Persistent Fault Injection in FPGA via BRAM Modification

Yiran Zhang[†], Fan Zhang*[‡§†], Bolin Yang[†], Guorui Xu[‡§], Bin Shao[†], Xinjie Zhao[¶] and Kui Ren[‡§]
[†]*College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, 310027, China*
[‡]*School of Cyber Science and Technology, Zhejiang University, Hangzhou, 310027, China*
[§]*College of Computer Science and Technology, Zhejiang University, Hangzhou, 310027, China*
[¶]*The Institute of North Electronic Equipment, Beijing, 100191, China*

***Abstract*—The feasibility of persistent fault analysis relies on special faults which can persist in all the rounds of block ciphers. This prerequisite can be positioned as a good fit into the FPGA scenario, which however has not been carefully exploited ever before. In this paper, we propose the persistent fault attack on the block cipher AES-128 implemented in FPGA where a new type of persistent fault is induced with the technique of Block RAM (BRAM) modification. The details of persistent fault injection are elaborated, especially on how the target bits of AES in BRAM can be identified and how they can be altered. Our experimental results show that: with the proposed attack, a simple statistical analysis can extract the secret key of AES-128 with S-Box implemented in BRAMs and protected by the countermeasure of inversive decryption based dual modular redundancy.**

***Keywords*-Persistent Fault, Persistent Fault Analysis (PFA), FPGA, Block RAM (BRAM).**

I. Introduction

With the prevailing communication, computation and hardware technologies, security problems are becoming increasingly serious for Internet of Things (IoT) and Cyber-Physical Systems (CPS). Traditional side channel attack (SCA) is a passive attack that exploits the leakages such as power, electromagnetic emission, and timing of encryption hardware to recover the secret key. In contrast, *Fault Attack* (FA) is an active attack which retrieves confidential information by intentionally injecting faults into the crypto system[1]. A fault attack in practice usually consists of online and offline stages. In the online stage, an adversary needs to perform a fault injection. The injection could be done by varying solutions, such as changing the power supply voltage or the frequency of external clocks, manipulating the temperature or exposing the circuits to lasers etc., for disturbing the key-dependent computations[2]. In the second offline stage, the adversary launches the subsequent fault analysis based on faulty output or other observable abnormal behaviors. One of the most popular fault analysis is differential fault analysis [3] which conducts cryptanalysis on correct/faulty ciphertext pairs. Other fault analysis can be statistical which are capable of performing key recovery basing on faulty ciphertext only [4].

For a traditional fault attack, it usually requires fault injections with certain precision level. The injection has to be conducted on the right timing of certain operations within a specific round. However, the tightly coupled requirement is difficult to satisfy and hence requires wide expertise.

The countermeasures against fault attacks have long been studied, therefore they have to be taken into consideration. One mainstream strategy is known as *redundant encryption-based Dual Modular Redundancy* (REDMR), which uses two identical logic rails to check the attacks by comparing the two outputs from both rails. Comparatively, *inversive decryption-based DMR* (IDDMR) consists of two serialized modules to encrypt and then decrypt, and triggers an alarm in case the resulting form the decryption is not equivalent to the original plaintext. Both countermeasures can be easily implemented at the cost of either logic size or computation time.

The *persistent fault analysis* (PFA) [5] is recently proposed for coping with the two aforementioned countermeasures, which allows the adversary to prepare the fault injection before the encryption meanwhile bypassing countermeasures such as DMR. In [5], Zhang et al. give the first illustration of persistent faults. It is a disturbance error in the DRAM caused by the *rowhammer* attack, which is usually regarded as a software-controlled hardware attack.

This paper illustrates another method for injecting faults compatible with PFA, where the fault is at the FPGA hardware level and can be induced through the technique of BRAM modification.

This paper is organized as follows: Section II gives a brief description of BRAM, AES and PFA. Section III introduces the attack strategy. Section IV describes the implementation of AES and BRAM modification on FPGA. Section V shows the fault analysis result. Section VI concludes the paper.

