Security Analysis of the RC4+ Stream Cipher

Subhadeep Banik[1], Santanu Sarkar[2], and Raghu Kacker[2]

[1] Applied Statistics Unit, Indian Statistical Institute,
203 B T Road, Kolkata 700 108, India
s.banik_r@isical.ac.in
[2] National Institute of Standards and Technology, 100 Bureau Drive, Stop 8930
Gaithersburg, MD 20899-8930, USA
santanu.sarkar@nist.gov

Abstract. The RC4+ stream cipher was proposed by Maitra and Paul at Indocrypt 2008. The authors had claimed that RC4+ ironed out most of the weaknesses of the alleged RC4 stream cipher and was only marginally slower than RC4 in software. In this paper we show that it is possible to mount a distinguishing attack on RC4+ based on the bias of the first output byte. The distinguisher requires around 2^{26} samples produced by different keys of RC4+. In the second part of the paper we study the possibility of mounting the differential fault attack on RC4 proposed by Biham et. al. in FSE 2005, on RC4+. We will show that that the RC4+ is vulnerable to differential fault attack and it is possible to recover the entire internal state of the cipher at the beginning of the PRGA by injecting around $2^{17.2}$ faults.

Keywords: Cryptanalysis, Differential Fault Attack, Distinguishing Attack, RC4, RC4+, Stream Cipher.

1 Introduction

There has been extensive research in recent years to come up with RC4-like stream ciphers that while marginally slower in software, would wipe out the known shortcomings of RC4. Many such ciphers like RC4A [10], NGG [9], GGHN [4], VMPC [14] have been proposed to fulfil this objective. However, all the aforementioned ciphers have had distinguishing attacks reported against them [7, 11–13]. RC4+ is another stream cipher that belongs to this family. The cipher was proposed by Maitra and Paul at Indocrypt 2008 [5]. The authors had claimed that RC4+ while marginally slower than RC4 in software, would resist all the known distinguishing and state recovery attacks against RC4. To the best of our knowledge, no cryptanalytic advance has been made against this cipher.

Description of the Cipher. The physical structure of RC4+ is the same as that of RC4. It consists of a permutation S of $N = 256$ elements from the integer ring $\mathbb{Z}_{256}$. It also uses two index pointers i, j of size 1 byte each. As in RC4, during the Key Scheduling Algorithm(KSA), S is initialized to the identity permutation

G. Paul and S. Vaudenay (Eds.): INDOCRYPT 2013, LNCS 8250, pp. 297–307, 2013.

and mixed using a Secret Key K of size l bytes (typically $l = 16$). Then, the array S is further scrambled using an l byte IV, after which another layer of zig-zag scrambling is performed. The exact details of the KSA are given in Table 1. Note that all addition operations are performed in $\mathbb{Z}_{256}$, and $\oplus$ denotes bitwise-XOR. The array V used in the KSA is defined as

$$V[i] = \begin{cases} IV[127 - i], & \text{if } 128 - l \leq i \leq 127, \\ IV[i - 128], & \text{if } 128 \leq i \leq 127 + l \\ 0, & \text{otherwise.} \end{cases}$$

Table 1. KSA routine for `RC4+`

Input: Secret Key K, Initial Vector IV
Output: Permutation S on $\mathbb{Z}_{256}$

for $i = 0$ *to* 255 **do**
  $S[i] = i$;
end
$j \leftarrow 0$
`Key Loading`
for $i = 0$ *to* 255 **do**
  $j \leftarrow j + S[i] + K[i \bmod l]$;
  Swap $S[i], S[j]$;
end
`IV Loading`
for $i = 127$ *to* 0 **do**
  $j \leftarrow$
  $(j + S[i]) \oplus (K[i \bmod l] + V[i])$;
  Swap $S[i], S[j]$;
end
for $i = 128$ *to* 255 **do**
  $j \leftarrow$
  $(j + S[i]) \oplus (K[i \bmod l] + V[i])$;
  Swap $S[i], S[j]$;
end

`Zig-Zag Scrambling`

for $y = 0$ *to* 255 **do**
  if $y \equiv 0 \bmod 2$ **then**
    $i = \frac{y}{2}$;
  end
  else
    $i = 128 - \frac{y+1}{2}$;
  end
  $j \leftarrow j + S[i] + K[i \bmod l]$;
  Swap $S[i], S[j]$;
end

The PRGA routine of `RC4+` deviates slightly from the simplistic structure of `RC4`. In order to protect against the well known second output byte bias of Mantin-Shamir [6] and the permutation recovery attack of Maximov and Khovratovich [8], the designers propose to make the output keystream byte functions of a few other locations of the permutation array S. The details of the PRGA routine are given in Table 2. Note that $\gg$ and $\ll$ denote right and left bitwise shifts respectively.

