Differential Fault Attack on Rocca

Ravi Anand[1] and Takanori Isobe[1,2,3]

[1] University of Hyogo, Kobe, Japan
ravianandsps@gmail.com, takanori.isobe@ai.u-hyogo.ac.jp
[2] National Institute of Information and Communications Technology (NICT), Tokyo, Japan
[3] PRESTO, Japan Science and Technology Agency, Tokyo, Japan

Abstract. Rocca is the first dedicated cipher for 6G systems accepted in ToSC 2021 and will be presented at *FSE* 2022. In this paper we show that Rocca is susceptible to differential fault attack under the nonce reuse scenario. The attack outcome results in a complete internal state recovery by injecting 4×48 faults in three out of eight internal state registers. Since the round update function of Rocca is reversible it also allows for key recovery. To the best of our knowledge this is the first third party analysis of Rocca.

Keywords: Fault Attack · Differential Fault Attack · Random Faults · Rocca · Side Channel Attack · AES SBox

1 Introduction

When it comes to implementing any cryptosystem on hardware, security of the cipher becomes a primary concern. The adversary can always take advantage of the cipher implementation by disturbing the normal operation mode of the cipher, and then trying to find the secrets of the cipher by restricting its computationally expensive search space to a smaller domain. By disturbing normal modes of operation of a cipher we mean causing glitches in the clock input, using focused laser beams to introduce bit flips, exposing the hardware to severe environments like high temperatures, over-voltage or anything that can change the internal state of the cipher. Once the changes are incorporated into the cipher and faulty ciphertexts are produced, the differences between fault-free and faulty ciphertexts are noted and we try to deduce the internal state of the cipher, and if possible, the secret key too. Since Boneh et al. [2] used fault attacks against an implementation of RSA. Since then, fault attacks have been widely used against many encryption algorithms, including DES [4] and AES [5].

Rocca [12] is the first dedicated cipher for 6G systems which was accepted in ToSC-2021 Issue 2 and will be presented at *FSE* 2022. Rocca [12] is an AES-based encryption scheme with a 256-bit key and 128-bit tag, which provides both a raw encryption scheme and an AEAD scheme. The design of this cipher is inspired by the work of Jean and Nikolic[8], in which the authors have described several constructions based on the AES round function which can be used as building blocks for message authentication code (MAC) and authenticated encryption with associated data (AEAD). The AEGIS family [16] and Tiaoxin-346 [11] are some of the ciphers which are also inspired by these constructions. These two ciphers were submitted to the CAESAR competition and AEGIS-128 has been selected in the final portfolio for high-performance applications. The round functions of the AEGIS family and Tiaoxin-346 are quite similar. Both these ciphers have been found vulnerable to differential faults attacks [15,6,1].

Our Contribution. In this paper we describe a fault attack on Rocca. To the best of our knowledge this is the first third party analysis of Rocca. The fault attack described here is a differential fault attack targeted on one byte at a time. We provide a theoretical analysis of the feasibility of this attack.

We show that the complete internal state of Rocca can be recovered by injecting faults in all the 48 bytes of 3 internal state registers out of the total 8 registers. The recovery of the values of these 48 bytes using differential fault attack reduces to differential fault attack of AES S-box due to the design of the cipher. The complete attack strategy is described in Section 3.

The fact that we could extend the recovery of these 3 internal state registers to complete internal state recovery shows certain flaws in the design of the cipher. In comparision, the recovery of complete internal state requires the fault to be injected in all the 8 state registers of AEGIS-128L and in all 13 state registers of Tiaoxin-346 in the random fault model [15]. However, the number of faults required in Rocca is more than that of AEGIS-128L due to the strong cryptographic properties of AES Sbox. The comparision between our attack on Rocca and fault attacks on similarly designed ciphers in terms of the fault model, number of faults required, and the number of state registers in which fault is injected is described in Table 1.

We also discuss the possible strategies to reduce the threat the our differential fault attack in Section 4.2. We believe that the threat of a differential fault attack cannot be completely abated, but these strategies can help make the attack computationally hard or impractical.

The attack presented in this paper assumes that the adversary can induce faults at precise location and timing. When we assume that an adversary can inject faults into the cipher, it is a strong assumption. Similarly, if the fault attack model assumes that the adversary can inject faults with precise location and timing, it is an another strong assumption. Thus, the assumption in this paper is very strong. It should also be noted that this attack requires the assumption of nonce-misuse.

Motivation. As stated above the attack on Rocca described in this paper assumes the nonce misuse settings. The authors of Rocca does not claim any security of the cipher in nonce-misuse setting.

However, the security evaluation of ciphers in the nonce misuse setting is very important. It is considered near-impossible to prevent nonce misuse in the presence of physical attacks [3]. In the practical applications, the case of nonce misuse might exist due to the poor implementations, misuse of cipher and technical flaw of nonce generation function etc. The impossibility of full robustness against nonce-misuse settings in practical applications of a cryptographic algorithm is considered a reality. There has been a lot of study regarding the security of cryptographic systems in the nonce misuse settings, such as in [4,14,9,10] to cite a few. The differential fault attacks on AEGIS and Tiaoxin family of ciphers [15,6,1] also have the requirement of nonce-reuse.