This work was supported in part by the Alibaba-Zhejiang University Joint Institute of Frontier Technologies, by the Research Institute of Cyberspace Governance in Zhejiang University, by the Provincial Key Research and Development Program of Zhejiang (Grant No. 2019C03133), by the Major Scientific Research Project of Zhejiang Lab (Grant No. 2018FD0ZX01), by the Open Fund of State Key Laboratory of Cryptology (Grant No. MMKFKT201805), by the National Natural Science Foundation of China (Grants No. 61571063 and 61772236), and by the Young Elite Scientists Sponsorship Program by CAST (Grant No. 17-JCJQ-QT-045).

*Corresponding author: Fan Zhang, Email: fanzhang@zju.edu.cn.

II. Background

This section recalls the target devices and the used attack technique.

A. Block RAM in FPGA

Field programmable gate array (FPGA) are programmable digital circuit. The architecture of a typical FPGA is composed of an array of configurable elements (as the Configurable Logic Block, CLB in short, in Xilinx series) in a mesh fashion. Combined with plenty of powerful features like rich block memory, precise clocking managers, efficient digital signal processing (DSP), etc, FPGA has become a popular solution for fast, low-cost, and recyclable IC solution in recent decades.

To enlarge the storage space of FPGA, particular for the EDK usages, numerous memory primitives, named *Block RAM* (BRAM), are embedded inside the chip for providing dedicated data storage elements. BRAM is basically a configurable module that can be easily instantiated in a design. Using BRAMs in security algorithms is advantageous over pure logical implementations due to its intrinsic immunity against side-channel attacks [6]. This is mainly due to the fact that memory is a hard macro with reduced leakage profile and regular structure, unlike logical implementation with CLB. Moreover, the BRAM features can further be exploited for strengthening other SCA countermeasures, such as masking and dual-rail precharge logic (DPL) [7].

B. AES Algorithm

The Advanced Encryption Standard (AES) was published as a standard for symmetric encryption by the National Institute of Standards and Technology (NIST) in 2001. AES has three versions, AES-128, -192 and -256 which has 10, 12, and 14 rounds, respectively. In this paper, we focus on the AES-128 where the block size and key size are all 128 bits, i.e., 16 bytes. During encryption, the 16-byte plaintext is copied into a 4×4 matrix called state. Then it goes through 10 rounds, and the final state is copied into ciphertext. Each round consists of 4 major operations. The first one is **SubBytes**. It is a simple table lookup to the so-called SBox. Each byte is substituted by the corresponding byte in SBox. The second one is **ShiftRows**. The third one is called **MixColumns**. The last operation is **AddRoundKey**. The state is XORed with round keys which are generated by **KeyExpantion** function with the master key. Besides, there is an additional AddRoundKey at the beginning of encryption and the MixColumns operation is skipped in the last round. As to the decryption of AES, it is a reverse procedure of encryption where an inversive table S^{-1}-Box is utilized.

C. Persistent Fault Analysis (PFA)

PFA is a novel type of fault analysis whose fault model differs from conventional fault models, i.e., transient and permanent. Persistent fault falls between transient and permanent faults, meaning that the fault persists from one encryption to another but disappears when the target device reboots. As a major feature, faults in PFA can be injected even before the target encryption starts, thus making the attack easier by relaxing the synchronization.

Statistical means are used in PFA for key recovery. For example, a modification of one S-Box element of AES will cause a biased distribution of ciphertexts, which could be utilized for key-recovery [5].

The threats from PFA can be viewed in two standpoints. One is that the coverage of scenarios is increasing a lot that is possibly a good fit for PFA. The other is that most of the scenarios are showing a trend that PFA can relax the attack assumptions and make the attack more practical.

III. Attack via BRAM Modification

This section clarifies the scenario and assumption first, and then gives the details of the BRAM modification attack.

A. Attack Scenario

Suppose the victim, denoted as V, is conducting the encryption on an FPGA device denoted as D. V sends plaintext P to the FPGA and the ciphertext C is returned. The secret key K is stored in the tamper-resistant zone and used for encryption. To prevent the fault attacks from adversary A, the device is deployed with the IDDMR countermeasure. For each P, V will actually ask D to encrypt it. Inside D, the original plaintext P is encrypted with K with one module to get the ciphertext C. Then C is decrypted to get the plaintext P'. If $P' = P$, the victim will consider the encryption in D is executed without errors. He will output the ciphertext C. If $P' \neq P$, V will consider that the abnormal encryption is due to the potential fault attacks. In this paper, the so-called *zero value output* (ZVO) strategy [5] is under discussion. That means, once the fault is detected, the device D will output a zero value string instead of faulty ciphertexts for preventing the analysis upon the faulty outputs. As shown in [5], the countermeasure is also vulnerable to PFA if no output or random output is chosen as a reaction to fault detection.

