Some Combinatorial Results towards State Recovery Attack on RC4*

Apurba Das[1], Subhamoy Maitra[1], Goutam Paul[2], and Santanu Sarkar[1]

[1] Applied Statistics Unit, Indian Statistical Institute,
Kolkata 700 108, India
{contactadasbesu,sarkar.santanu.bir}@gmail.com, subho@isical.ac.in

[2] Department of Computer Science and Engineering, Jadavpur University,
Kolkata 700 032, India
goutam.paul@ieee.org

Abstract. A stream cipher has an unobservable internal state that is updated in every step and a keystream output (bit or word) is generated at every state transition. State recovery attack on stream cipher attempts to recover the hidden internal state by observing the keystream. RC4 is a very widely used commercial stream cipher that has a huge internal state. No known state recovery attack on RC4 is feasible in practice and the best so far has a complexity of 2^{241} (Maximov et al., CRYPTO 2008). In this paper, we take a different approach to the problem. RC4 has a secret index j of size one byte. We perform a combinatorial analysis of the complexity of RC4 state recovery under the assumption that the values of j are known for several rounds. This assumption of knowledge of j is reasonable under some attack models, such as fault analysis, cache analysis, side channel attacks etc. Our objective is not to devise an unconditional full state recovery attack on RC4, but to investigate how much information of j leaks how much information of the internal state. In the process, we reveal a nice combinatorial structure of RC4 evolution and establish certain interesting results related to the complexity of state recovery.

Keywords: Cryptanalysis, RC4, State Recovery Attack, Stream Cipher.

1 Introduction

RC4 is one of the most popular stream ciphers with the following structure. It requires an array S of size N (typically, 256), which contains a permutation of the integers $\{0, \ldots, N-1\}$, two indices i, j and the secret key array K. Given a secret key k of l bytes (typically 5 to 32), the array K of size N is such that $K[y] = k[y \bmod l]$ for any y, $0 \le y \le N-1$.

* This paper is based on the M. Tech. (CS) dissertation work of the first author under the supervision of second author at Indian Statistical Institute, Kolkata.

The permutation S is initialized as the identity permutation. Then RC4 proceeds in two phases: the Key Scheduling Algorithm (KSA) and the Pseudo-Random Generation Algorithm (PRGA). The KSA uses the secret key to scramble the permutation and the PRGA uses the scrambled permutation to generate the keystream bytes that are bitwise XOR-ed with the plaintext bytes in the sender end (during encryption) and with the ciphertext bytes at the receiver end (during decryption).

Below we describe the KSA and the PRGA briefly. All additions performed are addition modulo N, unless otherwise specified.

```
KSA
Initialization:
        For $i = 0, \ldots, N-1$
                $S[i] = i$;
        $j = 0$;

Scrambling:
        For $i = 0, \ldots, N-1$
                $j = (j + S[i] + K[i])$;
                Swap($S[i], S[j]$);
```

```
PRGA
Initialization:
        $i = j = 0$;

Keystream Generation Loop:
        $i = i + 1$;
        $j = j + S[i]$;
        Swap($S[i]$, $S[j]$);
        $t = S[i] + S[j]$;
        Output $z = S[t]$;
```

RC4 can be completely broken if one can reconstruct the permutation S^G by observing the keystream output bytes. Such attacks are called *state recovery attacks*.

The RC4 state consists of two 8-bit indices i and j and a permutation of 256 possible 8-bit elements. Thus, the size of the state space is $2^8! \times (2^8)^2 \approx 2^{1700}$, making the exhaustive search completely infeasible.

In [1], it has been estimated that this kind of attack would require around 2^{779} complexity. Later in [6], an improved idea has been presented that estimates a complexity of 2^{731}. A much improved result [3] in this area shows that the permutation can be recovered in around 2^{241} complexity, rendering RC4 insecure when the key length is more than 30 bytes. Fortunately, this result does not affect RC4 for the typical secret key size of 5 to 16 bytes.