Hence, cryptanalyzing Rocca in the nonce-misuse setting is very important for complete understanding of its security and this analysis should be done before the ciphers' wide commercial deployments.

Organization. In this work we have presented a differential fault attack on Rocca under the nonce reuse scenario. In Section 2, we describe the basics of DFA and the specification of Rocca. In Section 3, we describe our attack on Rocca for complete internal state and key recovery. In Section 4 we present a comparision of Rocca with other ciphers from the point of view of DFA, and present possible countermeasures. We conclude our work in Section 5.

2 Preliminaries

2.1 Differential Fault Attack

Differential fault attacks can be thought of as a combination of the classical differential attack and the side channel attacks (SCAs). While the procedure to retrieve the secret state (key or the internal state) in DFA is similar to the classical differential cryptanalysis, the DFA also belongs to the set of very powerful SCAs. The injected faults can be specified as positively observable leaked information by capable attackers [13]. These attacks injects a well-timed and well-aimed faults which exploits the confusion and diffusion property of a cryptographic algorithm and this allows the attacker to obtain a desired difference distribution in the ciphertext.

The first fault attack was presented by Boneh, DeMillo and Lipton [2] when they published an attack on RSA signature scheme. Inspired by this attack, Biham and Shamir [4] described such attacks on symmetric ciphers and called these attacks Differential Fault Attack (DFA). DFA are the most commonly used fault technique. In these attacks the attacker induces faults in the cryptographic primitive, then obtains at least one correct ciphertext and one faulty ciphertext and uses this information to obtain some knowledge of the secret state. These faults induced can be described using the fault models, which includes the timing, location of the fault and the number of bits or bytes affected in the register in which the fault is induced. The fault model provides the attacker some information about the difference between certain states of the computation of the cipher.

For a general idea of DFA one can take the following example: for a function S, a attacker injects a fault ϵ and obtains a relation of the type: $S(x) \oplus S(x \oplus \epsilon) =$

δ; where the difference δ is known to the attacker, x is the unknown value that she wants to retrieve and the fault ϵ is either known (deterministic model) or unknown (random model) to the attacker.

For the attack in this work, the attacker is assumed to have full control on the timing and the location of the fault. She should also be able to induce not permanent, but transient faults in the random model.

2.2 Specification of Encryption Phase of Rocca

We here describe only the encryption phase, which is where we intend to inject the faults. For details of other phases, such as initialization, associated data processing, finalization and tag generations, the readers are requested to refer to the cipher specification description [12].

Rocca has an internal state with eight 128-bit registers $S[0], S[1], \ldots, S[7]$ and thus has a total state size of $8 \times 128 = 1024$ bits. The internal state is updated using a nonlinear state update function defined below:

$$\begin{aligned} S_{i+1}[0] &= S_i[7] \oplus X_0 & S_{i+1}[4] &= S_i[3] \oplus X_1 \\ S_{i+1}[1] &= R(S_i[0], S_i[7]) & S_{i+1}[5] &= R(S_i[4], S_i[3]) \\ S_{i+1}[2] &= S_i[1] \oplus S_i[6] & S_{i+1}[6] &= R(S_i[5], S_i[4]) \\ S_{i+1}[3] &= R(S_i[2], S_i[1]) & S_{i+1}[7] &= S_i[0] \oplus S_i[6] \end{aligned} \tag{1}$$

where $R(X)$ is defined as:

$$R(X) = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}(X) \tag{2}$$

Note that the transformation R is invertible. We denote the reverse of R by R^{-1}. This is an important property for state recovery, and also in the case of key recovery, as it allows the internal state to be clocked backwards.

This update function has two external outputs X_0 and X_1 and the non-linearity is provided by the the transformation R, which is applied to four out of the total eight registers. During the encryption phase, a block of $256-$bit plaintext $P_i = P_i^0 || P_i^1$ is encrypted to produce a 256-bit ciphertext $C_i = C_i^0 || C_i^1$ as defined below:

$$\begin{aligned} C_i^0 &= P_i^0 \oplus R(S_i[1]) \oplus S_i[5] \\ C_i^1 &= P_i^1 \oplus R(S_i[0] \oplus S_i[4]) \oplus S_i[2] \end{aligned} \tag{3}$$

After this step the internal state is updated using the update function described in Equation 1 and the next 256-bits of plaintext is encrypted as in Equation 3, i.e.

$$\begin{aligned} C_{i+1}^0 &= P_{i+1}^0 \oplus R(S_{i+1}[1]) \oplus S_{i+1}[5] \\ C_{i+1}^1 &= P_{i+1}^1 \oplus R(S_{i+1}[0] \oplus S_{i+1}[4]) \oplus S_{i+1}[2] \end{aligned} \tag{4}$$

