Low-cost Persistent Fault Attack on AES and Beyond Faulting S-box Table

Viet Sang Nguyen, Vincent Grosso and Pierre-Louis Cayrel

Laboratoire Hubert Curien, Université Jean Monnet, Saint-Étienne, France

Abstract. Persistent Fault Attack (PFA) has emerged as an active research area in embedded cryptography. This attack exploits faults in one or multiple constants stored in memory, typically targeting S-box elements. In practice, inducing such a memory fault relies on techniques like laser or electromagnetic injection, which require expensive equipment. In this paper, we demonstrate that a persistent fault can also be achieved using a low-cost technique, such as clock glitch injection. Specifically, we target AES implementations that dynamically generate the S-box table at runtime, before executing the first AES operation. We illustrate this with an attack on the AES implementation in the MbedTLS library, where a clock glitch is inserted during the S-box generation.
Second, we introduce, to our knowledge, the first PFA that targets a constant other than the S-box elements. In this attack, faulting a round constant in AES is sufficient to recover the key by a differential analysis. This approach significantly reduces the amount of data for analysis compared to previous PFAs, which require hundreds to thousands of ciphertexts. In contrast, our differential analysis needs only three correct-faulty ciphertext pairs. We showcase this attack through an experiment on the AES implementation in the MbedTLS library with a clock glitch in the round constant generation.

Keywords: Persistent Fault Attacks · Glitch · AES

1 Introduction

Fault injection attacks have become a major area of interest in embedded cryptography. These attacks take advantage of errors in the execution of cryptographic algorithms, caused by intentional fault injection, to extract the secret key. The impact of such attacks can be severe, leaving a system vulnerable. A fault attack typically involves two main steps: fault injection and fault analysis. In the first step, the attacker deliberately introduces faults into the target device to disrupt the algorithm's execution. This can be done using techniques like laser pulses, electromagnetic interference, voltage glitches, or clock glitches. In the second step, the attacker analyzes the faulty outputs collected from the device to recover the secret key.

The concept of fault attack was first presented by Boneh *et al.* [BDL97] with an application to RSA. Subsequently, Biham and Shamir [BS97] proposed Differential Fault Analysis (DFA) with an application to DES. Since then, DFA has become a common fault attack, applicable to many block ciphers such as AES [PQ03, DLV03], DES [Riv09]. Over time, numerous additional effective fault attacks have been developed using disparate techniques. Some examples are Ineffective Fault Analysis (IFA) [Cla07], Statistical Fault Analysis (SFA) [FJLT13], Differential Fault Intensity Analysis (DFIA) [GYTS14], Fault Sensitivity

E-mail: viet.sang.nguyen@univ-st-etienne.fr (Viet Sang Nguyen), vincent.grosso@univ-st-etienne.fr (Vincent Grosso), pierre.louis.cayrel@univ-st-etienne.fr (Pierre-Louis Cayrel)

Date of this document: 2025-02-10.

Analysis (FSA) [LSG+10], Statistical Ineffective Fault Analysis (SIFA) [DEK+18], Fault Template Attack (FTA) [SBR+20].

Depending on the duration of the effect, faults can be classified into three categories: *transient faults*, *persistent faults*, and *permanent faults*. A transient fault affects the execution in a very short period, typically during a single encryption. This means that a transient fault causes errors in only one execution and does not persist in subsequent executions. Most fault attacks mentioned above are proposed within the transient fault settings. A permanent fault, on the other hand, has a lasting effect on the target and cannot be erased. Persistent faults, on which we focus in this work, fall between the other two categories. A fault of this type persists across different executions but is erased once the device is reset. Compared to attacks based on transient faults, a persistent fault attack has several advantages: the attacker does not need to inject faults with live synchronization of different executions; it takes advantage of its inherent characteristic to bypass some redundancy-based countermeasures.

The concept of persistent faults was introduced by Schmidt *et al.* [SHP09] with an application to attacking AES. Recently, Persistent Fault Analysis (PFA) has received significant attention since the work of Zhang *et al.* [ZLZ+18] at CHES 2018. In this work, the authors developed a dedicated model for the persistent fault setting and proposed a technique for recovering the key of block ciphers, with an application to AES. The model assumes that the ciphers are implemented with a lookup table for the S-box and that the faults affect one or multiple S-box elements stored in memory (*e.g.*, ROM). Specifically, the faults result in a *biased faulty S-box*, where one or several S-box elements appear more frequently while one or several others disappear. These faults persist across different encryptions until the device is reset. Building on Zhang *et al.*'s model, many follow-up works have either aimed to reduce the number of required ciphertexts by more advanced analyses [CGR20, XZY+21, ZZJ+20, SBH+22, ZFL+22, ZHF+23] or to apply this model to attack different (protected) ciphers [PZRB19, GPT19, TL22, ZFL+22].

We underline that the above PFAs are all based on the model of Zhang *et al.* [ZLZ+18], *i.e.*, based on the assumption that one or multiple S-box elements stored in memory are faulted. The injection of memory faults is expensive as it usually requires costly equipment for laser or electromagnetic fault injection. Examples of the use of these costly equipment in PFA are the experiments of Schmidt *et al.* [SHP09], Zhang *et al.* [ZZJ+20], and Soleimany *et al.* [SBH+22]. With this observation, we pose the following research question: *Can we perform a PFA with a classical and cheap fault injection technique such as inserting a clock glitch?*

Our second observation is that the above PFAs all target to fault the S-box elements. Their analysis phases focus on exploiting the biased distribution of the ciphertexts as the consequence of the biased faulty S-box. This leads us to the second research question: *Are the S-box elements the only constants that the attacker can target to fault in the PFA context?*

Contributions. In this work, we provide affirmative answers to the two research questions. First, we show that PFA attacks on AES can be performed using a low-cost fault injection technique, such as clock glitch insertion. This approach is particularly effective against implementations that generate the S-box table at runtime before executing the first AES operation. In practice, some embedded cryptographic libraries, such as MbedTLS[1] and cryptlib,[2] offer this strategy for their AES implementations. We showcase an attack on the AES implementation in MbedTLS by using a clock glitch to skip an instruction, resulting in a faulty S-box that can be exploited to recover the key.

[1] https://github.com/Mbed-TLS/mbedtls, version 3.6.1
[2] https://www.cs.auckland.ac.nz/~pgut001/cryptlib/, version 3.4.8

Table 1: Comparison with previous PFAs on AES.