In this paper, we revisit the problem of state recovery from a combinatorial view point. We model the problem under different assumptions and investigate how the time complexity of performing full state recovery differs from one model to another.

Let S_t be the permutation, z_t be the keystream output byte and i_t, j_t be the indices after t many rounds of RC4 PRGA, $t \geq 1$. We also denote the initial values of these variables before the PRGA starts by S_0, i_0, j_0 (note that z_0 does not exist).

2 Previous Works on State Recovery

The works [1,4] independently discovered for the first time that a branch and bound strategy reduces the complexity for recovering the internal state much below that of exhaustive search.

The basic idea of [1] is as follows. At any point of time, there are four unknowns, namely, $j_r^G, S_r^G[i_r^G], S_r^G[j_r^G], S_r^{-1}[z_r]$. One can simulate the PRGA and guess these unknown values in order to continue when necessary. The recursion steps backward if a contradiction is reached, due to the previously wrong guesses. If some M (out of N) many permutation entries are a-priori known, the complexity is reduced further. For $N = 256$, the complete attack requires a complexity of around 2^{779}. The time complexity of the attack for various values of N and M are provided in Tables D.1 and D.2 in [2, Appendix D.4].

In [4], the cycle structures in RC4 are analyzed in detail and a "tracking" attack is developed that recovers the RC4 state, if a significant fraction of the full cycle of keystream bits is generated. For example, the state of a 5 bit RC4-like cipher can be obtained from a portion of the keystream using 2^{42} steps, while the nominal key-space of the system is 2^{160}.

The work [5] showed that Knudsen's attack [1] requires 2^{220} search complexity if 112 entries of the permutation are known and presents an improvement whereby state recovery with the same complexity requires prior knowledge of only 73 permutation entries in certain cases.

In [6], an improvement over [1] is presented using a tree representation of RC4. At time-step r, the nodes are distributed at $r + 1$ levels. Nodes at level h, $0 < h \leq r$, refer to the set of all possible positions in S_{r-h}^G where z_r can be found. The nodes are connected by the branches which represent the conditions to pass from one node to another. In order to find the internal state, such a tree of general conditions is searched by hill-climbing strategy. This approach reduces the time complexity of the full RC4 state recovery from 2^{779} to 2^{731}.

The best known result for state recovery appears in [3] that shows that the permutation can be recovered in around 2^{241} complexity. This establishes that RC4 is not secure when the key length is more than 30 bytes (240 bits). The basic idea of cryptanalysis in [3] is as follows. Corresponding to a window of $w + 1$ keystream output bytes, one may assume that all the j^G's are known, i.e., $j_r^G, j_{r+1}^G, \ldots, j_{r+w}^G$ are known. Thus w many $S_r^G[i_r^G]$ will be available from $j_{r+1}^G - j_r^G$. Then w many equations of the form $S_r^{G^{-1}}[z_r] = S_r^G[i_r^G] + S_r^G[j_r^G]$ will be found where each equation contains only two unknowns (instead of four unknowns $j^G, S^G[i^G], S^G[j^G], S^{G^{-1}}[z]$ as in [1]). Some precomputation is performed to identify a certain position in the keystream where the internal state is compliant to a specific pattern. A *d-order pattern* is a tuple $A = \{i, j, U, V\}$, where U and V are two vectors from Z_N^d with pairwise distinct elements. At time step r, the internal state is *compliant with* A if $i_r^G = i$, $j_r^G = j$, and d cells of S_r^G with indices from U have corresponding values from V. A pattern A is called w-generative if for any internal state compliant with A, the next w clockings allow to derive w equations of the form $S_r^{G^{-1}}[z_r] = S_r^G[i_r^G] + S_r^G[j_r^G]$, i.e., if consecutive w values of j^G are known. The strategy is to look for d-order w-generative patterns with small d and large w. Whenever the observed keystream indicates such patterns of the internal state, iterative recovery of the unknowns is done and the window w is dynamically expanded. A general time complexity estimate is performed in [3], and simulation results for scaled-down version of

RC4 (i.e. smaller N) are reported. The authors claim that the success rate of the full attack is at least 98%.

