



Practical Persistent Fault Attacks on AES with Instruction Skip

Viet Sang Nguyen , Vincent Grosso  and Pierre-Louis Cayrel 

Université Jean Monnet Saint-Etienne, CNRS, Institut d'Optique Graduate School, Laboratoire
Hubert Curien UMR 5516, Saint-Etienne, France

Abstract. Persistent Fault Attacks (PFA) have 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 the literature, such persistent faults primarily induced by bit flips in storage, often achieved through laser fault injection techniques. In this paper, we demonstrate that persistent faults can also be induced through instruction skips, which can easily be achieved with almost any fault injection methods (e.g., voltage/clock glitching, electromagnetism). Specifically, we target AES implementations that dynamically generate the S-box table at runtime, during the initialization phase, 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. Secondly, we introduce, to our knowledge, the first PFA that targets a constant other than the S-box elements. We show that faulting a round constant involved in the AES key schedule is sufficient to recover the key by a differential analysis. Compared to previous PFAs that rely on statistical analysis requiring hundreds to thousands of ciphertexts, our approach needs only three correct-faulty ciphertexts pairs. We showcase this attack with an experiment on the MbedTLS AES implementation, using a clock glitch in the round constant generation.

Keywords: Persistent Fault Attacks · Instruction Skip · Clock 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. A fault attack typically involves two main steps: fault injection and fault analysis. In the first step, the attacker deliberately injects faults into the target device to disrupt the algorithm's execution. This can be done using techniques like laser pulses, electromagnetic interference, voltage/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)



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.

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]. 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 than once 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]. Among the previous works, only a few focus on fault injection experiments [ZLZ⁺18, SHP09, ZZJ⁺20, SBH⁺22, GTB⁺24], while the others primarily focus on analysis based on assumptions about the faults.

We make the first key observation: existing PFAs are all based on the assumption that the S-box is implemented as a lookup table stored in memory, with one or more elements being faulted (as modeled by Zhang *et al.* [ZLZ⁺18]). In practice, this table is typically stored statically in FLASH/ROM and then copied to RAM in runtime. Some works target inducing faults directly in FLASH [GTB⁺24] or ROM [SHP09], while others focus on inducing faults in SRAM [ZZJ⁺20], DRAM [ZLZ⁺18], or during the table transfer from FLASH to RAM [SBH⁺22]. However, a static S-box table stored in FLASH is not always the case in practice. In embedded systems, there is often a question about using FLASH and RAM for table storage. Storing lookup tables in FLASH consumes a certain amount of permanent memory space. To reduce this, tables can be generated on-the-fly. Table generation helps save FLASH space, while the RAM usage remains unchanged, whether the table is copied from FLASH or generated at runtime. For that reason, many embedded systems use the table generation approach. In reality, some embedded cryptographic libraries, such as MbedTLS¹ and cryptlib,² offer this strategy for their AES implementations.

Our second key observation is that persistent faults in the literature are primarily induced by bit flips in memory, often achieved through laser-based techniques [ZZJ⁺20, SHP09, GTB⁺24]. Laser-based fault injection is highly complex and requires specialized, expensive equipment, costing hundreds of thousands of dollars [BH22]. It also demands high-precision laser pulses with proper intensity and focus, as well as chip decapsulation without causing damage. Another approach for inducing persistent faults is electromagnetic fault injection (EMFI), as demonstrated in [SBH⁺22]. The experimental setup in [SBH⁺22] is also quite complex. It requires precise control over high-voltage pulses (up to 200V) with very low rise time (less than 4ns) to avoid damaging the device, expertise in EM

¹<https://github.com/Mbed-TLS/mbedtls>, version 3.6.1

²<https://www.cs.auckland.ac.nz/~pgut001/cryptlib/>, version 3.4.8

probe customization, and accurate probe positioning for effective fault injection.

