Pushing the Limit of PFA: Enhanced Persistent Fault Analysis on Block Ciphers

Guorui Xu, *Student Member, IEEE*, Fan Zhang, *Member, IEEE*, Bolin Yang, *Student Member, IEEE*, Xinjie Zhao, Wei He, and Kui Ren, *Fellow, IEEE*

***Abstract*—Persistent fault analysis (PFA) is a newly proposed cryptanalysis for block ciphers. Although the injected fault is persistent during the entire encryption, the corresponding analysis is only applied to the last round in the original PFA. In this article, the enhanced PFA (EPFA) is proposed, which can push the limit of PFA by exploiting the fault leakage in deeper rounds and target to reduce the number of required ciphertexts as small as possible. EPFA is first introduced as a general method with a specific application to advanced encryption standard (AES). Then it is extended to other substitution–permutation network (SPN)-based block ciphers, such as LED and SKINNY, both of which have unique features that EPFA fits well. To improve the efficiency of EPFA, a parallel algorithm based on mixed radix numbers is developed, which fully utilizes the power of GPU. Our experimental results show that EPFA can reduce the number of required ciphertexts to be under 1000, which is only about 40% of the 2500 ciphertexts in previous PFA on AES. In contrast to the single-threaded implementation, the parallel EPFA can have a speedup roughly about 200 times.**

***Index Terms*—Advanced encryption standard (AES), fault analysis, fault attack, GPU-accelerated computing, LED, persistent fault analysis (PFA), SKINNY.**

Manuscript received March 31, 2020; revised July 27, 2020 and November 2, 2020; accepted December 15, 2020. Date of publication December 30, 2020; date of current version May 20, 2021. This work was supported in part by the National Key Research and Development (R&D) Program of China under Grant 2020AAA0107700; in part by the National Natural Science Foundation of China under Grant 62072398 and Grant 61772236; in part by the Open Fund of State Key Laboratory of Cryptology under Grant MMKFKT202013; in part by the Alibaba–Zhejiang University Joint Institute of Frontier Technologies; in part by the Zhejiang Key R&D Plan under Grant 2019C03133; in part by the Major Scientific Research Project of Zhejiang Lab under Grant 2018FD0ZX01; in part by the Young Elite Scientists Sponsorship Program by CAST under Grant 17-JCJQ-QT-045; in part by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang under Grant 2018R01005; and in part by the Research Institute of Cyberspace Governance in Zhejiang University. This article was recommended by Associate Editor A. Sengupta. *(Corresponding author: Fan Zhang.)*

Guorui Xu is with the School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China, also with the State Key Laboratory of Cryptology, Beijing 100878, China, and also with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou 310027, China.

Fan Zhang and Kui Ren are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 311127, China, and also with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou 310027, China (e-mail: fanzhang@zju.edu.cn; kuiren@gmail.com).

Bolin Yang is with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310058, China.

Xinjie Zhao is with the Institute of North Electronic Equipment, Beijing 100191, China.

Wei He is with the China Telecom BestPay Company, Ltd., Shanghai 200100, China.

Digital Object Identifier 10.1109/TCAD.2020.3048280

I. Introduction

FAULT attacks can inject deliberate faults to cause faulty computations and produce faulty results [1], [2]. Fault analysis is the offline stage of fault attacks, which focuses on how to use the faulty results to recover some secret information used in the computation [3].

It has been demonstrated that many block cipher cryptosystems are vulnerable to fault attacks. First in 1997, Biham and Shamir developed differential fault analysis (DFA), showing that DFA is applicable to almost any secret key cryptosystem [1]. The powerful cryptanalysis method is based on the differences between correct and faulty ciphertexts. Under a transient fault model, DFA can extract the full DES key by analyzing 50–200 ciphertexts [1]. Later in 2003, Pierre and Gilles presented DFA on the advanced encryption standard (AES) [3]. AES without faults is not easy to compromise. However, when the adversary can inject faults and collect some faulty ciphertexts, he is able to recover the secret key in a few minutes. This could be a great threat if the injection of faults and the collection of ciphertexts can be done at a low cost. One promising research direction is to make fault attacks more practical, either to simplify the injections with ease efforts and fewer engineering requirements or to push the limit of analysis costs like using as few ciphertexts or resources as possible.

In previous works, most fault models consider the fault to be transient, which means that the fault only appears for a very short period in the process (typically one clock cycle), or permanent, which causes the device to be damaged permanently. Zhang *et al.* [4] proposed a new type of fault named persistent faults in 2018, which can be persistent during many encryptions and disappear when the target gets refreshed (such as rebooted). In Zhang's work [4], the online injection part is done by the RowHammer technique, which injects a persistent bit-flip fault in the S-box. The offline analysis part is named as persistent fault analysis (PFA). Using such faults with persistency, they applied proper statistical methods to extract the secret key with less than 2500 faulty ciphertexts. PFA can be very promising in this way. However, how to reduce the number of required ciphertexts and how to fully utilize the fault leakages, have not been investigated yet.

In this article, we propose a powerful extension of PFA, named as enhanced PFA (EPFA). EPFA naturally inherits all the advantages of PFA, for example, it needs ciphertexts only, it does not damage the physical device and it is able to break some countermeasures. Moreover, EPFA requires much fewer ciphertexts than the original PFA (only 40% in AES) by utilizing the fault leakages as much as possible. To clarify the methodology of EPFA, we propose a general algorithm on substitution–permutation network (SPN)-based block ciphers first, and then apply this generic method to some specific SPN ciphers (i.e., LED [5] and SKINNY [6]) as examples.

A. Related Work

Fault attacks focus on the physical implementation of encryption algorithms. In fault attacks, an adversary uses various possible means, such as laser, clock/voltage glitch, EM waves, heat, to interfere the normal behaviors, such as storage I/O, time synchronization of the target device, and the execution of instructions. Then the adversary observes the impacts of the fault, thereby obtaining the extra information to reveal the secrets. Fault attacks can be applied not only to the encryption devices but also to some other targets providing authentication or personal identification. For instance, it can bypass some security restrictions in the smart cards [7].

With fault analysis, the adversaries will be able to exploit the impacts of the faults and recover the secret key. There are many analysis methods, including DFA [3], [8], statistical fault analysis (SFA) [9], [10], ineffective fault analysis (IFA) [11], [12], algebraic fault analysis (AFA) [13], [14], fault sensitivity analysis (FSA) [15], etc.

DFA on AES was proposed by Piret and Quisquater in 2003 [16]. They assumed that the fault occurs between the 8th and 9th MIXCOLUMN. Then they are able to break the AES-128 with only two faulty ciphertexts using DFA. In Piret's work, the fault must be introduced accurately between two specific rounds, which raises the extra sophisticated requirements for the adversary.

The first SFA is proposed in 2000, by Fluhrer and McGrew [9]. They demonstrate a method for distinguishing 8-bit RC4 from randomness, which introduces weaknesses in the keystream generation using RC4. Statistics-based fault analysis is often very powerful in ciphertext-only scenarios. For instance, Fuhr *et al.* applied a statistical method to the fault analysis on AES and recovered the key without knowing the corresponding plaintexts [2]. The adversary needs to inject faults in the last four rounds, and then collect the ciphertexts. This attack breaks some countermeasures which use a randomized mode of operation to prevent ordinary fault attacks. Dobraunig *et al.* proposed the statistical IFA (SIFA) on CHES 2018 [12] and Asiacrypt 2018 [17]. It is a black-box attack that exploits the unbalanced distribution in the ineffective ciphertexts. The authors also attacked the masked AES implementation with fault countermeasures.

PFA is a newly proposed cryptanalysis for block ciphers [4] in 2018. The adversary focuses on the direct output of the last S-box to build constraints. As a result, the adversary can recover the full key of AES-128 by analyzing 2500 collected ciphertexts [4]. Similar to Fuhr's work [2] and SIFA [12], PFA is also a ciphertext-only attack. When the adversaries collect enough ciphertexts, they are able to perform the analysis to recover the key with statistical methods. Unlike SIFA [12] which only focuses on the intermediate state distribution of the correct ciphertexts, PFA collects all ciphertexts no matter whether they are correct. It can even break the implementations with countermeasures (e.g., Dual Modular Redundancy) against fault attacks with 8–10 times more ciphertexts (i.e., 20k–25k ciphertexts) [4]. However, the original PFA only utilizes a part of the leaked information induced by the injected persistent fault. Thus, this attack is not optimal and can be improved.

B. Contribution

Our contributions can be summarized as follows.

1) We propose the EPFA as a powerful extension of PFA from a general view, which can be applied to all SPN-based block ciphers and use multiple rounds to build constraints.
2) We investigate the characteristics of EPFA on AES in detail. The proposed method can reduce the required number of ciphertexts from 2500 to below 1000, which is only 40% of the traditional PFA on AES-128.
3) We extend the EPFA method to some other block ciphers (i.e., LED and SKINNY). We manage to conquer some challenges that have not been explored.
4) We develop a parallel version of EPFA which can accelerate the attack significantly when the computation complexity is large. The speedup achieved through the parallelization can be up to 200 times under certain configurations.

II. PERSISTENT FAULT ANALYSIS

In this section, we will introduce the original *PFA* [4] with a demonstrative example.

A. Persistent Fault Analysis

As Section I suggested, PFA is a ciphertext-only cryptanalysis method. The attack model can be described as follows: 1) the adversary injects a persistent fault which will last for a relatively long time into a predefined substitution table (or constants); 2) the victim performs multiple encryptions using the faulty table and a fixed key; and 3) the adversary collects the ciphertexts and uses PFA to recover the secret key.

Note that the introduced fault will exist on a single cell of the substitution table. In the full process of computation, the faulty cell might be accessed or not. Therefore, some output ciphertexts will be faulty while others remain unaltered. As for a single ciphertext, it is difficult for the adversary to tell whether it is correct or not without knowing the plaintext. However, he can find a distribution change of the ciphertext bytes by collecting multiple ciphertexts, and this change can be exploited to recover the key. We will discuss the analysis process in detail with an example next.

B. Demonstrative Example of PFA

We will first apply the PFA on a very common and basic component that may appear in many block ciphers, as shown in Fig. 1(a). It substitutes one byte using a lookup table and then XORs it with a key byte. S denotes the bijective substitution

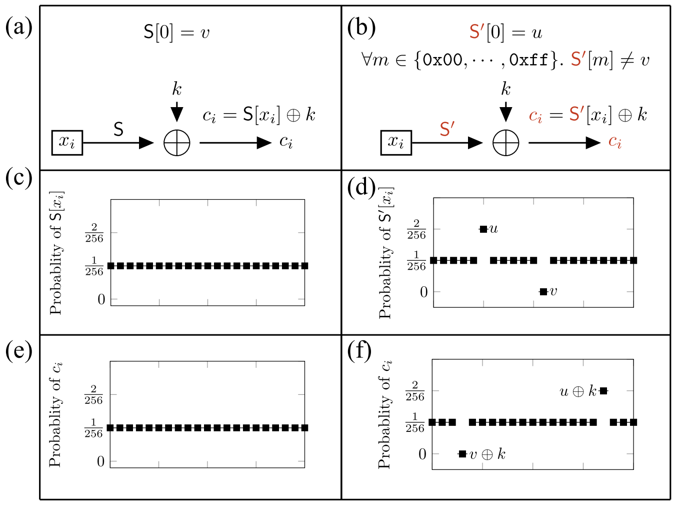

(a)
$\mathsf{S}[0] = v$
k
$c_i = \mathsf{S}[x_i] \oplus k$
x_i
S
c_i
(b)
$\mathsf{S}'[0] = u$
$\forall m \in \{\texttt{0x00}, \cdots, \texttt{0xff}\}.\ \mathsf{S}'[m] \neq v$
$c_i = \mathsf{S}'[x_i] \oplus k$
S'
(c)
Probability of $\mathsf{S}[x_i]$
$\frac{2}{256}$
$\frac{1}{256}$
0
(d)
Probability of $\mathsf{S}'[x_i]$
u
v
(e)
Probablity of c_i
(f)
Probablity of c_i
$u \oplus k$
$v \oplus k$

Fig. 1. PFA on a basic component. (a) Computation procedure of the basic component. (b) Distribution of $S[x_i]$. (c) Distribution of the output byte c_i. (b), (d), and (f) Procedure and distributions of $S[x_i]$ and c_i when using a faulty S-box, respectively.

table that maps one byte to another. k denotes a single key byte. The substituted $\mathsf{S}[x_i]$ and the key byte k will be XORed to produce the final element c_i in corresponding ciphertexts. Our goal is to recover the k by collecting the output bytes, with a persistent fault injected into the substitution table.

