On Side Channel Vulnerabilities of Bit Permutations in Cryptographic Algorithms

Jakub Breier[1], Dirmanto Jap[2], Xiaolu Hou[3] and Shivam Bhasin[2]
[1]School of Computer Science and Engineering, NTU, Singapore
[2]Physical Analysis and Cryptographic Engineering
Temasek Laboratories, NTU, Singapore
[3]School of Computing, National University of Singapore, Singapore
Email: jbreier@jbreier.com, {djap, sbhasin}@ntu.edu.sg, ho0001lu@e.ntu.edu.sg

Abstract—Lightweight block ciphers rely on simple operations to allow compact implementation. Thanks to its efficiency, bit permutation has emerged as an optimal choice for state-wise diffusion. It can be implemented by simple wiring in hardware or shifts in software. However, efficiency and security often go against each other. In this work, we show how bit permutations introduce a side-channel vulnerability that can be exploited to extract the secret key from the cipher. Such vulnerabilities are specific to bit permutations and do not occur in other state-wise diffusion alternatives. We propose Side-Channel Assisted Differential-Plaintext Attack (SCADPA) which targets this vulnerability in the bit permutation operation. SCADPA is first experimentally demonstrated on PRESENT-80 on an 8-bit microcontroller, with the best case key recovery in 17 encryptions. Additionally, we adjust SCADPA to state-of-the-art bit sliced implementation from CHES'17 with experimental evaluation on 32-bit microcontroller. The attack is then extended to latest bit-permutation based cipher GIFT, allowing full key recovery in 36 encryptions. Application for reverse engineering of secret Sboxes in PRESENT-like proprietary ciphers is also shown.

1 Introduction

The area of smart devices brings forward the question about energy efficiency. In the past, where desktop computers were the norm, chip manufacturers could afford to release devices with high computational power at the expense of the power consumption. However, this trend is changing rapidly, thanks to shrinking size of mobile devices and emergence of Internet of Things (IoT). Often, the raw computational power is lowered, while keeping the battery last longer. It is therefore crucial for the algorithm designers to develop smaller and more efficient designs that would fit such resource-constrained devices. This affects the area of cryptography as well, where we can see more efficient algorithms emerge every year. Recently, *NIST* has launched a call for proposal to standardize lightweight crypto algorithms [1].

This work focuses on lightweight block ciphers which use bit permutation based diffusion layer to achieve efficient implementations. Common examples of such block ciphers are PRESENT [2], which is an *ISO* standard, GIFT [3], etc. Bit permutation is an interesting design choice as it has negligible area footprint in hardware where it be implemented with only wires [2]. Thanks to this property, implementers constantly find ways to deploy such ciphers in software more efficiently as well [4], so that ciphers like PRESENT could be used universally. Other ciphers, such as GIFT, are optimized by design for both hardware and software. As a consequence, bit permutation based ciphers are becoming one of the preferred choices for IoT [5] where mostly software implementations are used.

In this paper, we bring forward a specific vulnerability of bit permutation based diffusion functions, which can be simply exploited using side-channel. The exploit arises from the simple structure of bit permutation and is not easily found in diffusion functions of standard ciphers (like MixColumns in AES). The demonstrated vulnerability is more serious in low-cost platforms like 8-bit microcontrollers due to serialized execution of the algorithm. Such design vulnerabilities further make the need of countermeasures critical, however, lightweight and countermeasures do not often go hand-in-hand. To resist theoretical attacks, cipher designers add extra rounds to avoid vulnerabilities due to simple diffusion layer. Since the proposed attack exploits all the information in the first round, extra rounds will not add any security.

A bit permutation layer diffuses the output bits of an Sbox (non-linear Substitution layer) to multiple Sboxes. By observing the numbers of affected Sboxes in a given round, the (key-dependent) Sbox output in the previous round can be determined, thus revealing information about the secret key. In this paper, we present *Side-Channel Assisted Differential-Plaintext Attack* (SCADPA) which exploits bit permutation construction for secret key retrieval through observed side-channel leakage. Here, SCADPA is chosen plaintext attack, where the plaintexts are chosen to effectively exploit the bit-permutation leakage. It observes the difference of propagation through side-channel, thus revealing the differential of Sbox output. With the knowledge of the plaintext, this differential can be solved to reveal the corresponding key candidates.

Unlike usual side-channel attacks (SCA [6]), SCADPA is not a statistical attack but rather a differential attack. For a carefully chosen set of plaintexts, it observes differential on an internal sensitive value. As we show later, these dif-

ferentials for bit permutation ciphers can be obtained by simply subtracting the power measurements. Once the internal differentials are known, the attack is a (classical) differential cryptanalysis for secret key retrieval. While cryptanalysis can be applied on a full cipher and handle high complexity, we manage to restrict the attack to a single round, thus keeping the complexity negligible. In some cases, multiple plaintext pairs might be needed to determine a unique key candidate. With the demonstrated experiments on 8-bit implementation PRESENT-80, SCADPA can reveal the key in 17 encryptions for the best case and 65 in the worst case. Additionally, we target a state-of-the-art bitsliced implementation from Reis et al. presented at CHES'17 [4], with experimental results on 32-bit ARM microcontroller. Further, we extend our attack on implementation of GIFT-64-128. The methodology is also capable of reverse engineering secret Sboxes in PRESENT-like ciphers with $17 - 46$ encryptions.

1.1 Related Works

Chosen plaintext side-channel attacks have previously been proposed in different context. A side-channel based collision attack [7] was proposed under chosen plaintext setting. It detects collision in some internal value of initial rounds of the cipher to retrieve the key. This attack was extended to break secret AES-like ciphers [8]. Some proposed attacks also used chosen plaintext setting to amplify power [9] or timing [10] side-channel leakage. Apart from block ciphers, chosen plaintexts attack were also applied to break public key cryptography [11] and hash functions (HMAC [12]). Recently, a chosen plaintext attack on DES third round was proposed in [13], exploiting the Feistel structure in the design. The key motivation of the attack was that most countermeasures protect only corner rounds for area-security trade-off, enabling attack on internal rounds. To the best of our knowledge, SCADPA is the first attack proposition to exploit the diffusion function in a block cipher.

Reverse Engineering via Side-Channel. Side-Channel Analysis for Reversed Engineering (SCARE) was first introduced in by Novak et al. [14], who proposed a recovery of one of the two secret Sboxes of algorithm A3/A8 used in GSM. Rivain and Roche [15] showed how to recover an equivalent representation of SPN cipher with a fixed first word of a round key. The most recent article up to date, published by Clavier et al. [16] shows a recovery of the secret cipher blocks of AES-like ciphers by both side-channel analysis and ineffective fault analysis.

1.2 Our Contribution

The contributions of this paper are as follows[1]:

- We identify a specific vulnerability in bit-permutation based diffusion functions exploitable through side-channel, and we propose a specific attack called SCADPA, which exploits the identified vulnerability.
- We present a practical demonstration of SCADPA on a low cost platform, using PRESENT-80 lightweight cipher as a target algorithm.
- We provide a comprehensive analysis of SCADPA in numerous test scenarios including application to 8-bit and 32-bit architectures.
- We further extend the application to bit sliced software implementations, including the current state-of-the-art implementation of PRESENT [4].
- We further evaluate SCADPA on GIFT, a recent bit-permutation cipher designed for optimal performance across platforms.
- We simulate the difference observation under different signal-to-noise ratio scenarios to show the practicality of SCADPA.
- As a minor contribution, we extend SCADPA methodology to reverse engineer secret Sbox in PRESENT-like ciphers. To the best of our knowledge, this is the first practical demonstration of reverse engineering of secret ciphers.

Organization. The rest of the paper is organized as follows. Section 2 describes the key methodology of SCADPA. Experimental validation of SCADPA on an 8-bit microcontroller is presented in Section 3. Section 4 describes the attack on block cipher GIFT. Section 5 shows how SCADPA can be extended to 32-bit architectures, with minor modifications. Some discussions are provided in Section 7, exploring attacks on countermeasures, different block cipher operation modes, vulnerability of other diffusion functions etc. Extension of SCADPA to recover secret Sboxes is provided in Section 8. Final conclusions are drawn in Section 9.

2 SCADPA Methodology

In this part we first focus on permutation layer of PRESENT, while detailing the properties that are exploited by SCADPA in Section 2.1. Then we explain how the SCADPA method works in Section 2.2, followed by steps to choose optimal plaintexts for the attack in Section 2.3. Next, we provide an attack example in Section 2.4. Finally, we state the attack complexity in Section 2.5. Although this section is based on PRESENT-80, the techniques are generally applicable on similar ciphers. Full description of PRESENT cipher [2] can be found in Appendix B.

2.1 Bit Permutation Properties of PRESENT-80

There are three main properties of the pLayer, resulting from the optimal diffusion requirement, that are exploited in the following:

1) Output of one nibble is distributed into four distinct nibbles.

1. This paper is an extended version of [17]. While the original paper outlines the basic method to use Side-Channel Assisted Differential Plaintext Attacks (SCADPA), this paper provides new methods, targets, results, and use cases. More specifically, we add the following:
   - extension of SCADPA to lightweight cipher GIFT,
   - extension of SCADPA to 32-bit architectures,
   - extension of SCADPA to bitsliced implementations supported by experimental results,
   - simulation of difference observation under different SNR scenarios,
   - new method to reverse engineer Sboxes, with results on PRESENT and GIFT.

