A Study of Persistent Fault Analysis

Andrea Caforio and Subhadeep Banik

LASEC, École Polytechnique Fédérale de Lausanne, Switzerland
{andrea.caforio,subhadeep.banik}@epfl.ch

Abstract. Persistent faults mark a new class of injections that perturb lookup tables within block ciphers with the overall goal of recovering the encryption key. Unlike earlier fault types persistent faults remain intact over many encryptions until the affected device is rebooted, thus allowing an adversary to collect a multitude of correct and faulty ciphertexts. It was shown to be an efficient and effective attack against substitution-permutation networks. In this paper, the scope of persistent faults is further broadened and explored. More specifically, we show how to construct a key-recovery attack on generic Feistel schemes in the presence of persistent faults. In a second step, we leverage these faults to reverse-engineer AES- and PRESENT-like ciphers in a chosen-key setting, in which some of the computational layers, like substitution tables, are kept secret. Finally, we propose a novel, dedicated, and low-overhead countermeasure that provides adequate protection for hardware implementations against persistent fault injections.

Keywords: Fault Analysis · PFA · Feistel Networks · Reverse Engineering · AES · PRESENT · Countermeasures.

1 Introduction

Fault injections and their accompanying analysis techniques rank amongst the most devastating attacks against cryptographic implementations. They saw their inception in 1996 when Boneh et al. demonstrated how to use computation errors during the CRT step of RSA to recover a prime factor of the public modulus [5]. In the following year, Biham and Shamir gave a method to exploit the difference between a faulty and correct DES ciphertext to gain information about the encryption key, this type of analysis became known as differential fault analysis [3]. Usually very few ciphertext pairs are needed to mount a DFA attack successfully; however, the faults have to be precisely targeted, often at rather small memory regions or specific registers, and during particular rounds of a block cipher computation. Other attacks assume a permanent fault model that is most commonly induced by defective hardware [9].

Persistent faults attempt to bridge the gap between short-lived and permanent faults as they remain intact over multiple encryptions but vanish once the device is rebooted. Persistent fault analysis gained traction at CHES 2018, in a work by Zhang et al. [16]. Their attack exploits the statistical imbalance in a collected set of ciphertexts, caused by one or more overwritten s-box elements, to recover the last round-key of substitution-permutation networks. The idea is based on the fact that in most SPN ciphers, like AES, a skewed substitution layer distribution translates directly into the ciphertexts. To see this, suppose the element u does not appear anymore in the s-box output due to the persistent fault injection, as a consequence, $u \oplus k$ is an impossible ciphertext word, where k is a last round-key word. Hence, after enough collected ciphertexts from the faulty device, k can be uniquely identified. The authors subsequently show that around 1500 ciphertexts are sufficient to recover the last round-key of AES in the presence of a single overwritten s-box element. They further demonstrate how to use the rowhammer attack [10] in order to provoke persistent fault injections in the s-box of vulnerable AES implementations.

In this paper, we show how persistent faults can be used to attack generic Feistel schemes where an altered s-box distribution is not directly visible in the collected ciphertext set. In a next step, we tackle the task of reverse engineering concealed parts of block ciphers. In particular, we demonstrate how to leverage persistent fault injections to recover a hidden PRESENT s-box and its permutation layer, as well as the substitution box of AES in a reduced-round setting. These reverse engineering attacks take place in the chosen-key setting and exploit particular behaviours within the key-schedule routines of both PRESENT and AES. Lastly, we propose a novel, low-overhead hardware countermeasure that adequately protects bijective substitution boxes against persistent fault injections.

2 Persistent Fault Analysis on Feistel Schemes

The standard techniques of persistent fault analysis do not apply to Feistel networks due to the fact that the both the left and right side of the output are masked by previous round function outputs. Indeed any Feistel round is a permutation over bit strings of length equal to the block size of the cipher, irrespective of whether the component s-box used in it is bijective or not. As a consequence, the skewed distribution of a faulty substitution box does not appear in the collected ciphertexts. However, if we loosen the ciphertext-only requirement, persistent fault can become a feasible danger. More specifically, we allow the attacker a single device reset and the possibility to re-encrypt plaintexts.

Consider a generic r-round Feistel scheme whose substitution layer consists of b identical or different $n \times m$ s-boxes $S_1, S_2, \ldots, S_b$. The last round of such a construction is depicted in Figure 1, with $x_l || x_r$ being the input to the last round and $y_l || y_r$ the corresponding ciphertext. Both $x_l || x_r$ and $y_l || y_r$ can be further decomposed into b blocks of m bits such that

$$\begin{aligned} x_l &= |x_l^1, \ldots, x_l^b|, \; x_r = |x_r^1, \ldots, x_r^b|, \\ y_l &= |y_l^1, \ldots, y_l^b|, \; y_r = |y_r^1, \ldots, y_r^b|. \end{aligned}$$

For most ciphers $n = m$. However, for block ciphers like DES, $n = 6$ and $m = 4$. So we first expand x_r using an expansion function to a string d_r of length nb

bits. Thereafter, we can write $d_r = |d_r^1, d_r^2, \ldots, d_r^b|$, where each d_r^i is of length n bits.

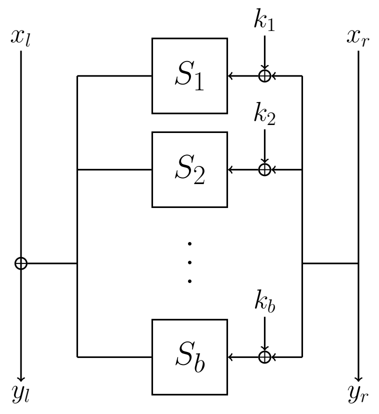

Fig. 1. Last Round of a Generic Feistel Scheme

2.1 Different S-Boxes

We first treat the case where $S_1, S_2, \ldots, S_b$ are pairwise different substitution boxes. The key observation is that in the presence of persistent faults some ciphertexts remain uncorrupted.

Suppose a single fault has been injected into one box. The probability that this element is not accessed in each round lies at $(1 - \frac{1}{2^n})^r$. Furthermore, the probability that the faulty element is only accessed in the very last round is given by $\frac{1}{2^n}(1 - \frac{1}{2^n})^{r-1}$.

Example 1 (DES). The data encryption standard [8] is a 16-round Feistel network whose substitution layer consists eight pairwise different 6×4 s-boxes. The probability that a faulty element in one box is not accessed in all rounds stands at $(1 - \frac{1}{2^6})^{16} \approx 0.777$. Additionally, the probability that the faulty element is only accessed in the last round is given by $\frac{1}{2^6}(1 - \frac{1}{2^6})^{15} \approx 0.0123$.

