IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 26, 2021, accepted April 12, 2021, date of publication April 21, 2021, date of current version May 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3074708

Persistent Fault Analysis Against SM4 Implementations in Libraries Crypto++ and GMSSL

QING GUO, ZHENHAN KE, SIYUAN WANG, AND SHIHUI ZHENG
School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China
Corresponding author: Shihui Zheng (shihuizh@bupt.edu.cn)

This work was supported in part by the National Science Foundation of China under Grant 61502048, in part by the National Science and Technology Major Project under Grant 2017YFB0803001, and in part by the Program of China Scholarships Council under Grant 201906475014.

ABSTRACT Compared to the injection of a transient fault, time synchronization and accuracy are not required for the injection process of a persistent fault. However, the known *persistent fault analyses* (PFAs) do not work on SM4 implementations because the linear transformation layer hides the position where an error occurs during the encryption process. We present the first *persistent fault analysis* against SM4 implemented with an S-box by combining the inverse linear transformation with differential techniques. In addition, we propose a locating algorithm to figure out not only where an error occurs during the encryption process but also where a fault is inserted in the lookup table. Consequently, the locating algorithm helps break SM4 implemented with a T-table. We validate our PFA on two open-source implementations of SM4 – Crypto++(v8.3) and GMSSL(v1.0.0). The experiments are performed on a PC and the analysis codes are written in C language. The experimental data shows that the probability of successfully recovering the encryption key approximates 1 when the number of normal-and-faulty-ciphertext pairs is 3000 on average. Namely, PFA can break the encryption system of SM4 in practice once valid faults are inserted. Finally, we apply the attack to protected SM4 implementations and prove that the E-and-D mode of the *dual modular temporal redundancy* (DMTR) can defeat our PFA.

INDEX TERMS SM4, persistent fault analysis, fault attack, Crypto++, GMSSL.

I. INTRODUCTION

Sm4 is designed to support the Chinese national standard for wireless LAN WAPI (*wired authentication and privacy infrastructure*) and is officially published by China as the commercial block cipher standard in 2012 [1]. A comprehensive analyses of SM4 have been proposed, such as such as linear attacks [2], [3], differential attacks [4], [5] and cache attack [6].

Fault analysis was firstly proposed by Boneh *et al.* [7] in 1996 and they pointed out that sensitive information may be leaked by accidentally or intentionally injecting faults during the execution of a cryptographic algorithm. One year later, Biham and Shamir [8] proposed a *differential fault analysis* (DFA) against the *Data Encryption Standard* (DES). They exploited the faults induced in the 14^{th},15^{th}, and 16^{th} rounds separately to disclose the encryption key of DES. Many DFAs against SM4 implementations have been published as well [9]–[11]. Zhang and Wu [9] firstly described a DFA against SM4 in 2006. A single-byte fault is injected at the intermediate state just before the last four rounds, and theoretically at least 32 times of fault injection are required to obtain the full encryption key. In 2008, byte-oriented faults are independently inserted in eight executions of the key schedule and then the full 128-bit key of SM4 is disclosed in [10]. In 2011, Li *et al.* [11] employed only one fault inside the intermediate state just before the 28^{th} round to extract partial key bits and the remaining unknown 22.11-bit key (on average) was recovered using a brute-force search.

The above fault analyses are based on transient faults which require accurate time synchronization to precisely trigger a fault at a particular position of the intermediate state during the encryption, such as flipping bits of the input state in the 29^{th} round during the encryption process of SM4 [9].

The associate editor coordinating the review of this manuscript and approving it for publication was Fangfei Li.

However, a persistent fault can be generally injected before the encryption of any plaintext so time precision is not necessary anymore. Zhang *et al*. [12] proposed a PFA against AES with a single-byte fault model at CHES-2018. The authors assumed a single-byte fault was inserted in the S-box, and the value and the position of the fault were known to the adversary. Moreover, only 2281 ciphertexts on average are used to recover the key. They also demonstrated the PFA against AES implementation with eight T-boxes, but the process of resetting the device and injecting a fault should be repeated four times to recover the full last round key and the number of required ciphertexts increases to 8200. Later, they presented a detailed analysis of the applicability of the PFA to several implementations of AES [13]. The improved PFA against AES implementation with the S-box does not require the knowledge of the value and location of the inserted fault [14]. Moreover, the PFA trial is first demonstrated on ATmega163L microcontroller in practice but the adversary is allowed to reset the device and collect the correct and incorrect ciphertexts corresponding to the same plaintext to determine whether a fault is inside the S-box. Caforio *et al*. [15] roughly described the concept of recovering the last round key of simplified Feistel ciphers whose round functions only consist of S-box lookup operations. However, if the permutation layer in the round function, such as the linear transformation of SM4, diffuses a single-byte error into every byte of the intermediate state, it is difficult to decide the error appears in which lookup operation. As a result, the last round key cannot be deduced even if the position and value of the inserted fault are known.

We present a revised PFA against SM4 which is based on a generalized Feistel structure. A single-byte fault is induced in the lookup table, and the value and the position of the fault are random and unknown to the adversary as well. Moreover, only one successfully fault injection is enough to leak the entire encryption key if SM4 is implemented either using the T-box or using the S-box. We launch the attack on the source codes of SM4 contained in standard cryptographic libraries Crypto++ [16] and GMSSL [17] separately. Our main contributions are as follows:

a) To the SM4 implementation using the S-box and the linear transformation, e.g., the SM4 code in Crypto++, we employ the inverse linear transformation L^{-1} [18] and the differential of two intermediate states to locate the position where the error occurs. Furthermore, we conclude the position of the inserted fault through three carefully chosen-ciphertext pairs.

b) To the SM4 implementation using a T-table, e.g., the SM4 code in GMSSL, we put forward a locating algorithm to locate the position where the error occurs and deduce the position of the faulty entry in the T-table. To our knowledge, this is the first fault attack breaking the SM4 implementation using a single T-table.

c) The locating algorithm also works on the SM4 implementation using the S-box.

d) We conduct experiments of the revised PFA against SM4 implementations protected by DMTR [19] and prove that the E-and-D mode can thwart the attack.

The rest of this paper is organized as follows: We first give a brief review of SM4 in Section II. Second, we depict the core idea and the process of PFA against SM4 implementation with the S-box in Section III, and we introduce the locating algorithm and PFA against SM4 implementation with a T-table in Section IV. Third, we give a theoretical evaluation of complexity of our attack in Section V. The experimental results of PFA against SM4 implementations involved in Crypto++ and GMSSL are shown in Section VI. Finally, we suggest the countermeasure to resist our PFA in Section VII and conclude the paper in Section VIII.

II. DESCRIPTION OF SM4 ALGORITHM

Both the block size and the key length of SM4 are 128 bits, and both the encryption and the key schedule are based on a generalized Feistel structure [1]. Due to the Feistel structure, the encryption and decryption process is the same, but the subkeys are applied in the reverse order in the decryption procedure. In this section, we briefly describe the encryption and the key schedule process of SM4 and introduce the implementation of SM4 based on a T-table.

A. ENCRYPTION

A 128-bit plaintext is split into four words (X_0, X_1, X_2, X_3) and fed into four 32-bit registers separately. The encryption process consists of 32-round iterations and a reverse transformation $\mathcal{R}$.