2) Input to one nibble consists of outputs from four distinct nibbles.
3) In `pLayer`, the cipher state can be split into four different groups of four nibbles, where one input group affects exactly one output group.

Examining these properties, it can be seen that by changing chosen four nibbles in the plaintext, it is possible to affect the whole cipher state after the first permutation.

Figure 1 shows this behavior by changing the first and the eighth nibble of the plaintext. The underlying implementation computes `sBoxLayer` nibble-wise while `addRoundKey` is computed byte-wise owing to the ALU support for bit-wise `xor`. This is to achieve the best speed-memory trade-off. Similarly, in 32-bit architectures, `sBoxLayer` would be implemented as a 4-bit or 8-bit look-up table and `addRoundKey` with `pLayer` would be done on 32-bit words, to achieve best speed-memory trade-off.

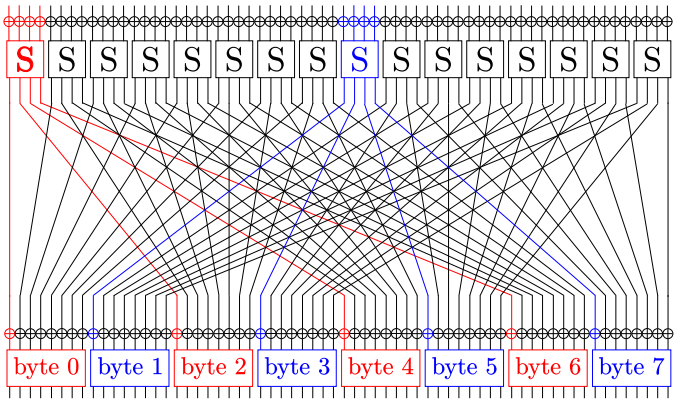


Fig. 1: Bytes in round 2 that could be potentially affected by changing the first and the eighth nibble of the plaintext.

By observing the changing nibbles in round 2, the change in Sbox output at round 1 can be determined. This value directly depends on the secret key which can be exploited for key retrieval. In the following, we use side-channel measurements to observe the changed nibbles.

2.2 SCADPA Procedure

Using the information from the previous part, we propose SCADPA. SCADPA exploits the permutation properties to observe changed nibbles and retrieve differential at Sbox output in round 1. The differential can be solved for key retrieval by using a standard differential attack on non-linear layer.

The attack steps can be summarized as follows:

Step 1: Encrypt a chosen plaintext p, by using an unknown secret key k, denoted as $E_k(p)$.
Step 2: Capture the power/EM leakage of the Device Under Test (DUT) during the second encryption round to get the trace t.
Step 3: Choose another plaintext $p' \neq p$ that differs exactly in one nibble at i^{th} position. The nibble at position i in plaintext p is denoted p_i and $x_i = p_i \oplus k_i$. Similarly, $x'_i = p'_i \oplus k_i$.
Step 4: Capture the leakage for $E_k(p')$ to get t'.
Step 5: Calculate $\Delta t = t - t'$.
Step 6: By examining Δt, get the Sbox output change $\Delta S_i = S(x_i) \oplus S(x'_i)$ of round 1 in the position where the plaintext had changed.
Step 7: Calculate all possible candidates for key nibble k_i such that with input pair p_i and p'_i, the change determined in Step 6 would appear.
Step 8: Repeat steps 3-7 with another p'_i, taking intersection of all the calculated key candidates, until there is just one candidate for k_i.
Step 9: Repeat steps 3-8 for all $i \in [0, 15]$ to recover the whole round key.
Step 10: Compute the remaining 16 bits of the secret key by exhaustive search or repeating SCADPA on the next round.

2.3 SCADPA Acceleration by Optimal Plaintext Choice

In order to reduce the key complexity and retrieve the secret key with fewer number of encryptions, it is possible to change the value of multiple nibbles in the plaintext. Based on the permutation layer, multiple nibbles can be changed without affecting same locations in the next round and hence, can be analyzed independently. The optimal methodology executes the following steps:

Step 1: Keep the record of nibbles of key that have not been recovered ($I = \{0, \ldots, 15\}$)
Step 2: Start by choosing one nibble ($n_i \in I$) and calculate all possible affected nibbles at the beginning of the next round (N_i).
Step 3: Choose another nibble ($n_j \in I$, $n_j \neq n_i$) and check if the affected nibbles interfere (check if $N_i \cap N_j = \emptyset$). If true, keep n_j, else, move to other nibble . Repeat, until no nibble remaining, then update $I \setminus \{n_i, n_j, ...\}$.
Step 4: Choose plaintext set that changes only on these nibble positions ($\{n_i, n_j, ...\}$).
Step 5: Repeat step 2-4, until $I = \emptyset$

Another option is to choose the pair difference in the plaintext that could minimize the key candidate. This is dependent on the Sbox used. In case of `PRESENT` the number of differences to identify a unique key nibble candidate is 2 (as can be seen from the extended DDT in Table 1).

A natural question would be – 'how many nibbles of `PRESENT` can we attack at once by using SCADPA?' As can be seen in Figure 1, one nibble can affect up to half of the state at the next round `addRoundKey`. Nibbles $0-7$ ("group 1") affect bytes $0, 2, 4, 6$, while nibbles $8-15$ ("group 2") affect the remaining bytes $1, 3, 5, 7$. Therefore, by combining one nibble from each group, we can retrieve two nibbles at the same time while avoiding the interference. This knowledge can help us to reduce the number of encryptions to half. Parallelization of attack to two nibbles is limited by the byte-wise granularity of `addRoundKey`. The following `sBoxLayer` with nibble-wise granularity would allow up to 4 nibbles in parallel, however at the cost of needed profiling.

2.4 Attack Example

We illustrate our method by a straightforward example that shows how to recover one nibble of the round key i.e. $i = 1$. We fix p_i = `0x0` and p'_i = `0xA`. After observing the Δt, we figure out that ΔS_i = `0x2`. By knowing that Δp_i = `0xA`, there can be two candidates for k_i: `0xF` and `0x5`. We make another experiment, now with p'_i = `0x3`. By capturing another trace, we determine ΔS_i = `0x6`, giving us four different candidates for k_i: `0xC`, `0xD`, `0xE`, and `0xF`. The only intersecting candidate is `0xF`, therefore we know that the key nibble has to be this value. The retrieval of ΔS_i from real power traces is explained in the following section.

2.5 Attack Complexity

In this section, we compute the attack complexity for single key nibble retrieval and full round key retrieval.

The single nibble retrieval complexity depends on the underlying Sbox function, being the only non-linear function in the derived differential equation. Table 1 shows the extended Difference Distribution Table (DDT) for `PRESENT` Sbox. Input difference to the Sbox is denoted as δ, while the output difference is denoted as Δ. From this table, it can be observed that two input differences are needed to uniquely identify the Sbox input. Therefore, to fully recover the first round key with a chosen plaintext model and one nibble recovery at a time, the attack requires one reference trace and 32 difference traces, resulting to 33 encryptions in total. However, the method from Section 2.3 provides more optimized way to mount the attack, with two nibbles at a time, resulting to 17 encryptions in total.

Please note that it is also possible to make a search on remaining candidates. For example, if we have 2 candidates for each nibble, we can determine the value with a brute-force search with complexity of 2^{16}. For such case, it would only require 9 encryptions in case we target two nibbles at a time. Similarly, operating at nibble wise granularity at the following `sBoxLayer` can recover the key in 9 encryptions as well.

3 Experimental Results

In this part, we show and discuss experimental results obtained by performing SCADPA on a microcontroller implementation of `PRESENT`-80 cipher [2]. Sections 3.1 and 3.2 provide an overview of the experimental setup and results, respectively.

3.1 Setup

As a DUT, we used a standard 8-bit microcontroller from Atmel, ATmega328P, mounted on Arduino UNO development board. We have measured an electromagnetic emanation with a Langer RF-U 5-2 probe. Signal was captured with LeCroy WaveRunner 610 Zi oscilloscope.

We have used an implementation of `PRESENT`-80 cipher, where `sBoxLayer` is computed nibble-wise and the rest of the operations are done byte-wise. Sampling rate was set at 500 MS/s, while the `addRoundKey` takes ≈7000 samples. A difference introduced in the first round could be observed during the second round. We chose to observe the difference at the second `addRoundKey` as in our case the start of round was easily identifiable by visual inspection of the trace. Choice of observing the difference on `addRoundKey` has an advantage and a disadvantage. The advantage being that precise profiling was not needed as compared to locating time instants for `sBoxLayer`. The disadvantage is that since `addRoundKey` is done byte-wise, the observed differences are limited to byte level. Observed differences on the following `sBoxlayer` can be nibble precise, given appropriate profiling.

3.2 Results

To support our method, we have conducted experiments showing the possibility of distinguishing ΔS.

Figure 2 shows differences in power consumption captured in the second `addRoundKey`, by calculating Δt. The implementation we used computes the `addRoundKey` in a reverse order, therefore the difference peaks follow this order. In order to improve signal-to-noise ratio and produce clear plots, both t and t' were averaged from 50 executions. Nevertheless, it was possible to see the difference in raw traces. By observing this difference, the output difference of `sBoxLayer` in round 1, i.e. ΔS_i, can be clearly distinguished. Once ΔS_i and Δp_i are known, it can be used to solve the value of secret key k_i, as shown in the previous section.