Meanwhile, instruction skip, a simple technique easily achievable with almost any fault injection techniques (*e.g.*, voltage/clock glitch, laser, EM), has not yet been explored for inducing persistent faults in the literature. In addition to being easy to implement, instruction skip can also be achieved with very low-cost equipment, for example, using clock glitch for about 130 dollars or EM for about 10 dollars (see [BH22]). This leads us to the following research question: *Can we perform a PFA attack with an instruction skip (that is easy to insert)?* Targeting faults in S-box elements stored in memory, as done in previous works, may seem infeasible with this approach. However, considering our first key observation about implementations that generate table at runtime, we could instead focus on skipping an instruction during the table generation phase.

Our third key observation is that existing 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 a PFA attack on AES can be performed using an instruction skip. This is particularly effective against implementations that generate the S-box table at runtime, during the initialization phase, before executing the first AES operation. 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.

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 constant other than S-box elements. Specifically, we consider a persistent fault induced on a round constant involved in the AES key schedule. 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 a statistical analysis on hundreds to thousands of ciphertexts. In contrast, our differential analysis needs only three correct-faulty ciphertext pairs.

Similar to previous PFAs with S-box table (based on Zhang *et al.*'s model), the round constants are assumed to be implemented as a lookup table stored in memory, which can be targeted by memory faults (*e.g.*, bit flips by laser as in [ZZJ⁺20, GTB⁺24]). However, as this work focuses on implementations that generate the tables at runtime, we use an instruction skip to demonstrate the attack on the AES implementation in MbedTLS, where round constants are generated during the initialization phase. By injecting a clock glitch to skip an instruction during the round constant generation, we obtain the desired faulty round constant for key recovery.

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 based on assumptions about the faults, 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://github.com/nvietsang/pfa-inskip>.

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. Section 5 discusses about countermeasures. Finally, Section 6 concludes our work.

Table 1: Comparison with previous PFAs on AES.

	Fault target	Fault injection technique
[ZLZ+18]	T-table elements stored in DRAM	Flipping bits by rowhammer
[ZZJ+20]	S-box table stored in SRAM	Flipping bits by laser
[SHP09]	S-box table stored in ROM	Flipping bits by laser
[GTB+24]	S-box table stored in FLASH memory	Flipping bits by laser
[SBH+22]	Transfer of S-box table from FLASH to RAM	Faults during transfer by electromagnetic
This work Section 3	S-box table generation	Skipping an instruction by clock glitch
This work Section 4	Round constant table generation	Skipping an instruction by clock glitch

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. Finally, we describe the setup for our experiments.

2.1 Related PFAs

Fault analysis 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. Caforio 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

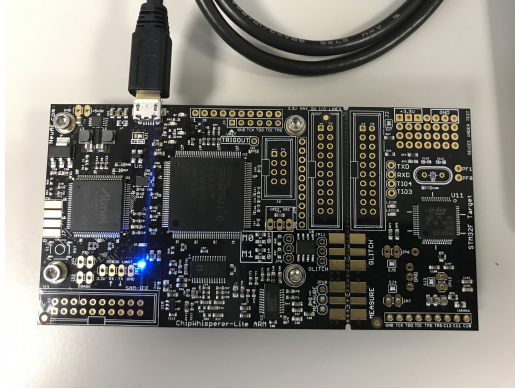


Figure 1: ChipWhisperer Lite used in our experiments.

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 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.

3 Attack with a fault on S-box generation

In practice, some public embedded cryptographic libraries, such as MbedTLS and cryptlib, support the S-box table generation at runtime. This feature is particularly useful when users aim to save ROM/FLASH 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 reset. 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 describing the fault model in Subsection 3.1. Next, we detail in Subsection 3.2 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 3.3.

3.1 Fault model

We consider AES implementations that generate the S-box table at runtime during the initialization phase, before the execution of the first AES encryption. The fault model is

summarized as follows:

- The attacker can fault an S-box element by an instruction skip in the S-box table generation during the initialization phase.
- The fault does not need to be precise either value or location. Knowing that an S-box value has changed is sufficient for the attack (we will later show how to verify that the fault injection is successful).
- The injected fault is persistent, *i.e.*, the affected S-box element remains faulty across encryptions until the device is reset.
- The attacker can collect the multiple ciphertexts.