Table 2. PRGA routine for RC4+

Input: Permutation S on $\mathbb{Z}_{256}$
Output: Output Keystream bytes Z

$i = j = 0$;
while *Keystream is required* **do**
  $i \leftarrow i + 1$;
  $j \leftarrow j + S[i]$;
  Swap $S[i], S[j]$;

  $t \leftarrow S[i] + S[j]$;
  $t' \leftarrow (S[i \gg 3 \oplus j \ll 5] + S[i \ll 5 \oplus j \gg 3]) \oplus$ 0xAA;
  $t'' \leftarrow j + S[j]$;

  $Z_i = (S[t] + S[t']) \oplus S[t'']$;
end

Our Contribution and Organization of the Paper. In this paper we will show that the first output byte produced by RC4+ is negatively biased towards 1. In fact we will prove that the probability that the first output byte is equal to 1 is around $\frac{1}{N} - \frac{1}{2N^2}$, where $N = 256$ is the number of elements of the array S used in the design. Using this observation we will mount a distinguishing attack against RC4+ that requires around 2^{26} output keystreams produced by **(a)** Secret Keys chosen uniformly at random or **(b)** any fixed Secret Key used with IVs chosen unformly at random. In the second part of the paper we revisit the Differential Fault Attack on RC4 proposed by Biham et. al. in FSE 2005 [1]. We explore the possibility of mounting such a fault attack on RC4+. We will show that by injecting around $2^{17.2}$ faults, it is possible to recover the internal state of the cipher efficiently.

2 Distinguishing Attack on RC4+

In this section we will prove that the first output byte Z_1 (when the value of the index $i = 1$) is negatively biased towards 1. We will prove that $\mathsf{Pr}(Z_1 = 1) = \frac{1}{N} - \frac{1}{2N^2}$. The initial state of the RC4+ PRGA is denoted by S_0.

Lemma 1. *Let S_0 be a random permutation on $\{0, 1, 2, \ldots, 255\}$. If $S_0[1] = 1$ and $S_0[2]$ is even, then Z_1 can never take the value 1.*

Proof. We refer to the PRGA algorithm in Table 2. Initially $i = j = 0$. After the increment operations the new values of i, j are as follows: $i = 0 + 1 = 1$ and $j = 0 + S_0[i] = 0 + S_0[1] = 1$. Since $i = j$ even after the increment operations, the subsequent swap operation does not bring about any change in the array S_0. Thereafter the values of t, t', t'' are calculated as follows:

$$t = S_0[i] + S_0[j] = 2 \cdot S_0[1] = 2.$$

$$\begin{aligned} t' &= (S_0[i \gg 3 \oplus j \ll 5] + S_0[i \ll 5 \oplus j \gg 3]) \oplus \texttt{0xAA} \\ &= (S_0[1 \gg 3 \oplus 1 \ll 5] + S_0[1 \ll 5 \oplus 1 \gg 3]) \oplus \texttt{0xAA} \\ &= (2 \cdot S_0[32]) \oplus \texttt{0xAA} \end{aligned}$$

Finally $t'' = j + S_0[j] = 1 + S_0[1] = 1 + 1 = 2$. Therefore we have $Z_1 = (S_0[2] + S_0[t']) \oplus S_0[2]$. Suppose that $Z_1 = 1$, then we will have

$$(S_0[2] + S_0[t']) \oplus S_0[2] = 1 \quad \Rightarrow \quad S_0[2] + S_0[t'] = S_0[2] \oplus 1$$

Since $S_0[2]$ is even, we must have $S_0[2] \oplus 1 = S_0[2] + 1$. Hence the previous equation reduces to:

$$S_0[2] + S_0[t'] = S_0[2] + 1 \quad \Rightarrow \quad S_0[t'] = 1$$

S_0 is a permutation and hence injective. So $S_0[t'] = S_0[1] = 1$ can only imply that $t' = 1$. Thus we have

$$(2 \cdot S_0[32]) \oplus \texttt{0xAA} = 1$$

The LHS of the above equation is clearly an even number whereas the RHS is odd. This gives rise to a contradiction, and therefore $Z_1 = 1$ can clearly not hold. □