In the fault injection phase, a single fault is injected into the S-box. Without loss of generality, we can take the first element of $\mathsf{S}[0]$ as an example. Its original value is denoted as $v = \texttt{0x63}$. In the faulty S-box S', the first element $\mathsf{S}'[0]$ is changed to $u = \texttt{0x61} \neq v$. Before the fault injection, the distribution of the output [Fig. 1(c)] and ciphertext [Fig. 1(e)] should be uniform due to the pseudorandomness of a well-designed block cipher. However, after the fault injection, the bijection is corrupted, leading to some faulty outputs of the substitution. The byte value u will be output with a larger probability (being mapped twice) and the byte value v will *never* appear. Therefore, the distribution of the output $\mathsf{S}'[x]$ [Fig. 1(d)] and the final ciphertexts [Fig. 1(f)] will be changed. The distribution of Fig. 1(f) is a simple permutation for that of Fig. 1(d) because the key k is XORed.

For each input byte x_i, we can build a constraint equations between the output byte c_i and the key byte k as (1). The first equation is from the injected fault, and the second equation is from the computation of c_i in Fig. 1(b)

$$\left.\begin{array}{r}\forall m, \mathsf{S}'[m] \neq v \\ c_i = \mathsf{S}'[x_i] \oplus k\end{array}\right\} \quad k \neq c_i \oplus v. \tag{1}$$

Since the fault is persistent, the substitution will be faulty during multiple encryptions. In this way, the adversary will be able to collect many bytes produced from the component using the same key byte k and the faulty table S'. The initial search space for that unknown key byte is the full space of an 8-bit data: $\{\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xff}\}$. When he collects one byte c_0, such as $c_0 = \texttt{0x0a}$, he knows that the actual key value cannot be $c_0 \oplus v = \texttt{0x69}$ according to the constraint (1). Therefore, the search space is reduced by one element $\texttt{0x69}$. By repeating the procedure, the key search space will be continuously reduced to only one possible key value remaining, which must be equal to the actual key byte k.

C. Extends to Full Key Recovery

As for a real-world example in AES, PFA will work similarly to what we have discussed previously. One difference is that we are not going to recover just one single byte but a full 128-bit AES key. According to the cipher design of AES, the state matrix contains 16 bytes and each byte has an independent component. Thus, the 16 ciphertext bytes will be analyzed independently and the full key can be recovered by simply combining all the recovered key bytes.

III. Enhanced Persistent Fault Analysis

In this section, we introduce the motivation and the core idea of EPFA by extending the example in Section II.

A. Motivation for Further Study on PFA

The original paper [4] shows that PFA is quite effective for block ciphers and has a good performance in breaking existing countermeasures. However, it can be improved and enhanced to recover the secret key with fewer efforts. Mainly there are three motivations for further study on PFA shown as follows.

1) In some scenarios, it is hard to collect enough ciphertexts required by PFA. For example, when using an aggressive key rotation policy, the encryption device will be forced to change the master key after encrypting 16-kb data. Since PFA requires 2500 ciphertexts on average, i.e., 2500×16 b $\simeq 39.06$ kb > 16 kb data, the recovery of the master key is impossible.
2) The time for the ciphertext collection can be very large. For example, using a covert channel to do transmission may have a data rate under 10 bits/s. In such a situation, using fewer ciphertexts to recover the key is essentially important because of the high cost of data collection.
3) Moreover, the original PFA is based on the output distribution of the S-box only in the last round. Since the fault persists in many rounds and datapaths, the original PFA has not fully utilized the fault leakages.

This article is an extension of the original PFA work. Next, we will show how to extend PFA to multiple-round analysis with an example.

B. Extend PFA to Two-Round Analysis

In many designs of AES-like block ciphers, the substitution table will be used multiple times in a full encryption. Fig. 2 shows an example of analyzing the last two rounds. k, k^* denote round key bytes where $k^* = g(k)$ (g is a transformation function). h denotes the transformation function for the intermediate byte y_i. Other parts are similar to those in Fig. 1(a).

As for the analysis in the rth round, it is almost identical to the PFA example shown in Section II, and we also have the constraint (1). The additional round introduces equations between the intermediate bytes (y_i, z_y) and the key bytes (k, k^*). Using these equations, we can build another constraint

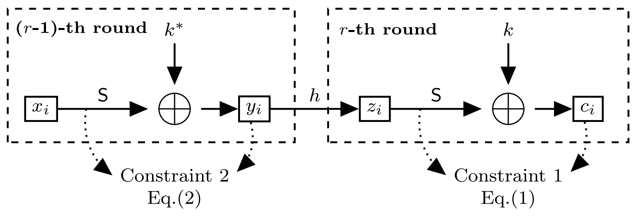


Fig. 2. Last two rounds for AES block cipher.

as follows:

$$\left.\begin{aligned} &\forall m, \mathsf{S}'[m] \neq v \\ &y_i = k^* \oplus \mathsf{S}'[x_i] \\ &z_i = h(y_i) \\ &k^* = g(k) \end{aligned}\right\} \quad g(k) \neq h^{-1}(\mathsf{S}^{-1}[c_i \oplus k]) \oplus v \qquad (2)$$

where h^{-1} and S^{-1} are the inverse of h and S.

Note that both (1) and (2) only require the output c_i and the key k, where k is the unknown. In this way, we have more equations and chances to reduce the search space of k with the collected output c_i.

C. Revisit the Demonstrative Example

From Fig. 2, we notice that the two-round analysis is a naive extension of the example in Section II. The newly introduced constraint can reduce the search space further with already collected ciphertexts. For example, if the search space is {0x76, 0x4f, 0x30, 0x0e} after the analysis in the rth round. By checking the new constraint (2), the adversary finds that 0x76 does not satisfy. Therefore, the search space is reduced by one more element. We will discuss the design of a complete EPFA in detail in the next section.

IV. Design of EPFA

In this section, we will introduce the model of a general SPN-based block cipher, generalize the two-round analysis mentioned before, and describe the design of EPFA in detail.

A. General SPN-Based Block Ciphers

Before we move onto the design of EPFA, we will discuss the general SPN-based block ciphers first.

In cryptography, a block cipher is a function that can map a data block of plaintext to ciphertext using a secret key. The block is often represented as a matrix (or array) and each element of the matrix/array can be a nibble or a byte (for simplicity, we skip the case for those at word levels). One type of important block cipher designs is the SPN, which consists of a series of basic operations, including as follows.

1) *Substitution:* This operation takes a data block as input, and substitutes the elements in it according to an S-box S. This substitution in SPN using S is usually a bijection, to make sure that it can be inversed in another direction. Such operation can be denoted as $\boldsymbol{Y} = \mathsf{S}(\boldsymbol{X})$, where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the input and output data block, respectively.
2) *Permutation:* This operation takes the full data block as input, and permutates the bits of the combined data by a P-box P. That is to say, permutation is not an element-wise operation, it directly operates the bits. $\mathsf{P}[i] = j$ means that the output $\boldsymbol{Y}$'s $(j+1)$th bit will be equal to the $(i+1)$th bit of the input $\boldsymbol{X}$ where the first bit has index 0. This operation can be denoted as $\boldsymbol{Y} = \mathsf{P}(\boldsymbol{X})$.
3) *Key Addition:* Normally, this is the part in SPN that directly uses the key-related information. It will XOR the input data block $\boldsymbol{X}$ with a round key $\boldsymbol{K}$, which can be denoted as $\boldsymbol{Y} = \boldsymbol{X} \oplus \boldsymbol{K}$.

Many block ciphers are based on SPN structure. A well-designed SPN with several rounds satisfies Shannon's confusion and diffusion properties. Such ciphers apply these three operations to the plaintext block round by round, and then produce the final ciphertext block (e.g., AES [18], PRESENT [19], LED [5], SKINNY [6], etc.). The operations that are applied multiple times are named *round functions*. A general SPN-based block cipher has two parts: 1) the key scheduling and 2) the encryption (decryption). The key scheduling part uses the master key to produce the round keys. The encryption (decryption) part takes round keys and plaintext (ciphertext) as inputs, and applies the round function with aforementioned three operations (or their inverse) round by round.

Recall the basic component we introduced in Section II, we note that this kind of component appears in each round. In this way, a typical SPN-based block cipher can be represented as the top half of Fig. 3. S and P represent the substitution and permutation, respectively. $\boldsymbol{P}$ and $\boldsymbol{C}$ denote the plaintext and ciphertext block, respectively. $\boldsymbol{I}_i$ denotes the intermediate data block in ith round, which is the output of the $(i-1)$th basic block key addition. The composition of P and S is denoted as $\mathsf{P} \circ \mathsf{S}$, where $(\mathsf{P} \circ \mathsf{S})(\boldsymbol{X}) = \mathsf{P}(\mathsf{S}(\boldsymbol{X}))$. Note that S and P can vary from round to round, so subscripts need to be added to differentiate Ss and Ps in different rounds.

B. Design of EPFA

Here, we introduce the attack model. The adversary (denoted as $\mathcal{A}$) injects a fault into the S-box. The following encryptions will use the faulty S-box (S') in the substitution operations as shown in the bottom part of Fig. 3. This inference caused by the faulty S-box can build constraint equations. In each round, S' will produce a subsequent state matrix that *does not* contain one specific byte value v (similar to the analysis in Section II of the original PFA). This value v then will be regarded as the "impossible value," is known to $\mathcal{A}$ if he knows the location of the injected fault.

For each round, $\mathcal{A}$ can build a group of constraint equations based on this impossible value v. Suppose in the first round of an SPN block cipher, the constraint equation takes the elements of the intermediate state $\boldsymbol{I}_1$ and the first round key $\boldsymbol{K}_0$ as inputs. From Fig. 3, we have $\boldsymbol{I}_1 = \mathsf{P}(\mathsf{S}'(\boldsymbol{P})) \oplus \boldsymbol{K}_0$. Then, the output of the S-box S' can be computed as $\mathsf{S}'(\boldsymbol{P}) = \mathsf{P}^{-1}(\boldsymbol{I}_1 \oplus \boldsymbol{K}_0)$. Thus, for each element of $\mathsf{S}'(\boldsymbol{P})$, it can never be equal to the impossible value v, which can be served as the constraints to build the first equation group.