For illustration purposes assume that entry e of S_b has been altered. The faulty s-box is denoted by S_b'. The injected element does not need to be known to the attacker. Let $y_l||y_r$ be a ciphertext from a faultless device, i.e. one with S_b, and $y_l'||y_r'$ the encryption of the same plaintext on a faulty device, i.e. one with S_b' that only accessed faulty element of S_b' in the last round. As a consequence we have that $y_l \oplus y_l'$ is of the form

$$y_l \oplus y_l' = |a_1, a_2, \ldots, a_b|,$$

where a_i is a block of m bits with the property that $a_i = 0$ for $0 \leq i < b$ and $a_b = S_b(y_r^b \oplus k_b) \oplus S'_b(y_r^b \oplus k_b)$. In other words, incorrect ciphertexts that accessed the faulty entry e of S'_b only in the last round round can be identified when given the corresponding correct ciphertext. In such a case k_b can be recovered via $k_b = y_r^b \oplus e$. In the case where we have l faulty elements $e_1, \ldots, e_l$, k_b can be recovered up-to l candidates. Note that to recover the remaining parts of the last round-key further injections into the other s-boxes are needed. Also note that there is a negligible probability that a false-positive, i.e. a ciphertext that accessed the faulty element in more than just the last round, is of the desired form.

The expected number of required ciphertexts pairs is given by the reciprocal $2^n(1 - \frac{1}{2^n})^{-(r-1)}$. For DES this value stands at $2^6(1 - \frac{1}{2^6})^{-15} \approx 82$. Algorithm 1 summarizes the developed ideas.

Algorithm 1: Feistel Scheme PFA Key-Recovery

```
1  $p, cl, cr \leftarrow (\cdot)$                         // Initialize empty lists
2  for $i \leftarrow 0;\ i < n;\ i \leftarrow i+1$ do
3      $p(i) \leftarrow$ random plaintext
4      $y_l || y_r \leftarrow E(p(i))$
5      $cl(i), cr(i) \leftarrow y_l, y_r$
6  Overwrite element $e$ of $S_t$ in $E$
7  for $i \leftarrow 0;\ i < n;\ i \leftarrow i+1$ do
8      $y'_l || y'_r \leftarrow E(p(i))$
9      $|a_1, \ldots, a_b| \leftarrow y_l \oplus y'_l$
10     if $(a_j = 0,\ j \in \{1, \ldots, b\} \setminus \{t\}) \wedge (a_t \neq 0)$ then
11         return $y_r^t \oplus e$
```

2.2 A Single S-Box

In the case where $S_1 = \cdots = S_b$ the probability that the faulty element is only accessed by y_r^b in the last round is now given by $\frac{1}{2^n}(1 - \frac{1}{2^n})^{br-1}$, which can significantly increase the number of required ciphertext pairs. However, unlike in the previous case, one fault injection is enough to recover the entire last round-key. Furthermore, the overwritten element does not need to be known by the attacker either and can be brute-forced. In summary, if the attacker is allowed slightly more powers persistent faults can be exploited to recover the last round-key of Feistel schemes as well.

3 Reverse Engineering

The idea of leveraging fault injections for reverse-engineering, (in short FIRE), was introduced by San Pedro et al. [14] in an attempt to use a corrupted last round computation of either AES or DES to recover their hidden s-boxes. The attack itself is differential in its nature and requires many thousands of faults in order to be successful. The attack was later improved by Le Boulder et al. [12], their attack requires fewer faults in the penultimate instead of the last round of DES to recover all eight hidden substitution boxes. The concept of ineffective fault analysis where a particular byte of the intermediate state is stuck at zero is used by Clavier et al. to recover a hidden s-box of AES [6]. Finally, Tiessen et al. [15] used integral cryptanalysis to retrieve a secret AES substitution box when the cipher is reduced to four rounds.

In this section, we present a chosen-key attack that, in combination with persistent faults, aims to recover hidden substitution boxes and permutations of block ciphers more efficiently than an ordinary exhaustive search. Consider a PRESENT-like [4] construction in which the s-box or the permutation layer are fixed but secret bijective functions over $\{0,1\}^4$ and $\{0,1\}^{64}$ respectively. We demonstrate how persistent faults can be used to reverse-engineer such a construction.

3.1 Brute-Force

We first consider the method of a simple exhaustive search in order to recover the substitution box. There are $2^n!$ bijective s-boxes over $\{0,1\}^n$, and hence the computational complexity of exhaustive search grows out of bound very quickly. For instance, there are $16! \approx 2^{44.25}$ possible arrangements for a small 4×4 s-box. However, already for a 8×8 s-box, as deployed in AES, there exist $256! \approx 2^{1684}$ possibilities. For non-bijective $n \times m$ substitution tables we have $\frac{2^n!}{(2^{n-m}!)^{2^m}}$ potential arrangements. For one of the 6×4 DES s-boxes this values stands at $\frac{64!}{(4!)^{16}} \approx 2^{222.64}$.

3.2 S-Box Recovery of 16-Round PRESENT

PRESENT is an ultra-lightweight block cipher designed by Bogdanov et al. [4] that operates on 64-bit blocks with a key size of either 80 or 128 bits meant for usage in low-energy and space-restricted devices. It operates over 31 rounds with a substitution layer consisting of a single 4×4 s-box that is applied on all 16 nibbles of the intermediate state. For the remainder, we will focus on the 80-bit version and in particular on its key schedule routine. Algorithm 2 depicts its key schedule procedure. Let $|k_{79}k_{78}\dots k_1k_0|$ be the individual bits of the master key in big endian notation. In each round the 64 most significant bits yield the current round-key. The key is then rotated 61 positions to the left, followed by the substitution of the four most significant bits $|k_{79}k_{78}k_{77}k_{76}|$ by $S(|k_{79}k_{78}k_{77}k_{76}|)$. Finally, the bits $|k_{19}k_{18}k_{17}k_{16}k_{75}|$ are xored with the binary representation of the round counter i.

As a warm-up we consider the s-box recovery in a reduced 16-round setting of PRESENT in the presence of persistent faults in the known-key setting such that the last round-key can be *exactly* determined. Assume the faults are injected into

Algorithm 2: 80-Bit PRESENT Key-Schedule

```
1 for $i = 1;\ i \leq 32;\ i \leftarrow i + 1$ do
2     $K_i \leftarrow |k_{79}k_{78} \dots k_{16}|$
3     $|k_{79}k_{78} \dots k_1 k_0| \leftarrow |k_{18}k_{17} \dots k_{20}k_{19}|$
4     $|k_{79}k_{78}k_{77}k_{76}| \leftarrow S(|k_{79}k_{78}k_{77}k_{76}|)$
5     $|k_{19}k_{18}k_{17}k_{16}k_{15}| \leftarrow |k_{19}k_{18}k_{17}k_{16}k_{15}| \oplus i$
```

the target device before the key schedule takes place and that the encryption key can be switched out without necessitating a reboot of the device, i.e. the injected faults do not disappear.

Due to its simplicity, the PRESENT key schedule exhibits some peculiarities. For instance, Hernandez et al. [11] showed that there exist keys that expand into very similar round-keys. This is partly due to the fact that some key bits only enter the substitution box during relatively late rounds and only appear in a few round-keys.

We want to stress another property of the key schedule routine. It is not hard to see that during the first 16 rounds no key bit enters the substitution box more than once, hence all s-box accesses during the second 16 rounds only depend on the values of the first 16 accesses. As a consequence, it is possible to compute keys that only access a single s-box element during the first half which in turn leads to the fact that the pattern of the latter half of s-box accesses is entirely determined by this single s-box value that was accessed during the first half.

