An Automated Framework for Exploitable Fault Identification in Block Ciphers – A Data Mining Approach

No Author Given

No Institute Given

Abstract

Characterization of all possible faults in a cryptosystem exploitable for fault attacks is a problem which is of both theoretical and practical interest for the cryptographic community. The complete knowledge of exploitable fault space is desirable while designing optimal countermeasures for any given crypto-implementation. In this paper, we address the exploitable fault characterization problem in the context of Differential Fault Analysis (DFA) attacks on block ciphers. The formidable size of the fault spaces demands an automated albeit fast mechanism for verifying each individual fault instance and neither the traditional, cipher-specific, manual DFA techniques nor the generic and automated Algebraic Fault Attacks (AFA) [10] fulfill these criteria. Further, the diversified structures of different block ciphers suggest that such an automation should be equally applicable to any block cipher. This work presents a completely automated framework for DFA identification, fulfilling all aforementioned criteria, which, instead of performing the attack, just estimates the attack complexity for each individual fault instance. A generic and extendable data-mining assisted dynamic analysis framework capable of capturing a large class of DFA distinguishers is devised, along with a graph-based complexity analysis scheme. The framework significantly outperforms another recently proposed one [6], in terms of attack class coverage and automation effort. Experimental evaluation on AES and PRESENT establishes the effectiveness of the proposed framework in detecting most of the known DFAs, which eventually enables the characterization of the exploitable fault space.

1 Introduction

The pervasive use of embedded electronic systems, with in-built cryptographic cores often tailored for resource constrained environments, has lent great impetus to the construction of optimal countermeasures against implementation based attacks, such as passive side channel attacks and active fault attacks. Ensuring security is, however, a nontrivial task, especially with the resource and performance constraints imposed, and designing precise countermeasures require proper knowledge of the possible attack space on the device. Differential Fault Analysis (DFA) attacks, the most widely explored class of fault attacks so far, are particularly interesting in this context given their (relatively) low data/fault complexity and easy-to-mount nature [1, 9, 3]. It is well-esteblished that even a single properly placed malicious fault is able to compromise the security of mathematically strong crypto-primitives in certain cases. However, discovering even a single such attack instance for a given cryptosystem is nontrivial, as not every possible fault may lead to a successful attack. 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. The problem becomes even more challenging when different ciphers with diverse structures are considered.

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 and the mathematical structure of the cipher), which eventually lead to a fault space of formidable size. Identifying exploitable faults in such a fault space demands a fast and completely autometed mechanism for characterizing each individual fault, which, on the other hand, should be equally applicable to all existing ciphers and different forms of DFAs. Although, automation of fault attacks has been addressed in recent past via the Algebraic Fault Analysis (AFA) framework [10], analyzing a single

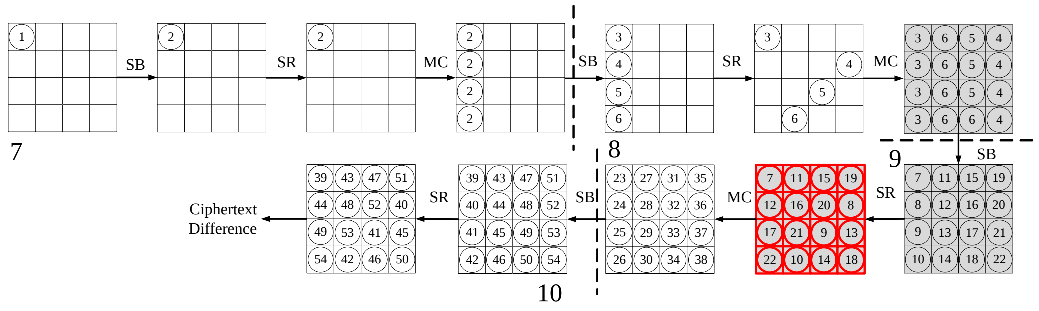

Figure 1: Motivating Example: Fault propagation in impossible differential fault attack on AES in terms of colors from XFC framework. The colors are represented by circled numbers.

fault instance there requires solving a SAT problem of prohibitive cost, which makes it a bad choice in the exploitable fault space characterization context.

The goal of this work is to figure out a generic albeit low cost mechanism for individual fault characterization in a cipher, which is suitable for scalably covering the complete fault space. A reasonable approach to achieve this goal could be to construct a methodology, which just estimates the attack complexity instead of doing the attack explicitly to recover the secret. In the light of this simple strategy, this paper presents a completely automated flow for quick characterization of individual faults in any given block cipher. We begin with a rigorous formalization of the cipher description and the DFA with an emphasis to the distinguisher identification, which, is the key step for any successful fault attack. Based on the formalization, a data-mining assisted, generic dynamic analysis framework is proposed to identify a large class of distinguishers, automatically. The data-mining scheme typically identifies complex relationships among fault variables, useful for reducing the key entropy, with sufficient amount of fault simulation data obtained at the initial phase, from the internal states of a cipher after fault injection, by varying the fault values, plaintexts and the keys. Next, we propose a graph-based algorithm which automatically estimates the evaluation complexity of the distinguisher, and in the process identifies the independent key extraction equation (or inequation) systems in an abstract form, by automatically figuring out a proper divide-and-conquer strategy. At the final step, the proposed framework estimates the size of the remaining key space, which, along with the distinguisher evaluation complexity, is a representative of the overall attack complexity.