Similarly, in Fig. 3, $\mathcal{A}$ will have at least $(r+1)$ groups of constraint equations to construct, which can be denoted as

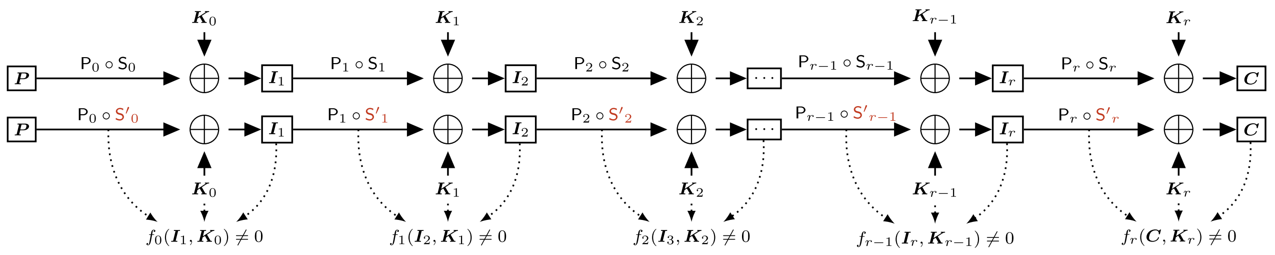


Fig. 3. r-round general block cipher with faults. The constraint equations $f_i(\cdot)$ can be built in each round, utilizing the fault in the substitution table S'_i. These equations take the candidates of next intermediate state $\boldsymbol{I}_{i+1}$ and current round key $\boldsymbol{K}_i$ as inputs.

$$f_0(\boldsymbol{I}_1, \boldsymbol{K}_0) \neq 0 \tag{3a}$$
$$\cdots$$
$$f_{r-1}(\boldsymbol{I}_r, \boldsymbol{K}_{r-1}) \neq 0 \tag{3b}$$
$$f_r(\boldsymbol{C}, \boldsymbol{K}_r) \neq 0. \tag{3c}$$

Note that these equations take the intermediate states and round keys in each round as inputs. To use these constraints, the adversary needs to compute (or guess) the values of some inputs with the knowledge he already has. For example, the ciphertext $\boldsymbol{C}$ is known to $\mathcal{A}$ in (3c), but the rth round key is not. So $\mathcal{A}$ guesses one candidate of the round key from the entire key search space. The guessed key and each collected ciphertext will be fed into (3c) to check if the constraint is satisfied. Wrong key guesses will not pass this check with a very high probability. In this way, $\mathcal{A}$ can reduce the key search of the corresponding round key.

Unfortunately, because of the data dependency among the intermediate states and round keys, multiple constraints are not suitable to be simultaneously utilized straightforwardly. In fact, solving many constraints at the same time will bring *very large* computation efforts and have *very large* search space. For instance, if $\mathcal{A}$ wants to use both (3b) and (3c) at the same time, the $(r-1)$th intermediate data block, together with the key candidates of both $(r-1)$th and rth round, should be known. However, since $\mathcal{A}$ only knows the ciphertexts, he can either merely use the last constraint equation, or have to compute all possible intermediate states and all corresponding possible round keys, which is computationally infeasible.

Thus, a more plausible way is to use these constraints round by round in sequence, instead of simultaneously solving them in parallel. At the very beginning, $\mathcal{A}$ applies (3c) and gets the candidate set of the rth round key (denoted as $\mathbb{K}_r$). Then, to initialize the inputs of the next constraint equation, the adversary $\mathcal{A}$ computes the corresponding intermediate states and round key. This process is shown in (4) and (5). The functions g_i and h_i used here only depend on the key derivation function and round function in the concrete cipher design

$$g_{r-1}(\boldsymbol{K}_r) = \boldsymbol{K}_{r-1} \tag{4}$$
$$h_{r-1}(\boldsymbol{C}) = (\mathsf{P}_r \circ \mathsf{S}_r)^{-1}(\boldsymbol{C} \oplus \boldsymbol{K}_r) = \boldsymbol{I}_r. \tag{5}$$

Then, the candidate set $\mathbb{K}_{r-1}$ for $(r-1)$th round key can be found by solving (3b). In this way, the constraints equations can be solved round by round in further analysis. As a special case of EPFA, PFA simply ends at (3c) (the last round) and does not use any other information leaked by the persistent fault in deep rounds.

V. EPFA ON AES

In this section, we instantiate the general EPFA on AES, which is one of the most well acknowledged block ciphers with SPN structure. First we use the constraints in the last round to get a smaller search space of the round key, and then we apply constraint equations in the penultimate round to further reduce the search space.

A. AES Cipher

AES was published by NIST in 2001 [18]. According to the general block cipher model, its major parts can be detailed as follows.

Key Schedule: The AES algorithm uses Rijndael's key schedule. It will perform some basic operations (subsititute 32-bit word byte by byte, XOR with round constants, etc.), to generate 128-bit round keys for each round. The round constants h_i are defined in the specification [18].

Encryption: In the encryption part, AES uses round operations to transform one state matrix to another. The round function includes four operations: SUBBYTES (SB), SHIFTROWS (SRs), MIXCOLUMNS (MC), and ADDROUNDKEY (AK). Especially in the last round, there is no MC operation according to the specification. The state matrix is sized 4×4, containing 16 bytes in total. SUBBYTES (SB) takes the matrix as input, and uses the same S-box to substitute each element in the matrix for every round. SR and MC reorder and mix the state matrix by row and column, respectively. They play the permutation role in this cipher. AK simply XORs the round key with the state matrix byte by byte.

Taking the plaintext as the input matrix, and then performing these operations round by round according to the specification [18], eventually we can get the corresponding ciphertext.

B. Instantiation for EPFA on AES

In the following analysis, the persistent fault is injected to the S-box of AES. Since all the ten rounds are accessing to the same S-box, such fault will persist in the entire encryption. It means that all the faulty S-boxes $S'_1, S'_2, \ldots, S'_{10}$ are the same in the AES instance, which can be denoted as S' where $S' = S'_1 = S'_2 = \cdots = S'_{10}$. The constraint equations in each round are based on the impossible value v caused by the persistent fault. They can be denoted as

$$f_i(\boldsymbol{I}_i, \boldsymbol{K}_{i-1}) = \left(\mathsf{P}^{-1}(\boldsymbol{I}_i \oplus \boldsymbol{K}_{i-1})\right)^{(j)} \oplus v \neq 0 \tag{6}$$

where $1 \leq i \leq 10$ and $1 \leq j \leq 16$.

As for the key schedule algorithm, the Rijndael's key schedule used in AES-128 guarantees that each round key can be directly computed from any other. Thus, the key computation equations (4) can be easily derived by reversing the key schedule. It is equivalent that the adversary $\mathcal{A}$ gets any round key or the master key.

C. Constraint Computation for Last Round

The constraint equations are based on the impossible byte value v of the S-box. Thus, they can be simplified as $\mathsf{S}(I_i^j) \neq v$ in all ciphertexts where $1 \leq i \leq 10, 1 \leq j \leq 16$. The original PFA only considers the last round, which means that only the equations $\mathsf{S}(I_{10}^j) \neq v$ are used in cryptanalysis. When considering those deeper rounds, more constraint equations can be constructed to improve the power of enhanced cryptanalysis.

Suppose the faulty ciphertext $\boldsymbol{C}$ and the 10-th round key $\boldsymbol{K}_{10}$ are denoted as

$$\boldsymbol{C} = \begin{bmatrix} c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \\ c_4 & c_8 & c_{12} & c_{16} \end{bmatrix} \quad \boldsymbol{K}_{10} = \begin{bmatrix} k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \\ k_4 & k_8 & k_{12} & k_{16}. \end{bmatrix}$$

For simplicity, we use k_i to denote the elements in $\boldsymbol{K}_{10}$, since they will be frequently used in further analysis. Then, the output state matrix of 10th S-box, denoted as $S(\boldsymbol{I}_{10})$, can be represented as

$$\begin{bmatrix} c_1 \oplus k_1 & c_5 \oplus k_5 & c_9 \oplus k_9 & c_{13} \oplus k_{13} \\ c_6 \oplus k_6 & c_{10} \oplus k_{10} & c_{14} \oplus k_{14} & c_2 \oplus k_2 \\ c_{11} \oplus k_{11} & c_{15} \oplus k_{15} & c_3 \oplus k_3 & c_7 \oplus k_7 \\ c_{16} \oplus k_{16} & c_4 \oplus k_4 & c_8 \oplus k_8 & c_{12} \oplus k_{12} \end{bmatrix}. \tag{7}$$

The term *candidate set* stands for the search space of key bytes or round keys. The candidate set for a key byte can be denoted as $\mathbb{D}_i$, which is initialized as $\mathbb{D}_i = \{0, 1, \ldots, 2^8 - 1\}$. The candidate set for a round key (equivalent to the master key on the computation) is computed as the Cartesian product of all $\mathbb{D}_i$. For the last round key $\mathbb{K}_{10}$, we have $\mathbb{K}_{10} = \prod_{i=1,2,\ldots,16}^{i} \mathbb{D}_i$ where $\prod$ is the Cartesian product.

If the specific byte value of the first ciphertext is c_1, then the adversary $\mathcal{A}$ knows that k_1 (the first byte in $\boldsymbol{K}_{10}$) cannot be $c_1 \oplus v$. The candidate set $\mathbb{D}_1$ will be reduced to $\mathbb{D}_1/\{c_1 \oplus v\}$ as shown in Algorithm 1. The operand "/" means to remove one element from the set. If enough ciphertexts are collected, $\mathbb{D}_1$ will be eventually reduced to a set containing only one element. This element must be the value of k_1. Recall that the goal of our improvement is to reduce the number of ciphertexts. When the required ciphertexts are not enough and no more ciphertexts are added to the analysis, $\mathbb{D}_i$ in Algorithm 1 may still contain several elements other than a single one.

There is an observation that the calculation to narrow down each $\mathbb{D}_i$ only requires one specific byte in the ciphertexts and has nothing to do with the other 15 values. Thus, all the 16 candidate sets $\mathbb{D}_i$ can be independently generated using

Algorithm 1 Compute One Candidate Set in Round 10

```
Input: The set of collected ciphertexts $\mathbb{C}$, the impossible value $v$ in S-box;
1: $\mathbb{D}_i \leftarrow \{0, 1, \cdots, 2^8 - 1\}$;
2: for all $C$ in $\mathbb{C}_i$ do
3:     if $C^i \oplus v \in \mathbb{D}_i$ then
4:         $\mathbb{D}_i \leftarrow \mathbb{D}_i/\{C^i \oplus v\}$;
5:     end if
6: end for
7: return $\mathbb{D}_i$
```

Algorithm 1. The candidate set for the last round key (i.e., $\mathbb{K}_{10}$) will be narrowed gradually.

Instead of increasing the number of ciphertexts that required, we can leverage more constraint equations to recover the round key using as few ciphertexts as possible. The process how to push the limit of PFA will be shown in the next section.

D. Constraint Computation for Penultimate Round

Following the procedures in Section II, the next stage is to use (6) in the penultimate round to further reduce the key search space. In these equations, the 9th round key $\boldsymbol{K}_9$ and the intermediate state $\boldsymbol{I}_{10}$ are required to be computed first.