A very recent work [7] revisits the method [3] and presents an iterative probabilistic reconstruction and discusses how one can practically approach the complexity of [3].

3 State Recovery with Known j: Theoretical Analysis

Our initial study on state recovery assumes that the index j is known for each round in the RC4 PRGA. If the index j is known at each round of the PRGA, then the value at updated location i of the S array before the current round is known. Therefore, after the swap operation in the current round of PRGA, the value at updated location j in the S array can be determined with probability 1. But that does not ensure that the value at updated location i would be determined after the swap operation, because of the fact that the value at the updated location j may not be known before the swap operation.

Therefore, the only problem here is to deterministically compute the value of the updated location j before the swap operation. For this, we use an auxiliary integer array *guess* of size N initially marked *EMPTY*. We use this array to simulate the hidden permutation S.

Our goal is to gradually fill the *EMPTY* locations of the array *guess* by the correct values of the permutation S. In the process, we perform swaps in the array *guess* in tandem with the swaps in S so that if the array *guess* becomes completely filled at some round r, then we can obtain $S_r[u]$ directly from the values $guess[u]$ for all u in $[0, N-1]$.

3.1 Without Using the Keystream Bytes

First, we attempt to recover the internal state without using any information about the keystream bytes z_t. Suppose, we observe the evolution of the cipher from round t onwards. At round $t+1$, the value of $S_t[i_{t+1}]$ is known. Therefore, at the end of the $(t+1)$-th round, the value of $S_{t+1}[j_{t+1}]$ will be known deterministically. Then that value will be placed in the array *guess* at location j_{t+1}. Before this update of array *guess*, if the value at location j_{t+1} in *guess* was not *EMPTY*, then that value is to be placed at location i_{t+1} of the array *guess*, otherwise the value at location i_{t+1} of the array *guess* should be updated to *EMPTY*.

If we repeat the above procedure for several rounds, the number of known entries in the array *guess* increases and eventually, at some round $t+m$, we derive $N-1$ entries of S. Since S is a permutation over $\{0, 1, \ldots, N-1\}$, knowledge of the values in any $N-1$ locations reveal the remaining value.

The above discussion is summarized in the form of Algorithm 1.

The complexity of the above algorithm can be expressed in terms of the number m of rounds that needs to be iterated to fill the array *guess*. The following theorem gives the expected value of m.

```
**Input**: $\{(i_{t+r}, j_{t+r}) : r = 0, 1, \ldots, M-1\}$.
**Output**: Permutation array $S_{t+m}$ for some $m \in [0, M-1]$.

1  $numKnown \leftarrow 0$;
2  for $u$ from $0$ to $N-1$ do
3      $guess[u] \leftarrow EMPTY$;
   end
4  $m \leftarrow 0$ ;
5  repeat
6      $guess[i_{t+m+1}] \leftarrow guess[j_{t+m+1}]$;
7      $guess[j_{t+m+1}] \leftarrow j_{t+m+1} - j_{t+m}$;
8      $m \leftarrow m + 1$;
9      $numKnown \leftarrow$Number of non-empty entries in the array $guess$;
   until $numKnown = N-1$ OR $m = M-1$ ;
10 if $numKnown = N-1$ then
11     Fill the remaining single EMPTY location of the array $guess$;
12     for $u$ from $0$ to $N-1$ do
13         $S_{t+m}[u] \leftarrow guess[u]$;
       end
   end
```

Algorithm 1. The algorithm for state recovery when j is known

Theorem 1. *The expected number of rounds of Algorithm 1 to recover S completely is* $N \cdot \sum_{k=2}^{N} \frac{1}{k}$.