4 SCADPA Applied on Lightweight Cipher GIFT

`GIFT` [3] is a lightweight block cipher, designed to improve the efficiency and correct the weaknesses of `PRESENT`. Similar to `PRESENT`, `GIFT` is based on Substitution-Permutation Network (SPN), where each round is composed of substitution layer (`SubCells`), permutation layer (`PermBits`) and key addition (`AddRoundKey`). `GIFT` has 128-bit key and supports either 64-bit or 128-bit plaintext. The design of 4-bit Sbox (`SubCells`, see Table 2) and the permutation layer of `GIFT` is different from `PRESENT`, however, the idea of SCADPA can still be applied.

Given the case of `GIFT`-64-128 (64-bit), the first 32 bits of the input to the permutation layer will go to bytes 0, 2, 4, and 6, whereas the other half will go to the remaining bytes. In general, the diffusion of each nibble after the `SubCells` can be formulated as shown in Table 3. In summary, each bit in the nibble will go to different bytes in the subsequent round and the order of the targeted byte is as denoted in the table. Here, $[i_0, ..., i_{j-1}]_{\gg n}$ denotes right circular shift by n elements in the array. In this case, SCADPA can be applied using the same rationale as for the case of `PRESENT`-80, described earlier, since the affected bytes follow the same diffusion pattern.

One of the challenges is due to the key addition performed in `GIFT`. Here, the key is separated into eight 16-bit words. Two words are used in each round, that is, for `GIFT`-64-128, there will only be 32-bit key for each round. For each round, the key is extracted as follows: take the first two words, k_1 and k_0 (referred to as U and V, respectively). At the next round, the key is rearranged as follows:

$$k_7||k_6||\dots||k_1||k_0 \leftarrow k_1 \ggg 2||k_0 \ggg 12||\dots||k_3||k_2.$$

Δ \ δ	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1			9a		36		078f				5e		1c		24bd
2						8e	34		09	5f		1d	67ab	2c	
3	cdef	46	12					3b		0a			58	79	
4			47		8d				35ac		0b		2f		169e
5		cdef		0145						2389		67ab			
6		9b	cdef	37		06	25		18					4a	
7	67ab		03	8c				5d				2e	49	1f	
8						17	ad		6f	4e	2389	0c		5b	
9	0145			9d	be			2a			7c	3f		68	
a		02	56	bf	9c					7d	1a	48	3e		
b			8b		27	35ac		169e			4f		0d		
c		8a		26	0145	9f	bc		7e					3d	
d	2389	57			af			4c		1b	6d			0e	
e		13		ae					24bd	6c		59			078f
f						24bd	169e	078f							35ac

TABLE 1: Extended difference distribution table for PRESENT Sbox. Columns represent input difference, rows represent output difference and entries are Sbox inputs.

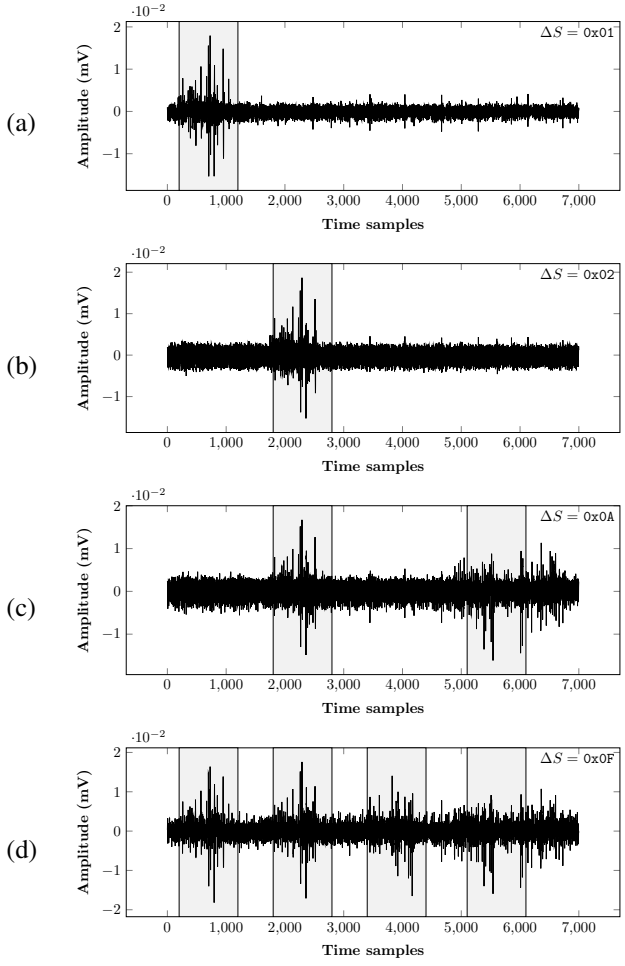


Fig. 2: Difference traces Δt showing addRoundKey of round 2, revealing the output difference of the Sbox at round 1. Bytes are processed in a reversed order, therefore the pattern showing the difference also has to be reversed. Difference between the Sbox outputs for the plots (highlighted by the gray background) is as follows: (a) 0x01, (b) 0x02, (c) 0x0A, and (d) 0x0F.

x	0	1	2	3	4	5	6	7
$S(x)$	1	A	4	C	6	F	3	9
x	8	9	A	B	C	D	E	F
$S(x)$	2	D	B	7	5	0	8	E

TABLE 2: GIFT SubCells.

Here, $\ggg i$ is an i bits right rotation within a 16-bit word. The round keys U and V are then xor-ed with the state as follows: $b_{4i+1} \leftarrow b_{4i+1} \oplus u_i$ and $b_{4i} \leftarrow b_{4i} \oplus v_i$, $\forall i \in \{0, ..., 15\}$. A single bit (set to 1) and 6-bit round constant are also added to bits 63/127 (most significant bit, depending on the version of algorithm), 23, 19, 15, 11, 7, 3. Thus, for the attack, in each round, only 32 bits of the key can be recovered.

Bits in targeted nibble i	Bytes affected in next round
$[b_0, b_1, b_2, b_3]_{0 \le i \le 7}$	$[0, 2, 4, 6]_{\gg (i \bmod 4)}$
$[b_0, b_1, b_2, b_3]_{8 \le i \le 15}$	$[1, 3, 5, 7]_{\gg (i \bmod 4)}$

TABLE 3: GIFT: affected bytes after bit permutation for each nibble.

In order to recover the rest of the key, the attack can then be performed on multiple consecutive rounds. Since the first round key has been recovered, to achieve the differential at the next round, the attacker can just "peel off" the first round and continue with SCADPA in the same way on the subsequent round. To recover the whole 128-bit key, this has to be repeated 4×, or 3× with brute-forcing the last 32 key bits. In Figure 3, we show the result of attacking GIFT-64-128. As can be seen in the plot, 68 traces are required to recover all the 128 key bits in case only one nibble is changed each time. In the best case, when two nibbles are targeted at a time, the attack requires 36 traces (4 reference traces and then 32 differential traces). This is due to the reason that for GIFT, on top of the longer key length, with each nibble we can only recover 2 bits, compared to 4 bits for PRESENT.

5 Extension of SCADPA to 32-bit Architectures

When considering the attack on 32-bit architectures, one has to distinguish between different possibilities of implementing the

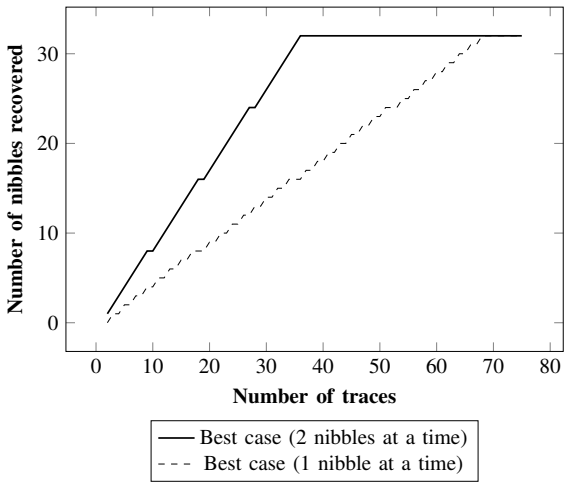


Fig. 3: The number of key nibbles retrieved by given number of traces attacking GIFT-64-128 targeting multiple rounds.

round function. In case the operations are computed on 4/8-bit blocks, the attack can be carried out in the same way as in Section 2.2. However, in case `addRoundKey` is computed as xor of 32-bit blocks, the attacker gains less information about the processed data and therefore, an updated attack strategy has to be used. This behavior is depicted in Figure 4, where one can easily observe that two output bits of each Sbox go to one word while the other two bits go to another word. In this part, we will describe the method of use for such case.

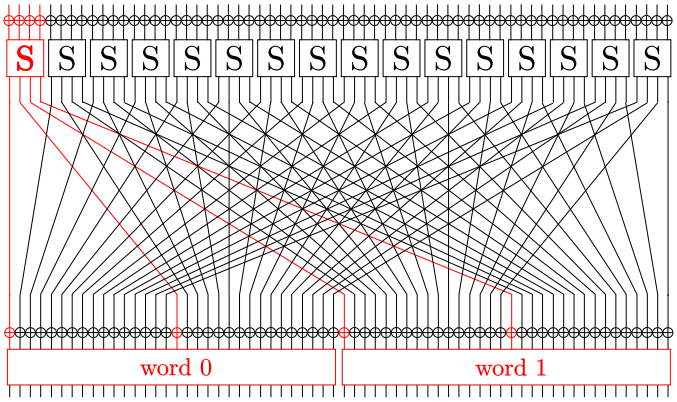


Fig. 4: `PRESENT` implemented on 32-bit architecture, where `addRoundKey` is computed on 32-bit words. One can easily see that from one Sbox, the output bits are split equally between the two words.