According to the AES specification [18], the round key $\boldsymbol{K}_9$ can be computed from $\boldsymbol{K}_{10}$, which is shown as

$$\begin{bmatrix} k_1 \oplus \mathsf{S}(k_{14} \oplus k_{10}) \oplus h_{10} & k_5 \oplus k_1 & k_9 \oplus k_5 & k_{13} \oplus k_9 \\ k_2 \oplus \mathsf{S}(k_{15} \oplus k_{11}) & k_6 \oplus k_2 & k_{10} \oplus k_6 & k_{14} \oplus k_{10} \\ k_3 \oplus \mathsf{S}(k_{16} \oplus k_{12}) & k_7 \oplus k_3 & k_{11} \oplus k_7 & k_{15} \oplus k_{11} \\ k_4 \oplus \mathsf{S}(k_{13} \oplus k_9) & k_8 \oplus k_4 & k_{12} \oplus k_8 & k_{16} \oplus k_{12} \end{bmatrix}$$

where $h_{10} = \texttt{0x36}$, defined in the round constants in the specification [18].

Based on the cipher structure, $\boldsymbol{I}_{10}$ can be derived using the final ciphertext $\boldsymbol{C}$ and the last round key $\boldsymbol{K}_{10}$

$$\mathsf{S}'(\boldsymbol{I}_{10}) = \mathsf{P}_{10}^{-1}(\boldsymbol{C} \oplus \boldsymbol{K}_{10}). \tag{8a}$$

$$\boldsymbol{I}_{10} = \mathsf{S}^{-1}(\mathsf{P}_{10}^{-1}(\boldsymbol{C} \oplus \boldsymbol{K}_{10})). \tag{8b}$$

Note that the inverse S-box S^{-1} is used in (8b). If the right-hand side (RHS) of (8a) contains impossible value v, the actual preimage of this byte in $\boldsymbol{I}_{10}$ cannot be determined. Thus, the intermediate state $\boldsymbol{I}_{10}$ cannot be computed in this case. Fortunately, the theoretical analysis in the original PFA paper [4] shows that this case occurs with a low probability. The only thing we need to do is to add a filtering step to filter out these ciphertexts that cannot be used to compute the previous state.

As for the next round, the intermediate state $\mathsf{S}'(\boldsymbol{I}_9)$ can be derived as $\mathsf{S}'(\boldsymbol{I}_9) = \mathsf{P}_9^{-1}(\boldsymbol{I}_{10} \oplus \boldsymbol{K}_9)$. Combined with the concrete permutation implementation in AES, the bytes in $\mathsf{S}'(\boldsymbol{I}_9)$ can be computed as follows:

$$\begin{aligned} \mathsf{S}'(I_9^{(1)}) = {} & (\mathsf{S}^{-1}(c_1 \oplus k_1) \oplus (k_1 \oplus \mathsf{S}(k_{14} \oplus k_{10}) \oplus h_{10}) \cdot 14) \\ & \oplus (\mathsf{S}^{-1}(c_{14} \oplus k_{14}) \oplus (k_2 \oplus \mathsf{S}(k_{15} \oplus k_{11})) \cdot 11) \\ & \oplus (\mathsf{S}^{-1}(c_{11} \oplus k_{11}) \oplus (k_3 \oplus \mathsf{S}(k_{16} \oplus k_{12})) \cdot 13) \\ & \oplus (\mathsf{S}^{-1}(c_8 \oplus k_8) \oplus (k_4 \oplus \mathsf{S}(k_{13} \oplus k_9)) \cdot 9) \end{aligned} \tag{9}$$

where "$\cdot$" means the multiplication in finite field GF_2^8.

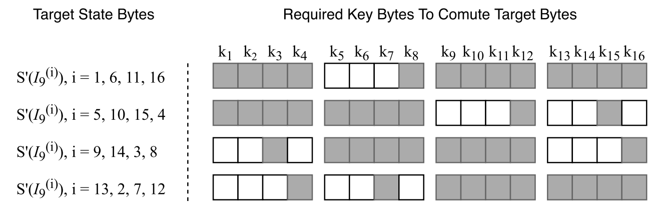

Target State Bytes
Required Key Bytes To Comute Target Bytes
k1 k2 k3 k4 k5 k6 k7 k8 k9 k10 k11 k12 k13 k14 k15 k16
S'(I9(i)), i = 1, 6, 11, 16
S'(I9(i)), i = 5, 10, 15, 4
S'(I9(i)), i = 9, 14, 3, 8
S'(I9(i)), i = 13, 2, 7, 12

Fig. 4. Required key bytes when computing the state in the penultimate round. The gray blocks on the right represent the required bytes to compute the corresponding $\mathsf{S}'(I_9^{(i)})$ on the left where $1 \le i \le 16$.

Other bytes can be computed in the same way. Note that to compute one element in the target state, only 10 or 13 key bytes are required, instead of using all 16 key bytes. For example, to compute $\mathsf{S}'(I_9^{(1)})$, $\mathsf{S}'(I_9^{(6)})$, $\mathsf{S}'(I_9^{(11)})$, $\mathsf{S}'(I_9^{(16)})$ in the penultimate round, those key bytes k_1, k_{14}, k_{10}, k_2, k_{15}, k_{11}, k_3, k_{16}, k_{12}, k_8, k_4, k_{13}, k_9 are required; to compute $\mathsf{S}'(I_9^{(5)})$, $\mathsf{S}'(I_9^{(10)})$, $\mathsf{S}'(I_9^{(15)})$, $\mathsf{S}'(I_9^{(4)})$, a different combination of key bytes is required.

The group of those required key bytes is termed as a key byte *tuple*. The order of the bytes in tuples shows their relative positions in the real round key. Using these key bytes tuples, the intermediate states [such as $\mathsf{S}'(I_9^{(\cdot)})$ in (9)] can be computed. When choosing a specific key byte tuple, if the computed state contains the impossible value v, the current key byte tuple is not the correct one due to it violated with the constraints. Thus, the current key byte tuple will be marked as invalid and discarded from the candidate set of key byte tuples. As a result of this stage, the adversary $\mathcal{A}$ obtains four different valid sets of key byte tuples since there are four different combinations of key bytes tuples. These candidates sets will be used to recover the full round key $\boldsymbol{K}_{10}$ in the following section.

E. Key Recovery

As mentioned before, each byte in $\mathsf{S}'(\boldsymbol{I}_9)$ is related to a tuple of key bytes. In Fig. 4, each row of blocks represents the 16 bytes in $\boldsymbol{K}_{10}$. The gray blocks represent the required bytes when computing $\mathsf{S}'(\boldsymbol{I}_9)$ (the target state bytes in Fig. 4). As an example, the computation of four bytes, $\mathsf{S}'(I_9^{(1)})$, $\mathsf{S}'(I_9^{(6)})$, $\mathsf{S}'(I_9^{(11)})$, $\mathsf{S}'(I_9^{(16)})$, requires the corresponding 13 bytes marked in gray at the first row, according to (9).

These tuples can be divided into four classes, as shown in Fig. 4, based on the positions (or index) of the key bytes. Thus, a *divide-and-conquer* strategy can be applied during the key recovery: first divide the tuples into four independent parts $\mathbb{G}_i$ ($1 \le i \le 4$), validate them separately and then merge the results together to build up the full key.

Validate the Tuples: This step is to validate the tuples, which is based on the results of $\mathsf{S}'(I_9^{(i)})$ ($1 \le i \le 16$) where the impossible byte v in S-box output is already known. The validation is derived from the additional constraint equations such as (8b). In this way these invalid tuples will be discarded, which is equivalent to reduce the key search space.

First, the adversary chooses one tuple and compute $\mathsf{S}'(I_9)$. If $\mathsf{S}'(I_9)$ contains the impossible value v, it shows that the corresponding tuple does not satisfy the constraint equations.

Algorithm 2 Key Recovery in Round 9 (AES-128)

```
Input: $\mathbb{D}_i$, $i \in \{1, 2, \cdots, 16\}$;
 1: for i ← 1 to 4 do
 2:     $\mathbb{V}_i \leftarrow \varnothing$
 3:     for all tuple $\in \mathbb{G}_i$ do
 4:         if tuple is valid then
 5:             $\mathbb{V}_i \leftarrow \mathbb{V}_i \cup \{\text{tuple}\}$
 6:         end if
 7:     end for
 8: end for
 9: $\mathbb{K}_{10,1} \leftarrow$ Merge $\mathbb{V}_1$, $\mathbb{V}_2$
10: $\mathbb{K}_{10,2} \leftarrow$ Merge $\mathbb{V}_1$, $\mathbb{V}_3$, $\mathbb{V}_4$
11: $\mathbb{K}_{10,3} \leftarrow$ Merge $\mathbb{V}_2$, $\mathbb{V}_3$, $\mathbb{V}_4$
12: $\mathbb{K}_9 \leftarrow \mathbb{K}_{9,1} \cup \mathbb{K}_{9,2} \cup \mathbb{K}_{9,3}$
13: return $\mathbb{K}_9$;
```

Then the chosen key tuple will be marked as invalid and then discarded. The adversary $\mathcal{A}$ will move to next tuple of key-byte candidates and repeat the validation. When all tuples in $\mathbb{G}_i$ ($1 \le i \le 4$) are validated, $\mathcal{A}$ can get four sets of valid tuples, denoted as $\mathbb{V}_i$ ($1 \le i \le 4$). The elements in these sets are the tuples of key bytes that satisfy the constraint equations.

Merge Tuples: In this step, the full key will be recovered by combining those valid tuples. $\mathcal{A}$ has obtained four sets of valid tuples, which only contain part of bytes in a round key as shown in Fig 4. To build up the full key, these tuples need to be merged together properly.

The principle of such merge is to make sure that every byte is included by at least one tuple. According to Fig. 4, it can be easily found that there are three types of possible combinations of the validated sets to achieve this goal: 1) use tuples in $\mathbb{V}_1$ and $\mathbb{V}_2$; 2) use tuples in $\mathbb{V}_1$, $\mathbb{V}_3$, and $\mathbb{V}_4$; and 3) use tuples in $\mathbb{V}_2$, $\mathbb{V}_3$, and $\mathbb{V}_4$. Since there are intersections between these tuples, such merge may not succeed all the time.

For example, when two tuples are selected from $\mathbb{V}_1$ and $\mathbb{V}_2$, however, they have different values in the k_1 block. These two tuples cannot be merged into one full key because of the conflict. $\mathcal{A}$ only needs to keep and merge these combinations that have no conflict.

After $\mathcal{A}$ has processed all four sets of valid tuples and merged them, he will obtain three sets of recovered keys from the three combinations. The correct full key is in the union of the final sets. Algorithm 2 shows the algorithm used in this step. This merging step further narrows down the key search space by finding conflicts. In the end, the search space for $\mathbb{K}_{10}$ can be very small. The experiments and results are shown in Section VII.

Note that in this section, we only target the S-box implementation. In the real-world AES, there are many other implementations. For example, the OpenSSL 1.1.1a uses four T-tables to accelerate the computation, each table sized 1024 bytes; and Libgcrypt 1.8.3 uses one T-table to generate the S-box used in all rounds in encryption. EPFA is also possible to reveal the secret under these implementations.