Proof. When k entries are filled, the probability that one more entry would be filled in the next step is equal to the probability that the difference in the consecutive j-values (that is a uniformly random number between 0 to $N-1$) computed in Step 7 is distinct from the already present k values. This probability is clearly $p_k = \frac{N-k}{N}$.

Let X_k denote the number of steps required to fill a new entry in *guess*, when k entries of *guess* are filled. So the total number of steps required to fill $N-1$ entries is given by $X = \sum_{k=0}^{N-2} X_k$. Each X_k follows a geometric distribution with probability p_k. Hence, $E(X_k) = \frac{1}{p_k} = \frac{N}{N-k}$. By linearity of expectation,

$$E(X) = \sum_{k=1}^{N-2} E(X_k) = \sum_{k=0}^{N-2} \frac{N}{N-k} = N \cdot \sum_{k=2}^{N} \frac{1}{k}.$$

□

Substituting $N = 256$ in the expression for $E(X)$, we get the theoretical expectation of the number m of rounds required as 1312. If $M < 1312$, then it is expected that we would have a partially recovered state. We experiment by fixing different values of M. For each M, we run RC4 with 100 randomly chosen secret keys and calculate the average number of permutations bytes recovered. The results are presented in Table 1.

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

Rounds M	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300
#Bytes Recovered	84	140	179	203	220	232	240	245	248	251	253	254	255

3.2 Using the Keystream Bytes

In the above strategy, information of the keystream bytes z_t has not been used. Knowledge of keystream is a standard assumption in *known plaintext attack model* of cryptanalysis. If we use the keystream information, it is likely that complete state recovery would be possible in smaller number of rounds.

```
   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 $m \in [0, M-1]$.
 1 $numKnown \leftarrow 0$;
 2 for $u$ from $0$ to $N-1$ do
 3     $guess[u] \leftarrow EMPTY$;
   end
 4 $m \leftarrow 0$;
 5 repeat
 6     $guess[i_{t+m+1}] \leftarrow guess[j_{t+m+1}]$;
 7     $guess[j_{t+m+1}] \leftarrow j_{t+m+1} - j_{t+m}$;
 8     if $(guess[i_{t+m+1}] \neq EMPTY)$ AND
       $guess[guess[i_{t+m+1}] + guess[j_{t+m+1}]] = EMPTY$ then
 9         $guess[guess[i_{t+m+1}] + guess[j_{t+m+1}]] \leftarrow z_{t+m+1}$;
       end
10     if $guess[i_{t+m+1}] = EMPTY$ AND $z_{t+m+1} = guess[v]$ for some $v$ then
11         $guess[i_{t+m+1}] \leftarrow v - guess[j_{t+m+1}]$;
       end
12     $m \leftarrow m + 1$;
13     $numKnown \leftarrow$ Number of non-empty entries in the array $guess$;
   until $numKnown = N - 1$ OR $m = M - 1$ ;
14 if $numKnown = N - 1$ then
15     Fill the remaining single EMPTY location of the array $guess$;
16     for $u$ from $0$ to $N-1$ do
17         $S_{t+m}[u] \leftarrow guess[u]$;
       end
   end
```

Algorithm 2. The algorithm for state recovery when j, z are known

We can use the keystream bytes to recover the state more efficiently in two ways. Assume that at round t, we know i_t, j_t and at round $t+1$, we know

$i_{t+1}, j_{t+1}, z_{t+1}$. First, we update the contents of the locations i_{t+1} and j_{t+1} of *guess*. Since $z_{t+1} = S_{t+1}[S_{t+1}[i_{t+1}] + S_{t+1}[j_{t+1}]]$, we check whether z_{t+1} is already present at some location v in the array *guess* after the update of the locations i_{t+1} and j_{t+1}. If so, then $S_{t+1}[i_{t+1}]$ is found from $v - S_{t+1}[j_{t+1}]$ and is placed in $guess[i_{t+1}]$. If however, z_{t+1} is not present but $S_{t+1}[i_{t+1}]$ is known, then we can fill z_{t+1} in location $guess[i_{t+1}] + guess_{[}j_{t+1}]$ of *guess*. The detailed procedure is explained in Algorithm 2.