For simplicity we assume the following in the rest of this section:

– $S_i[0] = [a_{i_0}, a_{i_1}, \ldots, a_{i_{15}}]$, where each $a^j_{i_l}$ is a byte.
– $S_i[1] = [b_{i_0}, b_{i_1}, \ldots, b_{i_{15}}]$, where each $b^j_{i_l}$ is a byte.
– $S_i[2] = [c_{i_0}, c_{i_1}, \ldots, c_{i_{15}}]$, where each $c^j_{i_l}$ is a byte.
– $S_i[3] = [d_{i_0}, d_{i_1}, \ldots, d_{i_{15}}]$, where each $d^j_{i_l}$ is a byte.
– $S_i[4] = [e_{i_0}, e_{i_1}, \ldots, e_{i_{15}}]$, where each $e^j_{i_l}$ is a byte.
– $S_i[5] = [f_{i_0}, f_{i_1}, \ldots, f_{i_{15}}]$, where each $f^j_{i_l}$ is a byte.
– $S_i[6] = [g_{i_0}, g_{i_1}, \ldots, g_{i_{15}}]$, where each $g^j_{i_l}$ is a byte.
– $S_i[7] = [h_{i_0}, h_{i_1}, \ldots, h_{i_{15}}]$, where each $h^j_{i_l}$ is a byte.

Also all the S_i's, can be represented as a 4×4 matrices. In our attack below we use this representation.

3 Key and Internal State Recovery of Rocca

In this section, we describe full state recovery attack on a block cipher Rocca using fault injections.

3.1 Attack Procedure

We describe here the process to recover the internal states $S_i[1]$, and $S_i[5]$. A similar procedure can be applied to recover the rest of the states. Let us assume that the adversary first encrypts a plaintext P_i^0 and obtains the corresponding ciphertext C_i^0,. Then the adversary repeats the encryption of the same plaintext after injecting a fault into b_{i_1}, such that this byte becomes faulty. Let ϵ is the fault injected and let the faulty state be denoted by $\tilde{S}_i[1]$. Thus the faulty state $\tilde{S}_i[1]$ differs from $S_i[1]$ only at the first byte, i.e. the first byte of $\tilde{S}_i[1]$ has the entry $b_{i_0} \oplus \epsilon$. This encryption with injection of the fault generates a faulty ciphertext, say $\tilde{C}_i^0$.

Without loss of generality, for the rest of the attack, we assume that the plaintext is all zeroes. The attack can easily be generalized for a randomly chosen plaintext. Now the difference between the fault-free ciphertext and faulty ciphertext is:

$$\begin{aligned} C_i^0 \oplus \tilde{C}_i^0 &= R(S_i[1]) \oplus S_i[5] \oplus R(\tilde{S}_i[1]) \oplus S_i[5] \\ &= R(S_i[1]) \oplus R(\tilde{S}_i[1]) \end{aligned} \tag{5}$$

The matrix representation of the above difference is:

$$\begin{bmatrix} \delta_0^0 & \delta_1^0 & \delta_2^0 & \delta_3^0 \\ \delta_4^0 & \delta_5^0 & \delta_6^0 & \delta_7^0 \\ \delta_8^0 & \delta_9^0 & \delta_{10}^0 & \delta_{11}^0 \\ \delta_{12}^0 & \delta_{13}^0 & \delta_{14}^0 & \delta_{15}^0 \end{bmatrix} = R(\begin{bmatrix} b_{i_0} & b_{i_1} & b_{i_2} & b_{i_3} \\ b_{i_4} & b_{i_5} & b_{i_6} & b_{i_7} \\ b_{i_8} & b_{i_9} & b_{i_{10}} & b_{i_{11}} \\ b_{i_{12}} & b_{i_{13}} & b_{i_{14}} & b_{i_{15}} \end{bmatrix}) \oplus R(\begin{bmatrix} b_{i_0} \oplus \epsilon & b_{i_1} & b_{i_2} & b_{i_3} \\ b_{i_4} & b_{i_5} & b_{i_6} & b_{i_7} \\ b_{i_8} & b_{i_9} & b_{i_{10}} & b_{i_{11}} \\ b_{i_{12}} & b_{i_{13}} & b_{i_{14}} & b_{i_{15}} \end{bmatrix}) \tag{6}$$

where δ_j's denote the difference in the fault-free ciphertext and faulty ciphertext. Now using the definition of $R(X)$ from Equation 2, we have:

1. Applying SubBytes we get

$$SubBytes(S_i[1]) = \begin{bmatrix} s(b_{i_0}) & s(b_{i_1}) & s(b_{i_2}) & s(b_{i_3}) \\ s(b_{i_4}) & s(b_{i_5}) & s(b_{i_6}) & s(b_{i_7}) \\ s(b_{i_8}) & s(b_{i_9}) & s(b_{i_{10}}) & s(b_{i_{11}}) \\ s(b_{i_{12}}) & s(b_{i_{13}}) & s(b_{i_{14}}) & s(b_{i_{15}}) \end{bmatrix} \text{ and}$$

$$SubBytes(\tilde{S}_i[1]) = \begin{bmatrix} s(b_{i_0} \oplus \epsilon) & s(b_{i_1}) & s(b_{i_2}) & s(b_{i_3}) \\ s(b_{i_4}) & s(b_{i_5}) & s(b_{i_6}) & s(b_{i_7}) \\ s(b_{i_8}) & s(b_{i_9}) & s(b_{i_{10}}) & s(b_{i_{11}}) \\ s(b_{i_{12}}) & s(b_{i_{13}}) & s(b_{i_{14}}) & s(b_{i_{15}}) \end{bmatrix}$$

   where $s(x)$ is the S-box value of x.

2. Applying ShiftRows we get

$$ShiftRows \circ SB(S_i[1]) = \begin{bmatrix} s(b_{i_0}) & s(b_{i_1}) & s(b_{i_2}) & s(b_{i_3}) \\ s(b_{i_5}) & s(b_{i_6}) & s(b_{i_7}) & s(b_{i_4}) \\ s(b_{i_{10}}) & s(b_{i_{11}}) & s(b_{i_8}) & s(b_{i_9}) \\ s(b_{i_{15}}) & s(b_{i_{12}}) & s(b_{i_{13}}) & s(b_{i_{14}}) \end{bmatrix} \text{ and}$$

$$ShiftRows \circ SB(\tilde{S}_i[1]) = \begin{bmatrix} s(b_{i_0} \oplus \epsilon) & s(b_{i_1}) & s(b_{i_2}) & s(b_{i_3}) \\ s(b_{i_5}) & s(b_{i_6}) & s(b_{i_7}) & s(b_{i_4}) \\ s(b_{i_{10}}) & s(b_{i_{11}}) & s(b_{i_8}) & s(b_{i_9}) \\ s(b_{i_{15}}) & s(b_{i_{12}}) & s(b_{i_{13}}) & s(b_{i_{14}}) \end{bmatrix}$$

3. Applying MixColumns we get

$$R(S_i[1]) = \begin{bmatrix} 2 \cdot s(b_{i_0}) \oplus 3 \cdot s(b_{i_5}) \oplus 1 \cdot s(b_{i_{10}}) \oplus 1 \cdot s(b_{i_{15}}) & z_1 & z_2 & z_3 \\ 1 \cdot s(b_{i_0}) \oplus 2 \cdot s(b_{i_5}) \oplus 3 \cdot s(b_{i_{10}}) \oplus 1 \cdot s(b_{i_{15}}) & z_6 & z_7 & z_4 \\ 1 \cdot s(b_{i_0}) \oplus 1 \cdot s(b_{i_5}) \oplus 2 \cdot s(b_{i_{10}}) \oplus 3 \cdot s(b_{i_{15}}) & z_{11} & z_8 & z_9 \\ 3 \cdot s(b_{i_0}) \oplus 1 \cdot s(b_{i_5}) \oplus 1 \cdot s(b_{i_{10}}) \oplus 2 \cdot s(b_{i_{15}}) & z_{12} & z_{13} & z_{14} \end{bmatrix} \text{ and}$$

$$R(\tilde{S}_i[1]) = \begin{bmatrix} 2 \cdot s(b_{i_0} \oplus \epsilon) \oplus 3 \cdot s(b_{i_5}) \oplus 1 \cdot s(b_{i_{10}}) \oplus 1 \cdot s(b_{i_{15}}) & z_1 & z_2 & z_3 \\ 1 \cdot s(b_{i_0}) \oplus 2 \cdot s(b_{i_5}) \oplus 3 \cdot s(b_{i_{10}}) \oplus 1 \cdot s(b_{i_{15}}) & z_6 & z_7 & z_4 \\ 1 \cdot s(b_{i_0}) \oplus 1 \cdot s(b_{i_5}) \oplus 2 \cdot s(b_{i_{10}}) \oplus 3 \cdot s(b_{i_{15}}) & z_{11} & z_8 & z_9 \\ 3 \cdot s(b_{i_0}) \oplus 1 \cdot s(b_{i_5}) \oplus 1 \cdot s(b_{i_{10}}) \oplus 2 \cdot s(b_{i_{15}}) & z_{12} & z_{13} & z_{14} \end{bmatrix}$$

   where each z_j's can be computed as the first column has been computed.