However, it requires the adversary to inject the fault into the *equivalent* S-boxes. As for OpenSSL 1.1.1a, for one byte in a state matrix, it may use one of the four T-tables to compute the next byte. This means that if we want to inject a fault to the equivalent S-box, we need to inject at least 1-bit fault to each byte in a specific element(like the first element, sized

32 bits) of each T-table (4×4 bits in total). This capability is much stronger than the original PFA. As for the Libgcrypt 1.8.3, the S-box used in all rounds is generated by the first T-table combined with a cyclic shift operation. Thus, we only need to inject the four bytes in one element in the first T-table (4 bits in total) to have the same effect on the S-box. In such ways, EPFA can then be applied to recover the secert key.

VI. EPFA on Lightweight Block Ciphers

We have successfully applied EPFA on AES-128 using constraint equations in deeper rounds. The enhanced method works efficiently and also reduces the number of required ciphertexts significantly. In this section, we will extend EPFA to other block ciphers.

One category contains those newly proposed lightweight SPN-based block ciphers, such as LED [5] and PRESENT [19]. We will take LED-64 as an example. Compared with AES-128, it has more rounds and smaller block size. Since LED-64 has no key schedule, the round keys are the same as the master key. The detailed analysis is shown in Section VI-A.

Another category worths to explore is those with complex key schedules, where the inverse calculation of the master key from the round keys is not easy. Such SPN ciphers need some computation across the intermediate states in different rounds. The original PFA cannot handle such cases since it only considers the last round. However, EPFA may work well as it initially takes the cross-round computation into consideration. The block cipher SKINNY [6] follows the TWEAKEY framework [20]. An important property of SKINNY is that its round keys in two adjacent rounds are independent, containing the irrelevant information derived from the master key separately. Thus, the fault analysis on SKINNY naturally requires at least two rounds to recover the master key. Even for the round-reduced SKINNY cipher (14 rounds, normally 32 rounds), the cryptanalysis time complexity is still very large ($> 2^{59}$) according to Sadegh Sadeghi's work in 2018 [21].

A. LED Block Cipher

LED is an AES-like lightweight block cipher with SPN structure. It was proposed in 2016 [5] and has an ultralight key schedule. Thus, it can be really fast and has very efficient software or hardware implementations.

In LED, the smallest data unit is sized as one nibble (4-bit). The size of state matrix is the same as AES, 4×4. Thus, the data block is 64-bit. As for the round functions, LED is very similar to AES. It has SubCells, SRs, MixColumnsSerial and AddRoundKey. SubCells is the only nonlinear operation and uses a predefined 4-bit bijective S-box to substitute these nibbles in the state matrix. A significant difference is that LED-64 uses 32 rounds, which is very large compared with AES. The AddRoundKey operation is optional, since LED cipher will only use the round keys per four rounds according to the specification [5]. Compared with AES cipher, LED cipher has an extra step which adds constants in a round. Since the constants is related to the round number and the current round key, we also need to add extra recovery step in our algorithms.

Instantiation: All permutations and substitutions used in these rounds are the same, which means $\mathsf{P}_1 = \mathsf{P}_2 = \cdots = \mathsf{P}_{32}$ and $\mathsf{S}_1 = \mathsf{S}_2 = \cdots = \mathsf{S}_{32}$. We can use the same fault model as what we used in Section V, which is to inject a fault into one cell of S-box. The derivation of the constraint equations is similar to (6) as well. We can write them as

$$f_i(\boldsymbol{I}_i, \boldsymbol{K}_{i-1}) = \left(\mathsf{P}^{-1}(\boldsymbol{I}_i \oplus \boldsymbol{K}_{i-1})\right)^{(j)} \oplus v \neq 0 \tag{10}$$

where $1 \leq i \leq 32$, $1 \leq j \leq 16$.

The key schedule used in LED will pick some specific nibbles in the full key based on key size (described in the specification [5]). Note that under this key schedule, the round keys are the same as the master key in LED-64, where 64 represents the master key size. What's more, LED-64 has 32 rounds and only 8 of them uses the round keys. Thus, the round keys $\boldsymbol{K}_0 = \boldsymbol{K}_4 = \cdots = \boldsymbol{K}_{32}$, and they are equal to the master key $\boldsymbol{K}$. As for other round keys, they can be denoted as $\boldsymbol{K}_{\text{others}} = \boldsymbol{Z}$ where $\boldsymbol{Z}$ is a zero matrix.

Key Recovery in the Last Round: Because of the linear permutation, we have following equations in the last round:

$$\boldsymbol{C} = \mathsf{P}(\mathsf{S}(\boldsymbol{I}_{32})) \oplus \boldsymbol{K} \tag{11}$$

$$\mathsf{P}^{-1}(\boldsymbol{C}) = \mathsf{S}(\boldsymbol{I}_{32}) \oplus \mathsf{P}^{-1}(\boldsymbol{K}). \tag{12}$$

The fault is injected in the S-box, and the faulty substitution will be denoted as S'. From (10) and (12), we can use Algorithm 3 to obtain the candidate sets of the impossible nibbles in $\mathsf{P}^{-1}(\boldsymbol{C})$ as a key-related matrix $\boldsymbol{K}'$, where the ith nibble is the value not shown in $\mathsf{P}^{-1}(\boldsymbol{C})^{(i)}$ for all collected $\boldsymbol{C}$.

Compared with AES, the difference when analyzing LED is at the MixColumnsSerial operation in the round function. If there is no such operation, $\boldsymbol{K}'$ will be exactly the XOR of $\boldsymbol{K}$ and the injected fault, resulting in a very trivial key recovery. However, with the MixColumnsSerial operation, the key can still be recovered due to its linearity. Note that the impossible nibbles in $\boldsymbol{K}'$ are caused by the impossible nibbles v in the faulty S-box. Due to the linearity of MixColumnsSerial operation, we have

$$\boldsymbol{K}' = \mathsf{P}^{-1}(\boldsymbol{K}) \oplus \boldsymbol{F} \tag{13}$$

$$\boldsymbol{K} = \mathsf{P}(\boldsymbol{K}' \oplus \boldsymbol{F}) \tag{14}$$

where $\boldsymbol{F}$ represents a special matrix with all cells having the same faulty value v.

Algorithm 3 shows the pseudo-code to find the candidate sets of nibbles. The inverse permutation operation P^{-1} is used and the candidate sets is made of nibbles which occur in the computation results. After Algorithm 3, 16 sets of impossible values for $\mathsf{P}^{-1}(\boldsymbol{C})$ will be obtained and denoted as $\mathbb{D}_i, 0 \leq i \leq 15$. Recall the key-related matrix $\boldsymbol{K}'$ mentioned in last paragraph, its candidate set can be computed as $\mathbb{K}' = \prod_{i=0,1,\ldots,15}^{i} \mathbb{D}_i$. Then, the adversary $\mathcal{A}$ will get the final candidate set of $\mathbb{K}_{32}$ by using (14). This step is described in line 2–6 in Algorithm 4.

Utilizing More Rounds: The current analysis on $\mathbb{K}_{32}$ only uses the constraint equations of the last round. The search space can be reduced with more constraint equations in the

Algorithm 3 Finding Candidate Sets of Nibbles (LED-64)

```
Input: Set of collected ciphertexts $\mathbb{C}$;
 1: count ← [[0; 16]; 16]
 2: $\mathbb{D}_i \leftarrow \varnothing, i \in \{1, 2, \cdots, 16\}$
 3: for all $\boldsymbol{C} \in \mathbb{C}$ do
 4:     for $i \in \{1, 2, \cdots, 16\}$ do
 5:         count[$i$][$\mathsf{P}^{-1}(\boldsymbol{C}[i])$]+=1;
 6:     end for
 7: end for
 8: for $i \in \{1, 2, \cdots, 16\}$ do
 9:     for $j \in \{1, 2, \cdots, 16\}$ do
10:         if count[$i$][$j$] $\neq 0$ then
11:             $\mathbb{D}_i \leftarrow \cup(\mathbb{D}_i, \{j\})$
12:         end if
13:     end for
14: end for
15: return $\mathbb{D}_i, i \in \{1, 2, \cdots, 16\}$
```

Algorithm 4 Utilizing Constraints in LED-64

```
Input: Ciphertexts $\mathbb{C}$, Candidate Sets $\mathbb{D}_i, i = 1, 2, \cdots, 16$;
 1: $\mathbb{K}' \leftarrow \mathbb{D}_1 \times \mathbb{D}_2 \cdots \times \mathbb{D}_{16}$
 2: $\mathbb{K}_{32} \leftarrow \varnothing$
 3: for all $\boldsymbol{K}' \in \mathbb{K}'$ do
 4:     $\boldsymbol{K} \leftarrow \mathsf{P}(\boldsymbol{K}' \oplus \boldsymbol{F})$
 5:     $\mathbb{K}_{32} \leftarrow \mathbb{K}_{32} \cup \{\boldsymbol{K}\}$
 6: end for
 7: $\mathbb{K}_{31} \leftarrow \mathbb{K}_{32}$
 8: for all $\boldsymbol{K} \in \mathbb{K}_{32}$ do
 9:     for all $\boldsymbol{C} \in \mathbb{C}$ do
10:         $\mathsf{S}'(\boldsymbol{I}_{32}) \leftarrow \mathsf{P}^{-1}(\boldsymbol{C} \oplus \boldsymbol{K})$
11:         if $\mathsf{S}'(\boldsymbol{I}_{32})$ contains $u$ then
12:             skip to next ciphertext
13:         end if
14:         $\mathsf{S}'(\boldsymbol{I}_{31}) \leftarrow \mathsf{P}^{-1}(\boldsymbol{I}_{32})$
15:         if $\mathsf{S}'(\boldsymbol{I}_{31})$ contains $v$ then
16:             $\mathbb{K}_{31} \leftarrow \mathbb{K}_{31}/\{\boldsymbol{K}\}$
17:             skip to next key candidate
18:         end if
19:     end for
20: end for
21: return $\mathbb{K}_{31}$;
```

penultimate round. Note that in LED, the round key is XORed with the state matrix per four rounds. In our general model described in Section II, $\boldsymbol{K}_{31}$ is used in round 31 at high levels. Since there is no actual key addition in the round 31 of LED, $\boldsymbol{K}_{31}$ can be equivalently represented as a zero matrix

$$\begin{aligned}\mathsf{S}(\boldsymbol{I}_{31}) &= \mathsf{P}^{-1}(\boldsymbol{I}_{32} \oplus \boldsymbol{K}_{\text{others}}) \\ &= \mathsf{P}^{-1}(\boldsymbol{I}_{32}) \\ \boldsymbol{I}_{31} &= \mathsf{S}^{-1}(\mathsf{P}^{-1}(\boldsymbol{I}_{32})).\end{aligned}$$

Note that the inverse SUBCELLS $\mathsf{S}^{-1}(\cdot)$ is used here. Due to the fault in S-box, the faulty value v will never appear in the output. Meanwhile, another value u will have two preimages. Thus, if the computed intermediate state $\boldsymbol{I}_{31}$ has value u, it cannot be distinguished whether the preimage corresponds to v or not. Therefore, this ciphertext has to be discarded, and $\mathcal{A}$ can move to next one. This filtering step is very similar to that aforementioned in (8b).

The overall algorithm used in this step is shown in Algorithm 4. As an example, Algorithm 4 only utilizes one additional round (Round 31). Line 4–12 in Algorithm 4 can be repeated to utilize more constraint equations. In this way, the key search space can be narrowed down round by round. The experimental results for attacking LED are shown in Section VII.