To support IDDMR, the FPGA is supposed to have both the encryption and decryption module for AES-128. For illustration and simplification, the AES S-Box implementation is adopted. The compact S-Box (S) is designed as a lookup table of 256 bytes. The whole encryption is implemented as combinational logic, except the substitution of S-Box (S) is implemented in BRAM. Since there are 16 table lookup operations in one round of AES in parallel, there are 16 copies of tables stored in BRAMs for S and S^{-1}, respectively.

B. Attack Assumption

The attack is applied in the following settings.

1) The adversary A is aware of the BRAM implementation of AES-128.
2) The readback function of PROM is enabled and that of FPGA is disabled. In fact, if the readback function of FPGA is enabled, our attack could be easier since we do not need to change the file type after reading PROM.
3) The CRC functionality of FPGA is enabled by default.
4) During the encryptions, the device D will be on for all the time, which serves as the carrier of persistent faults to be injected.
5) The encryption of FPGA bitstream is not enabled. The analysis of an encrypted bitstream is quite challenging and will be left as future work.

C. Attack Procedure

After the FPGA is on, the adversary A can physically access the device D. He can readback data from PROM on D through JTAG. Note that the encryption program is burned into the PROM and will be loaded from PROM to FPGA once the device is powered on. The data is actually a bitstream which is equivalent to a file, thus this data can be denoted as f_1. Then A needs to change the file f_1 to another file f_2 that is capable of reconfiguring D. Then, the most challenging task for A is to identify the location of BRAM section for 16 tables, find the correct bits to modify and then re-program it into FPGA.

(1) Identify the S-Box. To locate where the lookup table of S-Box is, the most straightforward method is to find the pattern of 256 bytes in f_2 which is repeated for exactly 16 times. The difficulty that the adversary A may meet is that first he may not be able to find it, second, if found, the pattern may not be the same as the S-Box as expected, which prevents him from further modifications.

In this attack on the hardware, the adversary can actually erase (reset) all the elements of tables to be '0's, even if he does not know the exact value of that found pattern. Theoretically, the subkey in the final round R_{10} of AES-128 will be simply outputted as ciphertexts and possibly observed by A. However, this adversarial behavior is easy to detect as all the ciphertexts are fixed and equal to the subkey of last round R_{10}.

The S-Box in f_2 has to be identified, as the information of bit locations in S is required to guide the injection of persistent fault.

(2) Modify the bits. In this step, the adversary has to modify two types of bits. The first is the CRC tag bits. By default, the CRC functionality of FPGA is always on. A has to identify this tag in f_2, modify it from 1 to 0, therefore turn off the CRC in order to modify more bits.

The second is some targets bits of S-Box in BRAM to inject a persistent fault. The principle of choosing those bits is that the fault countermeasures can be bypassed to some extent and the corresponding persistent fault analysis is easy to be conducted. The former requires persistent faults to be injected to all 16 tables simultaneously. The latter requires the injection to affect only one or a few bytes of lookup tables, making PFA relatively simple.

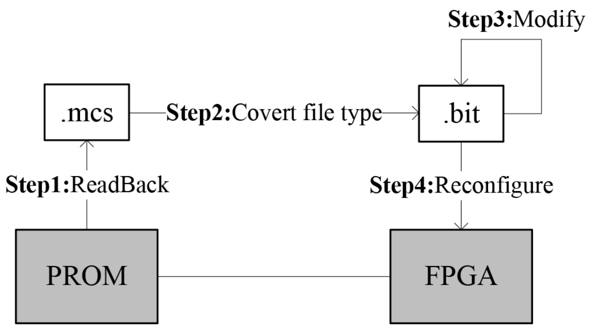


Figure 1. BRAM tampering.

(3) Writeback the bitstream After the modifications on f_2, A can get a tampered bitstream. A can reconfigure D with this bitstream thus the persistent fault has been injected.