The round function is defined as: $X_{i+4} = X_i \oplus \mathcal{F}(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i)$, $i = 0, 1, \ldots, 31$. Here, $RK_i \in \{0,1\}^{32}$ is the i^{th} round key, and $\mathcal{F}$ is composed of a nonlinear transformation τ and a linear transformation L, namely, $\mathcal{F}(\cdot) = L(\tau(\cdot))$. In the nonlinear transformation τ, there are four S-box lookups, denoted by S. Let $A_{i+1} = (X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i)$ represent the input state of the transformation $\mathcal{F}$, which is divided into four bytes $(a_{i+1,0}, a_{i+1,1}, a_{i+1,2}, a_{i+1,3})$, and let B_{i+1} and H_{i+1} be the output states of τ and L respectively. The transformations τ and L are expressed as follows:

$$
\begin{aligned}
B_{i+1} &= \tau(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i) \\
&= (S(a_{i+1,0})||S(a_{i+1,1})||S(a_{i+1,2})||S(a_{i+1,3})), \\
H_{i+1} &= L(B_{i+1}) = B_{i+1} \oplus (B_{i+1} \lll 2) \oplus (B_{i+1} \lll 10) \\
&\quad \oplus (B_{i+1} \lll 18) \oplus (B_{i+1} \lll 24).
\end{aligned}
$$

The reverse transformation $\mathcal{R}$ maps the internal state $X_{32}, X_{33}, X_{34}, X_{35}$ onto the ciphertext $C \in \{0,1\}^{128}$, i.e., $\mathcal{R}(X_{32}, X_{33}, X_{34}, X_{35}) = (X_{35}, X_{34}, X_{33}, X_{32})$.

In addition, the inverse linear transformation L^{-1} is defined as follows [11]:

$$
\begin{aligned}
B_{i+1} = L^{-1}(H_{i+1}) &= H_{i+1} \oplus (H_{i+1} \lll 2) \oplus (H_{i+1} \lll 4) \\
&\oplus (H_{i+1} \lll 8) \oplus (H_{i+1} \lll 12) \oplus (H_{i+1} \lll 14) \\
&\oplus (H_{i+1} \lll 16) \oplus (H_{i+1} \lll 18) \oplus (H_{i+1} \lll 22) \\
&\oplus (H_{i+1} \lll 24) \oplus (H_{i+1} \lll 30).
\end{aligned}
$$

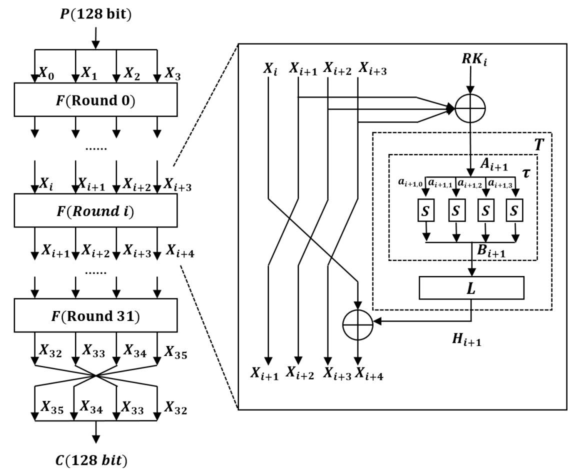

FIGURE 1. Structure of SM4 cipher.

B. KEY SCHEDULE

The encryption key, denoted by MK, is also split into four words (MK_0, MK_1, MK_2, MK_3) and XORed with four 32-bit constants (FK_0, FK_1, FK_2, FK_3), i.e.,

$$(K_0, K_1, K_2, K_3) = (MK_0 \oplus FK_0, MK_1 \oplus FK_1, MK_2 \oplus FK_2, MK_3 \oplus FK_3).$$

Afterwards, each 32-bit round key $RK_i (i \in \{0, 1, \ldots, 31\})$ is derived as follows:

$$RK_i = K_{i+4} = K_i \oplus \mathcal{F}'(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i),$$

$i = 0, 1, \ldots, 31$, where $(CK_0, CK_1, \ldots, CK_{31})$ are 32 constant parameters. The mixing transformation $\mathcal{F}'$ also consists of the nonlinear transformation τ and a simplified linear transformation L', i.e.,

$$B'_{i+1} = \tau(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i),$$
$$H'_{i+1} = L'\left(B'_{i+1}\right) = B'_{i+1} \oplus (B'_{i+1} \lll 13) \oplus (B'_{i+1} \lll 23).$$

In this paper, the adversary recovers the last four round keys $(RK_{31}, RK_{30}, RK_{29}, RK_{28})$ and calculates the encryption key using the inverse key schedule as follows: $(RK_{31}, RK_{30}, RK_{29}, RK_{28}) = (K_{35}, K_{34}, K_{33}, K_{32})$,

$$K_{35-i-4} = \mathcal{F}'(K_{35-i-3} \oplus K_{35-i-2} \oplus K_{35-i-1} \oplus CK_{31-i}) \oplus K_{35-i}, i = 0, 1, \ldots, 31,$$
$$MK = (MK_0, MK_1, MK_2, MK_3) = (K_0 \oplus FK_0, K_1 \oplus FK_1, K_2 \oplus FK_2, K_3 \oplus FK_3).$$

C. T-TABLE IMPLEMENTATION

Lang *et al.* [18] proposed a fast software implementation of SM4, which merges the nonlinear transformation τ and the linear transformation L into four lookup operations corresponding to four distinct T-tables, denoted by T_0, T_1, T_2, T_3. Each table $T_j (j \in 0, 1, 2, 3)$ contains 2^8 entries and the value of each entry is a 32-bit integer. In GMSSL [17], the transformation $\mathcal{F}$ is further optimized by looking up the same T-table, denoted by T. Consequently, the transformation $\mathcal{F}$ is expressed as:

$$H_{i+1} = \mathcal{F}(A_{i+1}) = [T(A_{i+1} \ \& \ 0xff) \ll 24] \oplus [T((A_{i+1} \gg 8) \ \& \ 0xff) \ll 16] \oplus [T((A_{i+1} \gg 16) \ \& \ 0xff) \ll 8] \oplus T[A_{i+1} \gg 24].$$

III. REVISED PERSISTENT FAULT ANALYSIS

This section explains the revised PFA in detail. At first, we will introduce the fault model and the core idea of our PFA. Secondly, we will assume the SM4 is implemented with the S-box and illustrate the concrete attack steps.

A. FAULT MODEL

The assumptions of our PFA are listed as follows:

1) The adversary can reboot the encryption system multiple times.
2) The adversary can inject a random single-byte fault into the lookup table (i.e., the S-box or the T-table). Let fp $(\in \{0, 1, \ldots, 255\})$ be the index of the faulty byte in the lookup table.
3) The injected fault is persistent, i.e., the affected entry stays faulty unless the encryption system is rebooted.
4) The adversary can feed chosen plaintexts into the encryption module, and obtain the corresponding (faulty or normal) ciphertexts.
5) The encryption key remains unchanged unless forced to alter.

B. CORE IDEA

First, the relationship between four words of the ciphertext $(X_{35}, X_{34}, X_{33}, X_{32})$ and internal states $(X_{31}, X_{30}, X_{29}, X_{28})$ involved in the last four rounds are as follows:

$$X_{35} = X_{31} \oplus \mathcal{F}(X_{32} \oplus X_{33} \oplus X_{34} \oplus RK_{31}),$$
$$X_{34} = X_{30} \oplus \mathcal{F}(X_{31} \oplus X_{32} \oplus X_{33} \oplus RK_{30}),$$
$$X_{33} = X_{29} \oplus \mathcal{F}(X_{30} \oplus X_{31} \oplus X_{32} \oplus RK_{33}),$$
$$X_{32} = X_{28} \oplus \mathcal{F}(X_{29} \oplus X_{30} \oplus X_{31} \oplus RK_{28}).$$

As shown in Figure 2, if an error occurs in the 32^{nd} round, X_{35} is affected so that the first words of the normal ciphertext C and the faulty ciphertext C' are distinct, i.e., $X_{35} \neq X'_{35}, X_{34} = X'_{34}, X_{33} = X'_{33}, X_{32} = X'_{32}$.