	Fault target	**Fault injection technique**	**Cost**
[**ZLZ+18**]	T-table elements stored in DRAM	Rowhammer	Low
[**ZZJ+20**]	S-box elements stored in SRAM	Laser	High
[**SHP09**]	S-box elements stored in ROM	Laser	High
[**SBH+22**]	Transfer of S-box elements from flash memory to RAM	Electromagnetic	Medium to High
[**GTB+24**]	S-box elements stored in flash memory	Laser	High
This work Section 3	Skip instruction in S-box generation	Clock glitch	Low
This work Section 4	Skip instruction in round constant generation	Clock glitch	Low

Second, we show that the S-box is not the only target for fault injection in PFA attacks. We propose, to our knowledge, the first PFA that exploits a fault in a different constant rather than the S-box elements. Specifically, we consider a persistent fault induced on a round constant of AES. We demonstrate that the key can be effectively recovered using a differential fault analysis. Our attack significantly reduces the amount of data required compared to many previous works based on Zhang *et al.*'s model, which typically involves analyzing hundreds to thousands of ciphertexts. In contrast, our differential analysis needs only three correct-faulty ciphertext pairs. Towards using a cheap fault injection technique, we showcase this with an experiment on the AES implementation in MbedTLS, where the round constants are generated in the initialization phase. To obtain the desired faulty round constant, a clock glitch is used to skip an instruction during the round constant generation in the initialization phase.

Table 1 presents a comparison between this work and the existing PFAs in the literature that include practical fault injection experiments. Other studies, such as [CGR20, XZY+21, ZFL+22, ZHF+23], which focus on the analysis phase, are excluded from this comparison.

For reproducibility, we publish the source code for our experiments and simulations. The experimental code is intended for those with access to the required hardware, while the simulation code can be used by those without the hardware. The code is available at

`https://anonymous.4open.science/r/RKA8-XLVC`

Outline. The paper is organized as follows. Section 2 provides the background necessary for this work. Section 3 details the attack that targets a fault in the S-box generation. Section 4 describes the attack that exploits a faulty round constant. Finally, Section 5 concludes our work and provide some perspectives.

2 Preliminaries

In this section, we first provide a brief overview of related PFAs in the literature. Next, we present the background on AES, the cipher used to showcase our attacks in this work.

We then present the fault model for the attacker. Finally, we describe the setup for our experiments.

2.1 Related PFAs

Fault anlysis phase. At CHES 2018, Zhang *et al.* [ZLZ+18] introduced a model dedicated to persistent faults and a novel analysis known as PFA. In this model, the S-box is assumed to be implemented as a lookup table and stored in memory. A single fault on an S-box element v alters this value to the faulty value $v' \neq v$. Since S-box is a permutation, v no longer appears in the S-box, while v' appears twice as often. Consequently, one value will never be observed in each ciphertext byte, and another value will be observed twice as often. This results in a non-uniform probability distribution for each ciphertext byte. If an attacker collects a sufficiently large number of ciphertexts, he can recover the last round key through a statistical analysis. Both the fault value (*i.e.*, $v \oplus v'$) and the fault location (*i.e.*, the position of v) are assumed to be known in this model.

Many follow-up works have been proposed based on the same model introduced by Zhang *et al.* [ZLZ+18]. Carré *et al.* [CGR20] reduced the number of ciphertexts needed for the analysis of AES by applying maximum likelihood estimation. Pan *et al.* [PZRB19] showed that PFA can break higher-order masking schemes with a single persistent fault and showcased on the masked implementations of the AES and PRESENT ciphers. Note that to apply PFA, they assumed that (part of) the masked S-box computation is realized as a lookup table. Gruber *et al.* [GPT19] applied PFA to the authenticated encryption schemes OCB, DEOXYS, and COLM. Xu *et al.* [XZY+21] enhanced PFA by extending the analysis to deeper middle rounds. Cafario and Banik [CB19] constructed PFA on generic Feistel schemes, with an additional requirement for the model that an attacker can collect both correct and faulty ciphertexts, *i.e.*, encrypt a set of plaintexts twice.

At CHES 2020, Zhang *et al.* [ZZJ+20] relaxed the assumption of knowing the fault value and the fault location for the case of a single fault with applications on the AES and PRESENT ciphers. For multiple faults, both [ZLZ+18] and [ZZJ+20] (double faults) presented analysis methods, however, the values and locations of the faults need to be known in both works. This assumption for the case of multiple faults was then relaxed by Engels *et al.* [ESP20] and Soleimany *et al.* [SBH+22] with applications on the AES and LED ciphers. Zheng *et al.* [ZLZ+21] and Zhang *et al.* [ZHF+23] proposed a collision analysis and chosen-plaintext analysis, respectively, which operate under a relatively relaxed model that does not require any information about the fault value, the fault location, or the number of faults.

Fault injection phase. Schmidt *et al.* [SHP09] reported that irradiating ultraviolet (UV) for a few minutes can flip bits from 0 to 1 in various types of non-volatile memory (EPROM, EEPROM, FLASH). They also demonstrated a real attack on AES by faulting an S-box element. In 2014, Kim *et al.* [KDK+14] exposed persistent bit flips on DRAM using the rowhammer technique, successfully inducing errors in most DRAM modules from major manufacturers. Using this technique, Zhang *et al.* [ZLZ+18] faulted the AES' T-tables stored in DRAM in their PFA attack. In [ZZJ+20], Zhang *et al.*'s experiment on the SRAM of an ATmega163L microcontroller showed that a single laser pulse can flip two adjacent bits. Soleimany *et al.* [SBH+22] experimented with electromagnetic fault injection (EMFI) on an STM32F407VG microcontroller. The EM pulse more likely affects multiple S-box elements (3-5 elements for the LED S-box and 4-6 elements for the AES S-box). Grandamme *et al.* [GTB+24] demonstrated that it is feasible to inject faults into S-box elements stored in flash memory, even when the device is powered off. Selmke *et al.* [SBHS15] demonstrated their experiments of flipping bits at precise locations into 90 nm and 45 nm SRAM cells.

2.2 Description of AES

AES [DR05] is a block cipher with a block size of 128 bits. A block (also called a *state*) is an array of 4×4 bytes indexed from 0 to 15. In this work, we consider the 128-bit key variant. It consists of 10 rounds where each round is the composition of the following transformations:

- **SubBytes** (SB): A substitution step where each byte is replaced using an S-box to introduce non-linearity.
- **ShiftRows** (SR): A transposition step where rows of the state are cyclically shifted by a certain number of positions to introduce diffusion.
- **MixColumns** (MC): A mixing step where columns are combined using linear algebra to further diffuse the state.
- **AddRoundKey** (AK): A 16-byte round key derived from the original key is XOR-ed to the state.

The process of deriving round keys from the original key is known as the *key schedule*. As this paper analyzes the fault effects in the key schedule (Section 4), we provide a detailed background of this process. An illustration of the key schedule can be found in Figure 3. The key schedule generates a total of 44 words (*i.e.*, 44 columns, each of 4 bytes) from the 128-bit master key. The master key is first divided into four 4-byte words (similar to the state), forming the initial round key (round 0).