B. SKINNY Block Cipher

SKINNY is another family of lightweight block ciphers proposed in 2011 [6]. It supports multiple key lengths and block sizes to meet requirements for different scenarios and adjustable security levels. We will apply EPFA to the cipher SKINNY-64-64. The suffix "-64-64" indicates that both the block size and the key size are 64-bit.

The round functions in SKINNY are similar to LED, including SUBCELLS, MIXCOLUMNS, ADDCONSTANTS, ADDROUNDTWEAKEY and SRS. This attack is also based on a fault injected to the S-box of nibbles. Note that the key-related operation ADDROUNDTWEAKEY is appeared in the middle of a round. To fit in the general model we proposed in Section II, we can add some precomputations and post-computations to shift the round operations properly.

However, when comparing with AES and LED, the key schedule in SKINNY is very different. SKINNY follows the TWEAKEY framework [20], takes a tweakey input instead of a key or a pair of key/tweak [6]. In SKINNY-64-64, each round key only uses half of the master key. Although the master key has 16 cells (4 × 4), the round key only picks 8 of them and sets other cells to 0. Note that (4) in Section II is closely related to the key schedule. Thus, the derivation of the round tweakeys should be discussed in details.

Key Schedule: At first, a tweakey array t sized 16 nibbles will be initialized with the 64-bit master key. Then, the first round key $\boldsymbol{K}_1$ will be formed like (15). In next round, a permutation P_T will be applied to the tweakey array: for all $1 \leq i \leq 16$, we set $t_i \leftarrow t_{P_T[i]}$ with $P_T = [10, 16, 9, 14, 11, 15, 13, 12, 1, 2, 3, 4, 5, 6, 7, 8]$.

Then the round key $\boldsymbol{K}_2$ will be constructed in the same way as $\boldsymbol{K}_1$ in (15). Thus, the $\boldsymbol{K}_2$ can be denoted as the following using the *initial* tweakey array:

$$\boldsymbol{K}_1 = \begin{bmatrix} t_1 & t_2 & t_3 & t_4 \\ t_5 & t_6 & t_7 & t_8 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \boldsymbol{K}_2 = \begin{bmatrix} t_{10} & t_{16} & t_9 & t_{14} \\ t_{11} & t_{15} & t_{13} & t_{12} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \quad (15)$$

Note that in the permutation, those cells t_i $(1 \leq i \leq 8)$ will be filled with the second half of the tweakey array. Thus, $\boldsymbol{K}_2$ is formed by other 8 nibbles which are irrelevant with $\boldsymbol{K}_1$. Each round key contains only *half* of the master key, therefore the full recovery requires an analysis of at least two rounds theoretically.

Key Recovery in Last Two Rounds: Since both round 32 and round 31 use independent round keys, we will first recover $\boldsymbol{K}_{31}$ and $\boldsymbol{K}_{32}$ separately, and then combine them together to get the candidate set of the master key.

Although the key schedule is different, the cipher structure is similar. We still have $\mathsf{S}(\boldsymbol{I}_{32}) = \mathsf{P}_{32}^{-1}(\boldsymbol{C} \oplus \boldsymbol{K}_{32})$. In the last round, applying the similar method shown in Algorithm 3 cannot recover $\boldsymbol{K}_{32}$ since it only contains 8 cells related to the master key. One more round is required in order to get the key candidate set $\mathbb{K}$. We have $\mathsf{S}(\boldsymbol{I}_{31}) = \mathsf{P}_{32}^{-1}(\boldsymbol{I}_{32} \oplus \boldsymbol{K}_{31})$. In this way, we can get two candidate sets $\mathbb{K}'_{32}$ and $\mathbb{K}'_{31}$.

Algorithm 5 Using Round 30 Constraint Equations (SKINNY)

```
Input: Ciphertexts $\mathbb{C}$, key candidate set $\mathbb{K}_{31}$;
1: $\mathbb{K}_{30} \leftarrow \mathbb{K}_{31}$
2: for all $K \in \mathbb{K}_{32}$ do
3:     Generate three round keys from the current guessed master key $K$.
4:     for all $C \in \mathbb{C}$ do
5:         $\mathsf{S}'(I_{32}) \leftarrow \mathsf{P}_{32}^{-1}(C \oplus K_{32})$
6:         if $S(I_{32})$ contains $u$ then
7:             skip to next ciphertext
8:         end if
9:         $\mathsf{S}'(I_{31}) \leftarrow \mathsf{P}_{31}^{-1}(I_{32} \oplus K_{31})$
10:        if $S(I_{31})$ contains $u$ then
11:            skip to next ciphertext
12:        end if
13:    end for
14: end for
15: return $\mathbb{K}_{30}$;
```

Utilizing More Rounds: Since we already have $\mathbb{K}'_{32}$ and $\mathbb{K}'_{31}$, we can invert the key schedule step in round 31, and recover the 31th tweakey array to get the candidate set $\mathbb{K}_{31}$.

For each candidate in $\mathbb{K}_{31}$, it satisfies the constraint equations in the last two rounds. We can utilize constraints from other rounds to further restrict the key search space. The algorithm using round 32, 31 and 30 is shown in Algorithm 5. Note that Line 9–12 can be repeated multiple times to use more constraint equations. The experimental results for SKINNY-64-64 are shown in Section VII.

VII. Experiments and Results

In this section, we will introduce our setups and designs of the experiments. The results of EPFA on three block ciphers, AES-128, LED-64 and SKINNY-64-64, are also presented.

A. Setup

In all three cases, the faults are injected by changing the value of one cell in the corresponding S-box. We implement the experiments at the software level on a computer which has 16-GB memory and an Intel i7-8700 CPU at 3.20 GHz.

The experiments generally follow the procedures below. First, many ciphertexts are generated to simulate the victim's encryption process. This step is done by encrypting random plaintexts using the faulty S-box. Second, the adversary $\mathcal{A}$ needs to collect the ciphertexts. A certain number of ciphertexts are randomly selected from the pregenerated ciphertexts. Then, EPFA is performed to recover the master key (or round key). Note that sometimes it may have more than one candidate key in the result. We use the *key residue entropy* to represent the size of the resultant key search space.

To retrieve the statistical results, we perform the experiments multiple times for each cipher and compute the average key residue entropy as the final results.

B. Attack on AES-128

Compared with the Round-10-only analysis in the original work [4], constraint equations in Round 9 have a significant effect on reducing the key search space. Fig. 5(a) shows that the original PFA cannot completely recover the key when the number of collected ciphertexts is less than 2500. In contrast, by using the leakages in Round 9, the limit can be pushed to be under 1000. In other words, with more than 1000 ciphertexts, the proposed EPFA can extract the full key with almost 100% success rate and zero residue entropy. The performance improvement is about 60% reduction on the number of required ciphertexts. Which is to say, the adversary has to collect 40kb encrypted data to perform the analysis using original PFA; while using EPFA, he may use only 16kb data to get the same result.

However, more constraints indicates more computations at the same time. In our experiments, the time consumption of the repeating experiments will not be affordable when the number of ciphertexts is smaller than 1000. It will take more than one day to perform an experiment and collect the statistics data. Thus, a parallel version of EPFA is proposed, which will be detailed in Section VIII.

C. Attack on LED-64

In LED-64, the elements in S-box is 4-bit. The target key can be recovered with a few ciphertexts due to the small size of S-box and the simple key schedule used in LED. Similar to AES, we perform the EPFA analysis on LED for 100 times, and compute the average key residue entropy.

Fig. 5(b) shows that: if the adversary $\mathcal{A}$ collects more than 75 ciphertexts, he can always recover the 64-bit master key when applying five rounds analysis using EPFA. While, the original PFA cannot recover the key even if $\mathcal{A}$ has more than 100 ciphertexts.

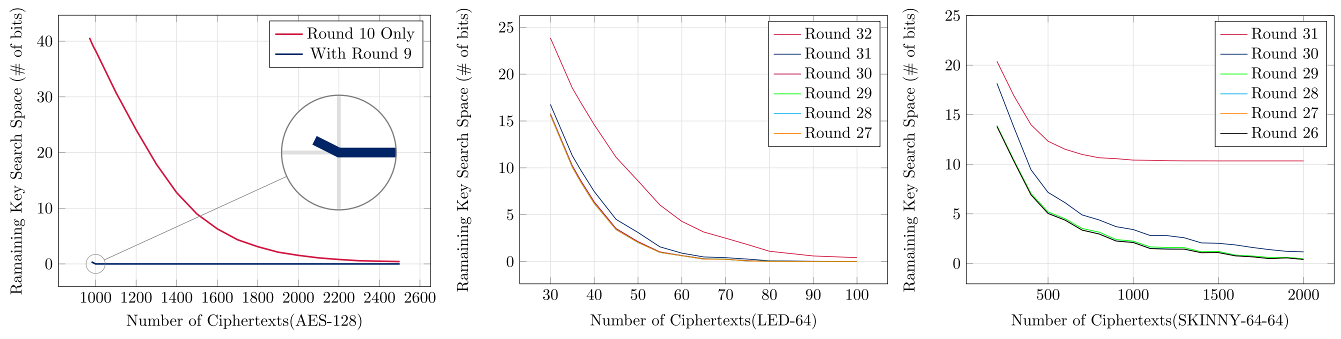
Another observation is that: as shown in Fig. 5(b), the newly added constraints will provide less capability to reduce the key search space when the analysis round is getting deeper. For example, the total key search space is 64 bits. When $\mathcal{A}$ has 30 ciphertexts, an analysis till round 32 gives a resultant search space sized 22 bits. It reduces 42 bits ($64 - 22 = 42$ bits), equivalently. In comparison, the analysis till round 31 only reduces 8 bits ($22 - 14 = 8$ bits). The analysis on round 30 is getting more limited, reducing less than 1 bit from round 29 as shown in Fig. 5(b). We will discuss the possible explanations in Section VII-E.

D. Attack on SKINNY-64-64

In SKINNY-64-64, since the recovery of full tweakey requires at least two rounds, we start our analysis from the ante-penultimate round (Round 31).

Compared with LED-64, it's more difficult to recover the 64-bit key of SKINNY because of the complex key schedule. Fig. 5(c) shows that the adversary $\mathcal{A}$ nearly needs more than 1200 ciphertexts to reduce the key residue entropy to be about 2 bits (4 candidates). As for LED-64, the number of ciphertexts that are required is only 75.

Fig. 5(c) shows that when EPFA is merely applied in Round 31 and 32 of SKINNY-64-64, the entropy of search space can only be reduced to about 10 bits. The red curve (the top one in Fig. 5(c)) is approaching to 10.34 bits on average. Note that even with more ciphertexts, such analysis has a lower-bound limit about 10 bits. However, once when the analysis is pushed



(a) (b) (c)

Fig. 5. EPFA experimental results. EPFA is applied on AES-128, LED-64, and SKINNY-64-64 multiple times and then the average results are collected. (a) EPFA on AES-128. (b) EPFA on LED-64. (c) EPFA on SKINNY-64-64.

TABLE I
EPFA on Different Ciphers

	PFA		EPFA	
	$\mathcal{H} \leq 1$ bit*	N	$\mathcal{H} \leq 1$ bit*	N
AES	Yes	2400 – 2500	Yes	<1000
LED	Yes	90 – 100	Yes	50 – 55
SKINNY	No	\	Yes	1500 – 1600

*: $\mathcal{H}$ is the residue key entropy. N is the number of traces.

to Round 30, the key search space will be reduced to be less than 3 bits. Further, when the analysis steps into Round 29, the residue key entropy is only about 0.5 bit. These observations show that the newly introduced constrain equations in deep rounds (such as Round 29-31) are quite effective and significant in reducing the key search space on the SKINNY block cipher.