Corollary 1. *The above Lemma would still hold if any even pad instead of* `0xAA` *were used in the design.*

Theorem 1. *Let S_0 be a random permutation on $\{0, 1, 2, \ldots, 255\}$. The probability that $Z_1 = 1$ is given by the equation $\mathsf{Pr}(Z_1 = 1) = \frac{1}{N} - \frac{1}{2N^2}$ (where $N = 256$).*

Proof. Let E denote the event: "$S_0[1] = 1$ and $S_0[2]$ is even". Then it is clear that $\mathsf{Pr}[\mathsf{E}] = \frac{\frac{N}{2} \cdot (N-2)!}{N!} \approx \frac{1}{2N}$. From Lemma 1, we have $\mathsf{Pr}[Z_1 = 1|\mathsf{E}] = 0$. By standard randomness assumptions, we have $\mathsf{Pr}[Z_1 = 1|\mathsf{E}^c] = \frac{1}{N}$ (this has been verified by extensive computer experiments with 2^{20} random keys). Therefore we have

$$\begin{aligned} \mathsf{Pr}[Z_1 = 1] &= \mathsf{Pr}[Z_1 = 1|\mathsf{E}] \cdot \mathsf{Pr}[\mathsf{E}] + \mathsf{Pr}[Z_1 = 1|\mathsf{E}^c] \cdot \mathsf{Pr}[\mathsf{E}^c] \\ &= 0 \cdot \frac{1}{2N} + \frac{1}{N} \cdot \left(1 - \frac{1}{2N}\right) = \frac{1}{N} - \frac{1}{2N^2}. \end{aligned}$$

□

We now state the following theorem from [6], which outlines the number of output samples required to distinguish two distributions X and Y.

Theorem 2. *(Mantin-Shamir [6]) Let X, Y be distributions, and suppose that the event e happens in X with probability p and in Y with probability $p(1 + q)$. Then for small p and q, $O\left(\frac{1}{pq^2}\right)$ samples suffice to distinguish X from Y with a constant probability of success.*

Distinguishing RC4+ from Random Sources. Let X be the probability distribution of Z_1 in an ideal random stream, and let Y be the probability distribution of Z_1 in streams produced by RC4+ for randomly chosen keys. Let the event e denote $Z_1 = 1$, which occurs with probability of $\frac{1}{N}$ in X and $\frac{1}{N} - \frac{1}{2N^2} = \frac{1}{N} \cdot \left(1 - \frac{1}{2N}\right)$ in Y. By using the Theorem 2 with $p = \frac{1}{N}$ and $q = -\frac{1}{2N}$, we can conclude that we need about $\frac{1}{pq^2} = 4 \cdot N^3 = 2^{26}$ output samples to reliably distinguish the two distributions.

Experimental Results. By performing extensive computer simulations with **(a)** one billion random keys, and **(b)** a fixed key with one billion random IVs, the probability $\mathsf{Pr}[Z_1 = 1]$ was found to be around $2^{-8} - 2^{-17.03}$. This is consistent with the theoretical value of $\frac{1}{N} - \frac{1}{2N^2}$ proven in Theorem 1.

3 Differential Fault Analysis of RC4+

In [1], a Differential Fault Attack and an Impossible Fault Attack of the RC4 stream cipher was proposed. The Impossible Fault Attack uses random faults on the i or j indices of the RC4 PRGA to drive the cipher into a special state called Finney state [3]. The Finney states are called impossible states because they can not occur under normal mode of operation of RC4 and hence the unusual name of the attack. By injecting around 2^{16} faults on either the i or j register, the cipher is expected to enter a Finney State. From observing the faulty output bytes of RC4 it is possible to assess if the cipher has indeed entered a Finney State. Since any Finney state cycles back after $255 \cdot 256 = 65280$ iterations of the cipher, the attacker selects one of the interleaved cycles in the output stream as the internal state. Once the internal state is obtained at some point in time, it is possible to backtrack and find the initial state at the beginning of the PRGA. Note that, since the PRGA update operations of RC4 and RC4+ are exactly similar, an impossible fault attack on RC4+ may also be carried out using the same techniques outlined in [1].