Provided that an error occurs in the i^{th} lookup operation in the 32^{nd} round, i.e., the i^{th} byte of the internal state $A_{32} = X_{32} \oplus X_{33} \oplus X_{34} \oplus RK_{31}$ hits the injected fault in the S-box, we can conclude that the i^{th} byte of the internal state equals the index of the fault (i.e., fp), namely $a_{32,i} = [X_{32} \oplus X_{33} \oplus X_{34} \oplus RK_{31}]_i = fp$. Consequently, the i^{th} byte of the last round key satisfies $rk_{31,i} = [X_{32} \oplus X_{33} \oplus X_{34}]_i \oplus fp$. Moreover, when i covers four byte-position of A_{32}, we can calculate the entire round key RK_{31}.

If the error appears in the 31^{st} round, the ciphertext pair (C, C') will be different at two words – X_{34} and X_{35}. We first execute the reverse transformation and one-round

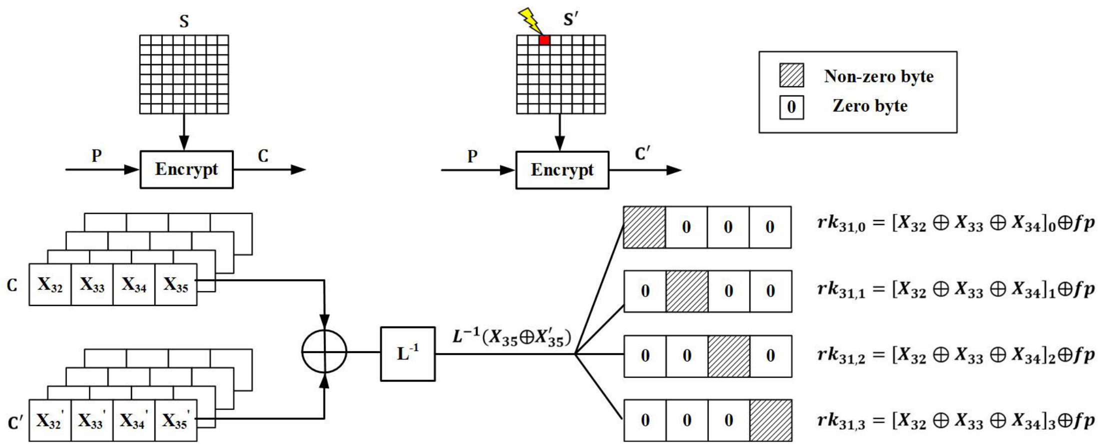

FIGURE 2. Overview of PFA against the implementation of SM4 with a S-box.

decryption on C to obtain internal state $X_{31}(= X_{35} \oplus \mathcal{F}(X_{32} \oplus X_{33} \oplus X_{34} \oplus RK_{31}))$. Next, we can similarly reveal the round key RK_{30} through $X_{31} \oplus X_{32} \oplus X_{33}$ and fp.

If the error occurs in the 30^{th} round, we collect ciphertext pairs (C, C') that the two ciphertexts are the same only at the last word X_{32}. Afterwards, we execute the reverse transformation and two-round decryption on C to obtain internal state X_{30}, and recover the round key RK_{29} in a similar way.

Finally, if the error occurs in the 29^{th} round, we cannot distinguish them directly through the ciphertext pairs (C, C'). Thereby, we first execute the reverse transformation and one-round decryption on C and C' separately to obtain internal states X_{31} and X'_{31}. If $X_{31} = X'_{31}$ holds, we know the error occurs in the 29^{th} round. Therefore, we execute two-round decryption on $(X_{34}, X_{33}, X_{32}, X_{31})$ to obtain X_{29} and disclose the round key RK_{28}.

Remark 1: How to locate the byte-position where an error occurs is not easy because of the diffusion effect of the linear transformation. When the SM4 is implemented with the S-box, we exploit the inverse linear transformation L^{-1} to calculate the difference between two internal states after the S-box lookups. Afterwards, the index of the non-zero byte of the difference corresponds to the error position. For instance, for a given ciphertext pair (C, C') of which the first words are distinct, i.e., $X_{35} \neq X'_{35}$, we compute $L^{-1}\left(X_{35} \oplus X'_{35}\right) = L^{-1}(X_{35}) \oplus L^{-1}(X'_{35}) = \tau\left(X_{32} \oplus X_{33} \oplus X_{34} \oplus RK_{31}\right) \oplus \tau'\left(X'_{32} \oplus X'_{33} \oplus X'_{34} \oplus RK_{31}\right)$. Consequently, if the i^{th} byte of $L^{-1}\left(X_{35} \oplus X'_{35}\right)$ is non-zero, we know the i^{th} lookup operation in the 32^{nd} round hits the injected fault.

Remark 2: How to know the position of the fault inside the S-box, i.e., the value of index fp, is another question. When the SM4 is implemented with the S-box, we can use some selected ciphertext pairs to filter out incorrect candidates of fp (see Phase 4 in Subsection III-C for details).

When the SM4 is implemented with the T-table, we present a locating algorithm to solve the above two problems (see Section IV for details).

Remark 3: Incorrect round keys may be generated because the lookup operation is also called in the key schedule. At this time, each ciphertext pair (C, C') satisfies $C \neq C'$, where C and C' are respectively the normal ciphertext and the incorrect ciphertext corresponding to the same plaintext P. In other words, we can conclude that no error appears in the key schedule once we detect at least one triple (P, C, C') such that $C = C'$.

C. REVISED PFA AGAINST THE IMPLEMENTATION OF SM4 WITH THE S-BOX

Our PFA includes 7 phases: ciphertexts online collecting in the first two phases and the encryption key offline extracting in other phases.

Phase 1: Obtain Correct Ciphertexts:

The adversary randomly generates some plaintexts $\mathcal{P} = \{P_j | j = 0, 1, \ldots, n\}$, individually encrypts each plaintext, and records the corresponding correct ciphertext $\mathcal{C} = \{C_j | j = 0, 1, \ldots, n\}$, where n is the number of plaintexts that is sufficient to recover the encryption key.

Phase 2: Inject a Fault and Obtain Faulty Ciphertexts:

Step 1: The adversary injects a single-byte fault in the S-box while rebooting the encryption system.

Step 2: The adversary encrypts each plaintext in set $\mathcal{P}$ and collects the corresponding faulty ciphertexts $\mathcal{C}' = \{C'_j | j = 0, 1, \ldots, n\}$.

Step 3: The adversary inspects those triples. If there exists a triple $\left(P_j, C_j, C'_j\right)$ $(j \in \{0, 1, \ldots, n\})$ such that C_j is identical to C'_j, the adversary continues to execute the phases 3-7. Otherwise, he restarts Phase 2.

Phase 3: Classify Triples Into Four Sets $\mathcal{C}'^{32}, \mathcal{C}'^{31}, \mathcal{C}'^{30}$, and $\mathcal{C}'^{*}$:

The adversary compares the two ciphertexts in each triple (P_j, C_j, C'_j), where C_j and C'_j are split into four words, i.e., $C_j = (X_{35}||X_{34}||X_{33}||X_{32})$ and $C'_j = (X'_{35}||X'_{34}||X'_{33}||X'_{32})$.