An iterative process is applied to derive the remaining 40 words. For the first word in each group of four, the process begins by taking the last word of the previous group and applying a series of transformations. These include **RotWord**, which cyclically shifts the bytes of the word, and **SubWord**, which substitutes each byte using the S-box. Following this, a round constant (**Rcon**) is XOR-ed with the first byte of the word. The result is then XOR-ed with the word located four positions earlier to produce the new word. For the subsequent words in the group, the process is simpler. Each word is generated by XOR-ing the previous word with the word from four positions earlier. This process is repeated until 40 words are created, which are grouped into 10 sets of four words to form the round keys.

2.3 Fault model

Unlike many previous PFAs, which assume that the S-box table resides in memory, we consider a scenario where the S-box table is dynamically generated at runtime during the initialization phase, before the execution of the first AES operation. Similarly, the round constants are also generated during this phase. An attacker is assumed to have the capability to skip an instruction involved in these generation processes by injecting a clock glitch.

In this paper, we analyze two types of attacks based on this assumption. The first is a *ciphertext-only attack*, where the attacker is limited to accessing the ciphertexts. The second is a *differential attack*, which requires the attacker to access both plaintexts and ciphertexts. Each of these attacks will be detailed in the subsequent sections.

2.4 Experimental setup

For our experiments, we use a ChipWhisperer Lite board featuring an STM32F303 32-bit ARM target microcontroller to realize the clock glitches. Figure 1 shows the ChipWhisperer setup used in this study. The device runs at its default clock frequency of 7.37 MHz and is connected to a MacBook Air M1 via a USB cable.

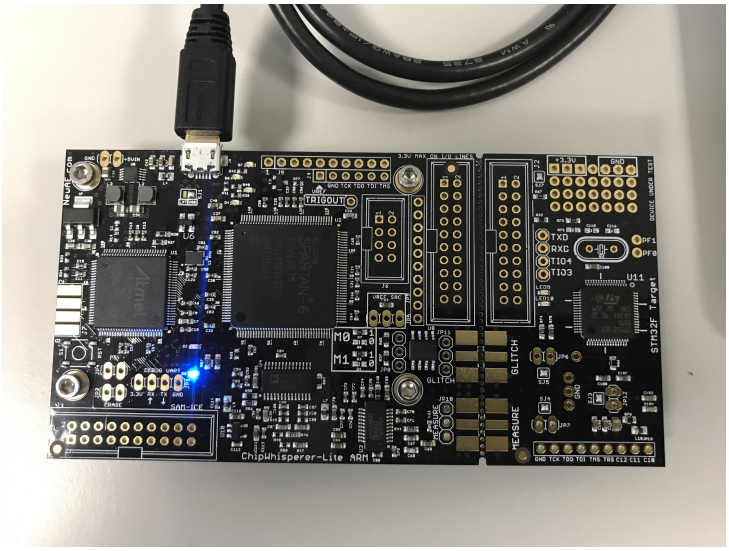

Figure 1: ChipWhisperer Lite used in our experiments.

3 Attack with a fault on S-box generation

In practice, some public embedded cryptographic libraries, such as MbedTLS and cryptlib support generating the S-box table at runtime. This feature is particularly useful when users aim to save ROM usage for statically storing the full table and benefit from faster RAM access. The generation occurs during the initialization phase before the first AES operation. Once generated, the S-box table is stored in RAM and reused across all subsequent AES operations as long as the device is not rebooted. This persistent reuse of the S-box introduces a potential vulnerability. If the S-box generation process is faulted, the resulting fault affects multiple AES operations. By collecting sufficient faulty outputs, it becomes feasible to perform a PFA to recover the key.

Since the analysis phase has been well investigated in the literature, *e.g.* [ZLZ+18, ZZJ+20], we refer to these works for the analysis of the key recovery. In this section, we focus on the injection of a fault in the S-box generation. We begin by detailing in Subsection 3.1 the S-box generation process with a concrete C implementation and point out where the fault can be injected. We then present the experimental results of the fault injection via clock glitching in Subsection 3.2.

3.1 S-box generation

We refer to [DR05] for the mathematical aspect of the S-box. Here, we only focus on the implementation aspect. We use the AES implementation provided by MbedTLS library[3] for our demonstration. Listing 1 shows the extracted C code for generating the forward S-box table (`FSb`). Two additional tables (`pow` and `log`) are involved in this process. The generation of each table is performed with a loop of 256 iterations (see lines 18 and 26). Skipping an instruction within any of these iterations will result in an error in an S-box element. Recall that a single faulty S-box element is sufficient for the key recovery analysis. Consequently, the injected fault does not need to be precise in value or location. Such a fault can be induced using low-cost methods such as a clock glitch injection.

[3]The source code can be found at https://github.com/Mbed-TLS/mbedtls/blob/71c569d44bf3a8bd53d874c81ee8ac644dd6e9e3/library/aes.c#L375. To use the table generation feature, we need to comment the default macros `MBEDTLS_AES_ROM_TABLES`, `MBEDTLS_HAVE_ASM` and `MBEDTLS_AESNI_C` in the configuration file https://github.com/Mbed-TLS/mbedtls/blob/v3.6.1/include/mbedtls/mbedtls_config.h.

In addition, Listing 1 illustrates the generation of four T-tables (`FT0, FT1, FT2, FT3`) derived from the S-box table `FSb` (line 37). These T-tables are instrumental in accelerating computation. Each 8-bit S-box element corresponds to four 32-bit elements distributed across the T-tables. Consequently, to have an equivalent effect as faulting an S-box element, we need to fault all four corresponding elements in the T-tables. This approach is inefficient because it requires targeting multiple elements. However, the generation of the last three T-tables is based on applying a cyclic shift to the first T-table (lines 46-48). Therefore, by faulting the generation process of the first T-table (lines 38-45), it is possible to achieve an equivalent effect, making the fault injection more efficient.

Listing 1: Implementation of S-box generation in C

```
1  #define ROTL8(x) (((x) << 8) & 0xFFFFFFFF) | ((x) >> 24)
2  #define XTIME(x) (((x) << 1) ^ (((x) & 0x80) ? 0x1B : 0x00))
3
4  // Forward S-box & tables
5  static uint8_t  FSb[256];
6  static uint32_t FT0[256];
7  static uint32_t FT1[256];
8  static uint32_t FT2[256];
9  static uint32_t FT3[256];
10
11 static void aes_gen_tables(void){
12     int i;
13     uint8_t x, y, z;
14     uint8_t pow[256];
15     uint8_t log[256];
16
17     // Compute pow and log tables over GF(2^8)
18     for (i = 0, x = 1; i < 256; i++) {
19         pow[i] = x;
20         log[x] = (uint8_t) i;
21         x ^= XTIME(x);
22     }
23
24     // Generate the forward S-box
25     FSb[0x00] = 0x63;
26     for (i = 1; i < 256; i++) {
27         x = pow[255 - log[i]];
28         y  = x; y = (y << 1) | (y >> 7);
29         x ^= y; y = (y << 1) | (y >> 7);
30         x ^= y; y = (y << 1) | (y >> 7);
31         x ^= y; y = (y << 1) | (y >> 7);
32         x ^= y ^ 0x63;
33         FSb[i] = x;
34     }
35
36     // Generate the forward T-tables
37     for (i = 0; i < 256; i++) {
38         x = FSb[i];
39         y = XTIME(x);
40         z = y ^ x;
41
42         FT0[i] = ((uint32_t) y) ^
43                  ((uint32_t) x <<  8) ^
44                  ((uint32_t) x << 16) ^
45                  ((uint32_t) z << 24);
46         FT1[i] = ROTL8(FT0[i]);
47         FT2[i] = ROTL8(FT1[i]);
48         FT3[i] = ROTL8(FT2[i]);
49     }
50 }
```