Fig. 1 describes the main flow for updating the BRAMs for PFA attack, which **starts** from reading back the bit file and **ends** at reconfiguring the tampered bitstream to the device under attack:

Step1: ReadBack function [8] (including BRAM frame) in Xilinx FPGA is used for extracting the `.mcs` file from PROM.
Step2: Convert `.mcs` file into a `.bit` file.
Step3: Modify specific bits in the converted bitstream to get the tampered bitstream.
Step4: Refresh BRAMs using the tampered bitstream.

IV. Implementation

Hardware implementations of cryptographic algorithms mainly pursue high throughput by achieving computations in parallel and reusing logics for each round to reduce the cost. The hardware configurable capability of FPGA renders it as a suitable platform for the block cipher implementation where blocks can be processed in parallel. Without loss of generality, we consider AES-128 as the target in this paper. The hardware for SubByte, ShiftRow, MixColumn, and AddroundKey are reused for each encryption/decryption round.

The BRAM equipped in modern Xilinx FPGAs is normally dual-ported block memories, which can be configured to read and write simultaneously [9]. The size of BRAM in Virtex-5, Spartan-6, or the newest 7 series FPGA is 36kb (denoted as RAMB36). Each BRAM can be further configured as two independent modules with 18kb size for each (denoted as RAMB18), as illustrated in Fig 2.

A. Implementation of Cipher

A common way to implement the AES circuit into FPGA is to leverage the BRAM to partially accommodate the non-

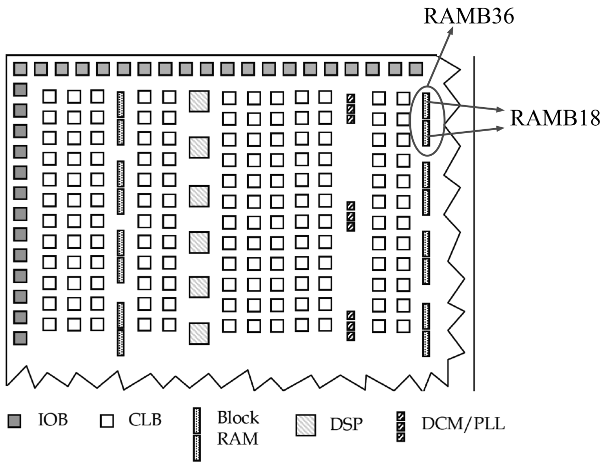


Figure 2. Illustration of Xilinx FPGA architecture.

linear computations, *i.e.*, S-Box into memories. The merits using BRAM reside on the identical signal delays, the lower fan-out, and the less consumed logical resources.

A standard AES S-Box has a size of $2^8 \times 8$ bits (256 bytes). For the RAM implementation of S-Box, 256 elements have to be addressed with bit-width of 8-bit, and each element has a depth of 8-bit for each S-Box. For the target FPGAs, the RAMB18 is basically designed as a dual-port RAM, where two groups of input ports (15-bit for each group) are available, hence a RAMB18 is sufficient to implement 2 S-Boxes. Since each RAMB36 module is comprised of 2 RAMB18, totally 4 S-Boxes can be allocated into each RAMB36.

Some optimization techniques have been previously proposed to compact the S-Box [10], [11], while in this paper we just adopted the straightforward approaches to implement the S-Box for the sake of simplicity. Precisely each RAMB18 accommodates 2 S-Boxes. The target AES implementation consists of both encryption and decryption, so totally 8 RAMB36 are occupied of 16 S-Boxes and 16 S^{-1}-Boxes. In the exemplary target Virtex-5 (xc5vlx50), there are 48 BRAMs available, which are sufficient to realize the circuit.

The costs of the implementation are summarized in Table I. We implemented the AES without special place and route constraints, but only forced the usage and placement of those tables S, S^{-1} into BRAMs. Since there are 16 S-Box and 16 S^{-1}-Box consumption, each BRAM (RAMB36) can accommodate 4 S-Box or S^{-1}-Box so the BRAM cost for this implementation is 8.

Table I
COST OF S-BOX AES IMPLEMENTATION (ALSO WITH S^{-1}).