Note that in Fig. 5(c), other experimental curves for Round 26 to 28 are highly overlapped with that for Round 29, even when the number of ciphertexts is increased from 500 to 2000. It implies the limit for such fault analysis and there is no need to conduct further analysis beyond Round 29, which is similar to the situations in LED experiments.

E. *Effect and the Limit of EPFA*

Table I shows that with EPFA can effectively reduce the number of required ciphertexts: when both attacks having the same residue entropy, in AES-128, it reduces $(2500 - 1000)/2500 = 60\%$ ciphertexts; and in LED case, it reduces $(100 - 50)/100 = 50\%$ ciphertexts. What's more, it can work on SKINNY that PFA cannot handle.

As aforementioned, there exists an analysis limit when EPFA is applied towards the deeper rounds. It is mainly due to the filtering step in Algorithms 4 and 5. This step discards some ciphertexts that cannot be used to infer the previous state round by round. For this reason, the accumulated leakage by these ciphertexts will be reduced gradually.

Another possible reason is the change of the distribution for those intermediate states. In the last round, the ciphertexts are uniformly distributed. With the filtering step and the inversed round functions, the distribution will be changed round by round. This change may decrease the effect of EPFA, which actually relies on the statistical property of the ciphertexts.

VIII. Parallelization and GPU Acceleration

In this section, we will apply the parallelization to improve the efficiency of some bottleneck parts in our analysis, which can push the limit of our EPFA further. Such acceleration is independent to specific cipher and key, thus therefore can be applied to all attacks discussed in previous sections.

A. *Feasibility and Necessity*

When the adversary $\mathcal{A}$ gets the candidate set of key bytes in the last round, the size of such candidate set is actually very large. For example in AES-128, the residue entropy can be nearly 50 bits in our experiments. It will require nearly $2^{50} \times 16$ bytes $\approx 1.68 \times 10^7$ GB space so it is infeasible if all the results are stored statically. Thus, the Cartesian product needs to be generated dynamically. However, the iteration on the Cartesian product still requires lots of computation resources, which brings certain of challenge in practice. Fortunately, the design of our analysis algorithms can be transplanted to the scenario of GPU, using parallelization to accelerate the iteration.

Note that the four groups of Cartesian product mentioned earlier are independent. Meanwhile, the iteration of the key tuples is also independent to each other, which means processing one tuple has no influence to others. Thus, two levels of parallelization can be conducted.

The first level parallelization is to perform all iterations on the four groups simultaneously. Any iteration in one group will be independent of the one in other groups, both of which can be executed in parallel. This enhancement is quite straightforward and can be done with very little efforts, which will not be discussed in details. A more challenging parallelization at a higher level is to traverse the generated Cartesian product, which will be discussed later in detail.

B. *Vanilla Parallelization*

In this section, how to parallelize a classic traversing of the dynamically generated Cartesian product will be elaborated.

Bottleneck of Iterations: As mentioned before, the bottleneck of the efficiency is at the traversing of the Cartesian product. As for a *static* array, a common way to accelerate its traversing is to partition the iteration task into multiple disjoint ones. For example, we can set multiple iterators at the beginning which have different initial values and the same step size. Each iterator can be put into a standalone thread. In this way we can achieve the parallelization of the traversing.

Naive Acceleration: Unfortunately, the idea above cannot be applied to the traversing of Cartesian product directly. This is because the elements are generated *dynamically*, and the step size is not fixed which cannot be determined in advance. A simple solution is to use many "for" loops. In the AES-128 case, there are 10 or 13 candidate sets. Thus, 10 or 13 "for" loops are required to implement. The traversing can be accelerated by assigning some inner loops to different GPU cores. For example, if the last candidate set contains 40 elements, the corresponding loop can be rewritten into small tasks to be executed in 40 different threads.

Disadvantages: This naive acceleration is easy to understand, however, it is not flexible and cannot fully utilize the power of GPU. For an ordinary array, the elements are stored statically and the step size can be set arbitrarily. However, as for a dynamically generated elements in Cartesian product, the step size is fixed to some numbers because it is highly related to the size of inner loops. Moreover, in different analyses, the size of candidate sets will be different and the task partition needs to be customized and reimplemented. Thus, a new strategy needs to be raised to implement the parallel algorithm.

C. Our Improvement

Our improved solution is to use an idea of mixed radix representation to bridge the index and the corresponding dynamic tuple of key bytes.

Recall the limit of the naive parallel traversing for a dynamically generated Cartesian product: the step size cannot be set arbitrarily. A straightforward idea to achieve this is to provide a conversion between the tuple and the corresponding index. It can be supported by the *mixed radix numbers* whose digits have different radices. For example, we consider tuples having 10 key bytes. Each tuple of the key bytes can be represented as $(\mathbb{D}_1[a_1], \mathbb{D}_2[a_2], \ldots, \mathbb{D}_{10}[a_{10}])$ where a_i is the index of the value in ith candidate set. The mixed radix representation of this tuple can be denoted as $\overline{a_1 a_2 \cdots a_{10}}$, and the radix for digit a_i is the size of $\mathbb{D}_i$ (denoted as s_i). From the mixed radix representation, the corresponding index is

$$\begin{aligned}\overline{a_1 a_2 \cdots a_{10}} &= \sum_{i=1}^{10} \left(a_i \times \prod_{k=i+1}^{10} s_k \right) \\ &= a_{10} + a_9 \times s_{10} + \cdots + a_1 \times s_{10} \times \cdots \times s_2.\end{aligned}$$

In another direction, the mixed radix representation can also be computed from an index. This is an intuitive conversion between an index and a tuple of key bytes. In real world implementations, the values of each candidate set are actually stored in memory. To get the values for a key tuple efficiently, we can compute their addresses using the mixed radix representation rather than copy them from one place to another.

This indexing method has several benefits: 1) it provides direct accesses to the data with little efforts since the pointers can be easily computed from the index; 2) the configuration of the step size is very flexibly which can fully utilize the power of GPU; and 3) there is almost no extra memory space requirements and little performance penalty.

As a result, we can set the step size to match with the thread number and perform many iterations simultaneously. In each iteration, the index number needs to be initialized first. Then it will be converted into the mixed radix representation in order to generate the corresponding key byte tuple. The final step is to validate the tuple and update the index.

TABLE II
EXPERIMENT RESULTS WITHOUT THRESHOLD

# of C	CPU			GPU			$\mathcal{H}$	Speedup
	$t_{\min}$	$t_{\max}$	t_{avg}	$t_{\min}$	$t_{\max}$	t_{avg}		
1500	0.01	0.95	0.06	0.02	0.00	0.03	0	2.0
1400	0.02	8.33	0.54	0.02	0.07	0.03	0	18.0
1300	0.10	85.57	5.65	0.02	0.56	0.05	0	113.0
1200	1.85	1236.94	96.95	0.02	6.67	0.54	0	179.5
1100	66.12	36392.09	3559.89	0.42	190.95	17.93	0	198.54

Note: $\mathcal{H}$ is the average residue key entropy in bits.

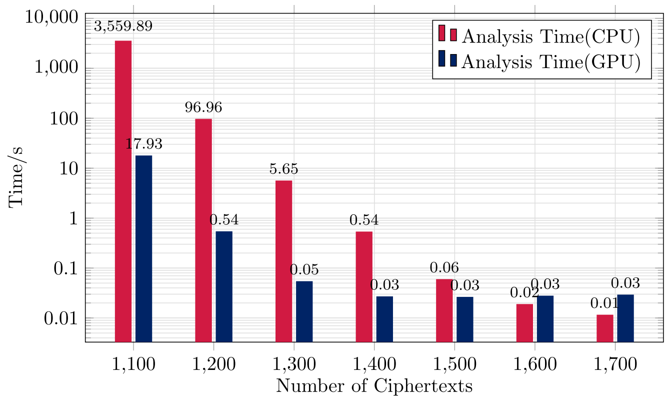


Fig. 6. EPFA performance comparison between CPU and GPU.

D. Parallelization Experiment Results

The parallel version of EPFA is implemented and applied to AES-128. The machine has an NVIDIA GP102 GPU, and the CPU is Intel Xeon E5-2678 v3 at 2.50 GHz. Since the analysis time depends on the ciphertexts, repeated experiments are conducted to get the statistics such as the average. The ciphertexts are randomly generated using a bunch of seeds. As for the experiments run on GPU or CPU only, the same seed is used to ensure the results is comparable.

In the first part of the experiments, EPFA is applied on the collections of ciphertexts for multiple times. Then, the average analysis time and key residue entropy for GPU and CPU version are recorded. The speedup of GPU for the average time consumed is shown in Table II. From this table, we can see that with fewer ciphertexts, the parallel version has a more significant improvement in efficiency compared with the ordinary algorithm. When the number of ciphertexts is about 1500, the speedup of the GPU is only 2. However, with 1100 ciphertexts, the speedup grows to nearly 200 times. We also illustrate the performance comparison in Fig. 6 to depict this issue. Note that the y-axis is in the logarithm mode. When the number

TABLE III
EXPERIMENTS WITH THRESHOLD

# of C	$t_{\min}$	$t_{\max}$	t_{avg}	$\mathcal{H}$	s
1100*	66.12	5599.05	1493.79	0	88%
1100	0.42	190.95	17.93	0	100%
1000	31.99	4411.62	750.48	0	92%
990	37.77	6586.22	1150.49	0.18	90%
980	39.44	6623.99	1750.25	0.34	87%
970	68.03	6496.65	2434.82	0.98	83%

Note: 1100* runs on CPU only, and others run on GPU.

of ciphertexts is large, e.g., 1700, the parallel version is even *slower* than the CPU-based one.

This is because in this case, the bottleneck (if any) of the execution progress is at the memory operations (copy and allocation) but not actual computations. CPU and GPU do not have shared memory by default. To initialize a computation in GPU, data must be transferred from host (CPU) memory to device (GPU) memory. Thus, the parallel version takes more time when the computation part does not take much time. However, when it really encounters the computation bottleneck, the parallelization will be very effective.

Although the effect of acceleration is very significant, Fig. 6 shows that analysis time will grow exponentially when we lower the number of ciphertexts. The volatile GPU utilization metric provided by `nvidia-smi` shows that our program achieved 100% utilization, which means that with a fine-tuned implementation, we fully utilized the power of GPU.

As the second part in our experiments, we performed multiple trials with time-outs as mentioned before. A time threshold $\tau = 7200$ s is set for all experiments. When the analysis of one experiment takes more than τ, it will be regarded as a failed case. The success rate and other metrics are listed in Table III. Note that the seeds used to generate the ciphertext collections are the same as that in the first part of experiments. Thus, the minimum value of analysis time for the 1000* case in Table II (66.12 s) is identical to the results in Table III. Other results are different because of the threshold. It shows that 88% experiments running on CPU (single-threaded) recover the round key successfully, while others exceed the timeout τ when the number of ciphertexts is 1100. By comparison, all the experiments with same number of ciphertexts (1100) running on GPU recover the round keys under the time threshold τ. It also shows that with fewer ciphertexts, the success rate starts to decrease. For example, with 1000 ciphertexts, the success rate is 92%. With 970 ciphertexts only 83% experiments recover the round keys successfully. It will become more infeasible if we continue decrease the number of ciphertexts. The infeasible limit cannot be eliminated by parallelization.