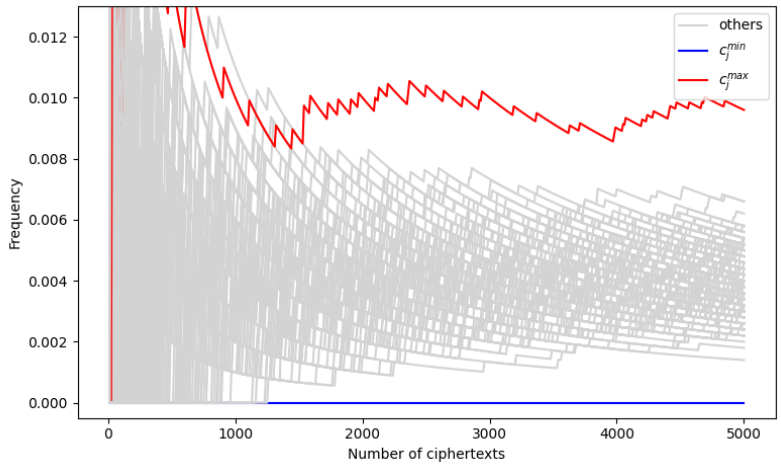

Figure 2: Occurrence probability for 256 values of a ciphertext byte.

3.2 Experiment

We use the ChipWhisperer to perform the fault injection via clock glitches. An additional fast clock cycle is inserted between two ordinary clock cycles during the execution. The width of the induced clock is chosen such that it is short enough to disrupt the correct execution of the current instruction but still recognized by the microprocessor. When the next clock edge arrives, the microprocessor starts executing the next instruction, effectively skipping the current one. Our fault targets the generation of the `pow`, `log` and `FSb` tables, as shown in Listing 1. Notably, we do not require a precise fault on a specific instruction or a specific value during the table generation. Any fault that causes an error in an S-box element is sufficient for the attack.

In this experiment, we use the AES encryption with Electronic Code Book (ECB) mode from the MbedTLS library. The results should be analogous for other modes, as the attack exploits the biased distribution of the ciphertext bytes. After performing the clock glitch, we collect a number of ciphertexts. It is important to note that we consider the PFA with a ciphertext-only scenario, meaning that the attacker does not need to control the plaintexts.

We then compute the occurrence probabilities for all 256 possible values of each ciphertext byte. Figure 2 shows this for a specific ciphertext byte, denoted as c_j, where $0 \leq j \leq 15$. If an S-box element is erroneous, the distribution of c_j reveals that one value (denoted by $c_j^{\min}$) never appears, while another value (denoted by $c_j^{\max}$) appears twice as often. In practice, if this pattern is not observed across all 16 ciphertext bytes $(c_0, \ldots, c_{15})$ after analyzing a sufficiently large number of ciphertexts, the fault injection was likely unsuccessful. In such cases, the fault injection process should be repeated.

In our experiment using the ChipWhisperer, it takes around 30 minutes to find a glitch configuration (offset and width) that causes the fault as desired. This duration is primarily dominated by the calculation of probabilities to check whether an S-box element is erroneous (as shown in Figure 2). However, we note that this time depends on the initial parameters of the glitch. If the initial configuration is already close to the successful one, the time required will be shorter, otherwise, it may take longer.

We now briefly recall the process of recovering the last round key. For further details and optimizations regarding the required number of ciphertexts, we refer to several related works, such as [ZZJ+20, CGR20, XZY+21]. Let $S[i]$ and $S'[i]$ denote the original S-box element that is faulted and its altered value, respectively. The fault value can be expressed

as $f = S[i] \oplus S'[i]$. Since $S[i]$ no longer appears in the S-box table and $c_j^{\min}$ is absent in the distribution of c_j, we deduce that $c_j^{\min} = S[i] \oplus k_j$, where k_j $(0 \leq j \leq 15)$ represents the j-th byte of the last round key. Thus, k_j can be recovered using the following equation:

$$k_j = c_j^{\min} \oplus S[i].$$

Note that the fault location i is unknown to the attacker. However, this can be resolved through brute force by testing all 256 possible locations in the S-box ($i \in [0, 255]$), resulting in 256 candidates for the last round key. We can derive the 256 corresponding master key candidates, as the AES key schedule relies solely on the forward (faulty) S-box table for both key expansion and key reversal. If a correct plaintext-ciphertext pair is available, identifying the correct key becomes straightforward. For ciphertext-only attacks, where such a pair is not accessible, obtaining the unique correct key candidate remains an open question. Many previous works [ZLZ+18, SBH+22, PZRB19, CGR20, XZY+21, GPT19, TL22] ignore this issue by assuming that the round keys are precomputed and unaffected by the faulty S-box. Zhang *et al.* [ZHF+23], however, addressed this issue in the context of a chosen-plaintext attack (not ciphertext-only). In the next section, we also tackle this challenge with an attack that involves injecting a fault into a round constant.

4 Attack with a fault on round constant

In this section, we show that by faulting the 8th round constant in AES, an attacker can recover the key using a differential fault analysis (DFA). We note that the effect of this fault attack is similar to Kim's work [Kim12], which presented a DFA with a fault in the AES key schedule. The main difference is that Kim considered a *transient fault* induced into the first column of the 8th round in the key schedule, whereas we consider a *persistent fault* induced into the 8th round constant. Our analysis is somewhat simpler since the fault value remains the same across executions due to the nature of a persistent fault. Nonetheless, we can directly apply Kim's analysis to recover the last round key. Therefore, we refer to the original analysis of Kim [Kim12] for the key recovery. In this section, we only present the effect of a persistent fault on the 8th round constant, which is the main difference from [Kim12].

4.1 Fault propagation

We consider a persistent fault induced into the 8th round constant in the key schedule. The fault value does not need to be known to the attacker. Suppose the attacker obtains N pairs of correct-faulty ciphertexts encrypted with the same plaintexts. We now describe how the differential effects of the fault propagate through the last three rounds of AES encryption.