AES style	RAMB36	Slice LUTs	Slice Registers	Occupied Slices
S-Box (S^{-1})	8	2630	2469	2131

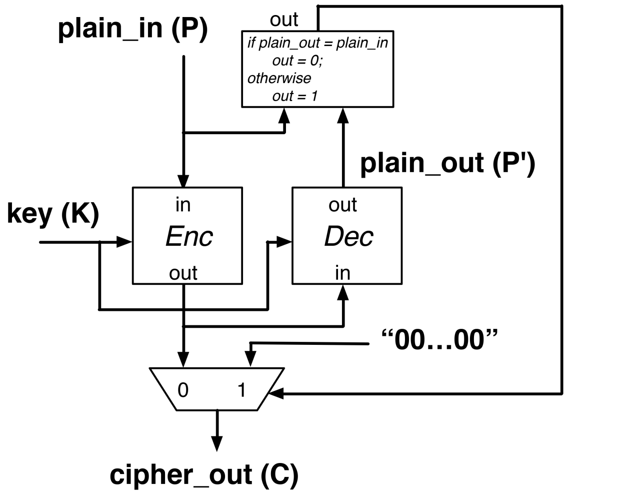


Figure 3. The sketch design of inversive decryption based DMR (IDDMR) AES implementation with zero value ouput (ZVO) scheme.

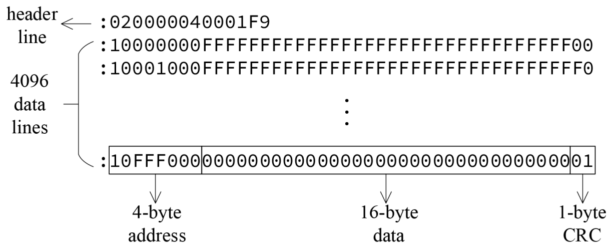
B. Implemenation of BRAM Modification

The BRAM modification is carefully investigated on a Xilinx Virtex5-xc5vlx50 FPGA with ISE 14.7 in Windows 7. As aforementioned, in order to modify data stored in the FPGA BRAM, the adversary A needs to retrieve programming data from the PROM associated with the target FPGA by using readback feature and get the `.mcs` file. Then, to reconfigure the FPGA with tools like IMPACT, A needs to make a `.bit` file using the retrieved `.mcs` file. The `.mcs` file is an ASCII file that is composed of blocks of data.

The organization of each block in Virtex5 is shown in Fig.4. A could get the `.bit` file by modifying the `.mcs` file with following steps:

1) Delete the header line.
2) Delete the ":" character, the 4-byte address and the 1-byte CRC in each data line.
3) Delete the lines consists of all "F"s at the end of the file, which can be found after a series of '20000000'.
4) Convert the ASCII values into HEX.
5) Add a header which is available from any other .bit file generated for the same-model FPGA.

With these steps, the adversary A can get a bitstream



```
:020000040001F9
:10000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00
:10001000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0
                    ⋮
:10FFF0000000000000000000000000000000000001
```

Figure 4. Block organization in mcs file.

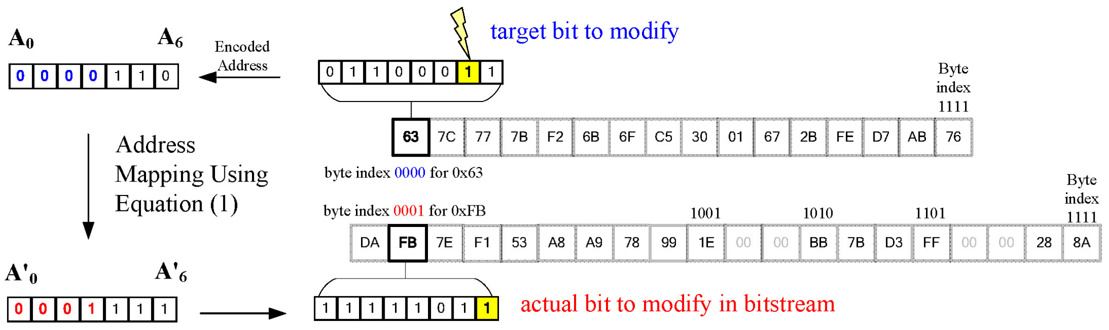


Figure 5. Data mapping between S-Box and BRAM. The target bit of modifying 0x63 to 0x61 has been highlighted.

which can be used to configure the target FPGA. Next, he should modify the BRAM data stored in this bitstream appropriately. Thus he needs to know how the S-Box (S^{-1}) data are stored in the block RAM. To find out this relation, dozens of programming files are generated, where only the block RAM data is modified while the rest parts of verilog code remain the same. The most important observation from the experiment is that the order of each 16-byte block in the bitstream remains unchanged, however, the index of those bits in each 16-byte block are bitwise shuffled with fixed correlation.