We note that the attack presented in this section is a *ciphertext-only attack*. Therefore, the attacker does not need access to the plaintexts.

3.2 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 library³ 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 i -th S-box element, FSb[i] for $1 \leq i \leq 255$, is computed based the two values log[i] and pow[$255 - \log[i]$]. The generation of the pow and log tables is performed with a loop of 256 iterations in line 18. The computation of the FSb is then performed with a loop of 255 iterations in line 26, except for the first element FSb[0], which is directly assigned in line 25. To obtain a faulty S-box element, one can cause an instruction skip during an iteration in the generation of the FSb table, or log table, or pow table.

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;

```

³The 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 deactivate 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.

```

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 }

```

3.3 Experiment

We use the ChipWhisperer to perform the fault injection via a clock glitch. 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. We now discuss how to verify, using these ciphertexts, whether the fault has been successfully injected. Let us consider what can be observed when an S-box element is faulted as intended. We 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, \dots, c_{15}) after analyzing a sufficiently large number of

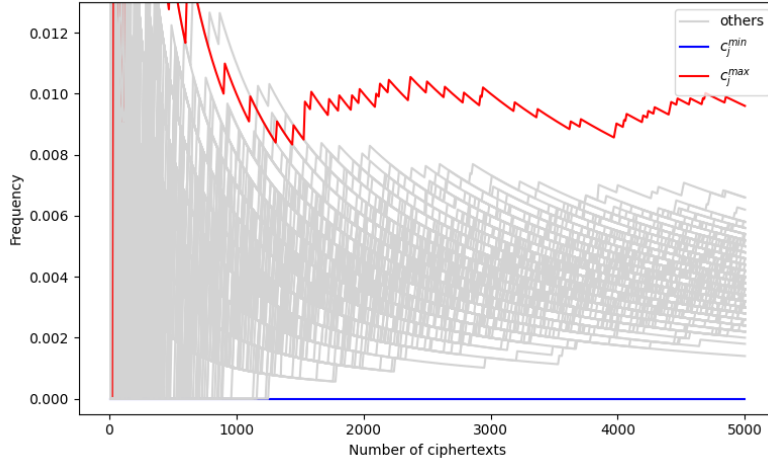


Figure 2: Occurrence probability for 256 values of a ciphertext byte.

ciphertexts, the fault injection was likely unsuccessful. In such cases, the fault injection process should be repeated with different parameters of glitch.

In summary, the attack is automated with the following steps:

1. Perform the glitch with chosen values for the parameters (delay, offset and width).
2. Request the device to do encryption N times and collect N ciphertexts.
3. Check if we can observe (c_j^{\min}, c_j^{\max}) as described above:
 - If yes, proceed with the key recovery.
 - If no, reset the device, go back to step 1 and try with different parameter values.

In our experiment using the ChipWhisperer, it takes around 30 minutes identify a glitch configuration 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 (step 3). However, we note that this time depends on the initial parameter values of the glitch. If the initial configuration is already close to the successful ones, 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 is a challenge.

Zhang *et al.* [ZHF⁺23] addressed this issue, but in the context of a chosen-plaintext attack (not ciphertext-only).

4 Attack with a fault on round constant

In this section, we show that S-box elements are not the only target for PFA attacks. We demonstrate 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 focus on the effect of a persistent fault on the 8th round constant, which is the main difference from [Kim12].

4.1 Fault model

As in previous PFAs with S-box table based on Zhang *et al.*’s model [ZLZ⁺18], the round constants are assumed to be stored as a lookup table in memory. The attacker can induce a memory fault in an element of the table, for example, by flipping bits using laser, as in [ZZJ⁺20, GTB⁺24].

In this work, we consider AES implementations that generate the tables at runtime, including the table of round constants. We still demonstrate a persistent fault using an instruction skip. The fault model is summarized as follows:

- The attacker can fault the 8th round constant by skipping an instruction during the round constant table generation in the initialization phase.