Definition 1 (Low-Diffusion Key). *A low-diffusion key $\widetilde{K}$ is a PRESENT master key that, if fed into the key schedule routine, causes only one element of the s-box table to be accessed during the first 16 key schedule rounds.*

Naturally, there can only be 16 low-diffusion keys in total $\widetilde{K}_0, \dots, \widetilde{K}_{15}$ one for each substitution box element, i.e. key $\widetilde{K}_i$ only accesses s-box entry i during the first 16 rounds of the key schedule. See Algorithm 7 in the appendix on how to calculate all the 16 low-diffusion keys.

The existence of low-diffusion keys immediately suggests a s-box recovery procedure in a reduced 16-round setting. Given a faulty device E, for each low-diffusion key $\widetilde{K}_i$, $0 \leq i < 16$ we recover the last round-key k through PFA then iterate over all possible values of $S(i) = j$ and compare whether an offline key schedule calculation is equal to k. Algorithm 3 depicts the described method. Note that we assume that the faults remain intact after re-keying, if this is not the case the persistent faults have to be injected again for each iteration. Further note that only a few dozens of ciphertexts are required to recover the last round-key through PFA in PRESENT with high probability.

3.3 S-Box Recovery of Full-Round PRESENT

The attack from the previous section does not directly apply to a full 31-round setting, however we can use persistent faults to engineer a new s-box recovery

Algorithm 3: 16-Round PRESENT S-Box Recovery

```
1 $S(i) \leftarrow 0$, for $0 \leq i < 16$
2 for $i = 0;\ i < 16;\ i \leftarrow i + 1$ do
3     $k \leftarrow$ PFA($E_{\widetilde{K}_i}$)                  // Recover last round-key through PFA
4     for $j = 0;\ j < 16;\ j \leftarrow j + 1$ do
5         $S(i) = j$                          // Assign $j$ to $i$-th s-box entry
          // Offline key schedule using s-box $S$
          // and low-diffusion key $\widetilde{K}_i$
6         if KeySchedule$_S(\widetilde{K}_i) = k$ then
7             break
```

algorithms applicable to different fault models. The intuition is still based on the particular behaviour of low-diffusion keys, especially their behaviour throughout the key schedule computation.

Definition 2 (Access Rate). *The access rate of a low-diffusion key, denoted by $r_{\widetilde{K}_i}(j)$, is the number of accessed s-box elements by the key $\widetilde{K}_i$ during the key schedule routine when $S(i) = j$.*

Table 1 depicts each low-diffusion key alongside their respective access rates.

Table 1. Low-Diffusion Keys and their Access Rates

		$r_{\widetilde{K}_i}(j)$															
i	$\widetilde{K}_i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0x037bf04d5c0567402460	16	15	13	13	11	11	11	11	15	14	12	13	11	11	11	11
1	0x026ae06f7e012300ace8	16	15	13	13	11	11	11	11	16	15	13	13	11	11	11	11
2	0x0159d009180defc13570	16	15	13	13	11	11	11	11	15	14	12	13	11	11	11	11
3	0x0048c02b3a09ab81bdf8	16	15	13	13	11	11	11	11	15	14	12	13	11	11	11	11
4	0x073fb0c5d41476420640	16	15	13	13	11	11	11	11	15	14	13	14	11	11	11	11
5	0x062ea0e7f61032028ec8	16	15	14	14	11	11	11	11	15	14	12	13	11	11	11	11
6	0x051d9081901cfec31750	16	16	14	14	11	11	11	11	15	14	13	14	11	11	11	11
7	0x040c80a3b218ba839fd8	16	15	14	14	11	11	11	11	15	15	13	14	11	11	11	11
8	0x0bf3715c4c2745446020	16	15	13	13	12	12	12	11	15	14	12	13	12	12	11	11
9	0x0ae2617e6e230104e8a8	16	15	13	13	12	12	11	11	15	14	12	13	12	12	12	11
10	0x09d15118082fcdc57130	16	15	13	13	12	12	11	12	15	14	12	13	12	12	11	11
11	0x08c0413a2a2b8985f9b8	16	15	13	13	12	12	11	11	15	14	12	13	12	12	11	12
12	0x0fb731d4c43654464200	16	15	13	13	12	11	12	12	15	14	12	13	11	11	12	12
13	0x0ea621f6e6321006ca88	16	15	13	13	11	11	12	12	15	14	12	13	12	11	12	12
14	0x0d951190803edcc75310	16	15	13	13	11	12	12	12	15	14	12	13	11	11	12	12
15	0x0c8401b2a23a9887db98	16	15	13	13	11	11	12	12	15	14	12	13	11	12	12	12

Definition 3 (Access Pattern). *Denote by $p_{\widetilde{K}_i}(j)$ the set of accessed s-box entries during the key schedule for low-diffusion key $\widetilde{K}_i$ with $S(i) = j$.*

Example 2. The access pattern for $\widetilde{K}_0$ and $S(0) = 12$ is given by

$$p_{\widetilde{K}_0}(12) = \{0, 1, 2, 3, 4, 5, 6, 7, 12, 14, 15\}.$$

We look at the case where the adversary manages to inject a single chosen fault at a precise position in the substitution table, i.e. one element is overwritten. The online stage for this attack is the same as the previous one; we use the persistent attack module with the set of known keys $\widetilde{K}_i$ to extract the last round-key.

Note that a brute-force search in this setting, to recover the secret s-box, requires in the worst case $15! \approx 2^{40.25}$ trials in order to recover the remaining s-box entries. This value can be significantly improved if the encryption key can be chosen. Let the injected fault be of the form $S(i) = j$ such that $r_{\widetilde{K}_i}(j) = 11$. In this case there are only 10 remaining s-box entries in the entire key schedule routine that are accessed for the low-diffusion key $\widetilde{K}_i$. The strategy is to randomly assign values to these 10 entries, after which it is possible to compute the last round-key from $\widetilde{K}_i$ using an offline key schedule computation: the assigned values are correct if this computed key matches the last round-key obtained through PFA. Now, there are $\frac{15!}{5!} \approx 2^{33.34}$ potential arrangements that have to be checked in the worst case since due to the low-diffusion key and the fault injection only 10 s-box entries need to be assigned from a set of 15 potential values. This results in a reduction by a factor of 120 compared to the brute-force approach. The remaining 5 elements can then be safely brute-forced. Thus the computational complexity of this method is around $2^{33.34} + 5! \approx 2^{33.34}$ offline key schedule computations. Algorithm 4 formally depicts the described strategy: it makes use of the following definition.