Applying the Differential Fault Attack (DFA) of [1] to RC4+, however, is not so straightforward. Before proceeding, we note that the PRGA of RC4 is exactly the same as that of RC4+, the only difference being that RC4 outputs $S[t]$ instead of $(S[t] + S[t']) \oplus S[t'']$. We will state in brief the DFA algorithm in [1].

A. Perform a key setup (KSA) with the unknown key and run the RC4 PRGA for around 1000 iterations, and record the output stream Z_i, $(1 \leq i \leq 1000)$ for later analysis.

B. Process the following 256 times with l being set from 0 to 255, giving 256 faulty output streams
   1. Restart the cipher and perform a key setup with the same unknown key.
   2. Make a fault in $S[l]$.
   3. Run the RC4 PRGA 30 steps, and record the faulty output stream $Z_i^1[l]$ for later analysis.

C. Repeat Step **B** with fault injection in k^{th} $(2 \leq k \leq 1000)$ PRGA iteration instead of just after key setup. Record the faulty keystream sequence $Z_i^k[l]$ in each case (thus $Z_i^k[l]$ is the faulty i^{th} keystream byte when the location $S[l]$ has been faulted at PRGA round k).

For any i, the output byte Z_i is a function of just 3 locations of the S array: $i,\ j,\ S[i]+S[j]$. So evidently, the output byte of all the $Z_i^i[l]$'s (note $Z_i^i[l]$ is the first output byte obtained after faulting $S[l]$ at round i), except for three of them, are the same as in the faultless output byte Z_i. The identification of these three streams leak the values of $i,\ j,\ S[i]+S[j]$, but not which is which. Of course, the value of i is always known, thus the only task is to identify which is j and which is $S[i]+S[j]$. After the values of $j,\ S[i]+S[j]$ are obtained for sufficiently many PRGA rounds i, a cascade guessing technique is employed in [1] to eliminate incorrect guesses of j from $j,\ S[i]+S[j]$ and thereafter reconstruct the initial permutation S. For more details, we refer the reader to [1].

However in `RC4+`, the output byte is a function of 7 locations of the S array: $i,\ j,\ S[i]+S[j],\ j+S[j],\ i \gg 3 \oplus j \ll 5,\ i \ll 5 \oplus j \gg 3, (S[i \gg 3 \oplus j \ll 5] + S[i \ll 5 \oplus j \gg 3]) \oplus$ `0xAA`. Therefore repeating the above procedure in the case of `RC4+` would leak a maximum of 7 indices in each round, of which only the value of i is known with certainty. The values of the other 6 indices can not be assigned with certainty. Thus, on the face of it, performing DFA on `RC4+` seems to be more difficult than `RC4`. However as we will see in Section 3.1, this is not so.

3.1 Inferring the Values of j in Each Round

As we have seen, performing steps **A, B, C** for `RC4+`, leaks the values of 6 indices. Although the attacker knows that these are the values of the indices $j,\ S[i]+S[j],\ j+S[j],\ i \gg 3 \oplus j \ll 5,\ i \ll 5 \oplus j \gg 3, (S[i \gg 3 \oplus j \ll 5] + S[i \ll 5 \oplus j \gg 3]) \oplus$ `0xAA`, he is unable to ascertain which of these 6 values correspond to which index. We will later see in Section 3.2, that if the attacker can correctly establish the value of only the index j, it will be enough to reconstruct the permutation S at the beginning of the PRGA. Before we outline our strategy to find the value of j, we will look at a result that will help us build the attack.

Lemma 2. *For any value of i, consider two values $j_1,\ j_2$. If $i \gg 3 \oplus j_1 \ll 5 = i \gg 3 \oplus j_2 \ll 5$, and $i \ll 5 \oplus j_1 \gg 3 = i \ll 5 \oplus j_2 \gg 3$, then $j_1 = j_2$.*

Proof. Rearranging the terms in both equations we get $(j_1 \oplus j_2) \ll 5 = 0 = (j_1 \oplus j_2) \gg 3$. Then, $j_1 \oplus j_2 = 0$ is the only solution to the equation and so $j_1 = j_2$.