Case 1: If $X_{35} \neq X'_{35}, X_{34} = X'_{34}, X_{33} = X'_{33}, X_{32} = X'_{32}$, he adds the pair (C_j, C'_j) to $\mathcal{C}'^{32}$.

Case 2: If $X_{35} \neq X'_{35}, X_{34} \neq X'_{34}, X_{33} = X'_{33}, X_{32} = X'_{32}$, he adds the pair (C_j, C'_j) to $\mathcal{C}'^{31}$.

Case 3: If $X_{35} \neq X'_{35}, X_{34} \neq X'_{34}, X_{33} \neq X'_{33}, X_{32} = X'_{32}$, he adds the pair (C_j, C'_j) to $\mathcal{C}'^{30}$.

Case 4: If $X_{35} \neq X'_{35}, X_{34} \neq X'_{34}, X_{33} \neq X'_{33}, X_{32} \neq X'_{32}$, he adds the pair (C_j, C'_j) to $\mathcal{C}'^{*}$.

Phase 4: (Locating Algorithm) Deduce the Round Key RK_{31} and the Value of fp:

Step 1: The adversary sets the value of fp to 0.

Step 2: For a ciphertext pair (C_j, C'_j) in $\mathcal{C}'^{32}$, the adversary computes $L^{-1}(X_{35} \oplus X'_{35})$. Next, he sets the value of the $i^{th}(i \in \{0, 1, 2, 3\})$ byte of the last round key RK_{31} to $[X_{32} \oplus X_{33} \oplus X_{33}]_i \oplus fp$ if the i^{th} byte of $L^{-1}\left(X_{35} \oplus X'_{35}\right)$ is non-zero.

Step 3: The adversary picks out another ciphertext pair from $\mathcal{C}'^{32}$ and repeats Step 2 until the variable i covers four possible positions. Consequently, he obtains a candidate of the round key RK_{31}.

Step 4: The adversary picks out a ciphertext pair from $\mathcal{C}'^{31}$, denoted by (C_{j_0}, C'_{j_0}) Here, each ciphertext C_{j_i} also composes of four words $(X_{35,j_i}||X_{34,j_i}||X_{33,j_i}||X_{32,j_i}), j_i \in \{0, 1, \ldots, n\}$. Then, he computes $L^{-1}(X_{34,j_0} \oplus X'_{34,j_0})$. Provided that the index of the first non-zero byte of $L^{-1}(X_{34,j_0} \oplus X'_{34,j_0})$ is l $(l \in \{0, 1, 2, 3\})$, he searches for two other ciphertext pairs (C_{j_1}, C'_{j_1}) and (C_{j_2}, C'_{j_2}) from $\mathcal{C}'^{31}$ satisfying that the l^{th} bytes of $L^{-1}(X_{34,j_1} \oplus X'_{34,j_1})$ and $L^{-1}(X_{34,j_2} \oplus X'_{34,j_2})$ are non-zero as well. The selected pairs are called as verification pairs, denoted by $(C_{v_0}, C'_{v_0})_l$, $(C_{v_1}, C'_{v_1})_l$ and $(C_{v_2}, C'_{v_2})_l$.

Step 5: The adversary runs the reverse transformation and separately decrypts C_{v_0}, C_{v_1}, and C_{v_2} one round using the above candidate of the round key RK_{31}. Furthermore, he individually calculates $X_{31,v_0} \oplus X_{32,v_0} \oplus X_{33,v_0}$, $X_{31,v_1} \oplus X_{32,v_1} \oplus X_{33,v_1}$, and $X_{31,v_2} \oplus X_{32,v_2} \oplus X_{33,v_2}$. If the l^{th} bytes of them are not equal, let $fp = fp + 1$ and go to Step 2. Otherwise, he outputs the candidate of the round key RK_{31} and the value of fp at the moment.

The corresponding pseudocode of Phase 4 is shown in Algorithm 1 in Appendix A.

Phase 5: Deduce the Round Keys RK_{30} and RK_{29}:

Step 1: The adversary picks out a ciphertext pair from $\mathcal{C}'^{31}$ and decrypts the correct ciphertext C_j one round to obtain X_{31}. Then, he calculates the difference $L^{-1}\left(X_{34} \oplus X'_{34}\right)$ and recovers the round key RK_{30} using a similar method as that in Steps 2 and 3 of Phase 4.

Step 2: The adversary picks out a ciphertext pair from $\mathcal{C}'^{30}$, and decrypts the correct ciphertext C_j two rounds to obtain X_{31} and X_{30}. Then, he deduces the difference $L^{-1}\left(X_{33} \oplus X'_{33}\right)$ and recovers the round key RK_{29} using the same method.

Phase 6: Classify the Set C'^{*} and Deduce the Round Key RK_{28}:

Step 1: For ciphertext pairs (C_j, C'_j) in $\mathcal{C}'^{*}$, the adversary decrypts C_j and C'_j one round to obtain the internal states X_{31} and X'_{31} separately. If $X_{31} = X'_{31}$, he puts the corresponding ciphertext pair (C_j, C'_j) into set $\mathcal{C}'^{29}$.

Step 2: The adversary picks out a ciphertext pair form $\mathcal{C}'^{29}$, and decrypts the correct ciphertext C_j three rounds to obtain X_{31}, X_{30}, and X_{29}. Then, he computes the difference $L^{-1}\left(X_{32} \oplus X'_{32}\right)$ and recovers the round key RK_{28} using the same method.

Phase 7: Deduce the Encryption Key MK:

The adversary derives the encryption key MK according to RK_{28}, RK_{29}, RK_{30} and RK_{31} using the inverse key schedule.

IV. PFA AGAINST THE SM4 IMPLEMENTATION WITH A T-TABLE

The input of the T-table lookup is a byte, but the output is four bytes [18]. Thus, only one of the four bytes is altered when a single-byte fault is inserted. Consequently, the combined techniques used in Phase 4 cannot correctly locate the position where an error occurs. Therefore, we develop a locating algorithm to accomplish this task (see Figure 3). Other analysis phases are similar to the phases in Subsection III-C.

Phase 4: (Locating Algorithm) Deduce the Round Keys RK_{31}, RK_{30} and the Value of fp:

Step 1: The adversary sets fp to 0.

Step 2: The adversary creates two empty sets $\mathcal{G}$ and $\bar{\mathcal{G}}$.

Step 3: For each pair (C_j, C'_j) in $\mathcal{C}'^{32}$, the adversary computes $G_j = X_{32} \oplus X_{33} \oplus X_{34} \oplus (fp||fp||fp||fp)$, where $C_j = (X_{35}||X_{34}||X_{33}||X_{32})$, and adds G_j to the set $\mathcal{G}$.

Step 4: The adversary selects four words $G_a = (g_{a,1}||g_{a,2}||g_{a,3}||g_{a,4})$, $G_b = (g_{b,1}||g_{b,2}||g_{b,3}||g_{b,4})$, $G_c = (g_{c,1}||g_{c,2}||g_{c,3}||g_{c,4})$, and $G_d = (g_{d,1}||g_{d,2}||g_{d,3}||g_{d,4})$ from $\mathcal{G}$ such that $g_{a,i} \neq g_{b,i} \neq g_{c,i} \neq g_{d,i}$, for each $i(\in \{1, 2, 3, 4\})$.

Step 5: The adversary randomly constructs a vector $(g_{a,i1}||g_{b,i2}||g_{c,i3}||g_{d,i4})$, where $i1, i2, i3, i4 \in \{1, 2, 3, 4\}$ and the four indexes are pairwise distinct. The constructed vector is a candidate of RK_{31} and totally 24 candidates of RK_{31} can be generated, denoted by $\varkappa = \{RK_{31,1}, RK_{31,2}, \cdots, RK_{31,24}\}$.