Definition 4 (m-Permutation). *Let L be a collection of n elements. A m-permutation of L, denoted by $\Pi_{n,m}(L)$, is the set of all possible ways to choose m elements from L without repetition.*

3.4 Permutation Layer Recovery of PRESENT

On paper, recovering the permutation layer appears to be a harder task due to the sheer amount $64! \approx 2^{296}$ of possibilities. Let $C = \{c_1, \ldots, c_n\}$ be a set of n ciphertexts from the faulty device. Denote by $|c(i), c(j), c(k), c(l)|$ the nibble that is created by extracting the bits i, j, k, l from a ciphertext c. Further, denote by $C(i, j, k, l)$ the set of nibbles that is generated by extracting bits i, j, k, l from each ciphertext, i.e.

$$C(i, j, k, l) = \{|c_1(i), c_1(j), c_1(k), c_1(l)|, \ldots, |c_n(i), c_n(j), c_n(k), c_n(l)|\}.$$

Suppose an random entry in the S-box is overwritten due to a fault. If each bit in a nibble i, j, k, l stems from the same s-box then we have necessarily

Algorithm 4: Single-Fault S-Box Recovery

```
1  Choose $i \in \{0,1,2,3,4,5,6,7\}$
2  Overwrite $S(i) = j$ such that $r_{\widetilde{K}_i}(j) = 11$
   // Recover last round-key $k$
   // and overwritten element $v$ through PFA
3  $k, v \leftarrow \texttt{PFA}(E_{\widetilde{K}_i})$
4  $L \leftarrow \{0, \ldots, 15\} \setminus \{v\}$
5  $S'(l) = 0,\ 0 \le l \le 15;\ S'(i) = j$
6  for each $\pi \in \Pi_{15,10}(L)$ do
7      $z \leftarrow 0$
8      for each $p \in p_{\widetilde{K}_i}(j)$ do
9          $S'(p) \leftarrow \pi(z),\ z \leftarrow z + 1$
10     if $\texttt{KeySchedule}_{S'}(\widetilde{K}_i) = k$ then
11         return $S'$
```

$|C(i,j,k,l)| < 16$ due to the persistent fault injection. This is obviously due to the fact that overwriting the entry of a bijective 4×4 s-box decreases the number of unique outputs to less than 16. For all other nibbles the set is of size 16 for a large enough n. In this fashion we recover the hidden permutation up to a reordering of the bits i, j, k, l of each nibble, which naturally gives rise to 4! possibilities for each nibble and hence $4!^{16}$ possibilities for the entire 16 nibbles. Furthermore a reordering of the 16 s-boxes is also required that gives rise to 16! possibilities, which leaves us with a remaining complexity of $24^{16} \times 16! \approx 2^{118}$ to recover the entire permutation. Algorithm 5 depicts this strategy.

Algorithm 5: PRESENT Permutation Recovery

```
1  $L \leftarrow \{0,1,2,\ldots,63\}$
2  $C \leftarrow \{c_1, \ldots, c_n\}$            // Set of $n$ ciphertexts from faulty device
   // Iterate over all permutations of size 4
3  for each $\pi_0, \pi_1, \pi_2, \pi_3 \in \Pi_4(L)$ do
4      if $|C(\pi_0, \pi_1, \pi_2, \pi_3)| < 16$ then
5          output $|\pi_0, \pi_1, \pi_2, \pi_3|$
6          $L \leftarrow L \setminus |\pi_0, \pi_1, \pi_2, \pi_3|$
```

The question is now, how large do we have to choose n in order to guarantee with high probability that only the correct nibbles are chosen? This is a classical instance of the coupon collector's problem where we want to quantify the number of uniform trials until some number of elements have been picked. In the case of Algorithm 5, how many ciphertexts are required until $|C(i,j,k,l)| = 16$ with high probability for an incorrect nibble? Let T the number of trials until

16 substitution box entries have occurred for a specific ciphertext nibble. The expected number of picks $E[T]$ is given by

$$\mathrm{E}[T] = 16H_{16} \approx 54.1,$$

where H_{16} is the 16-th Harmonic number. Similarly, we can quantify the variance $\mathrm{Var}[T]$, which is upper-bounded by

$$\mathrm{Var}[T] < \frac{\pi^2}{6}16^2 \approx 421.1.$$

Finally, we can use the Chebyshev's inequality to specify the bound on the error probability.

$$\begin{aligned}\Pr[T \geq kE[T]] &\leq \Pr\left[|T - \mathrm{E}[T]| \geq (k-1)\mathrm{E}[T]\right] \\ &\leq \frac{\mathrm{Var}[T]}{((k-1)\mathrm{E}[T])^2} \\ &< \frac{16\pi^2}{6(k-1)^2 H_{16}^2},\end{aligned}$$

where $k \geq 2$. Table 2 depicts $\Pr[T \geq kE[T]]$ for multiple choices of k. Evidently, a few hundred ciphertexts should suffice for a successful run of Algorithm 5.

Table 2. $\Pr[T \geq kE[T]]$ for several n

k	2	3	4	5	10	20
$\Pr[T \geq kE[T]]$	0.1439	0.0359	0.0159	0.0089	0.0017	0.0003

Knowing the nibbles that directly originated from one s-box, it is possible to further reduce the permutation space with multiple injections. Firstly, note that if 0 is overwritten then the last round-key bits $\kappa = |\kappa(i), \kappa(j), \kappa(k), \kappa(l)|$ is determined exactly (where we have already determined that bits i, j, k, l emanate from the same s-box). This is a direct consequence of the fact that all 4-bit permutations of 0 remain 0, hence $0 \oplus \kappa$ does not appear in the ciphertext nibble. The same also holds if 15 is overwritten (in that case $15 \oplus \kappa$ does not appear in the ciphertext nibble). Once the last round-key has been established, the last round substitution layer output bits $s = |\kappa(i) \oplus c(i), \kappa(j) \oplus c(j), \kappa(k) \oplus c(k), \kappa(l) \oplus c(l)|$ is also available. Figure 2 illustrates the situation.

Let L_a be the list of collected s-box outputs s for one nibble when entry a of the s-box has been overwritten. There exists a small set of potential 4-bit permutations that produce the values in L_a assuming element a never appears in the s-box output. For example let the entry 1 in the s-box table be overwritten. Then we know that the s-box never outputs 1. We are interested in the determining the order in which the output bits of some s-box is shuffled to map

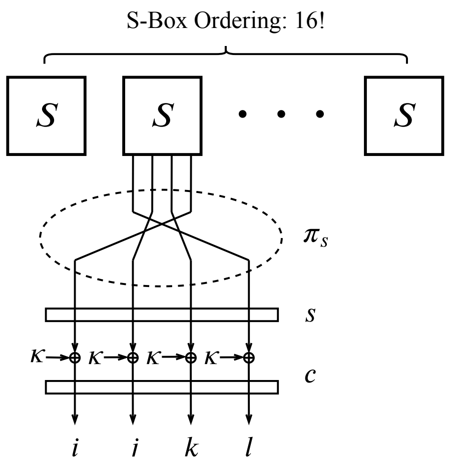

Fig. 2. Permutation Layer Recovery in PRESENT

to bits i, j, k, l of the ciphertext (denote this permutation by π_s). Now if some $s = s_0$ is not present in L_1, then we can reduce the search space for π_s to only those permutations that map 1 to s_0. For example if $s_0 = 8$ (the nibble which has 1 in the msb), then π_s is essentially the set of all 4 bit permutations that map the lsb to the msb.

It is not too difficult to see that repeating the above experiments for $a = 2, 4, 8$ gives us the unique 4-bit permutation π_s. This means with four injections we can retrieve the 4-bit permutation after each s-box, which reduces the overall search space down to a reordering of the s-boxes, i.e. $16! \approx 2^{44}$ possibilities. This set is small enough to be brute forced using appropriate computational resources.