Recently, Khanna, Rebeiro and Hazra has proposed solution to the exploitable fault problem [6] called XFC, which is also based on similar principles. 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 example 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. As it will be established in this paper, the proposed framework does not suffer from any of the issues found in XFC. It is also worth to mention that the data-centric view is sufficiently generic to be extended for other fault attack classes (Integral Fault Attacks [7] and Differential Fault Intensity Analysis attacks for example). The limitations of XFC are elaborated in the next section through proper examples to establish the relevance of our proposed methodology.

The rest of the paper is organized as follows. We start with a motivating example of an attack in Sec. 2, which the XFC framework clearly fails to detect. In Sec. 3, the cipher as well as the fault models are formalized. Next, we describe the complete scheme in Sec. 4. Proof-of-Concept evaluations on AES and PRESENT block ciphers will be presented in Sec. 5. Finally we conclude in Sec. 6.

2 Motivating Examples

In this section, we briefly discuss two attack examples which the XFC framework fails to detect. The first among these examples is the IDFA attack on AES [2], which targets the 7-round of the AES state. The second one is the fault attack on PRESENT at 28-th round [5].

2.1 Undetected Distinguishing Properties

The concept of impossible differential fault analysis stems from the fact that, a cipher state differential, in certain scenarios, may not attain certain values, which eventually results in the reduction of the entropy of the state differential, and is exploitable as a key distinguisher. Fig. 1 presents the generation of an impossible differential property in the case of AES, where the fault is injected at the beginning of the 7-th round of the cipher. The fault propagation path is represented in the context of XFC, with colors assigned to them according to Algorithm 3 in [6]. The XFC framework represents the propagation path of a fault by means of colors, with each color representing a new fault difference variable, which can take all possible values within its respective domain determined by its bit length. An induced fault assigned with a new color, as its correct value is always unknown. 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, the for the 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. The coloring is continued up to the generation of the ciphertext. The colors are actually the abstractions of word level variables.

The shaded states in Fig. 1 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 (marked by red). The key observation here is that the coloring framework of XFC does not provide any information regarding the existence of this impossible differential property, as according to XFC, each new variable (color) should assume all possible values in its range. So in this case, XFC will fail to detect the missing 0s. In general, the values that a fault difference variable may assume, actually depend on several complex cipher-dependent factors and no straightforward extension of the static-analysis based coloring framework can capture them. As a concrete example of this claim, the colors up to the 9-th round ShiftRow operation in the Fig. 1, will never assume the value 0, whereas after the 9-th round MixColumn operation, they can assume values within the full range, including 0. No trivial modification of XFC can capture such variation in generic sense. Another example of 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 the reason similar to the previous one fails to detect this.

2.2 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 [9], 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:

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

where, x_h and $x_h^{'}$ denote the correct and faulty ciphertext bytes, respectively; δ denotes some fault difference 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 (1) depending on the distinguisher chosen. For example, Fig. 2 corresponds to the impossible differential distinguisher of AES, where one shall obtain inequalities of much complex nature. Similar complex difference equations can be obtained for the attack on PRESENT in [5]. However, XFC only assumes equations of the form (1), which is clearly an oversimplification. From the perspective of an automated tool, it is desirable that the tool should generate such relations internally on-the-fly, which is possible only if a proper divide-and-conquer strategy can be chosen for keeping the attack complexity within reasonable levels. Automatic identification of the

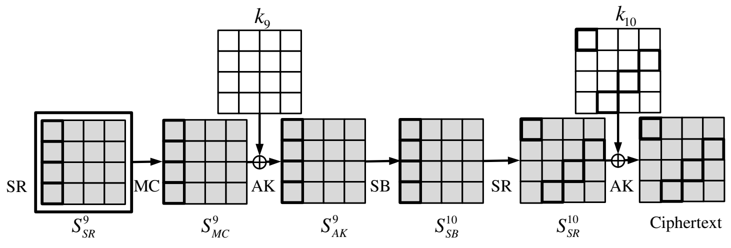

Figure 2: Motivating Example: Key Extraction in Impossible DFA

correct divide-and-conquer strategy is, however, nontrivial and demands algorithmic support. In this paper we address this issue for the first time. From the next section onwards, we present our framework, which efficiently handles these issues.

3 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.

3.1 Differential Fault Attack 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 a wrong key distinguisher, which unlike classical differential attacks, is deterministic in nature. Typically, a distinguisher in a block cipher context is a logical or statistical property affecting the essential Pseudorandom Permutation (PRP) nature of the cipher. According to the well-known wrong key assumption, a distinguisher state attains a uniform distribution with a wrong key guess and a non-uniform one with a correct key guess.
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. Identification of the subparts in a distinguisher (using a proper divide-and-conquer strategy), as well as the number of required key guesses for computing each subpart is, however, highly cipher dependent, and indeed a decisive factor for determining the practicality of the attack.
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 proper mathematical specification of the cipher and the faults to begin with. In the following subsections, we formalize the cipher and the differential fault attacks in this context.

3.2 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. Block ciphers are typically abstracted as PRPs (that is $\mathscr{P} = \mathscr{C}$) specified by the key $k \in \mathscr{K}$. Structurally, they can be represented as a tuple of invertible functions as follows:

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

Typically, for a given $p \in \mathscr{P}$ and a fixed $k \in \mathscr{K}$, there exists an 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 the a R round cipher. Further, each o_j^i can be represented as:

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

$$o_j^i(x_1, x_2, ...x_l) = \bigoplus_{h_1=1}^{h_1=2^l} a_{h_1} \cdot \prod_{h_2 \in I} x_{h_2}, \quad \text{if } o_j^i \text{ is nonlinear} \tag{4}$$

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.