When the attacker obtains the difference of the traces, Δt, she can distinguish three possible cases when observing the `addRoundKey`:

- only `word0` has changed, indicated by $Y|N$,
- only `word1` has changed, indicated by $N|Y$,
- both words have changed, indicated by $Y|Y$.

Step 6 in SCADPA procedure (Section 2.2) will be changed as follows:

Step 6: By examining Δ_t, get the change of Sbox output between $S(x_i)$ and $S(x_i')$, which is denoted by $Y(N)|Y(N)$ as indicated above.

For `PRESENT` Sbox, the information the attacker can get is stated in Table 4, which represents a "compressed" DDT with output difference following the three cases described above. Now, she can construct an optimal attack strategy that will require the lowest number of traces, depending on the chosen plaintext. She will iterate through possible combinations of input differences until she finds a combination that yields a unique solution for the Sbox input. To be more specific, for `PRESENT` Sbox, the minimal number of input differences needed is 4 (calculated based on Table 4). The calculation process can be done by exhaustive search, where we try all the possible combinations of different values of δ starting with 2 faults, and increasing the number of faults until a unique distribution of output differences can be extracted from Table 4 for each Sbox input value. One such combination is stated in Table 5, for differences `0x3, 0x7, 0xd, 0xe`. For example, the attacker first measures the trace for $p = 0$, which will serve as a reference trace. Then to attack ith nibble of the first round key she measures the traces for p_i =`0x3, 0x7, 0xd, 0xe`, respectively. The four differences between the four traces and the reference trace will uniquely determine the input of Sbox for the first encryption, i.e. $p_i \oplus k_i$. Considering that attacker needs one reference trace and 4 other traces per nibble, the total number of chosen plaintexts to fully recover the round key is 65.

5.1 More Precise Attacker Model

Now, let us consider a more powerful attacker where she is able to distinguish not only whether there is a change in each of the 32-bit words, but also what is the Hamming weight (HW) of such change. Such model requires the device to leak according to HW leakage model and the attacker to have equipment precise enough to determine the HW from the power/EM leakage. This knowledge can help her distinguish how many bits of the Sbox output were changed and therefore, she needs lower number of encryptions to recover the key. Let us denote this difference as $\Delta_{HW_j} = HammingWeight(word_j \oplus word_j')$, where $j \in 0, 1$ is index of the word. Thus for two different plaintext nibbles p_i, p_i', Δ_{HW_0} and Δ_{HW_1} are the Hamming weights of the first and last two bits of $S(k_i \oplus p_i) \oplus S(k_i \oplus p_i')$ respectively. Step 6 in SCADPA procedure (Section 2.2) will be changed as follows:

Step 6: By examining Δ_t, get the change of Sbox output between $S(x_i)$ and $S(x_i')$: Δ_{HW_0} and Δ_{HW_1}.

For `PRESENT` Sbox, Table 6 shows the DDT for this case – columns denoting the input difference and rows denoting Δ_{HW_i} for each word. One can easily see that for such case, the attacker only needs two difference traces on top of the reference trace to uniquely identify the Sbox input. Therefore, she needs 33 encryptions in total, same as the single nibble attack described for 8-bit architecture.

6 Extension to Bit-Sliced Implementations

To show the practicality of the attack, we extend its application to bit-sliced implementations. Bit slicing is currently the most popular way to implement block ciphers in software due to

w0|w1 \ δ	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$N\|Y$	cdef	46	129a		36	8e	03478f	3b	09	05af	5e	1d	15678abc	279c	24bd
$Y\|N$		8a	47	26	01458d	179f	abcd		3567acef	4e	02389b	0c	2f	35bd	169e
$Y\|Y$	012345 6789ab	012357 9bcdef	03568 bcdef	0134578 9abcdef	279a bcef	02345 6abcd	12569e	0124567 89acdef	1248bd	12367 89bcd	1467 acdf	234567 89abef	0349de	0146 8aef	0357 8acf

TABLE 4: Extended difference distribution table for PRESENT Sbox, where the columns follow the input difference, rows follow the change of the two output words and entries represent the Sbox inputs – w0 denotes word0, w1 denotes word1.

Input \ δ	3	7	d	e
0	■		■	■
1		■		■
2		■	■	
3	■		■	■
4	■		■	■
5	■	■		■
6	■	■		■
7	■			
8	■			■
9		■	■	
a		■		■
b	■	■		■
c	■	■		
d	■	■	■	■
e	■	■	■	■
f	■		■	■

TABLE 5: Example of output differences for input difference 0x3, 0x7, 0xd, 0xe. White color indicates $Y|N$, red color $N|Y$, and black color indicates both words changed – $Y|Y$. Each row is unique, and therefore, enables recognition of the Sbox input.

Algorithm 1: High-level overview of "PRESENT Runs Fast" implementation [4].

Input : A 64-bit block of plaintext B, a key K.
Output: A 64-bit block of ciphertext C.

1 $subkey :=$
  $(subkey_1, subkey_2, \ldots, subkey_{32}) \leftarrow keySchedule(K)$;
2 $C \leftarrow B$;
3 **for** *int* $i := 1$ *to* 15 **do**
4 $C \leftarrow C \oplus subkey_{2i-1}$;
5 $C \leftarrow P_0(C)$;
6 $C \leftarrow S_{BS}(C)$;
7 $C \leftarrow P_1(C)$;
8 $C \leftarrow C \oplus subkey_{2i}$;
9 $C \leftarrow S_{BS}(C)$;
10 $C \leftarrow C \oplus subkey_{31}$;
11 $C \leftarrow P(C)$;
12 $C \leftarrow S_{BS}(C)$;
13 $C \leftarrow C \oplus subkey_{32}$;
14 **return** C;

increased efficiency. In this section, we focus on current state-of-the-art bitsliced implementation of PRESENT, introduced at CHES 2017, entitled "PRESENT Runs Fast" [4].

6.1 PRESENT Runs Fast and SCADPA

This implementation is focused on 32-bit architectures with 16-bit registers. A high-level overview of the implementation is stated in Algorithm 1. Please note that in the algorithm we keep the notation used in the original paper to avoid confusion. In this implementation, the application of two original PRESENT permutations P is replaced by permutations P_0 and P_1 which satisfy the property $P_1 \circ P_0 = P^2$. Application of these permutations on state B is specified in Figure 5. After P_0, Sbox is applied in a bit-sliced fashion (S_{BS}), followed by P_1.

Listing 1: Implementation of permutation P_1 in C. X0 - X3 are 32-bit variables.

```
#define PRESENT_PERMUTATION_P1(X0,X1,X2,X3)
 1    t = (X0^(X1 >>4)) & 0x0F0F0F0F;
 2    X0 = X0^t;
 3    X1 = X1^(t<<4);
 4    t = (X2^(X3 >>4)) & 0x0F0F0F0F;
 5    X2 = X2^t;
 6    X3 = X3^(t<<4);
 7    t = (X0^(X2 >>8)) & 0x00FF00FF;
 8    X0 = X0^t;
 9    X2 = X2^(t<<8);
10    t = (X1^(X3 >>8)) & 0x00FF00FF;
11    X1 = X1^t;
12    X3 = X3^(t<<8);
```

Now we can adjust the method described in Section 5 so that it will be possible to capture the first Sbox output difference at the second key addition (line 8 in Algorithm 1).

The first thing to notice is that the permutation P_0 arranges input bits of Sboxes into columns, i.e. a nibble i is located in column $Col = i \times 4 \ mod \ 15$ after the application of P_0. The Sbox S_{BS} is then applied in a bit sliced fashion. This does not change the working principle of SCADPA as the Sbox output corresponds to a standard one. The adjustment has to be made after the second permutation P_1, where it is clear that the 32-bit words that serve as the input to the second key addition are scrambled (see $P_0 \circ P_1(B)$ in Figure 5). To efficiently implement this permutation in 32-bit architectures, the algorithm uses a macro stated in Listing 1. The SCADPA attack for this implementation observes the Sbox output differences during the P_1. For example, if we take the output of the first Sbox, it is distributed into the first bit of each variable and after P_1, it ends up in the first, fifth, ninth, and thirteenth bit of X0. This is illustrated in Figure 6. As will be explained later in Section 6.2, the changes in variables X0 - X3 can be observed based on the order of operations in which they are processed in the macro. As the Sbox output bits change the variable during the operation, it is important to trace this change. For example, the first bit of the first Sbox

$\Delta_{HW_0}\|\Delta_{HW_1}$ \ δ	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0|1			9a		36	8e	03478f		09	5f	5e	1d	167abc	2c	24bd
1|0			47		8d	17	ad		356acf	4e	02389b	0c	2f	5b	169e
1|1	0145	029bcdef	56cdef	0134579bdf	9bce	06	25	2a	18	23789d	17ac	34678abf	3e	468a	
0|2	cdef	46	12					3b		0a			58	79	
2|0		8a		26	0145	9f	bc		7e					3d	
1|2	67ab		038b	8c	27	35ac		1569de			4f	2e	049d	1f	
2|1	2389	1357		ae	af			4c	24bd	16bc	6d	59		0e	078f
2|2						24bd	169e	078f							35ac