Ascertaining j. For any round i, the attacker has with him 6 values corresponding to the indices $j,\ S[i]+S[j],\ j+S[j],\ i \gg 3 \oplus j \ll 5,\ i \ll 5 \oplus j \gg 3, (S[i \gg 3 \oplus j \ll 5] + S[i \ll 5 \oplus j \gg 3]) \oplus$ `0xAA`. Let us call these six values $k_1, k_2, \ldots, k_6$. He of course does not know the correspondence between the

$k_1, \ldots, k_6$ and the indices. Without loss of generality let k_1 be the correct value of j. Then evaluating the functions $i \gg 3 \oplus k_1 \ll 5$ and $i \ll 5 \oplus k_1 \gg 3$ will lead to two of the values in $k_2, k_3, \ldots, k_6$ i.e. those corresponding to $i \gg 3 \oplus j \ll 5$ and $i \ll 5 \oplus j \gg 3$. The probability that any other $k_a,\ 2 \leq a \leq 6$ will on evaluating $i \gg 3 \oplus k_a \ll 5$ and $i \ll 5 \oplus k_a \gg 3$ will lead to two elements of $\{k_1, k_2, \ldots, k_6\}$ is very low. Therefore given any i the strategy will be as follows

- For $a = 1$ to 6

    1. Compute $M_a = i \gg 3 \oplus k_a \ll 5$ and $N_a = i \ll 5 \oplus k_a \gg 3$.
    2. If $M_a, N_a \in \{k_1, k_2, k_3, k_4, k_5, k_6\}$ then $j = k_a$.

The strategy of the attacker will be to determine the values of j for around 602 consecutive values of i. As will be seen in Section 3.2, this will suffice to reconstruct the permutation S at the beginning of the PRGA.

Error Analysis. Lemma 2 guarantees that any value k_a different j, when used to calculate M_a, N_a will result in values $\neq i \gg 3 \oplus j \ll 5$ and $i \ll 5 \oplus j \gg 3$. Therefore, a confusion will only occur when some value $k_a \neq j$ on evaluating $i \gg 3 \oplus k_a \ll 5$ and $i \ll 5 \oplus k_a \gg 3$ also leads to two elements of $\{k_1, k_2, \ldots, k_6\}$ (which are not equal to $i \gg 3 \oplus j \ll 5$ and $i \ll 5 \oplus j \gg 3$). In such an event the attacker must guess one from the multiple values of j extracted by the algorithm. Experiments with 2^{20} random keys show that in the first 602 rounds there are around 5 to 6 confusions on average, and each confusion usually gives no more than 2 values of j to choose from. The attacker can simply guess the values of j during these rounds and use it in the algorithm for state recovery that will be discussed in the next subsection.

Fault Requirement. As we will see in the next subsection, around 602 values of j are required to reconstruct S. Since each round requires 256 faults, the total fault requirement is around $602 \times 256 \approx 2^{17.23}$.

3.2 Reconstructing the Permutation S

We will now present the Algorithm 1 that will be used to reconstruct the state S. The technique used here is similar to the algorithm presented in [2]. The algorithm works under the principle that if j_1, j_2 are the values of j in two successive PRGA rounds then the the value of $S[i_1]$ is given as $j_2 - j_1$.

We assume that the algorithm starts from PRGA round t armed with M values of j in consecutive PRGA rounds. First, a two dimensional array *acc* is used, whose r-th row contains the triplet (i_r, j_r, z_r). After each subsequent round $t + r$, the algorithm reverts to the initial round t and in the process uses new entries to check if the array *guess* (which is the temporary array used to guess the state S) can be populated further. Thereafter the algorithm again performs a *forward pass* up to the round $t + r + 1$ to further populate the array *guess* as much as possible. The strategy is formally presented in Algorithm 1.