Let a denote the difference in the output of the associated XOR operation caused by the fault. Figure 3 illustrates the propagation of a through the 8th, 9th and 10th round keys. Due to the SB transformation, this difference a results in two additional differential values, denoted b and c, in the 9th and 10th round keys. Since the fault is persistent, the values a, b, and c remain the same across all N correct-faulty ciphertext pairs.

Figure 4 depicts the influence of the differences a, b and c in the last three rounds of AES encryption. First, a spreads to the first row of the 8th round key, and thus the output state of the AK in the 8th round (state (5) in Figure 4). Next, the SB transformation causes the changes in the differences of the first row from (a, a, a, a) to (e, f, g, h) (state (6) in Figure 4). Since SB is a non-linear transformation, the differences e, f, g, and h are plaintext-dependent and vary among different correct-faulty ciphertext pairs. Then, the MC transformation spreads them to the four cells of its corresponding column. Now, there is a differential relation of the four cells in each column (state (8) in Figure 4), *e.g.*,

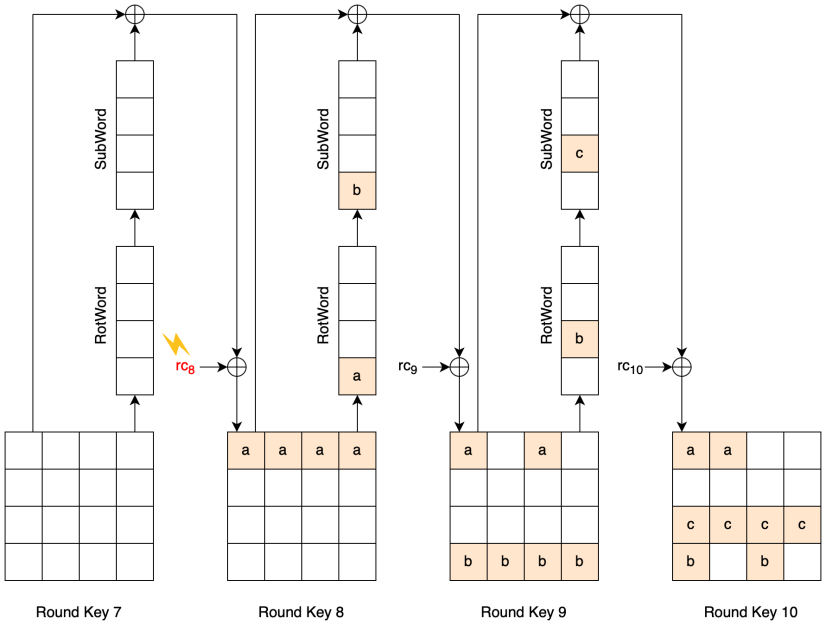

Figure 3: Differential propagation in the key schedule. This figure omits the XOR operations to recursively generate the 2-nd, 3-rd and 4-th columns of each round key for a succinct illustration.

$(2e, e, e, 3e)$ for the first column. The AK of the 9th round key then adds a to two cells of the first row and b to four cells of the last row (state (9) in Figure 4).

Given a pair of correct-faulty ciphertexts, we make hypotheses for bytes of the last round key and for the differences a, b, and c (we only need to make hypotheses for 2 bytes at a time, see [Kim12]). We then compute backward to the state at the beginning of the last round (state (9) in Figure 4). The correct hypothesis will lead to a match for the differential relation in this state. The analysis algorithm is provided in Appendix A, and a similar algorithm can be found in [Kim12].

According to Kim's analysis [Kim12], using $N = 2$ pairs of correct-faulty ciphertexts can reduce the search space of the key to 2^{24} candidates. However, Kim's approach assumes transient faults where the differences a, b, and c vary for each ciphertext pair. In contrast, we exploit a persistent fault where the differences a, b, and c remain consistent for all N pairs. Our simulation shows that with $N = 3$ pairs, we obtain a single candidate, which is the correct key. With $N = 2$ pairs, we obtain around 20 candidates. The correct key is then identified using a correct plaintext-ciphertext from the set of N pairs.

4.2 Discussion

Through this analysis, we reduce the PFAs based on statistical analyses from previous works (*e.g.*, [ZLZ+18, ZZJ+20, SBH+22]), which require hundreds to thousands of ciphertexts, to a DFA needing only 3 pairs of correct-faulty ciphertexts. This approach is much more efficient in terms of time and data complexity for the key recovery. However, DFA does not work in the ciphertext-only context as seen in the previous PFAs. Our attack assumes that the attacker can encrypt a plaintext twice, once with the fault present and once without, to collect a pair of correct-faulty ciphertexts. In practice, an attacker can collect a set of correct ciphertexts before injecting faults. This is common in the case of a fault by a clock glitch which we will present in the next section. Note that this assumption, where the