Step 6: The adversary draws an element G_j from $\mathcal{G}$ (without replacement) and compares it to each candidate $RK_{31,l}(l \in \{1, 2, \cdots, 24\})$ separately. If four corresponding bytes between G_j and $RK_{31,l}$ are all distinct, remove $RK_{31,l}$ from the candidate set $\varkappa$. The adversary keep checking the remaining candidates using each element in $\mathcal{G}$ until all elements in $\mathcal{G}$ are tried out.

Step 7: For each pair (C_j, C'_j) in $\mathcal{C}'^{31}$, the adversary decrypts C_j one round to obtain X_{31}, computes $\bar{G}_j = X_{31} \oplus X_{32} \oplus X_{33} \oplus (fp||fp||fp||fp)$, and adds $\bar{G}_j$ to the set $\bar{\mathcal{G}}$.

Step 8: The adversary recovers the round key RK_{30} using the method depicted in Steps 4, 5 and 6. If there is no candidate of RK_{30} left in the set $\varkappa$, i.e., the current value of fp is wrong, let $fp = fp + 1$ and go to Step 2. Otherwise, he outputs the candidate of the round keys RK_{31}, RK_{30}, and the value of fp at the moment.

The corresponding pseudocode of the above steps is shown in Algorithm 3 in Appendix B.

Remark 4: Since errors occur, at least one byte of G_j $(\in \mathcal{G})$ equals the corresponding byte of the last round key RK_{31}. Moreover, because the i^{th} $(i \in \{1, 2, 3, 4\})$ bytes of G_a, G_b, G_c, and G_d are pairwise distinct, each

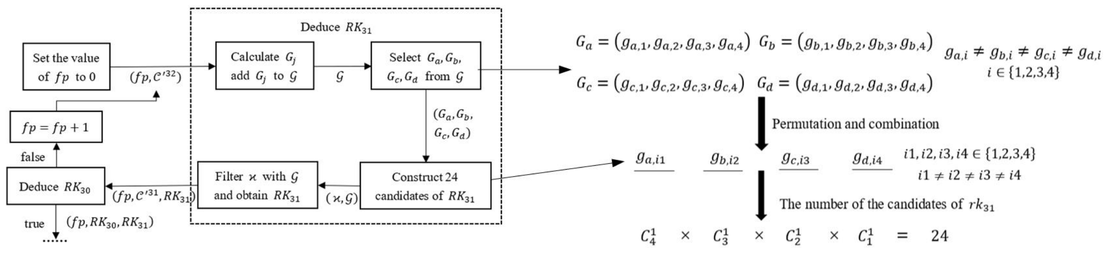

FIGURE 3. Process of the locating algorithm (taking the 32^{nd} round as an example).

$G_j (j \in \{a, b, c, d\})$ contains only one byte that equals the corresponding byte of RK_{31}. For instance, $rk_{31,0} = g_{a,1}$, $rk_{31,1} \neq g_{a,2}$, $rk_{31,2} \neq g_{a,3}$, and $rk_{31,3} \neq g_{a,4}$ hold, where $RK_{31} = (rk_{31,0}||rk_{31,1}||rk_{31,2}||rk_{31,3})$. Therefore, if we fill the first byte of a candidate with $g_{a,1}$, the second byte of the constructed candidate $g_{*,i2}$ only has three choices, i.e., $g_{b,2}$, $g_{c,2}$, and $g_{d,2}$. As a result, the total number of candidates is $24 = 4 \times 3 \times 2 \times 1$.

V. COMPLEXITY ANALYSIS

At first, there are $2n$ encryption operations in Phases 1 and 2 to obtain n triples. Second, he splits those triples into 4 sets in Phase 3. Since error may occur at every round, each of the first three sets involves $n/32$ triples on average. However, there are only n comparison operations in Phase 3, which can be negligible. Third, the main operation in Phase 5 is one-round and two-round decryption operations, and they are repeated for each element in sets $\mathcal{C}'^{31}$ and $\mathcal{C}'^{30}$ in the worst case. Thus, $n/32$ one-round and $n/32$ two-round decryption operations are required in the worst case. Forth, there are also $n/32$ triples that belong to set $\mathcal{C}'^{29}$ on average, but he should decrypt all triples in $\mathcal{C}'^{*}$ to identify them in the worst case. Consequently, $29/32 \times 2n$ one-round decryption operations and $n/32$ three-round decryption operations are needed in Steps 1 and 2 respectively. Fifth, Phase 7 only includes the key schedule process, i.e., there is one 32-round decryption operation. Also because the decryption procedure is the same as the encryption procedure, the worst-case complexity of the above phases approximate $(2 + 1/16)n$ encryption operations.

For the first locating algorithm (Algorithm 1), the adversary launches one-round decryption on each difference of the triple in set $\mathcal{C}'^{32}$ and one-round decryption on each ciphertext in the triple in $\mathcal{C}'^{32}$ in Step 2 and Step 4 separately. Furthermore, the number of iterations is 256 in the worst case. As a result, there are $3/4 \times n$ encryption operations at most.

For the second locating algorithm (Algorithm 4), Steps 2-6 and 8 only involves comparison operations, which is also negligible as well. However, Step 7 contains one-round decryption operations and the number of iterations of this step depends not only on the value of fp but also on the number of candidates left after Step 6. In the majority of our experiments, there is only one candidate when $n > 3000$, so the number of encryption operations is $n/32 \times 1/32 \times 2 \times 256 = n/2$ at most.

TABLE 1. Statistical results of the 992 experiments of successful attacks.

Number of Reboots	Number of Experiments	Average Elapsed Time(s)
1	616	0.0152
2	219	0.0158
3	93	0.0166
4	39	0.0172
5	15	0.0181
6	6	0.0188
7	4	0.0192
Total	992	\

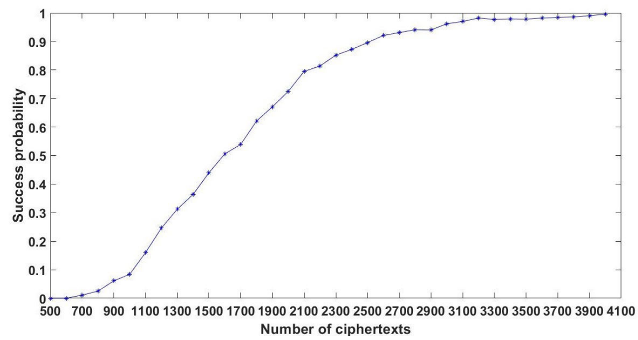

FIGURE 4. Success probability of the revised PFA against SM4 implementation in Crypto++.

In summary, the worst-case computational complexity of the analysis is $O(3n)$ encryption operations. Obviously, the data complexity is $O(n)$ plaintexts and the required memory is $O(3n \times 128)$ bits for all triples.

VI. EXPERIMENT RESULTS

We apply our PFA to software implementations of SM4. The source code of SM4 in Crypto++ is implemented using the S-box, and the code in GMSSL is implemented using the T-table. The experiments are performed on a PC with an Intel Core I7-8550U processor (1.8GHz) and the code is written in C language.

A. REVISED PFA AGAINST SM4 IMPLEMENTATION IN Crypto++

We progressively increment the number of plaintexts used in the attack procedure and repeat the experiment 1000 times for a given number.

It can be seen from Figure 4 that the encryption key is successfully recovered with a probability of more than 95% when the number of plaintexts approaches 3000. As mentioned

TABLE 2. Probability distribution of the number of reboots.