TABLE 6: Extended difference distribution table for PRESENT Sbox, where the columns follow the input difference and rows follow the Hamming weight of the two output words.

$$B = \begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 08 & 09 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{bmatrix},$$

$$P_0(B) = \begin{bmatrix} 00 & 16 & 32 & 48 & 04 & 20 & 36 & 52 & 08 & 24 & 40 & 56 & 12 & 28 & 44 & 60 \\ 01 & 17 & 33 & 49 & 05 & 21 & 37 & 53 & 09 & 25 & 41 & 57 & 13 & 29 & 45 & 61 \\ 02 & 18 & 34 & 50 & 06 & 22 & 38 & 54 & 10 & 26 & 42 & 58 & 14 & 30 & 46 & 62 \\ 03 & 19 & 35 & 51 & 07 & 23 & 39 & 55 & 11 & 27 & 43 & 59 & 15 & 31 & 47 & 63 \end{bmatrix},$$

$$P_1(B) = \begin{bmatrix} 00 & 01 & 02 & 03 & 16 & 17 & 18 & 19 & 32 & 33 & 34 & 35 & 48 & 49 & 50 & 51 \\ 04 & 05 & 06 & 07 & 20 & 21 & 22 & 23 & 36 & 37 & 38 & 39 & 52 & 53 & 54 & 55 \\ 08 & 09 & 10 & 11 & 24 & 25 & 26 & 27 & 40 & 41 & 42 & 43 & 56 & 57 & 58 & 59 \\ 12 & 13 & 14 & 15 & 28 & 29 & 30 & 31 & 44 & 45 & 46 & 47 & 60 & 61 & 62 & 63 \end{bmatrix},$$

$$P_0 \circ P_1(B) = \begin{bmatrix} 00 & 16 & 32 & 48 & 01 & 17 & 33 & 49 & 02 & 18 & 34 & 50 & 03 & 19 & 35 & 51 \\ 04 & 20 & 36 & 52 & 05 & 21 & 37 & 53 & 06 & 22 & 38 & 54 & 07 & 23 & 39 & 55 \\ 08 & 24 & 40 & 56 & 09 & 25 & 41 & 57 & 10 & 26 & 42 & 58 & 11 & 27 & 43 & 59 \\ 12 & 28 & 44 & 60 & 13 & 45 & 61 & 31 & 14 & 30 & 46 & 62 & 15 & 31 & 47 & 63 \end{bmatrix},$$

Fig. 5: Application of permutations P_0 and P_1 on state B according to fast software PRESENT-80 implementation specified in [4].

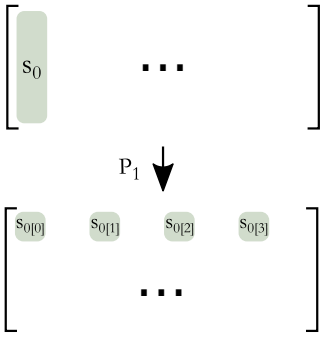


Fig. 6: Distribution of the first Sbox output after applying P_1.

TABLE 7: Sbox output difference observation based on processing the data by the macro in Listing 1. Details on how these numbers were determined are stated in Appendix A.

Sbox bit	Variable	Lines in Listing 1
0	X0	1, 2, 7, 8
1	X1	1, 2, 3, 7, 8
2	X2	4, 5, 7, 8, 9
3	X3	4, 5, 6, 7, 8, 9

is located in variable X0 at the beginning, and therefore its change can be determined during the processing of lines 1, 2, 7, and 8. The second bit is located in X1 at the beginning and therefore, can be observed from lines 1, 2, 3, 7, and 8. Full details are stated in Table 7. As one can see, the difference between the first and the second Sbox output might be hard to determine, since there is only one additional line of code for the second bit. However, if one can determine whether there was a change either in the first two bits or the second two bits, the same method as in Section 5 could be applied, following the DDT stated in Table 4.

6.1.1 Implementation

The bitsliced PRESENT was implemented on ARM Cortex M3 on Arduino DUE platform. Code was written in C, compiled with a standard Arduino compiler. The same measurement setup as in Section 3.1 was used. For positioning of the probe, standard profiling was performed to find the part of the chip with the best SNR. The trigger was placed so that the whole permutation P_1 could be captured.

6.2 Results

Results for the attack described in previous parts are stated in Figure 7. Colors follow the notation from Table 7, where the lines processing the first two bits are denoted by blue color, the lines processing the second two bits are denoted by red color, and the lines that process all the bits are denoted by black color.

In Figure 7(a) and Figure 7(b), we targeted the change in the first, and the first and the second Sbox output bits, respectively. Based on Listing 1 and Table 7, the difference for changes in variables X0 and X1 should be observed in peaks 1, 2, 3, 7, and 8. This is consistent with the plots, where we can see two distinct groups of peaks.

In Figure 7(c) and Figure 7(d), we targeted the change in the third, and the third and the fourth Sbox output bits, respectively. Based on Listing 1 and Table 7, the difference for changes in variables X2 and X3 should be observed in peaks 4, 5, 6, 7, 8, and 9. This is again consistent with

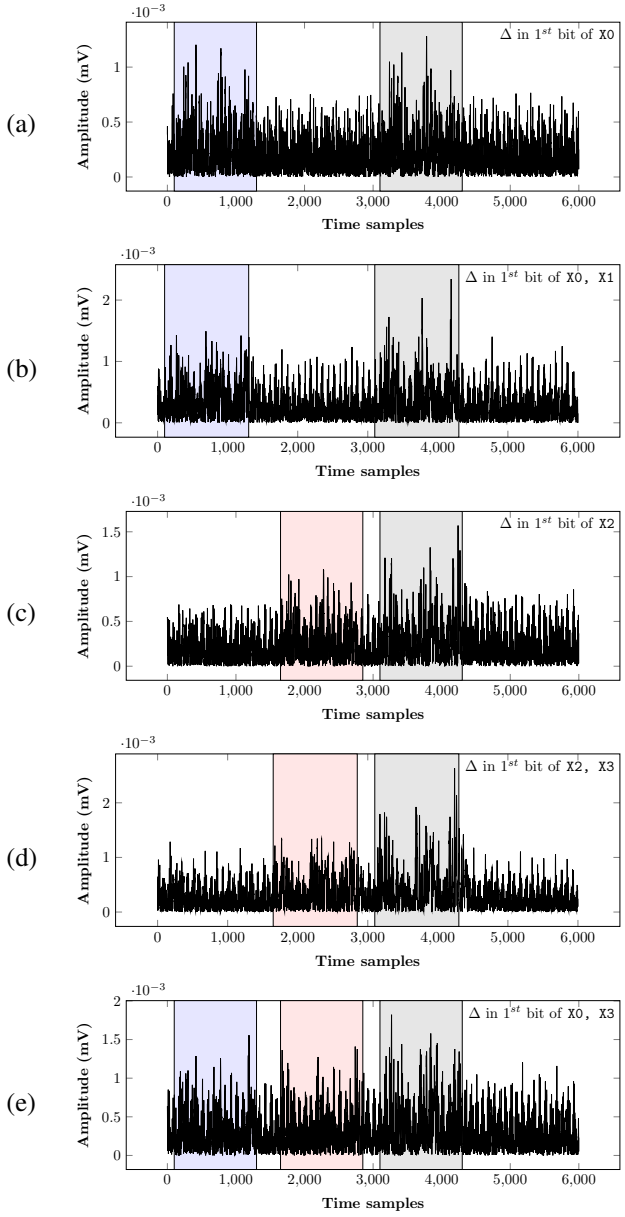


Fig. 7: Difference traces for implementation from [4] showing the first execution of P_1, revealing the output difference of the Sbox at round 1. Difference between the Sbox outputs for the plot is always in the first bit of the following variables: (a) `X0`, (b) `X0, X1`, (c) `X2`, (d) `X2, X3`, and (e) `X0, X3`.

the plots, this time there is no significant difference in the beginning part, corresponding to variables `X0` and `X1`, but one can notice the difference peaks right after this part. As the peak corresponding to Line 9 could not be clearly differentiated from the previous plots, we did not consider it during our analysis.

Finally, Figure 7(e) shows the change in the first and the fourth Sbox output bits. Therefore, all the three areas where one can spot the difference are active, which is in line with our attack method. We can therefore confirm that SCADPA methodology is capable of breaking the state-of-the-art bitsliced implementation from [4].

7 Discussion

In this part we will first discuss the difference observation under different SNR scenarios in Section 7.1 Later, we discuss how side-channel countermeasures affect the successful application of SCADPA in Section 7.2. Next, we show possibilities of attacking different block cipher modes of operation in Section 7.3. Finally, in Section 7.4 we state alternatives for permutation layer as well as countermeasures that prevent successful application of SCADPA.

7.1 Signal-to-Noise Ratio

An important question is what signal-to-noise ratio (SNR) is enough for distinguishing the difference with SCADPA. We simulated the behavior for different levels of SNR, considering a stochastic leakage model for the device under test. This model considers different contribution of every bit to the overall leakage, and was shown as a good approximation of the real device leakage. Before the simulation, we have obtained the leakage coefficients from ATmega328P chip for `EOR` instruction (Exclusive OR). The values for the coefficients are as follows: $\beta = \{-4.1343, -3.7208, -3.6601, -3.2879, -3.5317, -2.9721, -5.3845, -3.5531\}$.