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

$$\mathscr{E}_k = \langle p, s_1^1, s_1^2, ..., s_1^d, s_2^1, s_2^2, ..., s_2^d, ...s_R^1, s_R^2, ..., s_R^d \rangle \tag{5}$$

where each s_j^i represents the output 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 plaintext p and key k. Following the standard practice, we call each s_j^i an *internal state* or simply *state*. We also denote $\mathscr{E}_k$ as the *execution trace* of the cipher. The state sequence begins with the plaintext p and it is obvious that $s_R^d = c$. Each state s_j^i 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 and can be generated easily using some execution model constructed from the structural specification of the cipher described in Equation. (2).

3.3 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_j^i, the output of o_j^i will differ from its expected value. We provide a formal representation of a fault, which is similar to that of [10], as a tuple $F = \langle X, \lambda, wd, t, f \rangle$. Here, X represents the register (a datapath register, key register or round counter), where the fault is to be injected, λ denotes the width of the state, wd is the width of the fault, t is the fault location, and f is the value of the injected fault. Without loss of generality, we assume that X is the datapath register, and the fault is to be injected at round $r < R$. It is easy to follow that, X basically belongs to the set $\{s_j^i\}$ described in Equation. (5). Let us denote $s_j^i = \langle V_1, V_2, ...V_l \rangle$, where each V_z ($z \in \{i, 2, ..., l\}$) 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 denote the fault models for the ciphers. 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 fault model is determined, we can now formally describe the DFA attack on a cipher. Given a cipher $\mathscr{F}_k$ and a fault F on it, the DFA can be formally described as:

$$\mathscr{A} = \langle \mathscr{D}, \mathscr{T}, \mathscr{R} \rangle \tag{6}$$

where:

- $\mathscr{D}$ denotes the distinguisher, which could be a non-uniform distribution or a set mathematical/logical expressions, over the difference variables of some correct and faulty state, in the context of DFA. The evaluation of the distinguisher $\mathscr{D}$ 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.

- $\mathscr{T}$ is the enumeration algorithm for the key set $\mathscr{K}$ through the evaluation of the distinguisher. The main component of this algorithm is a divide-and-conquer strategy, which enables the evaluation of the distinguisher in parts. The time complexity of the enumeration algorithm is of particular interest, which is $O(2^n)$, with $n \leq \log_2(|\mathscr{K}|)$. For practical cases $n \lll \log_2(|\mathscr{K}|)$, 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. 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. Evidently, $\mathscr{R} = \mathscr{K}_{cr}$ and $|\mathscr{R}| \lll |\mathscr{K}|$ for an efficient fault attack. $\mathscr{R}$ is often represented as a system of equations or inequations, involving the keys and distinguisher variables, whose solution space enumerates $\mathscr{K}_{cr}$.

Algorithm 1 Procedure *RngChk*

```
Input: $T_{\delta^i_j} = \langle T_{w_1^{ij}}, T_{w_2^{ij}}, ..., T_{w_l^{ij}} \rangle$
Output: $\langle \{Rng_{w_z^{ij}}\}_{z=1}^{l}, H_{Ind}(\delta^i_j), Act \rangle$
 1: $H_{Ind}(\delta^i_j) := 0$
 2: $Act[z] := 1, \forall z \in \{1,2,...,l\}$
 3: for each $T_{w_z^{ij}} \in T_{\delta^i_j}$ do                    ▷ [1]
 4:     $Arr_z[q] := 0 \ \forall q \in \{0,1,2,...,2^m-1\}$          ▷ [2]
 5:     $Rng_{w_z^{ij}} := \phi$
 6:     for $i_1 \in \{0,1,2,...,|T_{w_z^{ij}}|\}$ do
 7:         $Arr_z[T_{w_z^{ij}}[i_1]] := Arr_z[T_{w_z^{ij}}[i_1]] + 1$
 8:     end for
 9:     $H_{Ind}(w_z^{ij}) := 0$
10:     for $(0 \leq q \leq 2^m - 1)$ do
11:         if $(Arr_z[q] > 0)$ then
12:             $Rng_{w_z^{ij}} := Rng_{w_z^{ij}} \bigcup \{q\}$
13:             $p_q^{'w_z^{ij}} := \frac{Arr_z[q]}{|T_{w_z^{ij}}|}$
14:         else
15:             $p_q^{'w_z^{ij}} := 0$
16:         end if
17:         if $(p_0^{'w_z^{ij}} == 1)$ then
18:             $Act[z] := 0$
19:         end if
20:         $H_{Ind}(w_z^{ij}) := H_{Ind}(w_z^{ij}) + p_q^{'w_z^{ij}} \log_2(p_q^{'w_z^{ij}})$
21:     end for
22:     $H_{Ind}(w_z^{ij}) := -H_{Ind}(w_z^{ij})$
23:     $H_{Ind}(\delta^i_j) := H_{Ind}(\delta^i_j) + H_{Ind}(w_z^{ij})$
24: end for
25: Return $\langle \{Rng_{w_z^{ij}}\}_{z=1}^{l}, H_{Ind}(\delta^i_j), Act \rangle$
```