The following result gives an estimate of the number of rounds that need to be iterated for full state recovery.

Theorem 2. *For Algorithm 2, let X_k denote the number of additional rounds required to fill the entire guess array, when k locations are already filled. Then*

$$E(X_k) = 1 + (1 - p_k)\Big(q_k E(X_{k+1}) + (1 - q_k)E(X_k)\Big)$$
$$+ p_k\Big(q_{k+1}E(X_{k+2}) + (1 - q_{k+1})E(X_{k+1})\Big),$$

where $p_k = \frac{N-k}{N}$ and $q_k = 2p_k(1 - p_k)$.

Proof. We call that a *success* has occurred in a step of Algorithm 2, if a new entry of *guess* is filled in that step. Note that the Conditions 8 and 10 cannot hold together.

We consider two different cases. When k entries are filled, Step 7 may give a success with probability $p_k = \frac{N-k}{N}$ or a failure with probability $1 - p_k$.

Case I: Failure in Step 7. After a failure in Step 7, which happens with probability $(1-p_k)$, we would have k entries filled. So, the probability that there would be a new success between Steps 8 and 12 is when either Step 9 gives a success (with probability p_k) and Step 11 gives a failure (with probability $1 - p_k$) or vice versa. Hence, after a failure in Step 7, the probability that there would be one more success between Steps 8 and 12 is given by

$$q_k = p_k(1 - p_k) + (1 - p_k)p_k = 2p_k(1 - p_k),$$

and if there is a success, we would have $k + 1$ entries filled. However, if there is a failure between Steps 8 and 12, which happens with probability $1 - q_k$, then after Step 12, we would have k entries filled. Thus, the contribution of this part to $E(X_k)$ is given by

$$(1 - p_k)\Big(q_k E(X_{k+1}) + (1 - q_k)E(X_k)\Big).$$

Case II: Success in Step 7. After a success in Step 7, we have $k + 1$ entries filled. So, the probability that there would be one more success between Steps 8 and 12 is when either Step 9 gives a success (with probability p_{k+1}) and Step 11 gives a failure (with probability $1 - p_{k+1}$) or vice versa. Hence, after a success in Step 7, the probability that there would be one more success between Steps 8 and 12 is given by

$$q_{k+1} = p_{k+1}(1 - p_{k+1}) + (1 - p_{k+1})p_{k+1} = 2p_{k+1}(1 - p_{k+1}),$$

and if there is a success, we would have $K+2$ entries filled. However, if there is a failure between Steps 8 and 12, which happens with probability $1-q_{k+1}$, then after Step 12, we would have $k+1$ entries filled. Hence, the contribution of this part to $E(X_k)$ is given by

$$p_k\Big(q_{k+1}E(X_{k+2})+(1-q_{k+1})E(X_{k+1})\Big).$$

In addition to the above two contributions, we need to add 1 to $E(X_k)$, as we have analyzed the situations after one more additional round. So,

$$\begin{aligned}E(X_k) = 1 &+ (1-p_k)\Big(q_kE(X_{k+1})+(1-q_k)E(X_k)\Big)\\ &+p_k\Big(q_{k+1}E(X_{k+2})+(1-q_{k+1})E(X_{k+1})\Big).\end{aligned}$$

□

Corollary 1. *The expected number of rounds required to completely recover the RC4 state using Algorithm 2 is given by* $E(X_0)$, *where* $E(X_{N-1}) = E(X_N) = 0$.

Experimental results show that the number m of rounds required to fill the array A using the improved algorithm is around 550, which is close to the theoretical value 531 obtained computing $E[X_0]$ as stated in Corollary 1. Table 2 shows the experimental results generated in the same method as in Section 3.1.