Replacing these values in Eqn 6, we get

$$\begin{bmatrix} \delta_0^0 & \delta_1^0 & \delta_2^0 & \delta_3^0 \\ \delta_4^0 & \delta_5^0 & \delta_6^0 & \delta_7^0 \\ \delta_8^0 & \delta_9^0 & \delta_{10}^0 & \delta_{11}^0 \\ \delta_{12}^0 & \delta_{13}^0 & \delta_{14}^0 & \delta_{15}^0 \end{bmatrix} = \begin{bmatrix} 2 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) & 0 & 0 & 0 \\ 1 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) & 0 & 0 & 0 \\ 1 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) & 0 & 0 & 0 \\ 3 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) & 0 & 0 & 0 \end{bmatrix}$$

Hence we have the following four equations

$$\begin{aligned} \delta_0^0 &= 2 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) \\ \delta_4^0 &= (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) \\ \delta_8^0 &= (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) \\ \delta_{12}^0 &= 3 \cdot (s(b_{i_0}) \oplus s(b_{i_0} \oplus \epsilon)) \end{aligned} \tag{7}$$

where the value of δ_i^0 is known and b_{i_0} and ϵ are unknown. Analyzing the Equation 7, we can recover the value of b_{i_0}. Similarly faults on other bytes will provide the information on other bytes of the state $S_i[1]$. Once $S_i[1]$ is recovered we can compute the value $S_i[5]$ as follows:

$$S_i[5] = R(S_i[1]) \oplus C_i^0$$

3.2 Analysis of Equation 7

Equation 7, can also be generalized as:

$$s(x) \oplus s(x+\epsilon) = c_i^{-1}\delta_i^0 \tag{8}$$

where $c_i = 1, 2$ or 3, (x and ϵ) are unknown variables, and δ_i^0 is a known constant. We know that the following two proposition holds for the AES S-box:

Proposition 1. *For any input difference ϵ of the AES S-box, a certain output difference δ always appears twice while the other output differences appear just once.*

Proposition 2. *For any input difference ϵ of the AES S-box, the number of possible output difference is always 127.*

Using Proposition 1, Proposition 2, and the analysis described in [7, Section 3.5] we can deduce that using four random S-box input differences i.e., four values of ϵ, and as the value of x remains the same, we can uniquely retrieve the value of x which satisfies the set of Equation 8.

These faults are induced in the internal state and not the plaintext, i.e. the faults do not depend on the plaintext. However, inducing these faults will generate 4 faulty ciphertexts for the same plaintext. Thus using 4 pairs of faultfree and faulty ciphertexts we can retrieve the value of b_{i_0}. Similarly the complete value of $S_i[1]$, which contains 16 bytes $b_{i_0}, \cdots, b_{i_{15}}$, can be retrieved using 16×4 faults and four pairs of faultfree and faulty ciphertext.

3.3 Internal State Recovery

We now describe how to recover the remaining internal states using the process described above.
From Equation 3 and the assumption that the plaintext is all zeroes we have

$$C_i^1 = R(S_i[0] \oplus S_i[4]) \oplus S_i[2] \tag{9}$$

If we inject a fault in the first byte of $S_i[4]$, then the faulty ciphertext generated would be

$$\tilde{C}_i^1 = R(S_i[0] \oplus \tilde{S}_i[4]) \oplus S_i[2] \tag{10}$$

From Equations 9 and 10 we get

$$C_i^1 \oplus \tilde{C}_i^1 = R(S_i[0] \oplus S_i[4]) \oplus R(S_i[0] \oplus \tilde{S}_i[4]) \tag{11}$$

Proceeding as the attack described above, we need to solve the following set of equations:

$$\begin{aligned} \delta_0^1 &= 2 \cdot (s(a_{i_0} \oplus e_{i_0}) \oplus s(a_{i_0} \oplus e_{i_0} \oplus \epsilon)) \\ \delta_4^1 &= (s(a_{i_0} \oplus e_{i_0}) \oplus s(a_{i_0} \oplus e_{i_0} \oplus \epsilon)) \\ \delta_8^1 &= (s(a_{i_0} \oplus e_{i_0}) \oplus s(a_{i_0} \oplus e_{i_0} \oplus \epsilon)) \\ \delta_{12}^1 &= 3 \cdot (s(a_{i_0} \oplus e_{i_0}) \oplus s(a_{i_0} \oplus e_{i_0} \oplus \epsilon)) \end{aligned} \tag{12}$$

and we can recover the value of first byte of $S_i[0] \oplus S_i[4]$ using four faults and four pairs of faultfree and faulty ciphertext. Applying the same strategy to the remaining 15 bytes we can recover the complete values of $S_i[0] \oplus S_i[4]$ and $S_i[2]$.

We have recovered the registers $S_i[1], S_i[5], S_i[2]$ and the value of $S_i[0] \oplus S_i[4]$. Recovering $S_i[4]$ will give the value of $S_i[0]$. Thus now we try to obtain the value of $S_i[4]$.