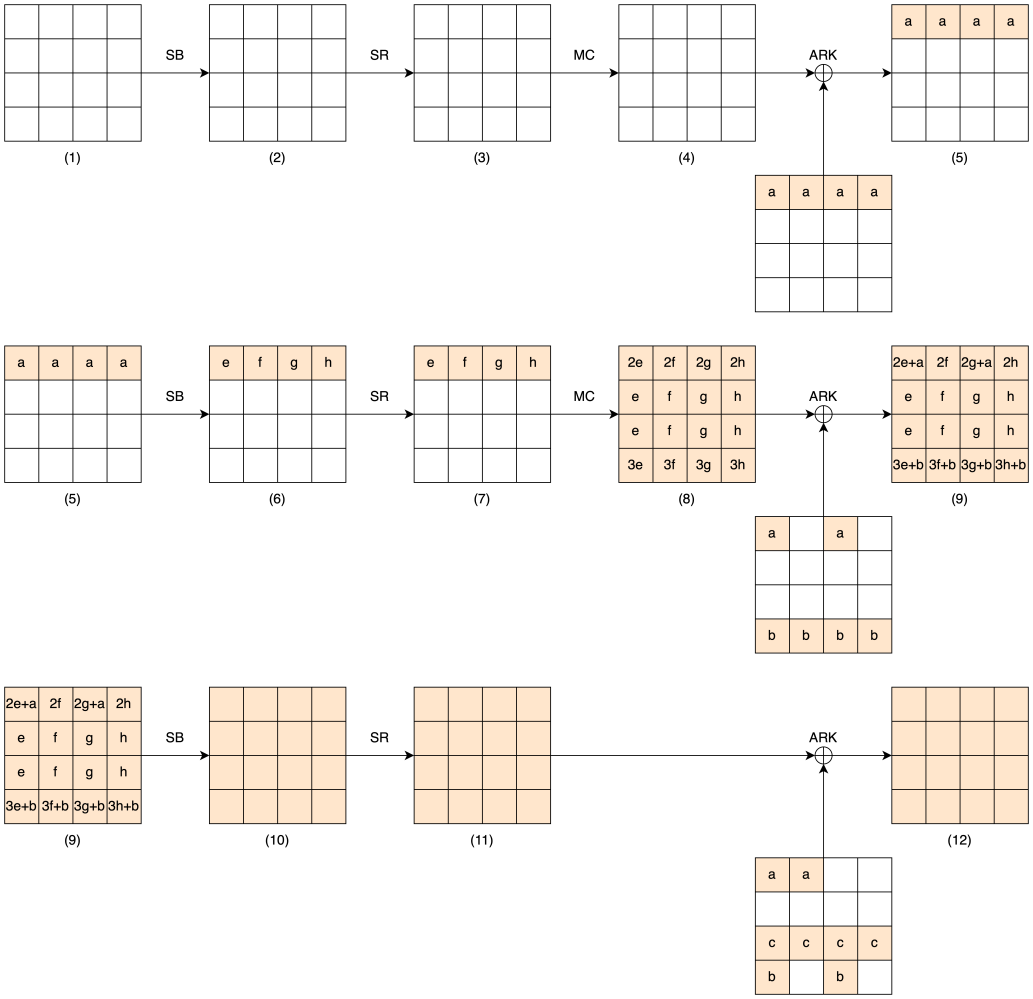

Figure 4: Differential propagation in the last three rounds of AES.

attacker has access to the plaintexts, is also used in [CB19, ZLZ+21, ZHF+23].

Another advantage of our analysis is that it relaxes the assumption in previous works. These works assumed that the key schedule is unaffected by the fault. In studies such as the PFA by Zhang *et al.* [ZLZ+18] and many follow-up works [SBH+22, PZRB19, CGR20, XZY+21, GPT19, TL22], it was assumed that the round keys are precomputed, ensuring that the faulty S-box does not impact the key schedule. This assumption guarantees that the recovered last round key is correct and simplifies the reversal of the master key. However, this does not always hold in practice. For example, each round key is often computed on the fly in smartcard implementations. Our DFA, which targets a fault in a round constant rather than S-box elements, is not affected by this limitation.

The third advantage is its applicability to implementations that do not use a lookup table for the S-box, as it targets a round constant instead. It is worth noting that the assumption of a table-based implementation is common in prior works (*e.g.*, [ZLZ+18, ZZJ+20, SBH+22, ZHF+23]). However, such implementations are known to pose significant risks of side-channel attacks when the CPU relies on caches [Ber05]. This vulnerability has led to the development of modern ciphers, such as Ascon. Recently selected by NIST as the lightweight cipher standard, Ascon is designed to support efficient bit-sliced implementations, avoiding reliance on a lookup table for S-box operations. For

AES, there are also implementations that do not use lookup tables, such as bit-sliced implementation [RSD06], threshold implementation [MPL+11].

4.3 Experiment

Similar to the S-box table, there are two approaches for implementing the round constant table. The first approach is to store it as a lookup table in memory, *e.g.*, ROM. In this scenario, an attacker can inject a persistent fault into the memory, as done in many PFAs in the literature (*e.g.*, [ZLZ+18, ZZJ+20]). The second approach is to generate the round constants at runtime during the initialization phase before the first AES operation. In this case, skipping an instruction by a clock glitch is sufficient to alter the targeted round constant. The altered constant remains faulty until the device is rebooted. In this experiment, we focus on the latter approach.

As before, we use the AES encryption from MbedTLS. Listing 2 shows the extracted implementation of the round constant generation from this library.[4] Recall that the target of our fault is the 8th round constant.

Listing 2: Code of round constant generation in C

```
#define XTIME(x) (((x) << 1) ^ (((x) & 0x80) ? 0x1B : 0x00))
static uint32_t round_constants[10];

for (i = 0, x = 1; i < 10; i++) {
    round_constants[i] = x;
    x = XTIME(x);
}
```

Before the fault injection phase, we encrypt three chosen (possibly random) plaintexts and collect their corresponding correct ciphertexts. The device is then rebooted. A clock glitch injection is performed during the initialization phase by inserting an additional fast clock cycle to skip a proper instruction. We subsequently encrypt the same three plaintexts again and collect their corresponding faulty ciphertexts. The three correct-faulty ciphertext pairs are then analyzed to recover the last round key. If the analysis does not yield a unique key candidate, the fault injection was likely unsuccessful in altering the targeted round key. In such cases, the attacker must repeat the fault injection.

In our experiment using the ChipWhisperer, it typically takes around 3 minutes to find a suitable configuration for the glitch (offset and width) that results in a successful fault injection. Note that this duration depends on the initial configuration's proximity to the successful configuration. If the initial parameters are close to the optimal values, the time to achieve a successful fault injection is shorter. However, if the starting configuration is far from the optimal settings, it may take longer to find the desired outcome.

5 Conclusion and perspectives

In this paper, we first demonstrated that a PFA attack on AES can be carried out using a low-cost fault injection technique, such as a clock glitch. To achieve this, we focused on the implementations that generate the S-box table at runtime before executing the first AES operation. Through an experiment on the ChipWhisperer platform, using the AES implementation in the MbedTLS library, we showed that skipping a single instruction via a clock glitch is enough to induce a faulty S-box element and enable the key recovery.

Second, we introduced the first PFA attack that does not rely on faulting the S-box table. Instead, we demonstrated that inducing a fault in a round constant can enable key recovery through a differential analysis. This approach significantly reduces the data

[4] The source code can be found at https://github.com/Mbed-TLS/mbedtls/blob/71c569d44bf3a8bd53d874c81ee8ac644dd6e9e3/library/aes.c#L394

requirements of previous works based on Zhang *et al.*'s model, which typically analyze hundreds to thousands of ciphertexts. In contrast, our differential analysis needs only three correct-faulty ciphertext pairs.

A potential direction for future work is to investigate efficient countermeasures against PFA. For implementations that rely on tables pre-stored in memory, Error Code Correction (ECC) might provide an effective solution. However, ECC is not capable of detecting errors in implementations where tables are generated at runtime. In such cases, an alternative approach could be to use implementations that do not rely on lookup tables, such as bit-sliced implementations. While this strategy may reduce vulnerability to PFA attacks, it could potentially slow down encryption.

References

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51. Springer, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_4.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525. Springer, Heidelberg, August 1997. doi:10.1007/BFb0052259.

[CB19] Andrea Caforio and Subhadeep Banik. A study of persistent fault analysis. In Shivam Bhasin, Avi Mendelson, and Mridul Nandi, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 13–33, Cham, 2019. Springer International Publishing.