Table 2. No. of rounds vs. average no. of bytes recovered for Algorithm 2

Rounds M	100	150	200	250	300	350	400	450	500	550
#Bytes Recovered	112	163	203	229	245	252	255	255.6	255.9	255.99

4 Heuristics for Further Improvement

The number of rounds (which is equal to the number of known j values) required in Algorithm 2 can be reduced further by applying some heuristics that we describe in this section.

In Algorithms 1 and 2, information of the new entries filled at any round r could not be used in any earlier round $t < r$. We introduce a concept of *backward pass* on the auxiliary array *guess*. Suppose the algorithm begin execution from round t. After each subsequent round r, we revert everything back to the initial round t and in the process use the new entries to check if the array *guess* can be populated further. After we reach round t, we again perform a *forward pass* up to the current round r to further populate the array *guess* as much as possible. The improved strategy is presented in Algorithm 3.

Algorithm 3 uses two subroutines. The subroutine $backtrack(r, t)$ presented in Algorithm 4 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 5 evolves the state information in the forward direction from a past round r to the current round $t > r$. Unlike the previous two algorithms, an additional two dimensional array *acc* is used, whose r-th row contains the triplet (i_r, j_r, z_r).

```
    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 $m \in [0, M-1]$.
 1  $numKnown \leftarrow 0$;
 2  $m \leftarrow 0$;
 3  for $u$ from $0$ to $N-1$ do
 4      $guess[u] \leftarrow EMPTY$;
    end
 5  $acc[0][0] \leftarrow i_t$;
 6  $acc[0][1] \leftarrow j_t$;
 7  for $u$ from $1$ to $M-1$ do
 8      $acc[u][0] \leftarrow i_{t+u}$;
 9      $acc[u][1] \leftarrow j_{t+u}$;
10      $acc[u][2] \leftarrow z_{t+u}$;
    end
11  repeat
12      $i_{t+m+1} \leftarrow acc[t+m+1][0]$;
13      $j_{t+m+1} \leftarrow acc[t+m+1][1]$;
14      $z_{t+m+1} \leftarrow acc[t+m+1][2]$;
15      if $guess[i_{t+m+1}] = EMPTY$ then
16          $guess[i_{t+m+1}] \leftarrow j_{t+m+1} - j_{t+m}$;
        end
17      $backtrack(t+m, t)$;
18      $processForward(t, t+m+1)$;
19      $m \leftarrow m+1$;
20      $numKnown \leftarrow$ Number of non-empty entries in the array $guess$;
    until $numKnown = N-1$ OR $m = M-1$ ;
21  if $numKnown = N-1$ then
22      Fill the remaining single EMPTY location of the array $guess$;
23      for $u$ from $0$ to $N-1$ do
24          $S_{t+m}[u] \leftarrow guess[u]$;
        end
    end
```

Algorithm 3. The algorithm for state recovery with backward and forward passes

```
    Subroutine $backtrack(r, t)$
 1  repeat
 2      $i_r \leftarrow acc[r][0]$;
 3      $j_r \leftarrow acc[r][1]$;
 4      $swap(guess[i_r], guess[j_r])$;
 5      $r \leftarrow r-1$;
    until $r = t$ ;
```

Algorithm 4. Subroutine *backtrack*

```
Subroutine $processForward(r, t)$
1  repeat
2      $i_r = acc[r][0]$;
3      $j_r = acc[r][1]$;
4      $z_r = acc[r][2]$;
5      $swap(guess[i_r], guess[j_r])$;
6      if $guess[i_r] \neq EMPTY$ then
7          $temp \leftarrow guess[i_r] + guess[j_r]$;
8          if $guess[temp] = EMPTY$ then
9              $guess[temp] \leftarrow z_r$;
           end
       end
10     if $guess[i_r] = EMPTY\ AND\ z_r = guess[v]$ then
11         $guess[i_r] \leftarrow v - guess[j_r]$;
       end
12     $r \leftarrow r + 1$;
   until $r = t$ ;
```

Algorithm 5. Subroutine *processForward*

4.1 Experimental Results

Theoretical analysis of Algorithm 3 is a challenging task. Since the theoretical analysis is yet open, we present some experimental evidences to support the improvements achieved. Experimental result showing the average number of bytes recovered (over 100 random simulations of RC4) against the number of rounds used is shown in Table 3.