Algorithm 2 Procedure *Miner*

```
Input: $T_{\delta^i_j} = \langle T_{w_1^{ij}}, T_{w_2^{ij}}, ..., T_{w_l^{ij}} \rangle, Act$
Output: $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}_{v=1}^{|VS_{\delta^i_j}|}, H_{Assn}(\delta^i_j) \rangle$
 1: $T'_{\delta^i_j} := \{T_{w_z^{ij}} \mid Act[z] \neq 0\}$
 2: $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}_{v=1}^{|VS_{\delta^i_j}|} \rangle :=$ Apriori$(T'_{\delta^i_j})$
 3: $H_{Assn}(\delta^i_j) := 0$
 4: for each $v \in VS_{\delta^i_j}$ do
 5:     $tot :=$ VarCount$(v) \times m$                              ▷ [3]
 6:     $p_q^v := \frac{|IS^v_{\delta^i_j}|}{2^{tot}}, \forall q \in IS^v_{\delta^i_j}$
 7:     $p_q^v := 0, \forall q \notin IS^v_{\delta^i_j}$
 8:     $H_{Assn}(v) := -\sum_{q=0}^{2^{tot}-1} p_q^v \log_2(p_q^v)$
 9:     $H_{Assn}(\delta^i_j) := H_{Assn}(\delta^i_j) + H_{Assn}(v)$
10: end for
11: Return $\langle VS_{\delta^i_j}, \{IS^v_{\delta^i_j}\}_{v=1}^{|VS_{\delta^i_j}|}, H_{Assn}(\delta^i_j) \rangle$
```

[1] $z = 1,2,...,l$
[2] m: bit length of w_z^{ij}
[3] `VarCount` returns the number of variables in a variable set

4 Proposed Framework for Exploitable Fault Characterization

In this section, we describe the proposed automated framework in detail. The following subsections, will provide generic algorithms for each of the components described in (6). 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.

4.1 Automatic Identification of Distinguishers $\mathscr{D}$

Let us consider the execution trace $\mathscr{E}_k$ of a cipher, as described in (5). The values assumed by the states s^i_j change, for every execution of the cipher with different p or k. To capture the effect of a fault $F = \langle X, \lambda, wd, t, f \rangle$, injected at the round $r < R$, we define a similar trace $\mathscr{E}'_k = \langle p, s^1_1, s^2_1, ..., s^d_1, s^1_2, s^2_2, ..., s^d_2, ..., s'^1_r, s'^2_r, ..., s'^d_r, ..., s'^1_R, s'^2_R, ..., s'^d_R \rangle$. Here each s'^i_j represents the faulty output of the i-th sub-operation in the j-th round ($r \leq j \leq R$) after the injection of the fault at round r. Before the r the round the states remain the same. The next step is to consider the XOR difference between the correct and faulty states to study the fault propagation up to the ciphertext. To capture this, we define the differential execution trace Δ_k as, $\Delta_k = \mathscr{E}_k \bigoplus \mathscr{E}_k^{'} = \langle 0,0,0,...,\delta_r^1,\delta_r^2,....,\delta_r^d,...,\delta_R^1,\delta_R^2,....,\delta_R^d \rangle$. For obvious reasons, the state differences before the r-th round are zero. Further, each δ_j^i is represented as, $\delta_j^i = s_j^i \bigoplus s'^i_j = \langle V_1^{ij} \oplus V_1^{'ij}, V_2^{ij} \oplus V_2^{'ij},...,V_l^{ij} \oplus V_l^{'ij} \rangle, \quad r \leq j < R$, where, V_z^{ij} denotes the z-th m-bit correct word of the state at j-th round and after i-th sub-operation, and $V_z^{'ij}$ denotes the faulty word for the same. For each such word $\oplus$ denote the bitwise XOR operation.

For convenience, now onwards we shall denote Δ_k as, $\Delta_k = \langle \delta_r^1,\delta_r^2,....,\delta_r^d,...,\delta_R^1,\delta_R^2,....,\delta_R^d \rangle$ by ignoring the zeros. We call this Δ_k as the *differential execution trace* after the injection of the fault, and each δ_j^i is denoted as *state differential*. Note that, valuations of Δ_k for different keys, plaintexts or fault values result in new trace vectors, which carry information regarding the fault propagation paths and the distinguishers. Each of the state differentials δ_j^i in Δ_k may potentially form a distinguisher. We now define the decision criterion by which we declare a state differential property as a distinguisher.

Definition 4.1. [**Entropy of a State Differential**] *The entropy of a state differential* $\delta_j^i = \langle w_1^{ij}, w_2^{ij},...,w_l^{ij} \rangle$, *where each* w_z^{ij} *is a discrete random variable with probability distribution* $p_z^{w^{ij}}$, *is defined as* $H(\delta_j^i) = H(w_1^{ij}, w_2^{ij},...,w_l^{ij})$, *that is the joint entropy of the random variables in the state differential.*

Definition 4.2. [**Maximum Entropy of a State Differential**] *The maximum entropy of a state differential* $\delta_j^i = \langle w_1^{ij}, w_2^{ij},...,w_l^{ij} \rangle$, *is defined as* $H_{max}(\delta_j^i) = \sum_{z=1}^{l} H_{max}(w_z^{ij}) = \sum_{z=1}^{l} (-\sum_{q=0}^{2^m-1} p_q^{w_z^{ij}} \log_2(p_q^{w_z^{ij}}))$, *where each* w_z^{ij} *is independent and uniformly distributed within the range* $[0, 2^m - 1]$, *given* m *is the bit width of each variable* w_z^{ij}.

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 basically 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, which will provide a decision metric to determine whether a state differential can be used as distinguisher or not.