- The fault value does not need to be known. However, the fault location must be precise, meaning the 8th round constant is specifically targeted. (We will later show how to verify the success of the fault injection.)
- The injected fault is persistent, *i.e.*, the affected round constant remains faulty across multiple encryptions until the device is reset.
- The attacker has access to both plaintexts and ciphertexts, as also assumed in previous PFAs [ZHF⁺23, ZLZ⁺21].
- The attacker can operate the device in two scenarios: with and without the fault, to collect correct and faulty ciphertexts. Note that this assumption is also used in previous PFA [CB19].

The attack presented in this section is a *differential attack*. The attacker needs to collect several pairs of correct-faulty ciphertexts encrypted with the same plaintexts.

4.2 Fault propagation

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 correct-faulty ciphertext pairs.

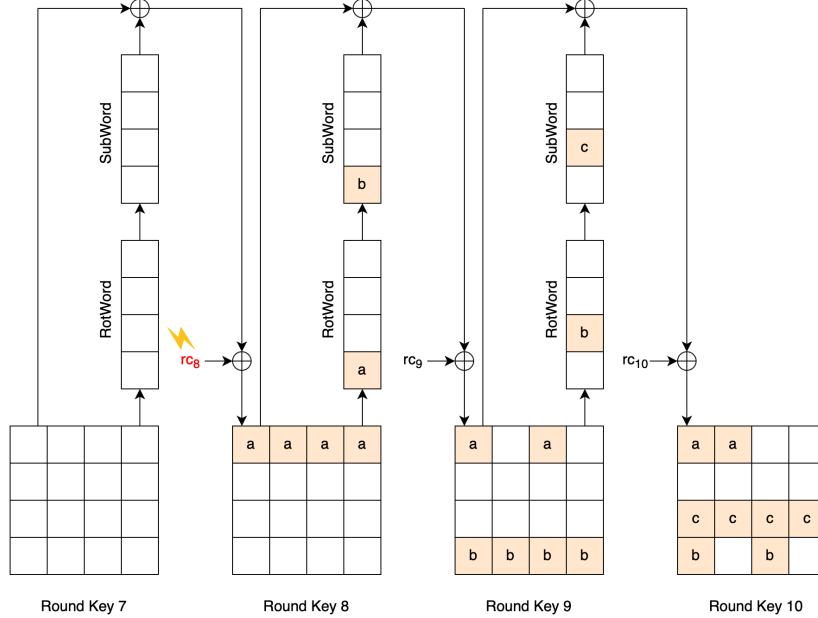


Figure 3: Differential propagation in the key schedule. This figure omits the XOR operations to recursively generate the 2-nd, 3-rd and 4-th columns of each round key for a succinct illustration.

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.*, $(2e, e, e, 3e)$ 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] or Appendix A). 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.

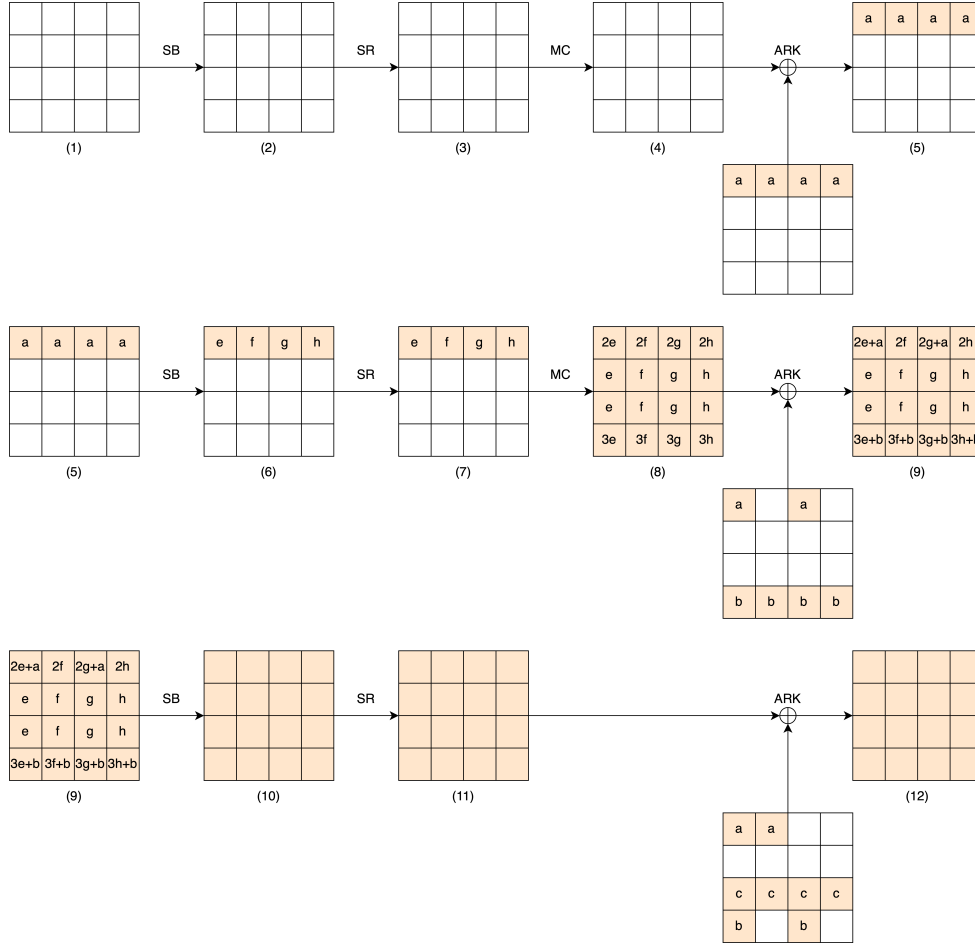


Figure 4: Differential propagation in the last three rounds of AES.

4.3 Experiment

As before, we use the AES implementation from MbedTLS for demonstration. Our goal is to skip an instruction in the generation of the round constants table during the initialization phase using a clock glitch. 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 ($i = 7$).

Listing 2: Code of round constant generation in C