Number of Reboots	Theoretical Probability	Practical Probability
1	0.605	0.621
2	$0.395 \times 0.605 = 0.239$	0.221
3	$0.395 \times 0.395 \times 0.605 = 0.0944$	0.0938
4	$0.395 \times 0.395 \times 0.395 \times 0.605 = 0.0373$	0.0393
5	$0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.605 = 0.0147$	0.0151
6	$0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.605 = 0.00582$	0.00605
7	$0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.395 \times 0.605 = 0.00229$	0.00403

in Remark 3, if the inserted fault affects the key schedule, the adversary will reboot the system to inject a new fault. When the number of plaintexts is 4000, the encryption key is successfully recovered in 992 experiments out of total 1000 experiments. In each experiment, we count the times that the system is rebooted, and record the elapsed time of each experiment. Table 1 lists the metrics of those experiments.

The second row of Table 1 shows that the adversary injects a single-byte fault only once in the majority of the experiments. Besides, the runtime of the attack increases slowly with the growth of the number of reboots. It is because that we only encrypt 100 plaintexts to determine whether the key schedule is affected by the inserted fault, i.e., the runtime of Step 3 in Phase 2 is much shorter than that of the entire encrypting and analysis process.

On the one hand, if a single-byte fault is injected in the S-box, the probability that the input of a lookup operation does not equal *fp* (the position of the fault) is $\frac{255}{256}$. On the other hand, the key schedule is composed of 32-round iterations, and the lookup operation is called four times in each iteration. Therefore, there are 128 lookup operations during the key schedule. Provided that the inputs of the 128 operations are independent of one another, the probability that no error occurs during the key schedule is $\left(\frac{255}{256}\right)^{128} \approx 0.605$ and the probability that at least one error occurs during the key schedule is $1 - 0.605 = 0.395$. Table 2 lists the practical probability and the theoretical probability that the number of reboots equals to a given integer. It can be seen that the theoretical prediction matches the experimental results.

B. REVISED PFA AGAINST SM4 IMPLEMENTATION IN GMSSL

In GMSSL, because the encryption uses the T-table but the key schedule uses the S-box, the fault in the T-table does not affect the key schedule. Therefore, the system is only rebooted once. In other words, as long as the adversary collects enough correct and incorrect ciphertext pairs, he is able to disclose the entire 128-bit encryption key.

We also progressively increment the number of plaintexts and repeat the experiment 1000 times for a given number. It can be seen from Figure 5 that the encryption key is successfully recovered with a probability of more than 99% when the number of plaintexts approaches 3000. We also

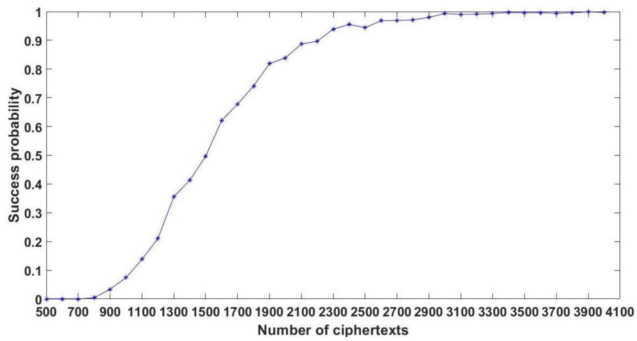

FIGURE 5. Success probability of the revised PFA against SM4 implementation in GMSSL.

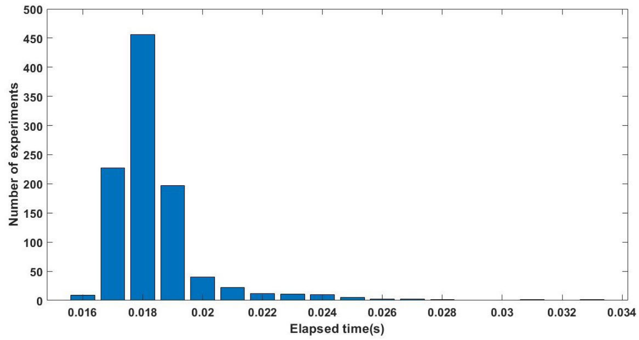

FIGURE 6. Distribution of the elapsed time for 1000 experiments of successful attacks.

record the elapsed time of each experiment independently when the number of plaintexts is 4000. Figure 6 shows the distribution of the elapsed time for 1000 experiments of successful attacks.

It can be seen from Figure 6 that the time to conduct most experiments is between 0.016 and 0.021 seconds and the longest runtime of the attack is less than 0.04 seconds. Since the round function only includes four T-table lookups and three XOR operations, the collecting ciphertexts procedure here is slightly sped up. However, the locating algorithm is more complicated than the inverse linear transformation. Consequently, the runtime of our PFA against the implementation of SM4 in GMSSL is a little longer than that of our PFA against the implementation of SM4 in Crypto++. Table 3 shows a comparison of PFAs against SM4 implementations in Crytpo++ and GMSSL. Obviously, if the adversary obtains enough plaintexts and the corresponding ciphertexts, the SM4 implemented with a T-table can be cracked with only one fault injection, but the SM4 implemented with the S-box

TABLE 3. Comparison of results for existing fault attacks against SM4.

Ref.	Target	Fault type	Position of the fault	Value of the fault	Length of the fault	Number of injections	Number of ciphertexts	Extra brute-force attack	Average elapsed time(s)	Success probability
[9]	SM4	Tran-sient	Date process (The input of $29^{th}-32^{nd}$ round)	Random	Byte	8	32 (Theoretical)	-	<0.001	-
[10]	SM4		Key schedule	Random	Byte	4	32 (Theoretical)	-	<0.001	-
[11]	SM4		Date process (The input of 28^{th} round)	Random	Byte	1	-	$2^{22.11}$ (Theoretical)	-	-
Our work	SM4 (Crypto++)	Persis-tent	Random at the S-box	Random	Byte	1.641 (The mean)	2760 (Average)	-	0.0172	0.992
	SM4 (GMSSL)		Random at the T-table			1 (The mean)	2789 (Average)	-	0.0184	1

needs more injections in most cases because the key schedule is affected by the injected fault as well. Therefore, the T-table speeds up the encryption calculation, but the security risk is higher.

VII. COUNTERMEASURE AGAINST THE PFA

We apply the revised PFA to SM4 implementations protected by DMTR, which includes the E-and-E mode and the E-and-D mode. In the E-and-E mode, a plaintext is encrypted twice. If the two encrypted states C and C' are identical, the procedure outputs C as the ciphertext. In the E-and-D mode, a plaintext P is encrypted to obtain the state C which is further decrypted to obtain another plaintext P'. If $P' = P$, the procedure outputs C as the ciphertext. However, if an error is detected, the procedure stops without any output (case 1) or outputs a 128-bit random number as the ciphertext (case 2).

Since the fault is persistent, two encrypted states are always the same in the E-and-E mode, i.e., errors cannot be detected. In our experiments, the success probability of our PFA approaches 1 when the number of plaintexts is more than 3000.

In the E-and-D mode, either the adversary cannot obtain faulty ciphertexts in case 1, or the output C'' is different to the genuine faulty ciphertext C' with high probability so that the deduced candidates of round keys are incorrect in case 2. Therefore, the probability that the encryption key is recovered with a negligible probability. In our experiments, we set the number of plaintext to 4000, and we fail to recover the encryption key in all 1000 experiments. Therefore, the experimental data matches the theoretical result. In summary, the DMTR countermeasure in the E-and-D mode thwarts our PFA.