Definition 4.3. [**Distinguisher Criteria**] *A state differential* δ_j^i *is called a distinguisher if the entropy* $H(\delta_j^i)$ *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. The data acquisition step is extremely low cost and takes only a couple of seconds. Let us denote the datasets corresponding to each state differential δ_j^i as $T_{\delta_j^i}$. Each $T_{\delta_j^i}$ is a table, each containing l, m-bit variables w_z^{ij} ($1 \leq z \leq l$) and data values, corresponding to each of them. For convenience, we further denote each column of a $T_{\delta_j^i}$ as $T_{w_z^{ij}}$. Given a fault simulation at some round r, we have many such tables corresponding to each state differential after the fault injection, and a subset of them actually qualifies as potential distinguishers. We denote $T_{\Delta_k} = \langle T_{\delta_r^1}, T_{\delta_r^2},....,T_{\delta_r^d},...,T_{\delta_R^1},T_{\delta_R^2},....,T_{\delta_R^d} \rangle$ as the set of the tables for the state differentials. Our data-based framework tests each δ_j^i separately and decides whether it constructs a distinguisher. At this point, we need to separately identify two cases, which typically occur in the course of the distinguisher identification.

4.1.1 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 $p_z^{'w^{ij}}$ ($z = 1, 2, ..., l$), and the joint state differential entropy

Table 1: 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

as $H_{Ind}(\delta_j^i) = \sum_{z=1}^{l}(-\sum_{q=0}^{2^m-1} p_z^{'w_q^{ij}} \log_2(p_z^{'w_q^{ij}}))$. Each column of the table $T_{\delta_j^i}$ (denoted as $T_{w_z^{ij}}$), corresponding to each variable w_z^{ij} 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_z^{ij}s can assume value 0, and as a result, the value 0 will be missing in the table $T_{w_z^{ij}}$ 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_z^{ij} in the set $Rng_{w_z^{ij}}$ (line no 12 in Algorithm 1). Interestingly, in several occations, the fault diffusion in a state may be incomplete, which will result in many state difference variables always taking value 0 in their respective tables. Indeed, such states may qualify as distinguishers if the non-zero part of them cause some entropy reduction. In Algorithm 1, such non-active variables are handled by maintianing a bit vector Act of length l, where $Act[z] = 0$ indicates the variable is not active and should not be considered during distinguisher construction (line no 2 and 17-19 in Algorithm 1). Typical examples of Case. 1 include the IDFA attack on AES and the attack on PRESENT described in [5] (both will be described in the case study.).

4.1.2 Case 2. The Variables are not Independent

The second case of the distinguisher identification problem deals with the scenerios where correlations exist between some of the variables within a state distinguisher, which eventually casue the reduction of state 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 under consideration.

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 occuring 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 occurence 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. 1. 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. 1, 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 imply that the variables v_4 and v_5 are statistically uncorrelated. Note that, setting the proper support is imperative, as otherwise one may obtain extremely 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 $T_{\delta_j^i}$ as input, which is the input to the `Apriori` function after some basic preprocessing (similar to that was done in case. 1 to deal with incomplete diffusion). 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 $\langle VS_{\delta_j^i}, \{IS^v_{\delta_j^i}\}_{v=1}^{|VS_{\delta_j^i}|} \rangle$, where $VS_{\delta_j^i}$ denote the set of all variable sets and $\{IS^v_{\delta_j^i}\}_{v=1}^{|VS_{\delta_j^i}|}$ denote the set of itemsets corresponding to each variable set. 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 $v \in VS_{\delta_j^i}$ (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 $H_{Assn}(\delta_j^i)$.

Setup for Apriori Algorithm:

Frequent itemset mining is crucially dependent on the support parameter of the miner. The implementation of the Apriori algorithm we used (from WEKA package [4]), 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 insufficient number of itemsets. 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. 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, which, in turn, significantly reduces the risk of generating spurious associations.

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 criterian), 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 4.2; ignores the 0 valued variables in the case of incomplete diffusion). Indeed, $\{H_j^i\} < H_{max}(\delta_j^i)$ is the essential criterian for a state differencial 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, $\mathscr{D}_j^i := \langle \{w_z^{ij}\}_{z=1}^l, \{Rng_{w_z^{ij}}\}_{z=1}^l, VS_{\delta_j^i}, \{IS^v_{\delta_j^i}\}_{v=1}^{|VS_{\delta_j^i}|} \rangle$. The description of the distinguisher here is more like a structure in a C-program, which indeed can be exploited for finding the correct keys if required. The pseudocode for the algorithm is rather straightforward and is omitted here

Determining the proper distinguisher:

The distinguisher identification step usually returns a set of potential distinguishers with their respective entropies specifying their qualities. However, the evaluation complexity of a given distinguisher plays a

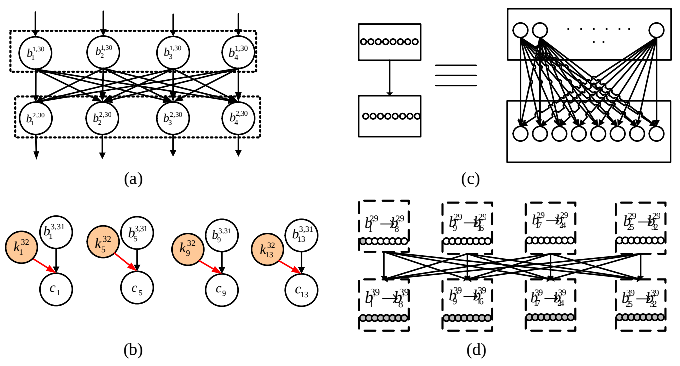

Figure 3: Example: Subgraphs corresponding to different sub-operations of a cipher

crucial role in its selection for a practical attack, as will be shown in the next subsection. Further, it is mandatory to have an S-Box between the 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. Nevertheless, in general, the distinguisher having the lowest entropy is the best for obvious reasons, only if it does not get rejected considering the above-mentioned criteria.