```

1 #define XTIME(x) (((x) << 1) ^ (((x) & 0x80) ? 0x1B : 0x00))
2 static uint32_t round_constants[10];
3
4 for (i = 0, x = 1; i < 10; i++) {
5     round_constants[i] = x;
6     x = XTIME(x);
7 }

```

Before the fault injection, we encrypt three chosen (possibly random) plaintexts and collect their corresponding correct ciphertexts. The device is then reset. A clock glitch injection is performed during the initialization phase by inserting an additional fast clock

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

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. This is also used as the verification to know whether the fault is successful injected as desired.

In summary, the attack is automated with the following steps:

1. Encrypt N chosen (possibly random) plaintexts and collect N corresponding correct ciphertexts.
2. Reset the device.
3. Perform the glitch with chosen values for the parameters (delay, offset and width).
4. Encrypt the same N plaintexts and collect N corresponding ciphertexts.
5. Check if we can obtain a unique key candidate when performing the key recovery on the N pairs of ciphertexts.
 - If yes, perform the inverse key schedule and return the recovered master key.
 - If no, go back to step 2 and try with different parameter values.

In our experiment using the ChipWhisperer, it typically takes around 3 minutes to find a suitable configuration for the glitch parameters (delay, 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 achieve the desired outcome.

4.4 Discussion

This analysis reduces 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 data complexity for the key recovery.

A limitation of this attack is that DFA does not work in the ciphertext-only context. Our attack assumes that the attacker can encrypt the same plaintext twice, once with the fault and once without, to collect a pair of correct-faulty ciphertexts. In practice, the attacker can first collect the correct ciphertexts, then reset the device, perform the fault injection and collect the faulty ciphertexts. This approach is feasible if the implementation allows re-initialization, where the tables are freed and then recreated in memory. Some configurations of the MbedTLS library allow this. However, in configurations where the initialization occurs only once, the attacker cannot collect the both correct and faulty ciphertexts. This difficulty in switching between faulty and non-faulty modes might restrict the applicability of DFA.

Another advantage of this DFA attack 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 [DEMS21]. 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 is also bit-sliced implementation that does not use S-box lookup tables [RSD06].

5 Countermeasure discussion

In this section, we discuss the applicability of common countermeasures against fault attacks in the PFA context. These countermeasures include Error Correction Codes (ECC) in memory and Dual Modular Redundancy (DMR).

5.1 ECC

For systems that use static tables of constants (S-box elements, round constants), ECC can be an effective method for detecting and correcting faults. In this method, redundant bits are stored alongside the data in the memory during the write process. These bits are typically derived from a linear relationship with the data, enabling the detection and correction of errors. When the data is read, the memory controller verifies this relationship and corrects any detected errors before passing the data to the processor. As a result, a fault persists only until the affected element is accessed in ECC-protected memory. No faulty ciphertexts are produced, making statistical analyses relying on faulty ciphertexts infeasible.

5.2 DMR

ECC may not be useful for systems that dynamically generate the tables at runtime, as the fault injection targets the generation process rather than the memory. In such cases, we consider a popular DMR countermeasure, where a ciphertext C is obtained by encrypting a plaintext P , then decrypted to yield P' for comparison with P . A fault is detected if the decrypted result does not match the original plaintext, *i.e.*, $P \neq P'$.

First, we consider the differential attack with a faulty round constant, as presented in Section 4. The round keys for both encryption and decryption are derived from the master key using the same round constant table, which contains the faulty element. Consequently, the verification $P = P'$ always holds, although the ciphertext C is faulty. Therefore, the countermeasure is not effective in this case.

Second, we consider the statistical attack with a faulty S-box element, as presented in Section 3. The AES S-box consists of 256 elements, and an encryption accesses the S-box 160 times. Note that S-box accesses during the key schedule are excluded from the consideration. This is because both encryption and decryption use the same (faulty) round keys derived from the forward S-box table. If the faulty S-box element is accessed only during the key schedule and not during encryption, the fault remains undetected.

Suppose a fault is injected into an element of the forward S-box table, while the inverse S-box used for decryption remains unaffected. There is a chance that the faulty element may not be accessed during encryption. This is because AES encryption only makes use of 160 out of the 256 elements in the S-box. As a result, the ciphertext could remain correct, and the fault is undetected by the DMR.