Input: $(i_t, j_t), \{(i_{t+r}, j_{t+r}, z_{t+r} : r = 1, \ldots, M-1)\}$.
Output: Permutation array S_{t+m} for some $m \in [0, M-1]$.
0.1 $numKnown \leftarrow 0$;
0.2 $m \leftarrow 0$;
0.3 **for** u *from* 0 *to* $N-1$ **do**
0.4 $\quad guess[u] \leftarrow EMPTY$;
end
0.5 $acc[0][0] \leftarrow i_t$;
0.6 $acc[0][1] \leftarrow j_t$;
0.7 **for** u *from* 1 *to* $M-1$ **do**
0.8 $\quad acc[u][0] \leftarrow i_{t+u}$;
0.9 $\quad acc[u][1] \leftarrow j_{t+u}$;
0.10 $\quad acc[u][2] \leftarrow z_{t+u}$;
end
0.11 **repeat**
0.12 $\quad i_{t+m+1} \leftarrow acc[t+m+1][0]$, $j_{t+m+1} \leftarrow acc[t+m+1][1]$, $z_{t+m+1} \leftarrow acc[t+m+1][2]$;
0.13 $\quad$ **if** $guess[i_{t+m+1}] = EMPTY$ **then**
0.14 $\quad\quad guess[i_{t+m+1}] \leftarrow j_{t+m+1} - j_{t+m}$;
$\quad$ **end**
0.15 $\quad backtrack(t+m, t)$;
0.16 $\quad processForward(t, t+m+1)$;
0.17 $\quad m \leftarrow m+1$;
0.18 $\quad numKnown \leftarrow$ Number of non-empty entries in the array $guess$;
until $numKnown = N-1$ *OR* $m = M-1$;
0.19 **if** $numKnown = N-1$ **then**
0.20 $\quad$ Fill the remaining single EMPTY location of the array $guess$;
0.21 $\quad$ **for** u *from* 0 *to* $N-1$ **do**
0.22 $\quad\quad S_{t+m}[u] \leftarrow guess[u]$;
$\quad$ **end**
end

Algorithm 1. The algorithm for state recovery with backward and forward passes.

Algorithm 1 uses two subroutines. The subroutine $backtrack(r, t)$ presented in Algorithm 2 performs a backward pass, tracing all state information back from the current round r to a previous round $t < r$. On the other hand, the subroutine $processForward(r, t)$, presented in Algorithm 3 evolves the state information in the forward direction from a past round r to the current round $t > r$. Note that Algorithm 1 returns the array S_{t+m} (the value of S at PRGA round $t+m$) where m is the minimal value for which S_{t+m} can be fully constructed. Thereafter the value of S at any previous round can be easily calculated as the state update of RC4+, like RC4, is one-one and invertible.

Subroutine $backtrack(r, t)$
1.1 **repeat**
1.2 $i_r \leftarrow acc[r][0]$;
1.3 $j_r \leftarrow acc[r][1]$;
1.4 $swap(guess[i_r], guess[j_r])$;
1.5 $r \leftarrow r - 1$;
until $r = t$;

Algorithm 2. Subroutine *backtrack*

Experimental Results. We present some experimental evidences. Experimental result showing the average number of bytes recovered (over 100 random simulations) against the number of rounds used is shown in Table 3. It shows that around 602 consecutive values of j are required to reconstruct the entire of S.

Table 3. No. of rounds vs. average no. of bytes recovered for Algorithm 1

Rounds M	100	200	300	400	500	602
#Bytes Recovered	84	144	194	233	249	255

4 Conclusion

The paper presents some weaknesses of the RC4+ stream cipher proposed by Maitra and Paul in Indocrypt 2008. Firstly, a distinguishing attack requiring around 2^{26} output samples is presented, based on the bias of the first output byte. In the second part of the paper, a Differential Fault Attack requiring around $2^{17.2}$ faults is reported against the cipher. The results show that designing reinforcements to strengthen RC4 is not an easy task. It would be worthwhile to discover a design paradigm that not only rids RC4 of its weaknesses but also preserves its innate simplicity.

Subroutine $processForward(r, t)$

2.1 **repeat**

2.2 $i_r = acc[r][0]$;

2.3 $j_r = acc[r][1]$;

2.4 $z_r = acc[r][2]$;

2.5 $t_r = S_r[i_r] + S_r[j_r]$;

2.6 $t'_r = (S_r[i_r \gg 3 \oplus j_r \ll 5] + S_r[i_r \ll 5 \oplus j_r \gg 3]) \oplus 0xAA$;

2.7 $t''_r = j_r + S_r[j_r]$;

2.8 $swap(guess[i_r], guess[j_r])$;