To explain this bitwise shuffling, the bit index of the data in a 16-byte block of S-Box can be encoded as $A = A_0A_1A_2...A_6$. $A_0A_1A_2A_3$ is the index of the byte and $A_4A_5A_6$ is the bit index in that byte (MSB first). The bit index of the data in the modified bitstream for BRAM is denoted as $A' = A'_0A'_1A'_2...A'_6$. The mapping between A and A' can be described as followings:

$$\begin{aligned} A'_3A'_4A'_5A'_6 &= \overline{A}_3\overline{A}_2\overline{A}_0\overline{A}_1 \\ A'_0 &= \overline{A}_4 \\ A'_1 &= A_4 \cdot \overline{A}_5 + \overline{A}_4 \cdot \overline{A}_6 \\ A'_2 &= \overline{A}_4 \cdot \overline{A}_5 + A_4 \cdot A_6 \end{aligned} \tag{1}$$

Besides, there are 4 bytes of zeros padded into each 16-byte block in the binary file. The first two bytes are inserted after the 10th byte and the other two are after the 14th byte.

With the information in Eq.(1), the adversary can modify the bitstream as he wishes and prepare the persistent fault. For example, A wants to update the first element of S-Box (0x63) to 0x61. Note that 0x63 is the first byte of the 16-byte block. Thus $A_0A_1A_2A_3 = 0000$. This modification requires the last but one bit of 0x63 to be altered from 1 to 0. Thus $A_4A_5A_6 = 110$. According to Eq.(1), $A'_0A'_1A'_2A'_3A'_4A'_5A'_6$ can be computed as 0001 111. Note that $A'_0A'_1A'_2A'_3 = 0001$ and $A'_4A'_5A'_6 = 111$. The adversary can target at the first block of the BRAM, modify the last bit of the second byte from 1 to 0, as shown in Fig.5.

To get the FPGA back to work, the adversary has to deal with the CRC problem. As to A, instead of recalculating the CRC value of the modified bitstream, he can just disable it. According to the user guide provided by Xilinx [8], this disabling can be fulfilled by setting the CRC_BYPASS bit to 1, which is the 29th bit from the last one in the Configuration Options Register (COR). The COR is located after a special word 0x30012001 which means "write one word to COR register".

After disabling the CRC, the adversary needs to change the two 4-bytes CRC values into 0x0000DEFC. Both of them can be found after a special word 0x30000001 which means "write one word to CRC register". After all these modifications, A could reprogram the FPGA and inject the persistent fault. Till now, the attacking environment is well prepared.

V. Evaluation of Fault Analysis

After the injection, the adversary A can inform the victim V to start the encryption. Since IDDMR with ZVO is employed by V, only a fraction of the ciphertexts can be output from D and the rest ciphertexts are all zeros.

The core idea of persistent fault analysis is very simple. For a specific byte of ciphertext, the distribution of output values is biased. This is due to the injection of persistent fault to the lookup table. For example, if the first element of S-box is changed from 0x63 to 0x61, the output of the SubBytes will never contain 0x63. Meanwhile, 0x61 will appear with double frequency. Since the MixColumns operation is skipped in the last round of AES and the ShiftRows operation doesn't affect byte values, each byte of ciphertext will never contain 0x63$\oplus k_i$ and 0x61$\oplus k_i$ will appear more frequently where k_i is one byte of the last round key. So, by analyzing the biased distribution, the last round key can be recovered thus the master key of AES. Considering the IDDMR with ZVO, the distribution of ciphertext is biased in the same way if the zero outputs are omitted. More details of PFA can be referred to as in [5].