3.5 Reduced-Round AES S-Box Recovery

The ideas developed so far can be adapted in order to recover a hidden AES s-box in a reduced-round setting. We consider the 128-bit Rijndael key schedule procedure as it is deployed in the final AES specification [7]. The routine acts in eleven rounds, one for each round-key, and works on 32-bit words. Let K_0, K_1, K_2 and K_3 denote the four 32-bit words of the master key with $W_0, \ldots, W_{43}$ being the 32-bit round-key output words. Further, let rc be a list of ten round constants.

The key schedule also makes use of two external transformations R and S. R designates the rotation of a 32-bit word, consisting of four bytes $|b_0, b_1, b_2, b_3|$, by one position to the left, such that

$$R(|b_0, b_1, b_2, b_3|) = |b_1, b_2, b_3, b_0|$$

while S is the substitution of each byte in a 32-bit word by its corresponding s-box value such that

$$S(|b_0, b_1, b_2, b_3|) = |S(b_0), S(b_1), S(b_2), S(b_3)|.$$

The key schedule then calculates each round-key word W_i as follows

$$W_i = \begin{cases} K_i, & i < 4 \\ W_{i-4} \oplus R(S(W_{i-1})) \oplus rc_{i/4}, & i \geq 4 \text{ and } i \equiv 0 \bmod 4 \\ W_{i-4} \oplus W_{i-1}, & \text{otherwise.} \end{cases}$$

In the PRESENT key schedule we had the property that during the first sixteen rounds the s-box access pattern was entirely determined by the master key thus special keys could be found that only access a single s-box entry during those initial rounds of the key schedule. It is obvious that no such property is given in the AES key schedule, simply because all intermediate key states are affected by the s-box accesses within one round. A single round contains four s-box accesses one for each byte of current 32-bit word, hence in total there are 40 accesses over all eleven rounds of the key schedule due to the first round not performing no lookup. However, it is possible to find keys for which all four lookups of a single round go to the same element for the first few rounds of the key schedule routine.

We consider the following adapted fault model, where the 0 is injected at at a *known* position i in the substitution table such that $S(i) = 0$, which has the following lemma as a consequence.

Lemma 1. *Given a AES master key of the form*

$$K_0 = \texttt{0x01000000},\ K_1 = \texttt{0x02000000},$$
$$K_2 = \texttt{0x02000000},\ K_3 = |a, a, a, a|,$$

where $a \in \{0,1\}^8$ are the individual bytes of the word, and a faulty substitution box S with $S(a) = 0$. The s-box access pattern during first six rounds of the key schedule is given by

$$\begin{aligned} W_3 &= |a,\ a,\ a,\ a|, & W_7 &= |a,\ a,\ a,\ a|, \\ W_{11} &= |a,\ a,\ a,\ a|, & W_{15} &= |a,\ a,\ a,\ b|, \\ W_{19} &= |a,\ a,\ c,\ d|, & W_{23} &= |a,\ e,\ f,\ g|, \end{aligned}$$

where $b, c, d, e, f, g \in \{0,1\}^8$ are bytes and given by

$$\begin{aligned} b &= a \oplus \texttt{0x06}, & c &= a \oplus S(b), \\ d &= a \oplus \texttt{0x08}, & e &= a \oplus S(a \oplus S(b)), \\ f &= a \oplus S(b) \oplus S(\texttt{0x08}), & g &= \texttt{0x1a}. \end{aligned}$$

This means that the set of input entries of the s-box accessed during the first 6 rounds is given by the byte-values a, b, c, d, e, f, g.

Proof. Due to space constraints, we present a proof in Appendix B.

By leveraging those keys we can device a s-box recovery attack on a reduced-round version of AES that is similar to the algorithms in Section 3.2 and Section 3.3. We use PFA to recover last round-key, then guess partial locations of

Algorithm 6: 5-Round AES S-Box Recovery

```
1  Inject 0 at position $a$ in substitution box of $E_K$
2  $S(i) \leftarrow 0$, for $0 \leq i < 256$
3  $a \in \{0, \dots, 255\}$
4  $K_0 \leftarrow \texttt{0x01000000},\ K_1 \leftarrow \texttt{0x02000000},\ K_2 \leftarrow \texttt{0x02000000},\ K_3 \leftarrow |a, a, a, a|$
5  $K \leftarrow K_0 || K_1 || K_2 || K_3$
   // Retrieve round-key $k$ and active s-box elements $L$ through PFA
6  $k, L \leftarrow \texttt{PFA}(E_K)$
7  for each $x \in L$ do
8      for each $y \in L \setminus \{x\}$ do
9          for each $z \in L \setminus \{x, y\}$ do
10             $S(a \oplus \texttt{0x06}) = x,\ S(a \oplus \texttt{0x08}) = y,\ S(a \oplus S(a \oplus \texttt{0x06})) = z$
11             if $\texttt{KeySchedule}_S(K) = k$ then
12                 return $S$
```

the s-box table to do an offline key schedule to calculate the last round-key and see if both the above keys match. Algorithm 6 shows a five-round s-box recovery, which recovers three elements of the substitution box.

The number of key schedule operations in Algorithm 6 depends the numbers of cipher rounds, i.e. since only one assignment is made to the s-box table for the 4 round attack, we require 255 key schedule operations. For the 5 round attack, 3 entries are assigned and so $\frac{255!}{252!} \approx 2^{24}$ offline key schedule computations. Similarly 6 assignments are required for the 6 round attack and so and $\frac{255!}{249!} \approx 2^{48}$ operations are required.

The above routine needs to be repeated for multiple values of a to recover the full S-box table. For example in both the 5/6 round attacks, we successfully assign 2 entries $(a \oplus \texttt{0x06})$ and $(a \oplus \texttt{0x08})$ in each execution of the above routine. Thus at most 128 executions of the above routine with judiciously chosen values of a are sufficient to extract the entire table.

4 Countermeasures

We propose a novel, low-overhead and dedicated countermeasure against persistent fault injections in bijective substitution boxes that is not susceptible to further injections through redundant lookup tables. The need for this arises through the fact that common fault injection countermeasures, like area redundancy and masking of intermediate values, have been shown to be ineffective [16, 13].

We assume the fault model where one or more entries of the s-box have been altered. In such a scenario there are necessarily at least two entries in the s-box that bear the same value such that $S(x) = S(y)$ with $x \neq y$, i.e. if two different values enter the substitution layer and are equal after the transformation an error is detected. This necessitates that all input and output are compared to each other. Figure 3 depicts such a construction. Once a fault has been detected the device can engage in remedy procedures such as outputting random ciphertexts until the next reboot. We will see that the number of encryptions is so low that no information about the last round-key can be inferred through persistent fault analysis techniques.

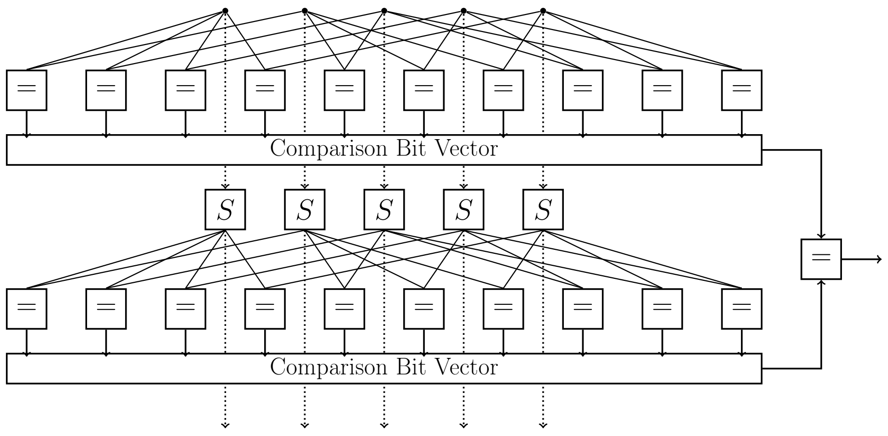

Fig. 3. Pairwise Comparison Network

If there is one or more overwritten elements in the substitution layer it is possible to quantify the probability that the fault is detected in a particular round by the following lemma.