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

Rounds M	100	150	200	250
#Bytes Recovered	146	218	240	255

In Figure 1, we plot the number of $S[i]$'s recovered vs. no. of j's known (on the left) and the ratio of the numbers of $S[i]$'s recovered and j's known (on the right). It is interesting to note that though the number of bytes recovered increases with number of known j's, the relationship between the two is not linear. When a few j's or a lot of j's are known, less number of bytes are recovered, compared to when moderate number (around 128) of j's are known. The reason behind this is as follows. When a few j's are known, the probability that more than one entry would be filled is very low (due to Theorem 2). Also, when many j's are known, most of the entries of *guess* are already filled, so the probability that a new entry computed is different from the already known ones is very low. Therefore, maximum information gain is achieved in between these two extreme cases. From our experiments, we find the maximum gain corresponding to the case when 116 many selected j values are known. A potential future work would be to guess such 116 j values and then devise a strategy to reconstruct the full state.

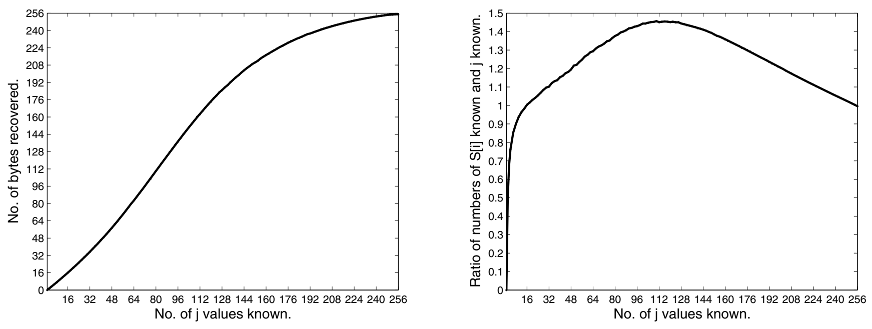

Fig. 1. Relationship between no. of permutation bytes recovered and no. of j's known for Algorithm 3

5 Conclusion

We show how the knowledge of the secret index j leaks information about the internal state of RC4 PRGA. Though our analysis does not immediately lead to a state recovery attack on RC4, it certainly gives insight into the interplay between the state variables and their dependencies. Full state recovery attack on RC4 in practically achievable complexity is still an open problem. Currently the best known state recovery attack requires 2^{241} complexity [3]. We believe our work may be extended further to investigate the possibility of RC4 state recovery in complexity less than 2^{241}.

References

1. Knudsen, L.R., Meier, W., Preneel, B., Rijmen, V., Verdoolaege, S.: Analysis Methods for (Alleged) RC4. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 327–341. Springer, Heidelberg (1998)
2. Mantin, I.: Analysis of the stream cipher RC4. Master's Thesis, The Weizmann Institute of Science, Israel (2001)
3. 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)
4. Mister, S., Tavares, S.E.: Cryptanalysis of RC4-like Ciphers. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 131–143. Springer, Heidelberg (1999)
5. Shiraishi, Y., Ohigashi, T., Morii, M.: An Improved Internal-state Reconstruction Method of a Stream Cipher RC4. In: Hamza, M.H. (ed.) Proceedings of Communication, Network, and Information Security, Track 440-088, New York, USA, December 10-12, pp. 440–488 (2003)
6. Tomasevic, V., Bojanic, S., Nieto-Taladriz, O.: Finding an internal state of RC4 stream cipher. Information Sciences 177, 1715–1727 (2007)
7. Golic, J., Morgari, G.: Iterative Probabilistic Reconstruction of RC4 Internal States. IACR Eprint Server, eprint.iacr.org, number 2008/348 August 8 (2008)