We now investigate the maximum number of correct ciphertexts, denoted by ℓ , that can be produced before the fault is detected. Suppose that with random plaintexts, each S-box element is accessed with uniform probability. In ℓ encryptions, there are $160 \cdot \ell$ S-box accesses in total. The probability that there exists an S-box element that is never accessed in ℓ encryptions is

$$\left(1 - \frac{1}{256}\right)^{160 \cdot \ell}.$$

The probability p that every S-box element is accessed at least once in ℓ encryptions is therefore

$$p = 1 - \left(1 - \frac{1}{256}\right)^{160 \cdot \ell}.$$

Figure 5 shows the probability p corresponding to different numbers of encryptions ℓ . We see that $p \approx 0.98$ when $\ell = 6$ and $p \approx 1.00$ when $\ell = 9$. In the worst case, the fault is detected after 9 encryptions.

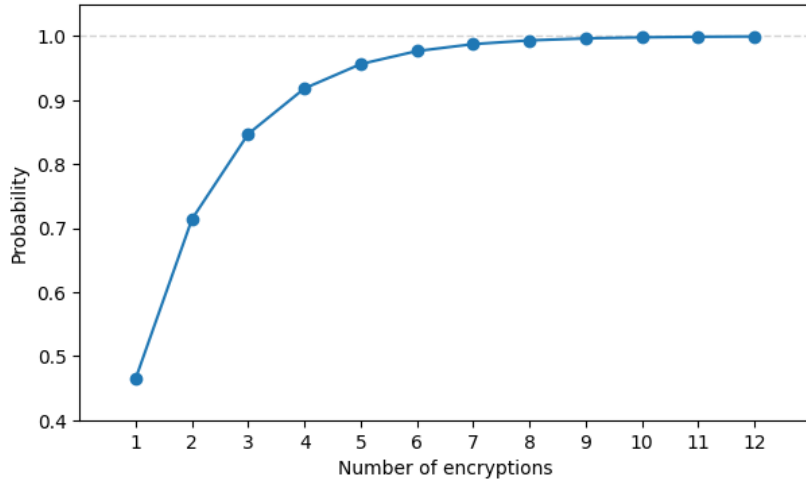


Figure 5: Probability p of different number of encryptions ℓ for PRESENT and AES

A key aspect of implementing the DMR countermeasure is defining the device’s response upon detecting a fault. Zhang *et al.* [ZLZ⁺18] demonstrated that returning no ciphertext, a zero-value ciphertext, or a random ciphertext upon fault detection does not prevent a PFA attack. Instead, it only introduces noise into the key recovery analysis, increasing the number of ciphertexts required for a successful attack. Note that this assumes the device continues to return correct ciphertexts in subsequent encryptions if no fault is detected during those encryptions. Therefore, aborting the operation and preventing the device from returning ciphertexts from the moment a fault is detected onwards could be a mitigation. In this case, the attacker can collect at most 8 correct ciphertexts (since with $\ell = 9$ encryptions, the fault is detected with a probability close to 1), which is insufficient for statistical analysis.

6 Conclusion

In this paper, we first demonstrated that a PFA attack on AES can be carried out using a simple instruction skip. 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 persistent fault in an 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. We demonstrated that inducing a fault in a round constant involved in the AES key schedule can enable key recovery through a differential analysis. This approach significantly reduces the data 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. Finally, we provide a discussion on the applicability of common countermeasures against fault attacks in the PFA context.

Acknowledgements. We sincerely thank the anonymous reviewers for their suggestion on the MbedTLS library and their valuable comments, which helped improve the paper. This work was supported by the French *Agence Nationale de la Recherche* through the grant ANR-22-CE39-0008 (project PROPHY).

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, Berlin, 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>.
- [BH22] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022. doi:10.1109/ACCESS.2022.3217212.
- [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, Berlin, 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 - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, 2019, Proceedings*, volume 11947 of *Lecture Notes in Computer Science*, pages 13–33. Springer, 2019. doi:10.1007/978-3-030-35869-3_4.
- [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, Cham, 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, Berlin, 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. URL: <https://tches.iacr.org/index.php/TCHES/article/view/7286>, doi:10.13154/tches.v2018.i3.547-572.
- [DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schl affer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, July 2021. doi:10.1007/s00145-021-09398-9.

- [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, Berlin, Heidelberg, August 2010. doi:10.1007/978-3-642-15031-9_22.
- [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, Berlin, 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, Berlin, 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, Berlin, Heidelberg, December 2006. doi:10.1007/11935070_14.
- [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, Cham, 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. doi:10.46586/tches.v2022.i2.289-311.
- [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. URL: <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. URL: <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

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

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}_6, \hat{c}) of 2^{16} possibilities, we compute

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

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

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

If $\Delta S_{15} = 3 \cdot \Delta S_{13} \oplus \hat{b}$ 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

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

If $\Delta S_{11} = 3 \cdot \Delta S_9 \oplus \hat{b}$ 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

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

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

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

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

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

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

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

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

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

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

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

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.