Lemma 2. *Denote by S a faulty $n \times n$ s-box with t altered entries $v_1, \ldots, v_t$ such that $S(v_i) = S(u_i)$ for some $u_i \notin \{v_1, v_2, \ldots, v_t\}$. Let $p_{n,m,t}$ be the probability that at least one pair of equal entries is accessed in one round. The value is given by*

$$\begin{aligned} p_{n,m,t} = 1 &- \left(\frac{2^n - 2t}{2^n}\right)^m \\ &- \sum_{i=1}^{t} \Bigg(2^i \binom{t}{i} \\ &\times \sum_{k_1}^{m} \sum_{k_2}^{m-k_1} \cdots \sum_{k_i}^{m-\sum_{j=1}^{i-1}} \binom{m}{k_1} \cdots \binom{m - \sum_{j=1}^{i-1} k_j}{k_i} \\ &\times \left(\frac{1}{2^n}\right)^{\sum_{j=1}^{i} k_j} \left(\frac{2^n - 2t}{2^n}\right)^{m - \sum_{j=1}^{i} k_j}\Bigg) \end{aligned}$$

Proof. Suppose there are t pairs of equal entries. An error is not detected if none of the $2t$ elements of these pairs are accessed in round, which happens with probability $\left(\frac{2^n-2t}{2^n}\right)^m$, or exactly one element is accessed from i pairs, which happens with probability

$$\sum_{k_1}^{m} \sum_{k_2}^{m-k_1} \cdots \sum_{k_i}^{m-\sum_{j=1}^{i-1}} \binom{m}{k_1} \cdots \binom{m-\sum_{j=1}^{i-1} k_j}{k_i}.$$

Summing over all $1 \leq i \leq t$ then yields $p_{n,m,t}$.

We can further calculate the probability that one or more faults are detected over r round function invocations in a block cipher.

Corollary 1. *Given a r-round block cipher whose substitution box bears t equalized entries. Denote by $p^r_{n,m,t}$ the probability that an error is detected. It is given by*

$$p^r_{n,m,t} = 1 - (1 - p_{n,m,t})^r.$$

The expected value of required encryption until an injected fault is detected lies at $\frac{1}{p^r_{n,m,t}}$. Table 3 shows the detection probabilities for a varying number of overwritten elements t for both AES $p^{10}_{8,16,t}$ and PRESENT $p^{31}_{4,16,t}$.

Table 3. $p^r_{n,m,t}$ for AES and PRESENT

	$p^r_{n,m,t}$					
	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$
$p^{10}_{8,16,t}$ (AES)	0.0341	0.0671	0.0990	0.1288	0.1597	0.1884
$p^{31}_{4,16,t}$ (PRESENT)	≈ 1	≈ 1	≈ 1	≈ 1	≈ 1	≈ 1

The sixteen substitution box accesses per AES round already yield a relatively good fault detection probability whose expected value is well below the number of ciphertexts that are required for a successful persistent fault analysis attack. However, it is possible to perform additional redundant accesses, i.e. increasing m, to further increase the detection probability.

4.1 Reducing the Hardware Cost

For both AES and PRESENT it is necessary to perform $\binom{16}{2} = 120$ pairwise byte or nibble comparisons. This necessitates a rather large overhead in hardware implementations. A naive circuit that compares bytes can be built out of 8 XNOR gates and 7 AND gates, thus doing 120 comparisons would result in a total of 1800 logic gates, which requires a significant amount of chip area. As an alternative construction, we propose a modification in which that only adjacent bytes are compared, as seen in Figure 4.

In such a pattern there are only fifteen comparisons, which in turn changes the detection probability, which is now lower-bounded by $\frac{p_{n,m,t}}{8}$. For AES this bound

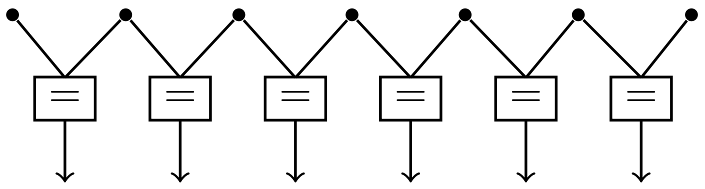

Fig. 4. Adjacent Comparison Network

is tight, the overall detection probability over 10 rounds then stands at $1-(1-\frac{p_{8,16,t}}{8})^{10} \approx 0.004335$. Hence in expectation we need $\frac{1}{0.004335} \approx 230$ encryptions until a faulty substitution box is detected. For PRESENT, this bound is not tight, it can be shown that in a single-fault setting the detection probability for one encryption is roughly 0.97, which means that in expectation 1 encryption suffices to detect the fault.

We measured the effectiveness of our countermeasure in the following experiment. We collected ciphertexts from a faulty device protected by our countermeasure until the fault is detected. Persistent fault analysis is then performed on those ciphertexts and the residual key-entropy of the last round key is evaluated. The experiment is further repeated for around 2^{20} random keys in order to obtain a probability for each entropy value. The results are tabulated as a function of the number of persistent faults t in Table 4 for AES and Table 5 for PRESENT. Our countermeasure offers a very strong protection of PRESENT already in the single-fault setting. For AES, it performs well on average but it is especially effective in the presence of more than one persistent fault injection.

Table 4. Probability of Residual Key Entropy (AES)

	$[0,15)$	$[15,30)$	$[30,45)$	$[45,60)$	$[60,75)$	$[75,90)$	$[90,105)$	$[105,120)$	$[120,128]$
$t=1$	0.00195	0.00361	0.00856	0.01699	0.03708	0.07811	0.16763	0.35617	0.32989
$t=2$	0.00000	0.00000	0.00008	0.00071	0.00343	0.01654	0.07612	0.35053	0.55261
$t=3$	0.00000	0.00000	0.00000	0.00002	0.00020	0.00261	0.02669	0.26949	0.70104
$t=4$	0.00000	0.00000	0.00000	0.00000	0.00001	0.00032	0.00860	0.18980	0.80127

Table 5. Probability of Residual Key Entropy (PRESENT)

	$[56,57)$	$[57,58)$	$[58,59)$	$[60,61)$	$[61,62)$	$[62,63)$	$[63,64)$	64
$t=1$	0.00000	0.00000	0.00002	0.00032	0.00067	0.03099	0.00000	0.96798
$t=2$	0.00000	0.00000	0.00000	0.00001	0.00001	0.00085	0.00000	0.99914