Using Equation 1, we can rewrite Equation 4 as follows, along with the assumption that the plaintext is all zeroes:

$$C_{i+1}^0 = R(R(S_i[0]) \oplus S_i[7]) \oplus R(S_i[4]) \oplus S_i[3] \tag{13}$$

and

$$C_{i+1}^1 = R(S_i[7] \oplus S_i[3]) \oplus S_i[1] \oplus S_i[6] \tag{14}$$

Inducing a fault in the first byte of $S_i[3]$ will provide a faulty ciphertext corresponding to Equation 14 and we have

$$\tilde{C}_{i+1}^1 = R(S_i[7] \oplus \tilde{S}_i[3]) \oplus S_i[1] \oplus S_i[6] \tag{15}$$

Thus we have

$$C_{i+1}^1 \oplus \tilde{C}_{i+1}^1 = R(S_i[7] \oplus S_i[3]) \oplus R(S_i[7] \oplus \tilde{S}_i[3]) \tag{16}$$

Using the same attack strategy we obtain the value of the first byte of $S_i[7] \oplus S_i[3]$ by solving the following equations:

$$\begin{aligned} \bar{\delta}_0^1 &= 2 \cdot (s(h_{i_0} \oplus d_{i_0}) \oplus s(h_{i_0} \oplus d_{i_0} \oplus \epsilon)) \\ \bar{\delta}_4^1 &= (s(h_{i_0} \oplus d_{i_0}) \oplus s(h_{i_0} \oplus d_{i_0} \oplus \epsilon)) \\ \bar{\delta}_8^1 &= (s(h_{i_0} \oplus d_{i_0}) \oplus s(h_{i_0} \oplus d_{i_0} \oplus \epsilon)) \\ \bar{\delta}_{12}^1 &= 3 \cdot (s(h_{i_0} \oplus d_{i_0}) \oplus s(h_{i_0} \oplus d_{i_0} \oplus \epsilon)) \end{aligned} \tag{17}$$

Inducing 4 faults and using four pairs of faultfree and faulty ciphertext for each 16 bytes of $S_i[3]$, we recover the values of $S_i[7] \oplus S_i[3]$. Since the value of $S_i[1]$

is already known, from Equation 14, we can recover the value of $S_i[6]$.

Since a fault was already induced in $S_i[4]$ and $S_i[3]$, thus from Equation 13 we obtain the value of the faulty ciphertext as:

$$\tilde{C}^0_{i+1} = R(R(S_i[0], S_i[7])) \oplus R(\tilde{S}_i[4]) \oplus \tilde{S}_i[3] \tag{18}$$

Hence we have

$$C^0_{i+1} \oplus \tilde{C}^0_{i+1} = R(S_i[4]) \oplus R(\tilde{S}_i[4]) \oplus S_i[3] \oplus \tilde{S}_i[3] \tag{19}$$

Since the value of the fault induced in $S_i[3]$ is known, hence the value of $S_i[3] \oplus \tilde{S}_i[3]$ is also known. Using the same attack strategy we obtain the value of $S_i[4]$, which then allows us to determine the value of $S_i[0]$.

Let us assume that $R(S_i[4]) = \alpha_0$ and $R(S_i[0]) = \alpha_1$, where α_0, α_1 are known constants. Replacing these values in Equation 13 we get

$$C^0_{i+1} = R(\alpha_1 \oplus S_i[7]) \oplus \alpha_0 \oplus S_i[3] \tag{20}$$

Since we already know the value of $S_i[7] \oplus S_i[3]$ and let $S_i[7] \oplus S_i[3] = \alpha_2$, Equation 20 can be written as:

$$\begin{aligned} C^0_{i+1} &= R(\alpha_1 \oplus S_i[7]) \oplus \alpha_0 \oplus S_i[7] \oplus \alpha_2 \\ &= R(S_i[7] \oplus \alpha_1) \oplus S_i[7] \oplus \alpha_0 \oplus \alpha_2 \end{aligned} \tag{21}$$

We can obtain the value of $S[7]$ by solving Equation 21.

Therefore we obtain the complete internal state.Once the entire internal state is known at a certain timestamp i, the cipher can be clocked forwards to recover all subsequent plaintext using known ciphertext and state contents.

3.4 Key Recovery

The round update function is reversible and thus the internal state can be clocked backwards to retrieve the value of the secret key. Consider that at any time stamp $i+1$ the complete internal state is known, i.e. $S_{i+1}[0], S_{i+1}[1], S_{i+1}[2], S_{i+1}[3], S_{i+1}[4], S_{i+1}[5], S_{i+1}[6], S_{i+1}[7]$, is known. Now the internal state at the previous time stamp i can be retrieved by the following done in order:

1. $S_i[7] = S_{i+1}[0] \oplus X_0$, where X_0 is either the known plaintext or the constant $Z_0 = 428a2f98d728ae227137449123ef65cd$
2. $S_i[0] = R^{-1}(S_{i+1}[1] \oplus S_i[7])$
3. $S_i[3] = S_{i+1}[4] \oplus X_1$, where X_1 is either the known plaintext or the constant $Z_1 = b5c0fbcfec4d3b2fe9b5dba58189dbbc$
4. $S_i[4] = R^{-1}(S_{i+1}[5] \oplus S_i[3])$
5. $S_i[5] = R^{-1}(S_{i+1}[6] \oplus S_i[4])$
6. $S_i[6] = S_{i+1}[7] \oplus S_i[0]$
7. $S_i[1] = S_{i+1}[2] \oplus S_i[6]$
8. $S_i[2] = R^{-1}(S_{i+1}[3] \oplus S_i[1])$

We can reverse clock the internal state until we reach the initial state and recover the secret key.