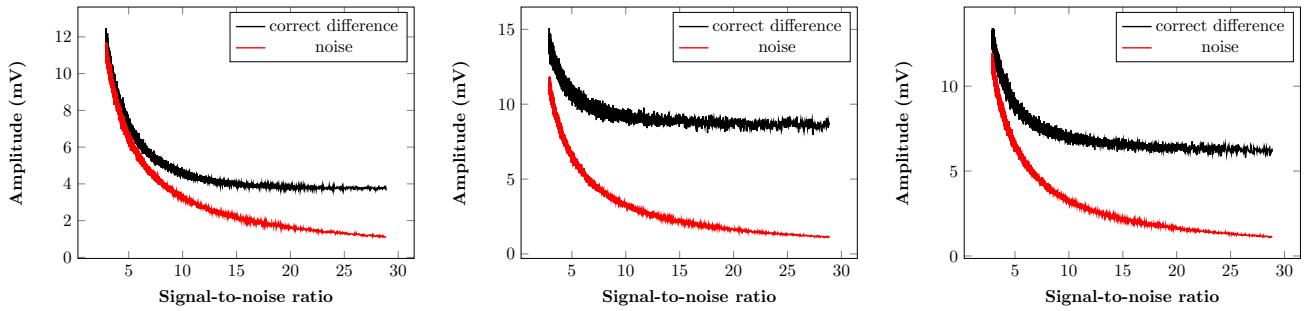
Results are shown in Figure 8. There are three scenarios in total, simulating different cases that can be observed during the attack. The worst case scenario in Figure 8(a) shows the evolution of correct difference and the noise in case the difference in the XOR computation was in one bit only. The best case scenario in Figure 8(b) shows the evolution of correct difference and the noise in case the difference in the XOR computation was in all 8 bits. Finally, the average case scenario in Figure 8(c) shows the evolution of correct difference and the noise in case the difference in the XOR computation was in 4 random bits.

The results show that even in low SNR scenario and the worst case for the attacker (1-bit difference), it is still possible to recognize the difference.

7.2 Side-Channel Countermeasures

As SCADPA exploits leakage of bit permutation through side-channel, any countermeasure which randomizes side-channel information and/or decreases signal-to-noise ratio can protect against such attacks. This includes both hiding [18] and masking countermeasures [19]. However, any bias in the implementation of the countermeasure can still render the attack possible. For instance, an unbalanced implementation of hiding will still allow to observe the difference. Similarly for masking, ΔS would depend upon $p_i \oplus m_i \oplus p'_i \oplus m'_i$. If masks m_i, m'_i are totally independent and uniformly distributed, the attack is not possible, however with biases in the mask, the attack can still be carried out with increased effort. Shuffling of the order of the Sboxes and/or key additions would make the attack harder since only the Hamming weight could be directly observed, instead of the difference value.

Nevertheless, countermeasures can incur significant overheads. Thus, for lightweight cryptography specially oriented for low-cost platforms, the functions must be carefully chosen to avoid such vulnerabilities.



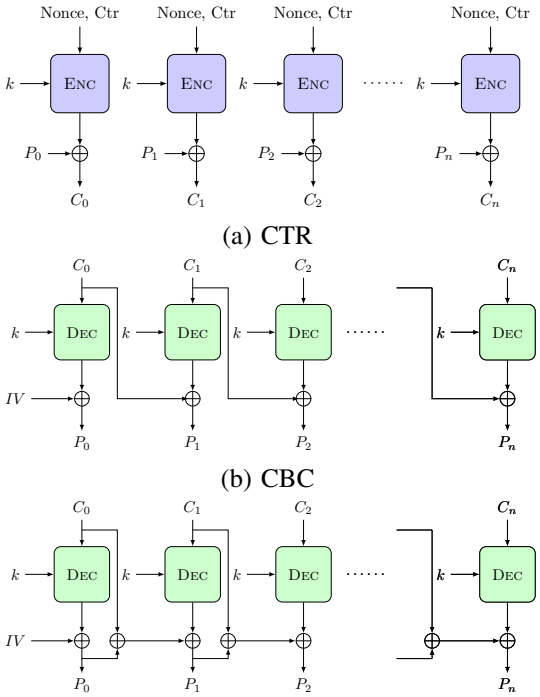
(a) Worst case (b) Best Case (c) Average case

Fig. 8: Signal-to-noise simulations of difference recognition using SCADPA for various scenarios.

7.3 Attack on Different Modes of Operation

Few block cipher operation modes are well oriented towards plaintext selection of SCADPA. When it comes to Counter (CTR) mode (Figure 9 (a)), the input to the encryption algorithm consists of a nonce and a counter. While the nonce is a random number, counter normally increases by 1 after each block. While the nonce stays fixed, the incrementing counter satisfies the chosen plaintext criteria of SCADPA as discussed in Section 2. The attack allows to recover few nibbles directly affected by the counter in the first round. For the remaining nibbles, chosen nonce or attack on second round can be carried out in a similar way.

On the other hand, when targeting Cipher Block Chaining (CBC) mode, one has to aim at the decryption module (Figure 9 (b)). This comes from the property of CBC where the plaintext is first `xor`-ed with the IV (first block) or with ciphertext from the previous block. In this case, chosen-plaintext attack changes to chosen-ciphertext, without the knowledge of plaintext. The same holds for Propagating Cipher Block Chaining (PCBC) mode.



(a) CTR

(b) CBC

(c) PCBC

Fig. 9: (a) Counter (CTR), (b) Cipher Block Chaining (CBC) and (c) Propagating Cipher Block Chaining (PCBC) modes of operation. SCADPA can be applied to encryption in case of CTR, and to decryption in case of CBC and PCBC.

7.4 Alternatives for Diffusion Layer

Certain diffusion function do not offer vulnerabilities that are exploited by SCADPA in bit permutation. `AES` [20], the *NIST* standard for symmetric key cryptography, is a relevant example. `AES` encrypts 128-bit data block with a 128/192/256 bits secret key in 10/12/14 rounds. The data is organised in a 4×4 matrix of bytes called state and the round function is applied upon it. A round comprises of four operations i.e. `SubBytes` (SB), `ShiftRows` (SR), `MixColumns` (MC) and `AddRoundKeys` (ARK). SB is a 8×8 non-linear table look up and ARK is the round key addition. We concentrate on diffusion functions i.e. SR and MC. SR applies a cyclic shift on the rows (1, 2, 3, 4) with offsets (0, 1, 2, 3). MC operates on four bytes of each column of the state. The four bytes are combined using an invertible linear transformation. When a difference is inserted at the input, irrespective of its value, the difference is always propagated on all four bytes, preventing ΔS leakage.

Recent trends show that it is possible to design a lightweight cipher with similar diffusion function and not only rely on bit permutations. `SKINNY` [21], `PRINCE` [22] are common examples. Other ultra-lightweight ciphers like `SIMON` and `SPECK` [23], use several bit shifts applied to a partial intermediate state to provide the diffusion. Furthermore, only a binary operation is used to provide non-linearity. Both operations combined would prevent a successful application of SCADPA.

8 Reverse Engineering of Secret Ciphers

As mentioned in the literature [16], one can still find solutions that are using secret cipher components despite the Kerckhoffs' principle. Often industry and agencies would use secret cipher for some specific applications. However, confidence in security of a cipher is only developed by detailed and lengthy analysis. This, for example, is the case of AES which has been extensively studied for over two decades without discovering a serious flaw. A common practice to design secret ciphers is to take

a well studied cipher and replace an operation with a secret operation of equivalent security properties. Replacing Sbox of a well studied cipher like AES or PRESENT, with a secret Sbox of same cryptographic strength as the original, results in a new secret cipher. A relevant example of such cipher is the Russian standard GOST [24], where the Sbox was not defined and is generally agreed between the communicating parties as a long term secret. In the remainder of the section, we investigate the adaption of proposed SCADPA methodology to recover secret Sbox from PRESENT-like ciphers. The reverse engineering works in a restrictive setting when only the Sbox is replaced while other components remain unaltered. To recover other components, other attack techniques (like fault injection) would be required and thus remain out of scope.

The reverse engineering of Sbox is carried out under the following assumptions:

- Only substitution layer with identical Sboxes is the secret component of the cipher. Identical Sboxes are important to keep the design lightweight.
- The attacker can feed chosen plaintext and observe the output ciphertext, without the knowledge of the secret key.

8.1 Recovery of Secret Sbox

Let us consider a PRESENT-like cipher using a secret Sbox, while the permutation layer is unchanged (or known). We describe a method that can reverse engineer the secret Sbox as well as recover the first round key with at most 46 measurements by using SCADPA – 16 for the first key nibble and 30 for the rest (this can be optimized to 17 according to Section 2.3). Following the above notations, let S denote the secret Sbox operation, let k_0 denote the first nibble of the first round key, and similarly, p_0 denotes the first nibble of the plaintext. The 16 measurements which use all the possible values of p_0 give us an array of Sbox output differences, say `dif`, such that

$$\texttt{dif}[i] = S(k_0) \oplus S(k_0 \oplus i).$$

Note that in the case $S(0) = 0$ and $k_0 = 0$, the array `dif` is the unique solution for the secret Sbox. All the possible solutions for the Sbox and corresponding k_0 can be calculated using Algorithm 2. Line 1 considers different values of k_0 and line 2 iterates through different values of $S(k_0)$. With each fixed pair of k_0 and $S(k_0)$, an Sbox solution is uniquely determined using `dif` (line 4). For each Sbox solution, there is a unique value for the first round key using the measurement results and SCADPA. This gives the secret Sbox as well as the secret key used in the original encryption algorithm.