5 Implementation

The countermeasure we suggest is efficiently implementable in ASIC platforms with minimal overhead in hardware. For a round based implementation of AES, the architecture is straightforward and is exactly as depicted in Figure 4. Comparing each of the adjacent bytes before and after the s-box operation to produce 15-bit vectors before and after the s-box, requires a total of $2 \cdot 15 = 30$ comparator blocks. Each such block can be constructed as the bitwise AND of the XNOR of the two input bytes, i.e.

$$c = \prod_{i=0}^{7} (a_i \oplus b_i \oplus 1),$$

where $\mathbf{a} = (a_0, a_1, \ldots, a_7)$ and $\mathbf{b} = (b_0, b_1, \ldots, b_7)$ are the input bytes and $c = 1$ if and only if $\mathbf{a} = \mathbf{b}$. The circuit to compare the two 15-bit vectors is similar, we perform a bitwise XNOR and compute the logical AND of the resultant bits to get the final fault integrity bit F. If $F = 1$ and all previous fault integrity checks have passed, the round function is allowed to output required result else it is replaced with the all 0 signal. We thus must have some way of ascertaining if all previous integrity checks have passed. To do that we introduce a fault integrity flip-flop, that is initialized to 1 at system reset, which is updated by ANDing the value of the current flip-flop state f_t with the current value of F, i.e $f_{t+1} = F \cdot f_t$. The advantage of this method is that once a fault integrity check fails, the integrity flip-flop is permanently set to 0, after which it becomes easy to replace s-box outputs with random bytes by xoring the term $(1 + f_t) \cdot \theta$, where θ is the output of a random byte generator. Since a bitwise AND of a t-bit signal requires $t - 1$ 2-input AND gates, the above comparison network requires $2 * 15 * 8 + 15 = 255$ two-input XNOR gates and $2 * 15 * 7 + 14 = 224$ two-input AND gates. The final randomizing of the round function output in case $f_t = 0$, requires a simple XOR of the function output with $(1+f_t) \cdot \theta$ and hence requires a further 128 two-input AND and XOR gates. For the round based implementation of PRESENT, the cost is similar, except that the comparison network is built over nibbles rather than bytes.

For a serial architecture, the implementation is even more efficient and requires minimal hardware overhead. take, for example the Atomic AES v 2.0 architecture proposed in [2], which performs one s-box operation in one clock cycle. The architecture has an internal round counter constructed with 5-bit full period LFSR that counts up from 0 to 30 for each round. Of these the 16 s-box operations are done in cycles 15 to 30, as the state bytes are shifted out serially through the first byte register implemented in the circuit. In each cycle labeled $t = 15+j$ for $j \in [0, 14]$ the s-box input and output are each driven into byte registers C_{IN} and $\mathsf{C}_{\mathsf{OUT}}$ respectively as shown in Figure 5. As a result at round $t+1$ one can make a comparison between adjacent byte inputs by simply comparing the values stored in C_{IN} and the current s-box input. A similar comparison can be made between the $\mathsf{C}_{\mathsf{OUT}}$ and the current s-box output. For fault integrity, the result of the input and output byte comparisons should be equal to each other which is again implemented by an XNOR gate: which naturally outputs 1 if the input-output pairs are both equal or both unequal. Moreover this equality must hold for the $15 \cdot 11 = 165$ comparisons made during an AES encryption operation. Thus we also have a fault integrity flip-flop which works exactly in the same manner as described in the round based circuit.

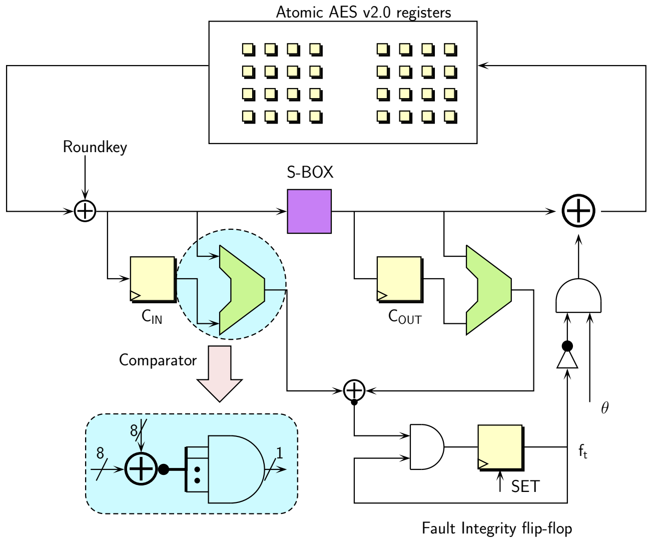

Fig. 5. Fault Integrity check for the Atomic AES v2.0 architecture

Table 6 tabulates the results of all implementations. The following design flow was used: first the design was implemented in VHDL. Then, a functional verification was first done using Mentor Graphics Modelsim software. The designs were synthesized using the standard cell library of the 90nm logic process of STM (CORE90GPHVT v 2.1.a) with the Synopsys Design Compiler, with the compiler being specifically instructed to optimize the circuit for area. A timing simulation was done on the synthesized netlist. The frequency of operation was fixed at 10 MHz as established in [1], because at this frequency, for the STM 90 nm logic process, the energy consumption of block ciphers was found to be frequency-independent. The switching activity of each gate of the circuit was collected while running post-synthesis simulation. The average power was obtained using *Synopsys Power Compiler*, using the back annotated switching activity. Energy is calculated as the product of average power and time taken for one encryption. The table clearly shows that the overhead in terms of area, except for the round based implementation of PRESENT is well under 8%. In terms of other performance metrics, we see reasonably competitive figures.

Table 6. Performance Comparison of circuits before and after implementing fault integrity countermeasures. B refers to the basic circuit without countermeasures, C refers to the circuit after implementing countermeasures, TP_{max} refers to maximum throughput achievable on hardware. Note that the figures do not include an RNG used for randomization

#	Architecture	Type	Area (GE)	Overhead (in %)	Latency (cycles)	Energy (nJ)	TP_{max} (Gbps)
	AES						
1	Round based	B	12876		11	0.72	2.371
		C	13615	5.7	11	0.78	1.555
2	8-bit Serial	B	2060		246	3.20	0.086
		C	2143	4.0	246	3.23	0.075
	PRESENT						
1	Round based	B	1316		33	0.19	1.598
		C	1713	30.2	33	0.29	1.050
2	4-bit Serial	B	892		564	2.50	0.052
		C	963	7.9	564	2.71	0.046

6 Conclusion

Persistent fault analysis has been shown to be an efficient and devastating attack against substitution-permutation networks. In this paper, we further broadened and investigated the range of these kinds of injections. In other words, we showed how Feistel schemes can also fall prey to persistent faults and demonstrate how they can be used to accelerated reverse engineering endeavors. Finally, we presented a low-overhead countermeasure that efficiently protects bijective substitution boxes against persistent fault injections.

In conclusion, persistent faults offer an exciting new perspective on fault attacks in various fields from key-recovery attacks to reverse engineering tasks and the development of efficient and adequate countermeasures. It is thus an interesting exercise in future works to see to what extent persistent faults can be further leveraged.