4 Discussion

This paper shows that a differential fault attack is possible in the nonce-reuse scenario on Rocca. When implementing Rocca care must be taken to avoid nonce reuse so that any attacker cannot obtain plaintext-ciphertext pairs using the same parameters. In this section we compare the security of Rocca with some other ciphers against DFA and present some countermeasures to protect Rocca against DFA.

4.1 Comparision

We compare the fault attack presented in this paper with that of the fault attacks on the other two ciphers, the Aegis family and Tiaoxin which was described in [15]and [6]. All these ciphers are AES based encryption scheme inspired by the work of Jean and Nikolic [8]. The comparision is described in Table 1.

The security any cipher against DFA is generally measured by the fault model used in the attacks and the number of faults required to implement the attack. It is also assumed that the location and timing of the faults are known to the attacker (however in some cases the attacker does not need to know about the location and timing of the attack, depending on the construction of the cipher). In our work as well as the work of [15], the fault model used is a *single fault* model, i.e. the fault injected has a width of $\leq b$, where b is the input size of the Sbox used in the cipher, AES Sbox in this case. However, the difference is the nature of this fault injected.

– *Random fault*: a fault which can have any value between 1 and $2^b - 1$ and all these values are equally possible. This model is considered to be the most practical fault model. The DFA described in [15] uses this model.
– *Known fault*: in this case the fault has a specific value as defined by the attacker. This paper uses this model of fault. This fault model requires a stronger adversary. However, it has been shown that such fault model is achievable.

The fault model used in [6] is:

– *Bit Flipping*: A specific internal bit of the cipher is flipped. In practice, it is more complex than the random fault, but it has been shown to be practical in several papers/attacks.

From Table 1, we cannot make a fair comparision of the security of the ciphers against DFA, still the following conclusions would be fair:

– For a state size of 1024, the requirement 4×48 faults is more than AEGIS-128L in random byte model. Thus we could say it is slightly more resistant towards DFA.

Cipher	Recovery	State Size	Fault Model	No. of Faults	Target*
AEGIS-128 [15]	State	540	Random	64	4/5
AEGIS-128 [6]	State	540	Bit Flipping	3×128	3/5
AEGIS-256 [15]	State	768	Random	80	5/6
AEGIS-256 [6]	State	768	Bit Flipping	4×128	4/6
AEGIS-128L [15]	State	1024	Random	128	8/8
AEGIS-128L [6]	State	1024	Bit Flipping	4×128	4/8
Rocca (Section3)	State and Key	1024	Random	4×48	3/8
Tiaoxin-346 [15]	State	1664	Random	208	13/13
Tiaoxin-346 [15]	Key	1664	Random	48	3/13

Table 1: Comparision of security against DFA of AES based ciphers. Target* implies the number of registers in which the faults were injected out of the total registers the complete internal state is divided into.

– Since the key recovery for Tiaoxin-346 requires only 48 faults and only 1 known plaintext, it can be considered slightly weak against DFA. However, with regards to state recovery Tiaoxin-346 is the most secure against DFA among the ciphers described in Table 1.

We describe below some countermeasure strategies for Rocca.

4.2 Mitigating the Differential Fault Attack on Rocca

We discuss here a few probable strategies to avoid differential fault attack or make it computationally hard or impractical.

While the construction of Aegis family, Tiaoxin and Rocca are all based on the AES round function, the basic difference is in the generation of the keystream or ciphertext. The Aegis family of ciphers and Tiaoxin has an output function that includes a bitwise AND operation but Rocca uses the AES round function even in its ciphertext generation phase. It was observed in [15] that the ciphers which has the following two properties can be attacked using fault attacks:

– The ciphertext output functions contains one quadratic term and is otherwise linear.
– The internal state transitions contain linear paths across different stages and do not have external input.

Based on these observations a few countermeasures were suggested.

However, the same countermeasures cannot be applied to Rocca as the output function is different from the these ciphers. The output function of Rocca makes use of AES round function on a state register and a XOR with another state register. Based on these observations and considering that the speed of the cipher should not be compromised, we observed that the potentially useful strategies for preventing partial recovery of state information from leading to full state recovery or key recovery include the following:

– Using more than one state registers while encrypting the plaintext. For example changing $C_i^0 = P_i^0 \oplus R(S_i[1]) \oplus S_i[5]$ to $C_i^0 = P_i^0 \oplus R(S_i[1] \oplus S_i[3]) \oplus S_i[5]$ will restrict the attacker to obtain values of combinations of state register $S_i[1] \oplus S_i[3]$ instead of obtaining the value of a single state register $S_i[1]$ directly. This will increase the required number of faults required.
– State registers used in the keystream generation on which the AES round function R is applied should again be non-linearly mixed when the state is updated. This will prevent the attacker from obtaining information of a state register by applying faults at different time stamps
– It is always a safer option to modify the state update function during intialization phase to be non-invertible without the knowledge of the secret key. This will prevent a state recovery attack to lead to key recovery.