2.9 **if** $\Big(guess[i_r] \neq EMPTY \wedge guess[j_r] \neq EMPTY \wedge guess[t_r] \neq EMPTY \wedge guess[i_r \gg 3 \oplus j_r \ll 5] \neq EMPTY \wedge guess[i_r \ll 5 \oplus j_r \gg 3] \neq \quad EMPTY \wedge guess[t'_r] \neq EMPTY\Big)$ **then**

2.10 **if** $guess[t''_r] = EMPTY$ **then**

2.11 $guess[t''_r] \leftarrow z_r \oplus \big(guess[t_r] + guess[t'_r]\big)$;

end

end

2.12 **if** $\Big(guess[i_r] \neq EMPTY \wedge guess[j_r] \neq EMPTY \wedge guess[t_r] \neq EMPTY \wedge guess[i_r \gg 3 \oplus j_r \ll 5] \neq EMPTY \wedge guess[i_r \ll 5 \oplus j_r \gg 3] \neq \quad EMPTY \wedge guess[t''_r] \neq EMPTY\Big)$ **then**

2.13 **if** $guess[t'_r] = EMPTY$ **then**

2.14 $guess[t'_r] \leftarrow \big(z_r \oplus guess[t''_r]\big) - guess[t_r]$;

end

end

2.15 **if** $\Big(guess[i_r] \neq EMPTY \wedge guess[j_r] \neq EMPTY \wedge guess[i_r \gg 3 \oplus j_r \ll 5] \neq EMPTY \wedge guess[i_r \ll 5 \oplus j_r \gg 3] \neq EMPTY \wedge guess[t'_r] \neq EMPTY \wedge guess[t''_r] \neq EMPTY\Big)$ **then**

2.16 **if** $guess[t_r] = EMPTY$ **then**

2.17 $guess[t_r] \leftarrow \big(z_r \oplus guess[t''_r]\big) - guess[t'_r]$;

end

end

2.18 $r \leftarrow r + 1$;

until $r = t$;

Algorithm 3. Subroutine $processForward$

References

1. Biham, E., Granboulan, L., Nguyen, P.Q.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 359–367. Springer, Heidelberg (2005)
2. Das, A., Maitra, S., Paul, G., Sarkar, S.: Some Combinatorial Results towards State Recovery Attack on RC4. In: Jajodia, S., Mazumdar, C. (eds.) ICISS 2011. LNCS, vol. 7093, pp. 204–214. Springer, Heidelberg (2011)
3. Finney, H.: An RC4 cycle that can't happen. Posting to sci.crypt (September 1994)
4. Gong, G., Gupta, K.C., Hell, M., Nawaz, Y.: Towards a General RC4-Like Keystream Generator. In: Feng, D., Lin, D., Yung, M. (eds.) CISC 2005. LNCS, vol. 3822, pp. 162–174. Springer, Heidelberg (2005)
5. Maitra, S., Paul, G.: Analysis of RC4 and Proposal of Additional Layers for Better Security Margin. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 27–39. Springer, Heidelberg (2008)
6. Mantin, I., Shamir, A.: A Practical Attack on Broadcast RC4. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 152–164. Springer, Heidelberg (2002)
7. Maximov, A.: Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of RC4 Family of Stream Ciphers. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 342–358. Springer, Heidelberg (2005)
8. Maximov, A., Khovratovich, D.: New State Recovery Attack on RC4. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 297–316. Springer, Heidelberg (2008)
9. Nawaz, Y., Gupta, K.C., Gong, G.: A 32-bit RC4-like Keystream Generator. IACR Cryptology ePrint Archive 2005, 175 (2005)
10. Paul, S., Preneel, B.: A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 245–259. Springer, Heidelberg (2004)
11. Paul, S., Preneel, B.: On the (In)security of Stream Ciphers Based on Arrays and Modular Addition. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 69–83. Springer, Heidelberg (2006)
12. Tsunoo, Y., Saito, T., Kubo, H., Shigeri, M., Suzaki, T., Kawabata, T.: The Most Efficient Distinguishing Attack on VMPC and RC4A. In: SKEW (2005), http://www.ecrypt.eu.org/stream/papers.html
13. Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T.: A Distinguishing Attack on a Fast Software-Implemented RC4-Like Stream Cipher. IEEE Transactions on Information Theory 53(9), 3250–3255 (2007)
14. Zoltak, B.: VMPC One-Way Function and Stream Cipher. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 210–225. Springer, Heidelberg (2004)