[CGR20] Sébastien Carré, Sylvain Guilley, and Olivier Rioul. Persistent fault analysis with few encryptions. In Guido Marco Bertoni and Francesco Regazzoni, editors, *COSADE 2020*, volume 12244 of *LNCS*, pages 3–24. Springer, Heidelberg, April 2020. doi:10.1007/978-3-030-68773-1_1.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 181–194. Springer, Heidelberg, September 2007. doi:10.1007/978-3-540-74735-2_13.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting ineffective fault inductions on symmetric cryptography. *IACR TCHES*, 2018(3):547–572, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7286. doi:10.13154/tches.v2018.i3.547-572.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on A.E.S. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer, 2003. doi:10.1007/978-3-540-45203-4_23.

[DR05] Joan Daemen and Vincent Rijmen. Rijndael/aes. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005. doi:10.1007/0-387-23483-7_358.

[ESP20] Susanne Engels, Falk Schellenberg, and Christof Paar. SPFA: SFA on multiple persistent faults. In *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, pages 49–56. IEEE, 2020. doi:10.1109/FDTC51366.2020.00014.

[FJLT13] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118, 2013. doi:10.1109/FDTC.2013.18.

[GPT19] Michael Gruber, Matthias Probst, and Michael Tempelmeier. Persistent fault analysis of OCB, DEOXYS and COLM. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*, pages 17–24. IEEE, 2019. doi:10.1109/FDTC.2019.00011.

[GTB+24] Paul Grandamme, Pierre-Antoine Tissot, Lilian Bossuet, Jean-Max Dutertre, Brice Colombier, and Vincent Grosso. Switching off your device does not protect against fault attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):425–450, 2024. URL: https://doi.org/10.46586/tches.v2024.i4.425-450, doi:10.46586/TCHES.V2024.I4.425-450.

[GYTS14] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential fault intensity analysis. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 49–58, 2014. doi:10.1109/FDTC.2014.15.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. doi:10.1109/ISCA.2014.6853210.

[Kim12] Chong Hee Kim. Improved differential fault analysis on aes key schedule. *IEEE Transactions on Information Forensics and Security*, 7(1):41–50, 2012. doi:10.1109/TIFS.2011.2161289.

[LSG+10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 320–334. Springer, Heidelberg, August 2010. doi:10.1007/978-3-642-15031-9_22.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, Heidelberg, May 2011. doi:10.1007/978-3-642-20465-4_6.

[PQ03] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 77–88. Springer, Heidelberg, September 2003. doi:10.1007/978-3-540-45238-6_7.

[PZRB19] Jingyu Pan, Fan Zhang, Kui Ren, and Shivam Bhasin. One fault is all it needs: Breaking higher-order masking with persistent fault analysis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2019. doi:10.23919/DATE.2019.8715260.

[Riv09] Matthieu Rivain. Differential fault analysis on DES middle rounds. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 457–469. Springer, Heidelberg, September 2009. doi:10.1007/978-3-642-04138-9_32.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In David Pointcheval, Yi Mu, and Kefei Chen, editors, *CANS 06*, volume 4301 of *LNCS*, pages 203–212. Springer, Heidelberg, December 2006.

[SBH+22] Hadi Soleimany, Nasour Bagheri, Hosein Hadipour, Prasanna Ravi, Shivam Bhasin, and Sara Mansouri. Practical multiple persistent faults analysis. *IACR TCHES*, 2022(1):367–390, 2022. doi:10.46586/tches.v2022.i1.367-390.

[SBHS15] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. Precise laser fault injections into 90 nm and 45 nm SRAM-cells. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2015. doi:10.1007/978-3-319-31271-2_12.

[SBR+20] Sayandeep Saha, Arnab Bag, Debapriya Basu Roy, Sikhar Patranabis, and Debdeep Mukhopadhyay. Fault template attacks on block ciphers exploiting fault propagation. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 612–643. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45721-1_22.

[SHP09] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical fault attacks on AES: A threat in violet. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 13–22, 2009. doi:10.1109/FDTC.2009.37.

[TL22] Honghui Tang and Qiang Liu. MPFA: An efficient multiple faults-based persistent fault analysis method for low-cost FIA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(9):2821–2834, 2022. doi:10.1109/TCAD.2021.3117512.

[XZY+21] Guorui Xu, Fan Zhang, Bolin Yang, Xinjie Zhao, Wei He, and Kui Ren. Pushing the limit of PFA: Enhanced persistent fault analysis on block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1102–1116, 2021. doi:10.1109/TCAD.2020.3048280.

[ZFL+22] Fan Zhang, Tianxiang Feng, Zhiqi Li, Kui Ren, and Xinjie Zhao. Free fault leakages for deep exploitation: Algebraic persistent fault analysis on lightweight block ciphers. *IACR TCHES*, 2022(2):289–311, 2022.

[ZHF+23] Fan Zhang, Run Huang, Tianxiang Feng, Xue Gong, Yulong Tao, Kui Ren, Xinjie Zhao, and Shize Guo. Efficient persistent fault analysis with small number of chosen plaintexts. *IACR TCHES*, 2023(2):519–542, 2023. doi:10.46586/tches.v2023.i2.519-542.

[ZLZ+18] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *IACR TCHES*, 2018(3):150–172, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7272. doi:10.13154/tches.v2018.i3.150-172.

[ZLZ+21] Shihui Zheng, Xudong Liu, Shoujin Zang, Yihao Deng, Dongqi Huang, and Changhai Ou. A persistent fault-based collision analysis against the advanced encryption standard. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1117–1129, 2021. doi:10.1109/TCAD.2021.3049687.

[ZZJ+20] Fan Zhang, Yiran Zhang, Huilong Jiang, Xiang Zhu, Shivam Bhasin, Xinjie Zhao, Zhe Liu(0), Dawu Gu, and Kui Ren. Persistent fault attack in practice. *IACR TCHES*, 2020(2):172–195, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8548. doi:10.13154/tches.v2020.i2.172-195.

A Key recovery algorithm for fault in round constant

Let $(C_j^i, \tilde{C}_j^i)$ be the i-th pair of correct-faulty ciphertext byte at index j, where $i \in [1, N]$ and $j \in [0, 15]$. Let ΔS_j be the difference of the j-th byte in the state at the beginning of the last round (state (9) in Figure 4). Let K_j and $\hat{K}_j$ be the correct value and the hypothesis for byte at index j of the last round key. To recover K, we perform the following algorithm:

1. Recover (K_{12}, K_9): For each candidate $(\hat{K}_{12}, \hat{K}_9)$ of 2^{16} possibilities, we compute

$$\Delta S_{12} = \mathsf{SB}^{-1}(C_{12}^i \oplus \hat{K}_{12}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{12} \oplus \hat{K}_{12})$$
$$\Delta S_{13} = \mathsf{SB}^{-1}(C_9^i \oplus \hat{K}_9) \oplus \mathsf{SB}^{-1}(\tilde{C}_9 \oplus \hat{K}_9)$$

   If $\Delta S_{12} = 2 \cdot \Delta S_{13}$ for every $i \in [1, N]$, then $(\hat{K}_{12}, \hat{K}_9)$ is a good candidate.