4.2 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, as the usability of a distinguisher depends on the complexity of evaluating it exhaustively. Typically, the candidates for the correct key are obtained by exhaustive evaluation of the distinguisher over all possible key guesses, which, in most of the occations is realized in the form of the complete solution space of a system of difference equations (or inequations). The time complexity of distinguisher evaluation as well as the size of the remaining keyspace (denotes the offline complexity of the attack) can be estimated once such systems are constructed, and the maximum among these two quantities denote the overall complexity of the attack. Further, the number of required fault injections for a practical attack can also be determined from these equations. However, knowing these relations apriori, is not a practical assumption for an automated tool, as it critically depends upon the distinguisher chosen. As already pointed out in Sec. 2.2, automatic construction of these equations require a proper divide-and-conquer strategy to be identified. This divide-and-conquer strategy allows one to parallelly evaluate each individual variable/variable set of the distinguisher by guessing small key parts exhaustively, instead of guessing the complete key at once.

In this work, we construct such equations systems automatically in an abstract form, which is suitable for the purpose of attack complexity evaluation and can be extended to concrete fault difference equations, if required. To automatically determine the divide-and-conquer strategy we propose an algorithm which typically, identify the key bits one need to guess to compute each variable/ variable set within a distinguisher by means of a graph-based abstraction of the cipher. Let us consider the data-centric view of the cipher in (5). Each state s^i_j here is represented as a set of binary variables as $s^i_j = \langle b^{ij}_1, b^{ij}_2, ..., b^{ij}_\lambda \rangle$, in contrary to the last subsection, where they (the states) were represented as vectors of variables of size m bits. Let us define a directed acyclic graph (DAG) $\mathscr{G}(Ver,\ E)$, where the set of the vertices Ver, consists of all the bit variables within the cipher. The set of directed edges E, on the other hand represents the effect of various sub-operations on the vertices, directed from the plaintext input towards the ciphertext outputs. More specifically, given two consecutive states s^i_j, and s^{i+1}_j, the edges represent the dependencies between them, imposed by the sub-operation o^{1+1}_j, at bit-level, considering the bit variables of s^i_j as inputs, and that of s^{i+1}_j as outputs, respectively. For a bit variable b^{ij}_e, $1 \leq e \leq \lambda$, the incoming edges on it, define on which bit variables, the value of b^{ij}_e actually depends on. We call such a graph as *Cipher Dependency Graph* (CDG).

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

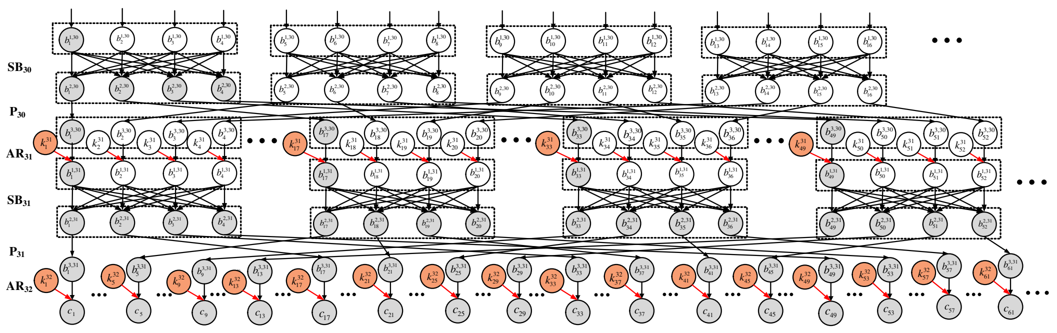

Figure 4: Example: Finding Key Parts for the Distinguisher Evaluations in PRESENT

the S-Box inputs (Fig. 3 (a)). The key addition operations are represented by structures shown in Fig. 3 (b). The nodes corresponding to the key variables are of specific interest, and special pointers to each of these variables are kept within the nodes they are incident to. 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 relevent to the attack is shown. It is interesting to observe that, the CDG is already divided into clearly identifiable levels.

The next step to the CDG construction is the identification of independent key parts. 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 form 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 by using the special key pointers maintained in $O(1)$ complexity.

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. 2, where the key bits corresponding to k_9 are not used for distinguisher evaluation, however, will still be detected by the BFS based search. The key reason behind this fact is that there are no nonlinear layer between the key variables in k_9 and the distinguisher in S^9_{SR}. As a result, these key variables get canceled out with the calculation of the differential. 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. A clear illustration of this is provided in the Sec. 5.2, for the AES example.

We provide an illustrative example at Fig. 4, which refers to a fault attack for PRESENT. The distinguisher is at the input of the S-Box layer at round 30 of the cipher. For clarity, we have only presented the BFS tree for a single node at round 30. The tree nodes and the key nodes are marked with separate colors. It can be seen that there are total 20 key bits, which actually affect the value of the first distinguisher bit. Careful observation reveals that the key set is same for the next three consecutive bits as well (that is, same for the complete nibble).

