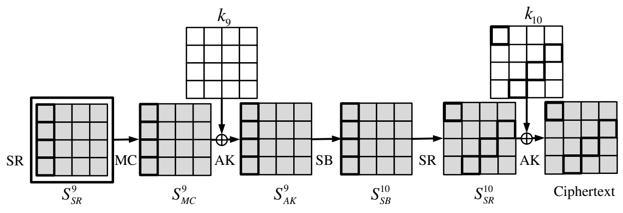


Figure 14: Motivating Example: Key Extraction in Impossible DFA

C.1.2 Lack of Automation and Fault Difference Equations

The practicality of a DFA attack lies within the efficient extraction of key parts in a divide-and-conquer manner, which is often realized in the form of small systems of equations (or inequations). For example, in the single fault based optimal attack on AES [17], the state differential after the 9-th round MixColumn operation is used as a distinguisher and keys can be extracted in 4-byte chunks with time complexity $O(2^{32})$. This actually results in 4 independent systems of difference equations (one system per 4-byte key part) each containing 4 equations of the following form:

$$a\delta = S^{-1}(x_h \oplus k_h) \oplus S^{-1}(x_h^{'} \oplus k_h) \tag{11}$$

where, x_h and $x_h^{'}$ denote the correct and faulty ciphertext bytes, respectively; δ denotes some state differential variable after the 9-th round MixColumn; k_hs denote the associated key bytes, and a denote some constant. In practice, the structures of resulting fault difference relations may vary significantly from (11) depending on the distinguisher chosen. For example, Fig. 14 corresponds to the key extraction strategy for the impossible differential distinguisher of AES described in Fig. 13. Here one shall obtain inequalities of much complex nature. The cipher states are represented by large gray boxes, whereas the associated round key states are represented with large white boxes. The state S^9_{SR} works as a distinguisher here. The bold boxes in each state of the figure represent the associated key bytes as well as the state bytes one need to guess/compute to calculate the first column of the distinguisher (shown in bold). The fault difference relations, which are inequations, in this case, can be represented as follows:

$$MC^{-1}|_b(S^{-1}(x_h \oplus k_h)) \oplus MC^{-1}|_b(S^{-1}(x_h^{'} \oplus k_h)) \neq 0 \tag{12}$$

In the set of relations represented by. 12, the $MC^{-1}|_b$ for $b \in \{1,2,3,4\}$ represent the effect of the inverse *MixColumn* operations on the 4 columns of the state S^9_{MC}. An important observation at this point is that the 9th round keys (denoted by state k_9 in the figure) are not included in the inequations described by 12, as they get canceled out during the calculation of the XOR differences. An automated tool is supposed to identify such simplification tricks and work according to that. Similar complex difference equations can be obtained for the attack on PRESENT in [13]. However, XFC only assumes equations of the form (11), which is clearly an oversimplification. Further, no automation was proposed to construct the equations. *The ExpFault framework proposed by us, however, addresses all these issues in a systematic manner through its powerful graph based abstraction. In particular, the divide-and-conquer strategy can be automatically identified and independently computable chunks can be extracted in an abstract form. Each of the independent computable chunk in ExpFault refers to an independently computable equation subsystem in classical DFA. It is also possible to construct the equations from the graphs.* All these facts clearly establish the superiority of ExpFault framework over the XFC.

C.2 Comparison with AFA

The AFA is a powerful method for automated fault analysis [18]. The main idea of AFA is to construct an algebraic equation system representing the cipher and injected faults. This system is then solved by means of state-of-the-art SAT solvers, which are sufficiently robust and powerful to handle such large problem instances. Although the AFA approach is fairly easy to implement and quite generic in nature, it is not very suitable for exploitable fault analysis. This is attributed to the fact that AFA has to explicitly perform an attack to evaluate the exploitability status of a fault instance, which can be extremely time-consuming. Evaluation of a fault instance in AFA requires solving a SAT problem. The time required for solving SAT problems often depends on the size of the search space. The key fact behind the success of an AFA (or any DFA attack) is that the size of keyspace of a cipher reduces significantly with the injection of faults. Solving an AFA instance sometimes suffers from serious scalability issues if the size of the keyspace after fault injection is still large. Moreover, computing the exact attack complexity in AFA requires enumerating all solutions of the corresponding SAT instance. This strategy may incur huge computational overhead. Although in [18] Zhang et. al. handled such scalability issues of AFA by assuming some of the key bits to be known, the complexity evaluation process still takes a significant amount of time. Perhaps, the most critical problem with AFA lies in its lack of interpretability. From the perspective of an evaluator, a clear understanding of the attack cause is essential, because it may help him to improve the cipher design or design good countermeasures. The CNF based abstraction used in AFA hides all structural information of the attack. In contrast, it is evident that from the outputs provided by the ExpFault one can have a precise understanding of each possible attack instance.

Recently, Saha et. al. [25] has proposed an alternative approach for exploitable fault characterization which combines AFA with Machine Learning to achieve the speedup desired for exploitable fault characterization. The scheme proposed by them utilizes a ML model to classify exploitable faults from unexploitable ones. The ML model is constructed by extracting features from CNF representations of certain exploitable and unexploitable fault instances already known for a given cipher. A small set of exploitable and unexploitable fault instances can always be constructed in an initial profiling phase of the cipher by means of AFA. This specific framework is somewhat complementary to the ExpFault proposed in this paper. In particular, the AFA-ML combination can explicitly characterize the exploitability status of different fault values corresponding to a specific fault location. This property is interesting for ciphers without MDS layers, as the exploitability of a fault instance for these ciphers critically depend upon the value of the fault. As a result, the average success rate of a fault attack corresponding to a specific fault location can be estimated by the AFA-ML framework exploiting this property, which is still not possible in the ExpFault framework. However, the attacks identified by the AFA-ML based framework lack interpretability, which is a strong point of ExpFault. Moreover, the calculation of attack complexity is not possible with the AFA-ML tool, which is the main goal of this paper.