2. Recover (K_6, c): For each candidate $(\hat{K}_{16}, \hat{c})$ of 2^{16} possibilities, we compute

$$\Delta S_{14} = \mathsf{SB}^{-1}(C_6^i \oplus \hat{K}_6) \oplus \mathsf{SB}^{-1}(\tilde{C}_6 \oplus \hat{K}_6 \oplus c)$$
$$\Delta S_{13} = \mathsf{SB}^{-1}(C_9^i \oplus K_9) \oplus \mathsf{SB}^{-1}(\tilde{C}_9 \oplus K_9)$$

   If $\Delta S_{14} = \Delta S_{13}$ for every $i \in [1, N]$, then $(\hat{K}_6, \hat{c})$ is a good candidate.

3. Recover (K_3, b): For each candidate $(\hat{K}_3, \hat{b})$ of 2^{16} possibilities, we compute

$$\Delta S_{15} = \mathsf{SB}^{-1}(C_3^i \oplus \hat{K}_3) \oplus \mathsf{SB}^{-1}(\tilde{C}_3 \oplus \hat{K}_3 \oplus b)$$
$$\Delta S_{13} = \mathsf{SB}^{-1}(C_9^i \oplus K_9) \oplus \mathsf{SB}^{-1}(\tilde{C}_9 \oplus K_9)$$

   If $\Delta S_{15} = 3 \cdot \Delta S_{13} \oplus b$ for every $i \in [1, N]$, then $(\hat{K}_3, \hat{b})$ is a good candidate.

4. Recover (K_5, K_{15}): For each candidate $(\hat{K}_5, \hat{K}_{15})$ of 2^{16} possibilities, we compute

$$\Delta S_9 = \mathsf{SB}^{-1}(C_5^i \oplus \hat{K}_5) \oplus \mathsf{SB}^{-1}(\tilde{C}_5 \oplus \hat{K}_5)$$
$$\Delta S_{11} = \mathsf{SB}^{-1}(C_{15}^i \oplus \hat{K}_{15}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{15} \oplus \hat{K}_{15})$$

   If $\Delta S_{11} = 3 \cdot \Delta S_9 \oplus b$ for every $i \in [1, N]$, then $(\hat{K}_5, \hat{K}_{15})$ is a good candidate.

5. Recover (K_8, a): For each candidate $(\hat{K}_8, \hat{a})$ of 2^{16} possibilities, we compute

$$\Delta S_9 = \mathsf{SB}^{-1}(C_5^i \oplus K_5) \oplus \mathsf{SB}^{-1}(\tilde{C}_5 \oplus K_5)$$
$$\Delta S_8 = \mathsf{SB}^{-1}(C_8^i \oplus \hat{K}_8) \oplus \mathsf{SB}^{-1}(\tilde{C}_8 \oplus \hat{K}_8)$$

   If $\Delta S_8 = 2 \cdot \Delta S_9 \oplus a$ for every $i \in [1, N]$, then $(\hat{K}_8, \hat{a})$ is a good candidate.

6. Recover K_2: For each candidate $\hat{K}_2$ of 2^8 possibilities, we compute

$$\Delta S_9 = \mathsf{SB}^{-1}(C_5^i \oplus K_5) \oplus \mathsf{SB}^{-1}(\tilde{C}_5 \oplus K_5)$$
$$\Delta S_{10} = \mathsf{SB}^{-1}(C_2^i \oplus \hat{K}_2) \oplus \mathsf{SB}^{-1}(\tilde{C}_2 \oplus \hat{K}_2 \oplus c)$$

   If $\Delta S_{10} = \Delta S_9 \oplus a$ for every $i \in [1, N]$, then $\hat{K}_2$ is a good candidate.

7. Recover (K_4, K_1): For each candidate $(\hat{K}_4, \hat{K}_1)$ of 2^{16} possibilities, we compute

$$\Delta S_4 = \mathsf{SB}^{-1}(C_4^i \oplus \hat{K}_4) \oplus \mathsf{SB}^{-1}(\tilde{C}_4 \oplus \hat{K}_4 \oplus a)$$
$$\Delta S_1 = \mathsf{SB}^{-1}(C_1^i \oplus \hat{K}_1) \oplus \mathsf{SB}^{-1}(\tilde{C}_1 \oplus \hat{K}_1)$$

   If $\Delta S_4 = 2 \cdot \Delta S_1 \oplus b$ for every $i \in [1, N]$, then $(\hat{K}_4, \hat{K}_1)$ is a good candidate.

8. Recover (K_{14}, K_{11}): For each candidate $(\hat{K}_{14}, \hat{K}_{11})$ of 2^{16} possibilities, we compute

$$\Delta S_6 = \mathsf{SB}^{-1}(C_{14}^i \oplus \hat{K}_{14}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{14} \oplus \hat{K}_{14} \oplus c)$$
$$\Delta S_7 = \mathsf{SB}^{-1}(C_{11}^i \oplus \hat{K}_{11}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{11} \oplus \hat{K}_{11} \oplus b)$$

   If $\Delta S_7 = 3 \cdot \Delta S_6 \oplus b$ for every $i \in [1, N]$, then $(\hat{K}_{14}, \hat{K}_{11})$ is a good candidate.

9. Recover (K_0, K_{13}): For each candidate $(\hat{K}_0, \hat{K}_{13})$ of 2^{16} possibilities, we compute

$$\Delta S_0 = \mathsf{SB}^{-1}(C_0^i \oplus \hat{K}_0) \oplus \mathsf{SB}^{-1}(\tilde{C}_0 \oplus \hat{K}_0 \oplus a)$$
$$\Delta S_1 = \mathsf{SB}^{-1}(C_{13}^i \oplus \hat{K}_{13}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{13} \oplus \hat{K}_{13})$$

   If $\Delta S_0 = 2 \cdot \Delta S_1 \oplus a$ for every $i \in [1, N]$, then $(\hat{K}_0, \hat{K}_{13})$ is a good candidate.

10. Recover (K_{10}, K_7): For each candidate $(\hat{K}_{10}, \hat{K}_7)$ of 2^{16} possibilities, we compute

$$\Delta S_2 = \mathsf{SB}^{-1}(C_{10}^i \oplus \hat{K}_{10}) \oplus \mathsf{SB}^{-1}(\tilde{C}_{10} \oplus \hat{K}_{10} \oplus c)$$
$$\Delta S_3 = \mathsf{SB}^{-1}(C_7^i \oplus \hat{K}_7) \oplus \mathsf{SB}^{-1}(\tilde{C}_7 \oplus \hat{K}_7)$$

    If $\Delta S_3 = 3 \cdot \Delta S_2 \oplus b$ for every $i \in [1, N]$, then $(\hat{K}_{10}, \hat{K}_7)$ is a good candidate.