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. As for most of the time, we are dealing with m bit distinguisher variables, it is trivial to combine the key bits corresponding to each m bit variable, and one should also consider combining the keys for the variable sets. While evaluating any of these variables/variable sets, the corresponding keys must be guessed. At this point, certain other things are

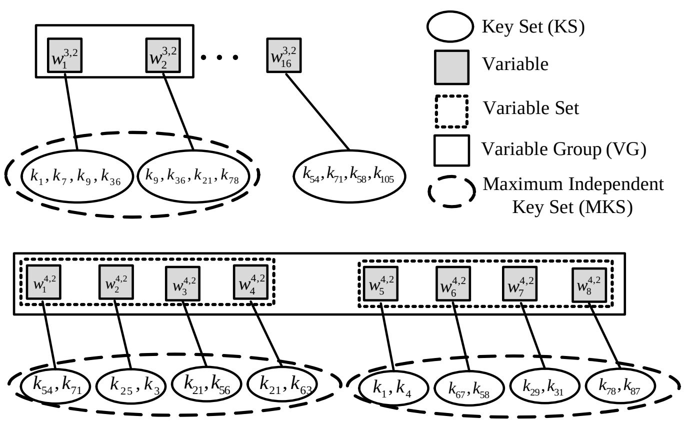

Figure 5: Illustration of the Relationships between Key and Distinguisher Variables

to be taken care of. Let us consider a distinguisher $\delta_j^i = \langle w_1^{ij}, w_2^{ij}, ..., w_l^{ij} \rangle$. Corresponding to each w_z^{ij}, there exists a set of key bit variables. An obvious way is to view the relationships as a bipartite graph, as shown in Fig 5. 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_z^{ij} 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 32-th round. Such overlaps are extremely important from the divide-and-conquer point of view. To deal with such cases, we define *Maximum Independent Key Sets* (MKS), which are the maximal subsets of keys having no overlap. They can be constructed easily by taking the union of overlapping KS_zs. Each MKS_h also impose a grouping on the corresponding w_z^{ij}s attached to its component KSs (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 items. Variable sets can be members of variable groups.). Intuitively, such groupings in keys and distinguisher variables imply the sets of independent equations to be solved for the key extraction.

Calculation of the distinguisher evaluation complexity 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 H such VG_hs along with their corresponding MKS_hs. The time complexity of computing each of them is given as $\mathbb{T}(h) = 2^{|MKS_h|}, \quad 1 \leq h \leq H$. 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 $max_h(\mathbb{T}(1), \mathbb{T}(2), ..., \mathbb{T}(H))$. Certain optimizations are possible in the above mentioned CDG searching scheme, by reducing the number of BFS traversals corresponding to each distinguisher, through the proper exploitation of the symmetries, usually present in the cipher constructions. Certain enhancements are also feasible. For example, the number of key guesses for a distinguisher can be significantly reduced by reordering the linear layers of the cipher, and this fact is utilized for certain attacks (see [2] for a concrete example). Although, our framework efficiently handles such enhancements, the details are omitted here due to the lack of space.

4.3 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 after the fault injection. Remaining search space complexity $\mathscr{R}$, along with the distinguisher enumeration complexity $\mathscr{T}$, determines the total complexity of a fault attack. 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. The 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. A summary of DC values for different ciphers can be found in [6].

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 $\mathscr{D}^i_j$, and a set of tuples with cardinality H, which contains the MKSs and corresponding VGs. As an additional component, the DC characteristic of the S-Boxes $\mathscr{H}^h_S$ corresponding to each MKS_h, VG_h pair is also supplied. The $\mathscr{H}^h_S$ 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 $\mathscr{H}^h_S$ 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 $\mathscr{H}^h_S$. Values of the $\mathscr{H}^h_S$ 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.

Algorithm 3 Procedure *EVAL_ SEARCH_ SPACE*

```
Input: $\mathscr{D}^i_j$, $\{\langle MKS_h, VG_h, \mathscr{H}^h_S \rangle\}_{h=1}^{H}$
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 $(VS_{\delta^i_j} == \phi)$ then      ▷ $\mathscr{D}^i_j$ includes no variable sets; so $VG_h$ contains independent variables
6:             $count := |Rng_{g_h}|$
7:             $b_c := m$
8:         else      ▷ $\mathscr{D}^i_j$ includes variable sets; so each $VG_h$ contains variables sets
9:             $count := |IS^{g_h}_{\delta^i_j}|$
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$(MKS_h)$      ▷ BitCount returns the number of bit variables in $(MKS_h)$
15:    $|\mathscr{R}|_{VG_h} := 2^{k_{size}} \times \mathbb{P}[VG_h] \times (\mathscr{H}^h_S)^{|VG_h|}$
16:    $|\mathscr{R}| := |\mathscr{R}| \times |\mathscr{R}|_{VG_h}$
17: end for
18: Return $|\mathscr{R}|$
```

5 Proof-of-Concept Evaluation: AES and PRESENT

In this section, we provide proof-of-concept evaluations of the proposed framework on two well-known ciphers – AES-128 (128-bit block size, 128-bit key, 10 rounds, and 8×8 S-Box) and PRESENT-80 (64-bit block size, 80-bit key, 32 rounds, and 4×4 S-Box), both of which are Substitution-Permutation-Networks (SPN). The diffusion layers of the ciphers are significantly different – AES has an MDS matrix in its diffusion layer, whereas PRESENT uses a simple bit permutation for the diffusion. We decide 2^{50} as the limit for practical exhaustive search. Using the proposed framework, we analyzed all possible byte and bit faults up to 6-th round for AES. For PRESENT, bit, nibble, and byte, and 2-byte faults were analyzed up to 26-th round. The fault locations were assumed to be known, which is a reasonable assumption in the evaluation mode. It was observerd 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 the following subsections, we describe three typical DFA use-cases in detail to emphasize the necessity of each of the components of the framework.