In the following, we describe the steps to reverse engineer the Sbox in detail:

Step 1: Run SCADPA for 16 different values of the first plaintext nibble p_0.

Step 2: Run SCADPA in a normal way for the rest of the nibbles $p_i; 1 \leq i \leq 15$, i.e. with 2-3 different values of p_i.

Step 3: After recovering the differences, obtain the `dif` array and run Algorithm 2.

Algorithm 2: Calculate all Sbox candidates and corresponding k_0

Input : `dif`: array of output differences
Output: ARRAY `A` of `(Sbox,key)`: array of tuples of candidate Sboxes and corresponding k_0 value

```
1 for int k:= 0 to 15 do
2     for int m:= 0 to 15 do
3         for int i:= 0 to 15 do
4             newSbox[i ⊕ k] = dif[i] ⊕ m;
5         A.add(newSbox,k);
6 return A;
```

Step 4: For each pair of Sbox and first nibble of the first round key returned by Algorithm 2, using the information from Step 2, we can construct a cipher. In total we have 256 potential cipher solutions. In this way, each candidate for Sbox will also yield a unique candidate for the first round key.

Step 5: Run each of the ciphers with the given plaintext and compare the ciphertext with the one obtained from the original algorithm. If the ciphertexts are equal, the Sbox value and the first round key value can be determined.

When it comes to reverse engineering of secret Sbox used in `GIFT`, the measurement part works in the same way as explained before. However, there is an advantage for the attacker – `GIFT` does not use pre-whitening key, therefore, the input to `SubCells` layer in the first round is known to the attacker. This reduces the search space from 256 to 16 in Step 3.

8.2 Recovery of Secret Sbox on 32-bit Architectures

Now let us consider the reverse engineering problem described in previous section on 32-bit architectures. In this case, the array `dif` cannot be obtained directly. However, the attacker can still calculate different candidates for `dif` using the same amount of SCADPA measurements as detailed in Step 1 and 2 of Section 8.1. In this section, we illustrate how to achieve this for each of the two attacker capabilities discussed above.

In case the attacker can observe the change of `word0` and `word1` as described in the beginning of this section, instead of `dif`, the attacker can obtain an array $\texttt{dif}_{YN}$ of length 15, such that

- $\texttt{dif}_{YN}[i]$ =Y|N if the first two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are non-zero and the second two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are zero;
- $\texttt{dif}_{YN}[i]$ =N|Y if the first two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are zero and the second two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are non-zero;
- $\texttt{dif}_{YN}[i]$ =Y|Y if the first two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are non-zero and the second two bits of $S(k_0) \oplus S(k_0 \oplus i)$ are non-zero.

Due to the fact that Sbox is a bijective function on $\mathbb{F}_2^4$, for any Sbox S, the values $S(k_0) \oplus S(k_0 \oplus i)$ for $i = 1, 2, \ldots, 15$ are

always a permutation of the 15 values of $1, 2, 3, \ldots, 15$. Thus, in the array $\texttt{dif}_{YN}$, we will have exactly 3 of $Y|N$, 3 of $N|Y$ and 9 of $Y|Y$. Furthermore, for any i, we have the following observations:

- if $\texttt{dif}_{YN} = Y|N$, there are 3 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $1000, 0100, 1100$;
- if $\texttt{dif}_{YN} = N|Y$, there are 3 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0010, 0001, 0011$;
- if $\texttt{dif}_{YN} = Y|N$, there are 9 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0101, 0110, 0111, 1001, 1010, 1011, 1101, 1110, 1111$.

Hence, for the array $\texttt{dif}_{YN}$, the attacker has $3^3 \times 3^3 \times 9^9 = 3^{24}$ possible solutions for dif. Then she can continue with steps 3-5 as in Section 8.1. In this way, the number of ciphers she needs to check is $3^{24} \times 2^8 \approx 2^{46}$. This still stays within the reasonable brute force complexity.

Now, we consider the case the attacker can observe Hamming weight of the differences as described in Section 5.1. Then, from the SCADPA measurements she can obtain an array $\texttt{dif}_{HW}$ of length 15, such that $\texttt{dif}_{HW}[i] = \Delta_{HW_{i0}} | \Delta_{HW_{i1}}$, where $\Delta_{HW_{i0}}$ and $\Delta_{HW_{i1}}$ are the Hamming weights of the first and last two bits of $S(k_i) \oplus S(k_i \oplus i)$ respectively.

By a similar argument as above, since Sbox is a bijective function on $\mathbb{F}_2^4$, for any Sbox S, the values $S(k_0) \oplus S(k_0 \oplus i)$ for $i = 1, 2, \ldots, 15$ are always a permutation of the 15 values of $1, 2, 3, \ldots, 15$. Consequently, in the array $\texttt{dif}_{HW}$, we have exactly $4, 2, 2, 2, 2, 1, 1, 1$ appearances of $11, 12, 21, 10, 01, 20, 02, 22$ respectively. Furthermore, for any i, we have

- if $\texttt{dif}_{HW}[i] = 11$, there are 4 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0101, 1010, 0110, 1001$;
- if $\texttt{dif}_{HW}[i] = 12$, there are 2 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0111, 1011$;
- if $\texttt{dif}_{HW}[i] = 21$, there are 2 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $1101, 1110$;
- if $\texttt{dif}_{HW}[i] = 10$, there are 2 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0100, 1000$;
- if $\texttt{dif}_{HW}[i] = 01$, there are 2 possibilities for $S(k_0) \oplus S(k_0 \oplus i)$: $0001, 0010$;
- if $\texttt{dif}_{HW}[i] = 20$, there is 1 possibility for $S(k_0) \oplus S(k_0 \oplus i)$: 1100;
- if $\texttt{dif}_{HW}[i] = 02$, there is 1 possibility for $S(k_0) \oplus S(k_0 \oplus i)$: 0011;
- if $\texttt{dif}_{HW}[i] = 22$, there is 1 possibility for $S(k_0) \oplus S(k_0 \oplus i)$: 1111.

For each array $\texttt{dif}_{HW}$,, the attacker has $4^4 \times 2^2 \times 2^2 \times 2^2 \times 2^2 \times 1 \times 1 \times 1 = 2^{16}$ possible solutions for dif. Then she can continue with steps 3-5 as in Section 8.1. And the number of ciphers she needs to check is $2^{16} \times 2^8 = 2^{24}$.

9 Conclusions

In this paper, we identified a vulnerability in bit permutation based lightweight ciphers (PRESENT, GIFT, etc) and developed a side-channel assisted methodology called SCADPA to exploit it. With a practical case study on low-cost microcontroller running PRESENT-80, we were able to practically recover the secret key with as low as 17 encryptions and an exhaustive search with complexity of 2^{16}. In case of GIFT, the number of encryptions in the best case was 36. We extended the methodology to enable the recovery of secret Sboxes in PRESENT-like ciphers. Several attacker models were presented, with different complexities of retrieving the key. To avoid the presented attacks, usage of more complex yet low-cost diffusion function is encouraged. The recently launched NIST competition on lightweight cryptography [25] invites candidate submission for cryptographic algorithms optimised for resource constrained devices like 8-bit and 32-bit micro controllers. Out of 56 submissions, at least 8 adopt a PRESENT-like structure (some directly use GIFT as the building block). Thus, the reported vulnerabilities in this paper can help designers perform a fair security analysis and choose better design parameters.

There are several directions that would be worth investigating in the future work. First, it would be interesting to look at possibilities of exploiting different side-channel countermeasures. Especially, if randomness in masking is biased or the leakage characteristics of hiding are not uniform. Another important direction of the future work would be to look at the case of low SNR, to investigate what is the minimum SNR required for SCADPA on different devices. Finally, extending this method to hardware implementations would be a challenging task, but of high interest of the side-channel community.

Acknowledgment

The authors acknowledge the support from the Singapore National Research Foundation ("SOCure" grant NRF2018NCR-NCR002-0001 – www.green-ic.org/socure).

References

[1] K. A. McKay, L. Bassham, M. S. Turan, and N. Mouha, "Report on lightweight cryptography," *NIST DRAFT NISTIR*, vol. 8114, 2016.

[2] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *CHES*, vol. 4727. Springer, 2007, pp. 450–466.

[3] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT: A Small Present," *Cryptographic Hardware and Embedded Systems-CHES*, pp. 25–28, 2017.

[4] T. B. Reis, D. F. Aranha, and J. López, "Present runs fast," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 644–664.

[5] M. A. Philip *et al.*, "A survey on lightweight ciphers for iot devices," in *2017 International Conference on Technological Advancements in Power and Energy (TAP Energy)*. IEEE, 2017.

[6] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in cryptology - CRYPTO'99*. Springer, 1999, pp. 789–789.

[7] K. Schramm, T. Wollinger, and C. Paar, "A new class of collision attacks and its application to DES," in *FSE*, vol. 2887. Springer, 2003, pp. 206–222.

[8] C. Clavier, Q. Isorez, and A. Wurcker, "Complete SCARE of AES-Like Block Ciphers by Chosen Plaintext Collision Power Analysis," in *INDOCRYPT*, vol. 8250. Springer, 2013, pp. 116–135.