However, these mitigation strategies leaves room to determine if they introduce some vulnerabilities which can be exploited by other attackers.

5 Conclusion

In this paper we demonstrate a differential fault attack on Rocca under the nonce reuse scenario. We show that we can recover the complete internal state using 4×48 faults. Since the round update function is reversible, internal state recovery also allows key recovery. In this attack we have assumed that the adversary has the knowledge of the location of the fault induced. It would be interesting to study this attack in the model where the adversary has no knowledge of the location of the fault induced.

6 Acknowledgement

Takanori Isobe is supported by JST, PRESTO Grant Number JPMJPR2031, Grant-in- Aid for Scientific Research (B)(KAKENHI 19H02141) for Japan Society for the Promotion of Science, and Support Center for Advanced Telecommunications Technology Research (SCAT).

References

1. Bartlett, H., Dawson, E., Qahur Al Mahri, H., Salam, M., Simpson, L. and Wong, K.K.H., 2019. Random fault attacks on a class of stream ciphers. Security and Communication Networks, 2019.
2. Boneh D., DeMillo R.A., Lipton R.J. (1997) On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy W. (eds) Advances in Cryptology EUROCRYPT 97. EUROCRYPT 1997. Lecture Notes in Computer Science, vol 1233. Springer, Berlin, Heidelberg.
3. Berti, F., Guo, C., Pereira, O., Peters, T. and Standaert, F.X., 2019. TEDT, a Leakage-Resilient AEAD mode for High (Physical) Security Applications. in IACR Transactions on Cryptographic Hardware and Embedded Systems, vol 2020, num 1, pp 256-320.

4. Biham E., Shamir A. (1997) Differential fault analysis of secret key cryptosystems. In: Kaliski B.S. (eds) Advances in Cryptology CRYPTO '97. CRYPTO 1997. Lecture Notes in Computer Science, vol 1294. Springer, Berlin, Heidelberg.
5. Blomer J., Seifert JP. (2003) Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In: Wright R.N. (eds) Financial Cryptography. FC 2003. Lecture Notes in Computer Science, vol 2742. Springer, Berlin, Heidelberg.
6. Dey, P., Rohit, R.S., Sarkar, S. and Adhikari, A., 2016, September. Differential fault analysis on Tiaoxin and AEGIS family of ciphers. In International Symposium on Security in Computing and Communication (pp. 74-86). Springer, Singapore.
7. Dusart, P., Letourneux, G. and Vivolo, O., 2003, October. Differential fault analysis on AES. In International Conference on Applied Cryptography and Network Security (pp. 293-306). Springer, Berlin, Heidelberg.
8. Jean, J. and Nikoli, I., 2016, March. Efficient design strategies based on the AES round function. In International Conference on Fast Software Encryption (pp. 334-353). Springer, Berlin, Heidelberg.
9. Khairallah, M., Bhasin, S. and Chattopadhyay, A., 2019, June. On misuse of nonce-misuse resistance: Adapting differential fault attacks on (few) CAESAR winners. In 2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI) (pp. 189-193). IEEE.
10. Khairallah, M., Hou, X., Najm, Z., Breier, J., Bhasin, S. and Peyrin, T., 2019, July. SoK: On DFA Vulnerabilities of Substitution-Permutation Networks. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (pp. 403-414).
11. Nikolic, I. Tiaoxin-346 (version 2.1). CAESAR competition. Available from https://competitions.cr.yp.to/round3/tiaoxinv21.pdf
12. Sakamoto, K., Liu, F., Nakano, Y., Kiyomoto, S. and Isobe, T., 2021. Rocca: An Efficient AES-based Encryption Scheme for Beyond 5G. IACR Transactions on Symmetric Cryptology, pp.1-30.
13. Sakiyama, K., Li, Y., Iwamoto, M. and Ohta, K., 2011. Information-theoretic approach to optimal differential fault analysis. IEEE Transactions on Information Forensics and Security, 7(1), pp.109-120.
14. Song, L., Tu, Y., Shi, D. and Hu, L., 2021. Security analysis of Subterranean 2.0. Designs, Codes and Cryptography, pp.1-31.
15. Wong, K.K.H., Bartlett, H., Simpson, L. and Dawson, E., 2019, December. Differential random fault attacks on certain CAESAR stream ciphers. In International Conference on Information Security and Cryptology (pp. 297-315). Springer, Cham.
16. Wu, H., Preneel, B. AEGIS: A Fast Authenticated Encryption Algorithm (v1.1) CAESAR competition. Available from https://competitions.cr.yp.to/round3/aegisv11.pdf