VIII. CONCLUSION

In this paper, we firstly present a revised PFA against SM4 which is based on a generalized Feistel structure and validate our PFA on the source codes of SM4 from standard cryptographic libraries Crypto++ and GMSSL separately. The experiments show that when the number of ciphertext pairs is 3000, the probability of successfully recovering the SM4 encryption key within libraries Crypto++ and GMSSL reach 95% and 99% separately. Table 3 lists the results of our work and previous fault attacks against SM4. It can be seen that the PFA only requires one or two fault injections before encryptions. Especially, our PFA against SM4 can practically recover the secret key in a very short time. Thus, the PFA is a great threat to the implementations of SM4. At last, we further prove that the DMTR countermeasure in the E-and-D mode can thwart our PFA. The attack exploits the characteristics of the Feistel structure, i.e., part of the 128-bit internal state keeps unchanged within one round

Algorithm 1 Deduce the Round Key RK_{31} and the Value of fp

Input: $\mathcal{C}'^{32}, \mathcal{C}'^{31}$;
Output: fp, RK_{31};
for $fp = 0, \ldots, 255$ **do**
  for $j = 0, \ldots, sizeof(\mathcal{C}'^{32})$ **do**
    $X_{35}||X_{34}||X_{33}||X_{32} = \mathcal{C}'^{32}[j][0]$; %normal ciphertext
    $X'_{35}||X_{34}||X_{33}||X_{32} = \mathcal{C}'^{32}[j][1]$; %incorrect ciphertext
    $flag[4] = 0$; %initial a flag array to index each obtained byte of RK_{31}
    for $i = 0, \ldots, 3$ **do**
      if $(L^{-1}(X_{35} \oplus X'_{35})[i] \neq 0$ and $flag[i] == 0)$ **then**
        $RK_{31}[i] = [X_{32} \oplus X_{33} \oplus X_{34}][i] \oplus fp$; %calculate the i^{th} byte of RK_{31},i.e.,$rk_{31,i}$
        $flag[i] = 1$; %the i^{th} byte of RK_{31} are obtained
      end if
    end for
    if ($flag[0]$ == 1 and $flag[1]$ == 1 and $flag[2]$ == 1 and $flag[3] == 1$) **then**
      break; %four bytes of RK_{31} are obtained
    end if
  end for
  $C_v[3] = 0$; %initial an array of verification pairs
  C_v=Find_verification_pairs($fp, \mathcal{C}'^{31}, RK_{31}$); %call the subroutine to generate verification pairs
  for $i = 0, 1, 2$ **do**
    $X_{31,i}$=Dec_oneround $(RK_{31}, C_v[i])$; %one round decryption using the above RK_{31}
  end for
  if $((X_{31,0} \oplus X_{32,0} \oplus X_{33,0})[l] == (X_{31,1} \oplus X_{32,1} \oplus X_{33,1})[l]$ and $(X_{31,1} \oplus X_{32,1} \oplus X_{33,1})[l] == (X_{31,2} \oplus X_{32,2} \oplus X_{33,2})[l])$ **then**
    return RK_{31} and fp; %the candidate of RK_{31} pass the check
  end if
end for

Algorithm 2 Find_verification_pairs

```
Input: C'^31, RK_31;
Output: C_v;
n = 0;
C_v[3] = 0;
for j = 0, ..., sizeof(C'^31) do
    X_35||X_34||X_33||X_32 = C'^31[j][0];   %Normal ciphertext
    X'_35||X'_34||X_33||X_32 = C'^31[j][1];   %Faulty ciphertext
    l = 0;
    if (n == 0) then
        while (L^-1 (X_34 ⊕ X'_34) [l] == 0) do
            l++;   %Find the index l of the first non-zero byte
        end while
    end if
    if (n < 3 and L^-1 (X_34 ⊕ X'_34) [l] ≠ 0) then
        C_v[n] = C'^31[j][0];   %Find the n^th verification pair
        n+ = 1;
    else if n == 3 then
        return C_v;
    end if
end for
```

Algorithm 3 Deduce the Round Key RK_{31},RK_{30} and the Value of *fp*

```
Input: C'^32, C'^31;
Output: fp, RK_31, RK_30;
for fp = 0, ..., 255 do
    G, Ḡ=empty   %initial set G and Ḡ for round keys RK_31 and RK_30 respectively
    for j = 0, ..., sizeof(C'^32) do
        X_35||X_34||X_33||X_32 = C'^32[j][0];   %normal ciphertext
        X'_35||X_34||X_33||X_32 = C'^32[j][1];   %incorrect ciphertext
        G[j] = X_32 ⊕ X_33 ⊕ X_34 ⊕ (fp||fp||fp||fp)   %Add G_j to set G
    end for
    G_a, G_b, G_c, G_d=Find_Ga_Gb_Gc_Gd(G);   %call subroutine to generate the seeds of candidates of round key RK_31
    RK_31=Get_roundkey(G, G_a, G_b, G_c, G_d);   %call subroutine to generate and filter candidates of round key RK_31
    if (RK_31 ≠ error) then   %only one filtered candidate of round key RK_31 left
        for j = 0, ..., sizeof(C'^31) do
            X_35||X_34||X_33||X_32 = C'^31[j][0];   %normal ciphertext
            X'_35||X'_34||X_33||X_32 = C'^31[j][1];   %incorrect ciphertext
            X_31=Dec_oneround (RK_31, C'^31[j][0]);
            Ḡ[j] = X_31 ⊕ X_32 ⊕ X_33 ⊕ (fp||fp||fp||fp);   %Add Ḡ_j to set Ḡ
        end for
        Ḡ_a, Ḡ_b, Ḡ_c, Ḡ_d= Find_Ga_Gb_Gc_Gd(Ḡ);   %call subroutine to generate the seeds of candidates of round key RK_30
        RK_30= Get_roundkey(Ḡ, Ḡ_a, Ḡ_b, Ḡ_c, Ḡ_d);   %call subroutine to generate and filter candidates of round key RK_30
        if (RK_30 ≠ error) then   %only one filtered candidate of round key RK_31 left
            return fp, RK_31, RK_30;
        end if
    end if
end for
```

of encryption. However, whether the attack can be generalized to break ciphers based on other structures is an interesting topic for future works. Besides, we put forward the first fault analysis against SM4 implemented with the T-table. As the T-table is widely embedded in the software implementations of block ciphers, extending the core idea of our PFA to the analysis of other ciphers is of great significance.

Algorithm 4 Find_Ga_Gb_Gc_Gd

```
Input: G;
Output: G_a, G_b, G_c, G_d;
G_a = G[0];   %set the first element of G to G_a
for j = 1, ..., sizeof(G) do
    flag[3] = 0;   %initial a flag array to index the G_b, G_c, and G_d respectively
    if (flag[0] == 0 and G[j][0] ≠ G_a[0] and G[j][1] ≠ G_a[1]
            and G[j][2] ≠ G_a[2] and G[j][3] ≠ G_a[3]) then
        G_b = G[j];   %select required G_b from G or Ḡ
        flag[0] = 1;
    else
        continue;
    end if
    if (flag[1] == 0 and G[j][0] ≠ G_a[0] and G[j][1] ≠ G_a[1]
            and G[j][2] ≠ G_a[2] and G[j][3] ≠ G_a[4]
            and G[j][0] ≠ G_b[0] and G[j][1] ≠ G_b[1]
            and G[j][2] ≠ G_b[2] and G[j][3] ≠ G_b[3]) then
        G_c = G[j];   %select required G_c from G or Ḡ
        flag[1] = 1;
    else
        continue;
    end if
    if (flag[2] == 0 and G[j][0] ≠ G_a[0] and G[j][1] ≠ G_a[1]
            and G[j][2] ≠ G_a[2] and G[j][3] ≠ G_a[4]
            and G[j][0] ≠ G_b[0] and G[j][1] ≠ G_b[1]
            and G[j][2] ≠ G_b[2] and G[j][3] ≠ G_b[3]
            and G[j][0] ≠ G_c[0] and G[j][1] ≠ G_c[1]
            and G[j][2] ≠ G_c[2] and G[j][3] ≠ G_c[3]) then
        G_d = G[j];   %select required G_d from G or Ḡ
        flag[1] = 1;
    else
        continue;
    end if
    if (flag[0] == 1 and flag[1] == 1 and flag[2] == 1) then
        return G_a, G_b, G_c, G_d;   %output the selected four seeds of candidates of the round key
    end if
end for
```

APPENDIX A
REVISED PFA AGAINST IMPLEMENTATION OF SM4 WITH THE S-BOX

See Algorithms 1 and 2.