E. Analysis Time Distribution

The distribution of the analysis time under certain number of ciphertexts is also interesting.

Fig. 7 shows the histogram chart of the analysis time with/without GPU accelerations, when 1100 ciphertexts are used. As for experiments running on GPU, most of the attacks can be done in relatively little time (under 20 s). The ratio of such attacks is about 80% (i.e., 60 + 20 = 80% for the first two bars). But there are still some attacks requiring much more time, i.e., greater than 180 s. As for CPU case, the situation is similar: the analysis time for 80% of attacks is less than 3600 s, and some analysis time can be up to 36000 s. Table III shows that the analysis time diverges in a wide range. For example, when initiated with 990 ciphertexts, the analysis time ranges from 41 to 6581.42 s.

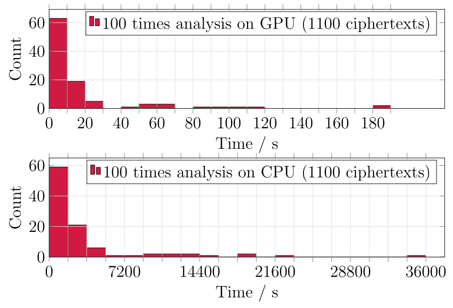


Fig. 7. Histogram count of analysis time.

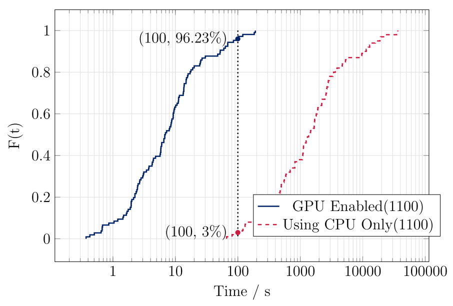


Fig. 8. Empirical CDF of analysis time.

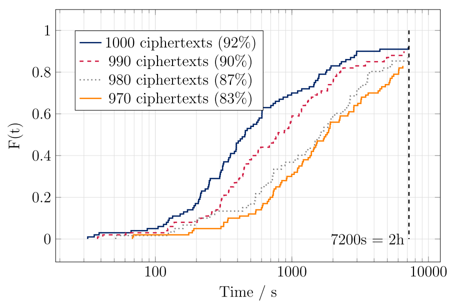


Fig. 9. Partial CDF of analysis time.

From a different point of view, the performance of CPU-only EPFA and GPU-enabled ones can be compared via the cumulative distribution function (CDF) as shown in Fig. 8. It plots the empirical CDF of the experiments when the number of ciphertexts is 1100. The y-axis represents the CDF function value that up to 1. Fig. 8 shows that: for more than 96% of experiments when GPU is enabled, the analysis time is less than 100 s, while for the rest 3% of experiments which only uses CPU, the analysis cannot complete within 100 s. The speedup provided by the acceleration is significant when the number of ciphertexts is small.

Since the results from experiments whose running time exceeds the threshold are not collected, the complete CDF figures cannot be plotted for these experiments. However, we can normalize the attack time distribution we have collected and then plot the *partial* empirical CDF figure as Fig. 9. The dashed vertical line on the right in Fig. 9 represents the time threshold τ where $\tau = 7200$ s. Because of the normalization, the CDF value for each curve finally approximate to the corresponding success rate when the time is increased to τ.

In general, EPFA is efficient and very practical. However, it may take more time to finish the analysis in some cases. Moreover, the average analysis time will grow exponentially as the number of ciphertexts decreases. Hence, collecting enough ciphertexts is still important in EPFA.

IX. Conclusion

In this article, we presented the EPFA, which will utilize more fault leakages deducted from deeper rounds. EPFA requires fewer ciphertexts and can handle some ciphers whose key recovery needs cross-round computation (e.g., SKINNY).

We explore several classical and modern block ciphers and apply EPFA successfully on them. In the experiment evaluation part, we compare the performance of EPFA. We also discuss the limit of EPFA and the possible reasons for it. In addition, we propose a fine-tuned parallel version of our EPFA to perform the attack more efficiently based on an idea of mixed radix number.

References

[1] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Proc. 17th Annu. Int. Adv. Cryptol. Conf. (CRYPTO)*, Santa Barbara, CA, USA, Aug. 1997, pp. 513–525.

[2] T. Fuhr, É. Jaulmes, V. Lomné, and A. Thillard, "Fault attacks on AES with faulty ciphertexts only," in *Proc. Workshop Fault Diagn. Tolerance Cryptogr.*, 2013, pp. 108–118.

[3] P. Dusart, G. Letourneux, and O. Vivolo, "Differential fault analysis on AES," in *Proc. 1st Int. Conf. Appl. Cryptogr. Netw. Security (ACNS)*, Kunming, China, Oct. 2003, pp. 293–306.

[4] F. Zhang *et al.*, "Persistent fault analysis on block ciphers," *IACR Trans. Cryptogr. Hardw. Embedded Syst.*, vol. 2018, no. 3, pp. 150–172, 2018.

[5] J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw, "The LED block cipher," in *Proc. 13th Int. Workshop Cryptogr. Hardw. Embedded Syst. (CHES)*, 2011, pp. 326–341.

[6] C. Beierle *et al.*, "The SKINNY family of block ciphers and its low-latency variant MANTIS," in *Proc. 36th Annu. Int. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, 2016, pp. 123–153.

[7] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proc. IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006.

[8] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Proc. 5th IFIP WG Int. Workshop Inf. Security Theory Pract. Security Privacy Mobile Devices Wireless Commun.*, 2011, pp. 224–233.

[9] S. R. Fluhrer and D. A. McGrew, "Statistical analysis of the alleged RC$_4$ keystream generator," in *Proc. 7th Int. Workshop Fast Softw. Encryption*, 2000, pp. 19–30.

[10] C. Dobraunig, M. Eichlseder, T. Korak, V. Lomné, and F. Mendel, "Statistical fault attacks on nonce-based authenticated encryption schemes," in *Proc. 22nd Int. Conf. the Theory Appl. Cryptol. Inf. Security (ASIACRYPT)*, 2016, pp. 369–395.

[11] C. Clavier and A. Wurcker, "Reverse engineering of a secret aes-like cipher by ineffective fault analysis," in *Proc. Workshop Fault Diagn. Tolerance Cryptogr.*, 2013, pp. 119–128.

[12] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "SIFA: Exploiting ineffective fault inductions on symmetric cryptography," *IACR Trans. Cryptogr. Hardw. Embedded Syst.*, vol. 2018, no. 3, pp. 547–572, 2018.

[13] X. Zhao, S. Guo, F. Zhang, T. Wang, Z. Shi, and K. Ji, "Algebraic differential fault attacks on LED using a single fault injection," *IACR Cryptol. ePrint Archive*, vol. 2012, p. 347, Jun. 2012.

[14] F. Zhang, X. Zhao, S. Guo, T. Wang, and Z. Shi, "Improved algebraic fault analysis: A case study on piccolo and applications to other lightweight block ciphers," in *Proc. 4th Int. Workshop Construct. Side Channel Anal. Secure Design*, 2013, pp. 62–79.

[15] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta, "Fault sensitivity analysis," in *Proc. 12th Int. Workshop Cryptogr. Hardw. Embedded Syst. (CHES)*, 2010, pp. 320–334.

[16] G. Piret and J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," in *Proc. 5th Int. Workshop Cryptogr. Hardw. Embedded Syst. (CHES)*, 2003, pp. 77–88.

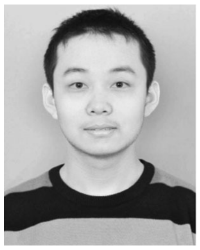
[17] C. Dobraunig, M. Eichlseder, H. Groß, S. Mangard, F. Mendel, and R. Primas, "Statistical ineffective fault attacks on masked AES with fault countermeasures," in *Proc. 24th Int. Conf. Theory Appl. Cryptol. Inf. Security (ASIACRYPT)*, 2018, pp. 315–342.

[18] *Advanced Encryption Standard*, FIPS Standard 197, 2001, p. 51.

[19] A. Bogdanov *et al.*, "PRESENT: An ultra-lightweight block cipher," in *Proc. 9th Int. Workshop Cryptogr. Hardw. Embedded Syst. (CHES)*, 2007, pp. 450–466.

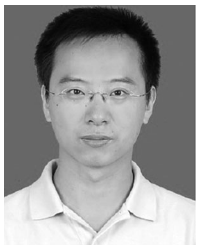
[20] J. Jean, I. Nikolic, and T. Peyrin, "Tweaks and keys for block ciphers: The TWEAKEY framework," in *Proc. 20th Int. Conf. Theory Appl. Cryptol. Inf. Security (ASIACRYPT)*, 2014, pp. 274–288.

[21] S. Sadeghi, T. Mohammadi, and N. Bagheri, "Cryptanalysis of reduced round SKINNY block cipher," *IACR Trans. Symmetric Cryptol.*, vol. 2018, no. 3, pp. 124–162, 2018.



Guorui Xu (Student Member, IEEE) received the bachelor's degree in information engineering from the Zhejiang University, Hangzhou, China, in 2019. He is currently pursuing the Ph.D. degree with the School of Cyber Science and Technology, Zhejiang University.

He is also with the State Key Laboratory of Cryptology, Beijing, China, and with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou. His current research focus is on cryptography and formal methods.



Fan Zhang (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science and Engineering, University of Connecticut, Mansfield, CT, USA, in 2001, 2004, and 2011, respectively.

He is currently a Full Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, and also with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou. His research interests include system security, hardware security, cryptography, and computer architecture.

Bolin Yang (Student Member, IEEE) received the bachelor's degree from the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China, in 2019, where he is currently pursuing the Doctoral degree.

His research interests include hardware security and cryptography.

Xinjie Zhao was born in 1986. He received the B.S., M.S., and Ph.D. degrees from Ordnance Engineering College, Shijiazhuang, China, in 2006, 2009, and 2012, respectively.

He is currently with the Institute of North Electronic Equipment, Beijing, China. His main research interests include side channel analysis, fault analysis, and combined analysis in cryptography.

Dr. Zhao won the best paper award in COSADE 2012 and the Outstanding Doctoral Dissertation Award in the province of Hebei.

Wei He received the bachelor's and master's degrees from Beijing Jiaotong University, Beijing, China, in 2009 and 2006, respectively, and the Ph.D. degree from the Universidad Politecnica de Madrid, Madrid, Spain, in 2014.

He was a Research Scientist with Nanyang Technological University, Singapore, from 2014 to 2016. He has been worked as a Senior Researcher with Huawei International, Singapore, since January 2019. He is currently leading the Blockchain Research Institute, China Telecom Bestpay Company, Ltd., Shanghai, China. His main research topics include hardware design, crypto security, and blockchain.

Kui Ren (Fellow, IEEE) received the B.S., M.S., and Ph.D. degrees from the Worcester Polytechnic Institute, Worcester, MA, USA, in 1998, 2001, and 2007, respectively.

He is currently a Professor of Computer Science and Technology and the Director of the Institute of Cyberspace Research, Zhejiang University, Hangzhou, China. His current research interests span cloud and outsourcing security, wireless and wearable system security, and artificial intelligence security.

Prof. Ren was a recipient of the IEEE CISTC Technical Recognition Award 2017 and the NSF CAREER Award in 2011. He is a Fellow of ACM.