5.1 AES: Fault Injection at the Beginning of the 8-th Round

In this attack, a byte fault of unknown value is injected at the beginning of the 8-th round of AES. Below, we describe the steps followed by the proposed framework in detecting this attack.

Distinguisher Identification:

The distinguisher identification step identifies all the differential states (total 9) on the fault propagation path showing some entropy reduction. However, the first 4 from the fault injection point are rejected due to their excessive evaluation cost, while the last two are rendered useless by the absense of a nonlinear layer between them and the ciphertext. Among the rest, δ_9^4, which is the output of the 9-th round MixColumn shows the smallest entropy value and is eventually selected as the potential distinguisher for the attack. Next, we elaborate the entropy calculation for this state differential. The state differential $\delta_9^4 = \langle w_1^{49}, w_2^{49}, ..., w_l^{49} \rangle$ contains 16 variables, each with bit-width $m = 8$. The maximum entropy here is $H_{max}(\delta_9^4) = 128$. However, the function `Miner` reveals variable associations. More specifically, there are 4 variable sets $(w_1^{49}, w_2^{49}, w_3^{49}, w_4^{49})$, $(w_5^{49}, w_6^{49}, w_7^{49}, w_8^{49})$, $(w_9^{49}, w_{10}^{49}, w_{11}^{49}, w_{12}^{49})$, and $(w_{13}^{49}, w_{14}^{49}, w_{15}^{49}, w_{16}^{49})$ (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 $H_{Assn}(\delta_9^4) = 4 \times 7.99 = 31.96$.

Distinguisher Evaluation Complexity:

Evaluating the time complexity for this attack requires the construction of the CDG and a series of BFS traversals starting from the distinguisher bits. 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, and with the existence of variable sets of cardinality 4, one must consider $8 \times 4 = 32$ key bits simultaneously, for guessing. Further, the key bytes associated with each distinguisher variables are independent, and hence the MKS and corresponding VGs will contain only single elements. Overall, the complexity of distinguisher evaluation becomes 2^{32}.

Evaluation of the Remaining Search Space Complexity:

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 distinguisher evaluation step and the exhaustive search step thus remains 2^{32}.

Special Notes on the Two Step Attacks Described in [9]:

Tunstall et. al. presented a 2 step approach for the same 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_8^4 which was previously costly to evaluate on the complete keyspace. The proposed mechanism here detects both the distinguishers. However, the two-step attack requires the existence of the inverted key schedule equations. The proposed tool can easily simulate the attack using the same conditions used by the XFC framework for this purpose. However, one should notice that the complexity of the attack still remains 2^{32}.

5.2 AES: Fault Injection at the Beginning of the 7-th Round

While XFC is unable to detect this attack, we find that the proposed framework can figure out an impossible differential attack for this case. It is evident from Fig. 1 as well as from the outcome of our tool, that the fault injection at the 7th round results in impossible differentials at the states δ_8^4, δ_9^1, and δ_9^2 and δ_9^2 is exploited for the attack. Althogh, the state differential δ_9^2 does not possess any association rules, the `RngChk` function detects the absence of 0 in each of the variables, and as a result, the entropy becomes $H_{Ind}(\delta_9^4) = 127.90$, which makes the state differential qualify as a distinguisher. The distinguisher evaluation in this case demands some special attention. It is observed that each byte of the distinguisher depends on 32 key bits from the 10-th round. Note that, the BFS tree obtained from the CDG will also include the 9-th round keys here. However, the 9-th round keys are discarded following the rule that there are no S-Boxes between them and the distinguisher. The distinguisher evaluation complexity becomes 2^{32}. Finally, the complexity of the remaining key space is evaluated, which in this case, turns out to be $|\mathscr{R}|_{VG_1} = 2^{32} \times (\frac{255}{2^8})^4$ (roughly equal to $2^{32} - 2^{26}$). The large size of the remaining key space indicates the need of multiple fault injection. Although, the estimation of the required number of faults here is slightly nontrivial due to the inequalities involved, it can be trivially estimated using the construction from [2].

5.3 PRESENT: 2-Byte Fault Injection at the Beginning of the 28-th Round

The third test case is quite distinct from the first two cases, owing to the fact that, PRESENT does not use any MDS matrix in its diffusion layer. The attack described here is similar to that proposed in [5]. The distinguisher identification algorithm, in this case, identify the input state of the S-Box of the $30-th$ round (δ^1_{30}) as the best distinguisher. The `RngChk` function identifies that the variables in this state differential can take only 2 values among 16 possible values (although the values may change depending on the fault locations), and as a result, the entropy becomes $H_{Ind}(\delta^1_{30}) = 16$. As it can been 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. As a result, the evaluation complexity becomes 2^{32}. For each of the 4 (MKS_h, VG_h) pair, which involves 2^{32} keys, the probability of occurrence of the distinguishing criterion is $\mathbb{P}[VG_h] = (2^{-3})^4 = 2^{-12}$, and the remaining key is $|\mathscr{R}|_{VG_h} = 2^{20}$. With a single fault injection, thus the remaining keyspace reduces to 2^{80} from 2^{128} 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).

6 Conclusion

In this paper, we have proposed a completely 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 is better than another recently proposed framework, in terms of its coverage of different fault cases and degree of automation. Three step-by-step case studies on different ciphers and fault attack instances were presented to establish the claims. 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 and MitM attacks [7, 8] which also seems to be feasible in this data-analysis based framework. Further, automated design of countermeasures with the assistance from this tool could be another direction of research.

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

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

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

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

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

[8] 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.

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

[10] 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.