Phase 4. Deduce the Round Key RK_{31} and the Value of fp:

- Subroutine to generate verification pairs.

APPENDIX B
REVISED PFA AGAINST IMPLEMENTATION OF SM4 WITH THE T-TABLE

See Algorithms 3, 4, and 5.

Phase 4. Deduce the Round Key RK_{31},RK_{30} and the Value of fp:

- Subroutine to generate the seeds of candidates of a round key.
- Subroutine to generate and filter candidates of a round key.

ACKNOWLEDGMENT

(Qing Guo and ZhenHan Ke contributed equally to this work.)

Algorithm 5 Get_roundkey

Input: G, G_a, G_b, G_c, G_d;
Output: RK;
K=empty; %initial set $\varkappa$
$n = 0$;
for $i = 0, 1, 2, 3$ **do**
  for $j = 0, 1, 2, 3$ **do**
    if $(j \neq i)$ **then**
      for $k = 0, 1, 2, 3$ **do**
        if $(k \neq i$ and $k \neq j)$ **then**
          for $l = 0, 1, 2, 3$ **do**
            if $(l \neq i$ and $l \neq j$ and $l \neq k)$ **then**
              $K[n] = G_a[i] || G_b[j] || G_c[k] || G_d[l]$;
              $n++$; %Compute 24 candidates of the round key
            end if
          end for
        end if
      end for
    end if
  end for
end for
$flag[24] = 0$; %initial a flag array to index each candidate of the round key
for $i = 0, \ldots, 23$ **do**
  for $j = 0, \ldots, sizeof(G)$ **do**
    $tmp = G[j] \oplus K[i]$;
    if $(tmp[0] \neq 0$ and $tmp[1] \neq 0$ and $tmp[2] \neq 0$ and $tmp[3] \neq 0)$ **then**
      $flag[i] = -1$; %set the flag of the candidate of the round key which didn't pass the check to -1
      break;
    end if
  end for
end for
for $i = 0, \ldots 23$ **do**
  if $(flag[i] == 0)$ **then**
    return $K[i]$; %return the remaining candidate of the round key
  end if
end for
return *error* %return "error" if no candidate left

REFERENCES

[1] *SM4 Block Cipher Algorithm*, Standard GB/T 32 907–2016 GM/T 0002-2012, 2016.

[2] J. Etrog and M. J. Robshaw, "The cryptanalysis of reduced-round SMS4," in *Proc. Int. Workshop Sel. Areas Cryptogr.* Sackville, NB, Canada: Springer, 2008, pp. 51–65.

[3] M.-J. Liu and J.-Z. Chen, "Improved linear attacks on the Chinese block cipher standard," *J. Comput. Sci. Technol.*, vol. 29, no. 6, pp. 1123–1133, Nov. 2014.

[4] L. Zhang, W. Zhang, and W. Wu, "Cryptanalysis of reduced-round SMS4 block cipher," in *Proc. Australas. Conf. Inf. Secur. Privacy*, vol. 5107, Jul. 2008, pp. 216–229.

[5] B.-Z. Su, W.-L. Wu, and W.-T. Zhang, "Security of the SMS4 block cipher against differential cryptanalysis," *J. Comput. Sci. Technol.*, vol. 26, no. 1, pp. 130–138, Jan. 2011.

[6] X. Lou, F. Zhang, G. Xu, Z. Liang, X. Zhao, S. Guo, and K. Ren, "Enhanced differential cache attacks on sm4 with algebraic analysis and error-tolerance," in *Information Security and Cryptology*. Nanjing, China: Springer, 2020, pp. 480–496.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.* Springer, 1997, pp. 37–51.

[8] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Proc. Annu. Int. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, 1997, pp. 513–525.

[9] L. Zhang and W. L. Wu, "Differential fault analysis on SMS4," *Chin. J. Comput.*, vol. 29, no. 9, pp. 1596–1602, 2006.

[10] W. Li and D. W. Gu, "Differential fault analysis on the SMS4 cipher by inducing faults to the key schedule," *J. Commun.*, vol. 10, p. 135, Aug. 2008.

[11] R. Li, B. Sun, C. Li, and J. You, "Differential fault analysis on SMS4 using a single fault," *Inf. Process. Lett.*, vol. 111, no. 4, pp. 156–163, Jan. 2011.

[12] F. Zhang, X. Lou, X. Zhao, S. Bhasin, W. He, R. Ding, S. Qureshi, and K. Ren, "Persistent fault analysis on block ciphers," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, Aug. 2018, pp. 150–172.

[13] F. Zhang, G. Xu, B. Yang, Z. Liang, and K. Ren, "Theoretical analysis of persistent fault attack," *Sci. China Inf. Sci.*, vol. 63, no. 3, pp. 1–3, Mar. 2020.

[14] F. Zhang, Y. Zhang, H. Jiang, X. Zhu, S. Bhasin, X. Zhao, Z. Liu, D. Gu, and K. Ren, "Persistent fault attack in practice," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, Mar. 2020, pp. 172–195.

[15] A. Caforio and S. Banik, "A study of persistent fault analysis," in *Proc. Int. Conf. Secur., Privacy, Appl. Cryptogr. Eng.* Gandhinagar, India: Springer, 2019, pp. 13–33.

[16] *Crypto++: A Free C++ Class Library of Cryptographic Schemes (Version 8.3)*. [Online]. Available: https://www.cryptopp.com/docs/ref/class_s_m4_1_1_enc.html

[17] *GMSSL: An Open Source Cryptographic Toolbox (Version 1.0.0)*. [Online]. Available: https://github.com/guanzhi/GmSSL

[18] H. Lang, L. Zhang, and W. L. Wu, "Fast software implementation of SMS4," *J. Univ. Chin. Acad. Sci.*, vol. 35, pp. 180–187, Dec. 2018.

[19] M. Joye and M. Tunstall, *Fault Analysis Cryptography*, vol. 147. Springer, 2012.

QING GUO is currently pursuing the B.S. degree with the School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China. Her main research interests include cryptography, intrusion detection, and attack investigation.

ZHENHAN KE is currently pursuing the B.S. degree with the School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China. His main research interests include cryptography and intrusion detection.

SIYUAN WANG is currently pursuing the B.S. degree with the Beijing University of Posts and Telecommunications, China. Her main research interests include cryptography and security and privacy issues in wireless communication networks.

SHIHUI ZHENG received the Ph.D. degree from Shandong University, China, in 2006. From 2006 to 2008, she held a postdoctoral position with the School of Information Engineering, Beijing University of Posts and Telecommunications (BUPT), China. In 2008, she joined the School of Cyberspace Security and National Engineering Laboratory, BUPT, for Disaster Backup and Recovery. Her current research interests include cryptographic scheme design and side-channel attacks.