[9] N. Veyrat-Charvillon and F.-X. Standaert, "Adaptive chosen-message side-channel attacks." in *ACNS*, vol. 6123. Springer, 2010, pp. 186–199.

[10] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *Computer Security - ESORICS 98*, pp. 97–110, 1998.

[11] B. den Boer, K. Lemke, and G. Wicke, "A DPA Attack against the Modular Reduction within a CRT Implementation of RSA," in *CHES*, vol. 2523. Springer, 2002, pp. 228–243.

[12] L. Guo, L. Wang, D. Liu, W. Shan, Z. Zhang, Q. Li, and J. Yu, "A chosen-plaintext differential power analysis attack on HMAC-SM3," in *Computational Intelligence and Security (CIS), 2015 11th International Conference on*. IEEE, 2015, pp. 350–353.

[13] O. Reparaz and B. Gierlichs, "A first-order chosen-plaintext dpa attack on the third round of des," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2017, pp. 42–50.

[14] R. Novak, "Side-channel attack on substitution blocks," in *ACNS*, vol. 2846. Springer, 2003, pp. 307–318.

[15] M. Rivain and T. Roche, "SCARE of secret ciphers with SPN structures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2013, pp. 526–544.

[16] C. Clavier, Q. Isorez, D. Marion, and A. Wurcker, "Complete reverse-engineering of aes-like block ciphers by scare and fire attacks," *Cryptography and Communications*, vol. 7, no. 1, pp. 121–162, 2015.

[17] J. Breier, D. Jap, and S. Bhasin, "Scadpa: Side-channel assisted differential-plaintext attack on bit permutation based ciphers," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1129–1134.

[18] P. Hoogvorst, J.-L. Danger, and G. Duc, "Software Implementation of Dual-Rail Representation," in *COSADE*, 2011, Darmstadt, Germany.

[19] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 142–159.

[20] N. F. Pub, "197: Advanced encryption standard (AES)," *Federal information processing standards publication*, vol. 197, no. 441, p. 0311, 2001.

[21] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim, "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS," Cryptology ePrint Archive, Report 2016/660, 2016, http://eprint.iacr.org/2016/660.

[22] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger *et al.*, "PRINCE–a low-latency block cipher for pervasive computing applications," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2012, pp. 208–225.

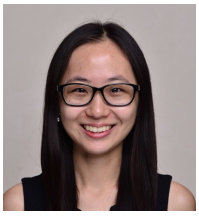
[23] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[24] V. Dolmatov, "Gost 28147-89: Encryption, decryption, and message authentication code (mac) algorithms," Tech. Rep., 2010.

[25] "Lightweight Cryptography," https://csrc.nist.gov/projects/lightweight-cryptography.

Jakub Breier is currently a Principal Research Fellow at School of Computer Science and Engineering, Nanyang Technological University, Singapore. Before that, he was a security analyst at Underwriters Laboratories, Singapore. Between 2013-2018, he was a hardware security researcher at PACE Lab, Nanyang Technological University. He received his PhD in Applied Informatics from Slovak University of Technology (STU), Slovakia in 2013, Master's in Information Technology Security from Masaryk University, Czech Republic in 2010, and Bachelor's in Informatics from STU, Slovakia in 2008. He was also a visiting researcher at Fraunhofer AISEC, Germany, in 2014. His research topics include fault and side-channel analysis methods and countermeasures, advanced fault injection techniques, and deep learning security.

Dirmanto Jap is currently a Research Scientist at PACE Lab, Temasek Laboratories, Nanyang Technological University (NTU), Singapore. He previously received his Ph.D in Mathematics from NTU in 2016. His main research topics include physical attacks (side-channel and fault attacks) and countermeasures, practical laser/EM fault injection, application of machine learning and deep learning for side-channel attacks and hardware Trojan detection, as well as security of deep learning.



Xiaolu Hou works as a Research Fellow at National University of Singapore since 2019, in the field of neural network security. She finished her PhD in Mathematics from Nanyang Technological University (NTU), Singapore in 2017. After her PhD she joined Cyber Security Laboratory, School of Computer Science and Engineering, NTU, as a Research Fellow, where her research focused on fault injection and side-channel attacks. She had also worked at Acronis, Singapore, focusing on secure multi-party computation. With a wide range of research interests, she has published her work at top venues within various fields, ranging from mathematics to computer security.

Shivam Bhasin is a Senior Research Scientist and Programme Manager (Cryptographic Engineering) at Centre for Hardware Assurance, Temasek Laboratories, Nanyang Technological University Singapore. He received his PhD from Telecom Paristech in 2011, Master's from Mines Saint-Etienne, France in 2008. Before NTU, Shivam held position of Research Engineer in Institut Mines-Telecom, France. He was also a visiting researcher at UCL, Belgium (2011) and Kobe University (2013). His research interests include embedded security, trusted computing and secure designs. He has co-authored several publications at recognized journals and conferences. Some of his research now also forms a part of ISO/IEC 17825 standard.

Appendix A
More Details on Difference Recognition from [4]

In this part we explain how the Sbox output bits propagate when the code from Listing 1 is being executed. As our experiments in Section 6.2 focus on the first Sbox, we will use it as an example. The outputs from other Sboxes can be done in a similar fashion. The bit position of k a variable is denoted by "_k". Therefore, the output bits of the first Sbox before the P_1 are stored in `X0_0`, `X1_0`, `X2_0`, and `X3_0`.

A.1 `X0_0` – First Bit of the First Sbox

First bit of `X0` shows in following lines:

- Line 1: shows in variable `X0`, but does not propagate into variable `t` as the first four bits are zeroes.
- Line 2: shows in variable `X0`, and is written back into `X0_0` since there is xor operation.
- Line 7: shows in variable `X0`, but does not propagate into variable `t` as the first eight bits are zeroes.
- Line 8: shows in variable `X0`, and is written back into `X0_0` since there is xor operation.

A.2 `X1_0` – Second Bit of the First Sbox

First bit of `X1` shows in following lines:

- Line 1: shows in variable `X1`, and propagates into variable `t` as `t_4`, xor-ed with `X0_4`.
- Line 2: shows in variable `t`, and is written into `X0_4`.
- Line 3: shows in variable `X1`, and is removed from that variable as a result of xor-ing with `t`.
- Line 7: shows in variable `X0`, but does not propagate into variable `t` as the first eight bits are zeroes.
- Line 8: shows in variable `X0`, and is written back into `X0_4` since there is xor operation.

A.3 `X2_0` – Third Bit of the First Sbox

First bit of `X2` shows in following lines:

- Line 4: shows in variable `X2`, but does not propagate into variable `t` as the first four bits are zeroes.
- Line 5: shows in variable `X2`, and is written back into `X2_0` since there is xor operation.
- Line 7: shows in variable `X2`, and propagates into `t_8`.
- Line 8: shows in variable `t`, and is written into `X0_8` since there is xor operation.
- Line 9: shows in variables `X2` and `t` and is rewritten by an xor.

A.4 `X3_0` – Fourth Bit of the First Sbox

First bit of `X3` shows in following lines:

- Line 4: shows in variable `X3`, and propagates into variable `t` as `t_4`, xor-ed with `X2_4`.
- Line 5: shows in variable `t`, and is written into `X2_4`.
- Line 6: shows in variable `X3`, and is removed from that variable as a result of xor-ing with `t`.
- Line 7: shows in variable `X2`, and propagates into `t_12`.
- Line 8: shows in variable `t`, and is written into `X0_12` since there is xor operation.
- Line 9: shows in variables `X2` and `t` and is rewritten by an xor.

Appendix B
PRESENT Cipher

PRESENT is a lightweight block cipher based on Substitution-Permutation Network (SPN) [2]. Therefore, it consists of three operations: `addRoundKey` is a bit-wise `xor` of the state with the round key; `sBoxLayer` is a nibble-wise nonlinear substitution; `pLayer` is a bit permutation. The structure of one round of the cipher is depicted in Figure 10. PRESENT consists of 31 rounds, followed by a post-whitening `addRoundKey` at the end. The variant used in our experiments, PRESENT-80, has a secret key of size 80 bits and a block size of 64 bits. Table 8 shows PRESENT Sbox that is executed on all 16 nibbles of the PRESENT-80 state during the `sBoxLayer`. The Sbox function is further denoted as $S(.)$.

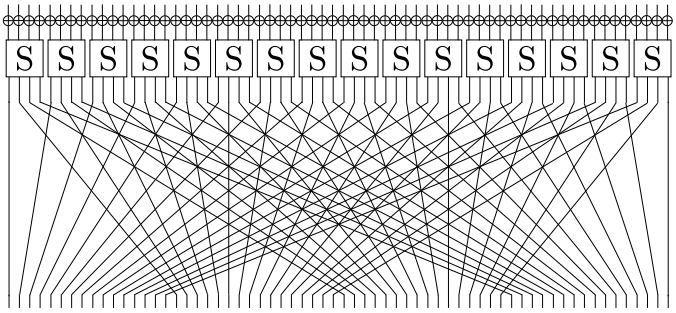


Fig. 10: Structure of one round of PRESENT-80 cipher.

x	0	1	2	3	4	5	6	7
$S(x)$	C	5	6	B	9	0	A	D
x	8	9	A	B	C	D	E	F
$S(x)$	3	E	F	8	4	7	1	2

TABLE 8: PRESENT Sbox.