References

1. Banik, S., Bogdanov, A., Regazzoni, F.: Exploring Energy Efficiency of Lightweight Block Ciphers. In: Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers. pp. 178–194 (2015). https://doi.org/10.1007/978-3-319-31301-6_10, https://doi.org/10.1007/978-3-319-31301-6_10
2. Banik, S., Bogdanov, A., Regazzoni, F.: Compact circuits for combined AES encryption/decryption. J. Cryptographic Engineering **9**(1), 69–83 (2019). https://doi.org/10.1007/s13389-017-0176-3, https://doi.org/10.1007/s13389-017-0176-3

3. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) Advances in Cryptology — CRYPTO '97. pp. 513–525. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
4. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007. pp. 450–466. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
5. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) Advances in Cryptology — EUROCRYPT '97. pp. 37–51. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
6. Clavier, C., Isorez, Q., Damien, M., Wurcker, A.: Complete reverse-engineering of AES-like block ciphers by SCARE and FIRE attacks. Cryptography and Communications - Discrete Structures, Boolean Functions and Sequences (Issue 1) (Mar 2015)
7. Daemen, J., Rijmen, V.: The Block Cipher Rijndael. In: Quisquater, J.J., Schneier, B. (eds.) Smart Card Research and Applications. pp. 277–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
8. Des: Data Encryption Standard. In: In FIPS PUB 46, Federal Information Processing Standards Publication. pp. 46–2 (1977)
9. Dutta, A., Paul, G.: Deterministic Hard Fault Attack on Trivium. In: Yoshida, M., Mouri, K. (eds.) Advances in Information and Computer Security. pp. 134–145. Springer International Publishing, Cham (2014)
10. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 300–321. Springer International Publishing, Cham (2016)
11. Hernandez-Castro, J.C., Peris-Lopez, P., Aumasson, J.P.: On the Key Schedule Strength of PRESENT. In: Garcia-Alfaro, J., Navarro-Arribas, G., Cuppens-Boulahia, N., de Capitani di Vimercati, S. (eds.) Data Privacy Management and Autonomous Spontaneus Security. pp. 253–263. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
12. Le Bouder, H., Guilley, S., Robisson, B., Tria, A.: Fault Injection to Reverse Engineer DES-Like Cryptosystems. In: Danger, J.L., Debbabi, M., Marion, J.Y., Garcia-Alfaro, J., Zincir Heywood, N. (eds.) Foundations and Practice of Security. pp. 105–121. Springer International Publishing, Cham (2014)
13. Pan, J., Bhasin, S., Zhang, F., Ren, K.: One Fault is All it Needs: Breaking Higher-Order Masking with Persistent Fault Analysis. Cryptology ePrint Archive, Report 2019/008 (2019), https://eprint.iacr.org/2019/008
14. San Pedro, M., Soos, M., Guilley, S.: FIRE: Fault Injection for Reverse Engineering. In: Ardagna, C.A., Zhou, J. (eds.) Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication. pp. 280–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
15. Tiessen, T., Knudsen, L.R., Kölbl, S., Lauridsen, M.M.: Security of the AES with a Secret S-Box. In: Leander, G. (ed.) Fast Software Encryption. pp. 175–189. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
16. Zhang, F., Lou, X., Zhao, X., Bhasin, S., He, W., Ding, R., Qureshi, S., Ren, K.: Persistent Fault Analysis on Block Ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 150–172 (2018)

Appendix

A Calculation of Low-Diffusion Keys

Algorithm 7 depicts the routine that calculates all 16 low-diffusion keys for PRESENT.

Algorithm 7: Calculation of Low-Diffusion Keys

```
1 for $i = 0;\ i < 16;\ i \leftarrow i + 1$ do
2     $\widetilde{K}_i \leftarrow 0$
3     for $j = 0;\ j < 16;\ j \leftarrow j + 1$ do
4         $r \leftarrow (15 + 19j) \bmod 80$
5         $\widetilde{K}_i = \widetilde{K}_i \oplus ((i \oplus j) \lll r)$
6 return $\widetilde{K}_i,\ 0 \leq i < 16$
```

B Proof of Lemma 1

Proof. The access pattern follows from a simple calculation of the intermediate round key words. Set $K_0 = \texttt{0x01000000}$, $K_1 = \texttt{0x02000000}$, $K_2 = \texttt{0x02000000}$, $K_3 = |a, a, a, a|$ and $S(a) = 0$.

$$\begin{aligned}
W_0 &= K_0 = \texttt{0x01000000} \\
W_1 &= K_1 = \texttt{0x02000000} \\
W_2 &= K_2 = \texttt{0x02000000} \\
W_3 &= K_3 = |a, a, a, a| \\
W_4 &= W_0 \oplus S(R(W_3)) \oplus rc_1 = \texttt{0x00000000} \\
W_5 &= W_1 \oplus W_4 = \texttt{0x02000000} \\
W_6 &= W_2 \oplus W_5 = \texttt{0x00000000} \\
W_7 &= W_3 \oplus W_6 = |a, a, a, a| \\
W_8 &= W_4 \oplus S(R(W_7)) \oplus rc_2 = \texttt{0x02000000} \\
W_9 &= W_5 \oplus W_8 = \texttt{0x00000000} \\
W_{10} &= W_6 \oplus W_9 = \texttt{0x00000000} \\
W_{11} &= W_7 \oplus W_{10} = |a, a, a, a| \\
W_{12} &= W_8 \oplus S(R(W_{11})) \oplus rc_3 = \texttt{0x06000000} \\
W_{13} &= W_9 \oplus W_{12} = \texttt{0x06000000} \\
W_{14} &= W_{10} \oplus W_{13} = \texttt{0x06000000} \\
W_{15} &= W_{11} \oplus W_{14} = |a \oplus \texttt{0x06}, a, a, a| \\
W_{16} &= W_{12} \oplus S(R(W_{15})) \oplus rc_4 = |\texttt{0x0e}, 0, 0, S(a \oplus \texttt{0x06})| \\
W_{17} &= W_{13} \oplus W_{16} = |a \oplus \texttt{0x08}, 0, 0, S(a \oplus \texttt{0x06})| \\
W_{18} &= W_{14} \oplus W_{17} = |a \oplus \texttt{0x0e}, 0, 0, S(a \oplus \texttt{0x06})| \\
W_{19} &= W_{15} \oplus W_{18} = |a \oplus \texttt{0x08}, a, a, a \oplus S(a \oplus \texttt{0x06})| \\
W_{20} &= W_{16} \oplus S(R(W_{19})) \oplus rc_5 = |\texttt{0x1e}, 0, S(a \oplus S(a \oplus \texttt{0x06})), S(\texttt{0x0b})| \\
W_{21} &= W_{17} \oplus W_{20} = |\texttt{0x16}, 0, S(a \oplus S(a \oplus \texttt{0x06})), S(\texttt{0x0b}) \oplus S(a \oplus \texttt{0x06})| \\
W_{22} &= W_{18} \oplus W_{21} = |\texttt{0x18}, 0, S(a \oplus S(a \oplus \texttt{0x06})), S(\texttt{0x0b})| \\
W_{23} &= W_{19} \oplus W_{22} = |\texttt{0x1a}, a, a \oplus S(a \oplus S(a \oplus \texttt{0x06})), a \oplus S(a \oplus \texttt{0x06}) \oplus S(\texttt{0x0b})|
\end{aligned}$$