The target AES cipher is implemented on a Virtex5-xc5vlx50 FPGA, and the last but one bit of the first element of the AES S-Box is modified (i.e., it is changed from 0x63 to 0x61) as detailed in Sec. IV. 20000 ciphertexts are then collected for analysis, and the distribution of the first byte of ciphertexts is shown in Fig.6. The x-axis is the number of

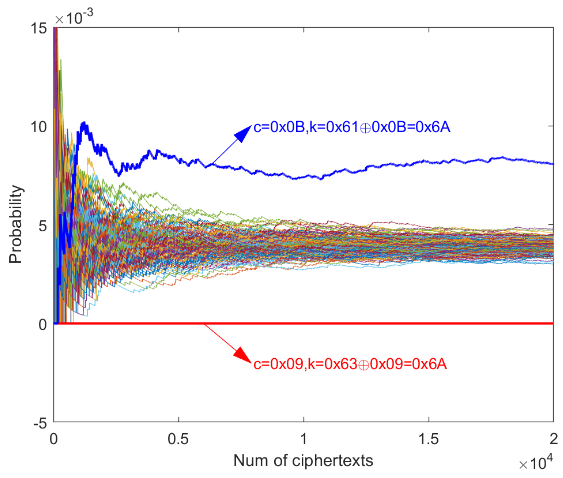


Figure 6. Attack result of PFA on IDDMR with ZVO.

ciphertexts that V receives (including those all zeros). The y-axis is the probability of each value of the first byte in the ciphertext. According to Fig.6, around 2000 ciphertexts are enough to differentiate the key byte from others. The byte value 0x0B has a larger probability which is approaching 0.0081, and it is consistent with the theoretical value $\frac{2}{256}$. Meanwhile, the byte value 0x09 has a zero probability. Both of them can be used to deduce the secret key byte since 0x63$\oplus k_1$ should be 0x09 and 0x61$\oplus k_1$ should be 0x0B. So the first byte of the last round key k_1 should be:

$$k_1 = 0x61 \oplus 0x0B = 0x63 \oplus 0x09 = 0x6A \qquad (2)$$

All 16 bytes of the last round key can be recovered in the same way, and the deduced master key is identical to the one used in the implementation.

VI. Conclusion

In this paper, we propose a new type of persistent fault attack, which can inject persistent faults to AES-128 implemented on FPGA with the technique of BRAM modification. The primary focus of this paper is to elaborate on how the target bits of AES in Block RAM can be identified and how they can be altered. It demonstrates that the persistent faults can be found in the scenario of the hardware level, and the corresponding persistent fault analysis is also feasible. Experimental results show that the attack can break the AES-128 implementation even if it is hardened with certain countermeasures against fault attacks, such as IDDMR. Note that the work in this paper can also be applied to T-Box implementations.

References

[1] M. Joye and M. Tunstall, *Fault Analysis in Cryptography*. Springer Berlin Heidelberg, 2012.

[2] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology — EUROCRYPT '97*, W. Fumy, Ed. Springer Berlin Heidelberg, 1997, pp. 37–51.

[3] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual international cryptology conference*. Springer, 1997, pp. 513–525.

[4] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard, "Fault attacks on AES with faulty ciphertexts only," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 2013, pp. 108–118.

[5] F. Zhang, X. Lou, X. Zhao, S. Bhasin, W. He, R. Ding, S. Qureshi, and K. Ren, "Persistent fault analysis on block ciphers," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 150–172, 2018.

[6] R. Velegalati and J.-P. Kaps, "Techniques to enable the use of block RAMs on FPGAs with dynamic and differential logic." in *ICECS*. Citeseer, 2010, pp. 1244–1247.

[7] S. Bhasin, J.-L. Danger, S. Guilley, and W. He, "Exploiting FPGA block memories for protected cryptographic implementations," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 3, pp. 16:1–16:16, May 2015.

[8] Xilinx, *Virtex-5 FPGA Configuration User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[9] ——, *Virtex-5 FPGA User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[10] C.-J. Chang, C.-W. Huang, H.-Y. Tai, M.-Y. Lin, and T.-K. Hu, "8-bit AES FPGA implementation using block RAM," in *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*. IEEE, 2007, pp. 2654–2659.

[11] A. Aziz and N. Ikram, "Memory efficient implementation of AES S-boxes on FPGA," *Journal of Circuits, Systems, and Computers*, vol. 16, no. 04, pp. 603–611, 2007.