How to Find Short RC4 Colliding Key Pairs

Jiageng Chen * and Atsuko Miyaji**

School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{jg-chen, miyaji}@jaist.ac.jp

Abstract. The property that the stream cipher RC4 can generate the same keystream outputs under two different secret keys has been discovered recently. The principle that how the two different keys can achieve a collision is well known by investigating the key scheduling algorithm of RC4. However, how to find those colliding key pairs is a different story, which has been largely remained unexploited. Previous researches have demonstrated that finding colliding key pairs becomes more difficult as the key size decreases. The main contribution of this paper is proposing an efficient searching algorithm which can successfully find 22-byte colliding key pairs, which are by far the shortest colliding key pairs ever found.

1 Introduction

The stream cipher RC4 is one of the oldest and most wildly used stream ciphers in the world. It has been deployed to innumerable real world applications including Microsoft Office, Secure Socket Layer (SSL), Wired Equivalent Privacy (WEP), etc. Since its debut in 1994 [1], many cryptanalysis works have been done on it, and many weaknesses have been exploited, such as [5] [6] and [7]. However, if RC4 is used in a proper way, it is still considered to be secure. Thus it is still considered to be a high valuable cryptanalysis target both in the industrial and academic world.

In this paper, we focus on exploiting the weakness that RC4 can generate colliding key pairs, namely, two different keys will result in the same keystream output. This weakness was first discovered by [2] and later generalized by [3]. For any ciphers, the first negative effects that this property could bring is the reducing of the key space. It seems that it is not very dangerous if the colliding key pairs are not so many. However, [4] demonstrated a key recovery attack by making use of this weakness, and the complexity of the attack depends heavily on how fast we can find those colliding key pairs. In [2], it has demonstrated that the shorter the key is, the harder it is to find the colliding key pairs. A searching algorithm was proposed in [2] and a 24-byte colliding key pair was the shortest one that experimentally found. Finding short colliding key pairs has its practical

* This author is supported by the Graduate Research Program.
** This work is supported by Grant-in-Aid for Scientific Research (B), 20300003.

meaning mainly because that the key size deployed in most of the applications are short ones which are between 16 bytes to 32 bytes, and also the link between the attacks like [4].

Our main contribution is proposing a searching algorithm that can find short colliding key pairs efficiently. 22-byte colliding key pair is experimentally found by using our algorithm in about three days time while a 24-byte colliding key pair was found in about ten days time in [2]. We also analyze the complexity of both our algorithm and the one in [2] to support our experimental result from a theoretical point of view, so that we can understand the new searching techniques clearly.

This paper is organized as follows. In section 2, we give a short introduction on RC4 algorithm and the details on the key collisions. In section 3, we review the previous searching techniques including brute force searching and the one proposed in [2]. Section 4 covers the new techniques we propose to reduce the searching complexity followed by the new algorithm in section 5. Complexity evaluations are described in section 6.

2 RC4 key collision

First we shortly describe the RC4 algorithm. The internal state of RC4 consists of a permutation S of the numbers $0, ..., N-1$ and two indices $i, j \in \{0, ..., N-1\}$. The index i is determined and known to the public, while j and permutation S remain secret. RC4 consists of two algorithms: The Key Scheduling Algorithm (KSA) and the Pseudo Random Generator Algorithm (PRGA). The KSA generates an initial state from a random key K of k bytes as described in Algorithm 1. It starts with an array $\{0, 1, ..., N-1\}$ where $N = 256$ by default. At the end, we obtain the initial state S_{N-1}.
Once the initial state is created, it is used by PRGA. The purpose of PRGA is to generate a keystream of bytes which will be XORed with the plaintext to generate the ciphertext. PRGA is described in Algorithm 2. Since key collision is only related to KSA algorithm, we will ignore PRGA in the rest of the paper.

Algorithm 1. KSA

1: **for** $i = 0$ **to** $N-1$ **do**
2: $\quad S[i] \leftarrow i$
3: **end for**
4: $j \leftarrow 0$
5: **for** $i = 0$ **to** $N-1$ **do**
6: $\quad j \leftarrow j + S[i] + K[i \bmod l]$
7: $\quad \text{swap}(S[i], S[j])$
8: **end for**

Algorithm 2. PRGA

1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: **loop**
4: $\quad i \leftarrow i + 1$
5: $\quad j \leftarrow j + S[i]$
6: $\quad \text{swap}(S[i], S[j])$
7: keystream byte $z_i = S[S[i] + S[j]]$
8: **end loop**

We focus on the key collision pattern discovered in [2], which can generate shorter colliding key pairs than other patterns discovered in [3]. In [2], it clearly described how two keys K_1 and K_2 with the only one difference $K_2[d] = K_1[d]+1$ can achieve a collision. It traced two KSA procedure and two S-Box states

generated by the two keys, and pointed out how two S-Box states become equal to each other at the end of the KSA. Actually, the essence of the key collisions is only related to some j values at some specific locations. If these conditions once satisfied, a collision is expected. Thus we prefer to use another way to explain the collision by listing all the j conditions. In this way, we only need to exam the behavior of one key, since once the j values generated by this key satisfy all the conditions, then deterministically, there exists another key that they form a colliding key pair. To simplify, we check whether a given key K_1 has a related key K_2 such that $K_2[d] = K_1[d] + 1$ and K_1 and K_2 can achieve a collision. Then all we need is to confirm whether K_1's j behaviors satisfy the conditions in Table 1.

Table 1. j conditions required to achieve a collision

Round	Round Interval	Class 1	Class 2
1	$[0, d+1]$	$j_d = d$, $j_{d+1} = d + k$	$j_{0\sim d-1} \neq d, d+1$
2	$[d+2, d+k]$	$j_{d+k} = d + 2k$	$j_{d+2\sim d+k-1} \neq d + k$
... ...	... ...	... ...	... ...
t	$[d+(t-2)k+1,$ $d+(t-1)k]$	$j_{d+(t-1)k} = d + tk$	$j_{d+(t-2)k+1\sim d+(t-1)k-1} \neq$ $d + (t-1)k$
... ...	... ...	... ...	... ...
$n-1$	$[d+(n-3)k+1,$ $d+(n-2)k]$	$j_{d+(n-2)k} = (d-1) + (n-1)k$	$j_{d+(n-3)k+1\sim d+(n-2)k-1} \neq$ $d + (n-2)k$
n	$[d+(n-2)k+1,$ $d+(n-1)k-1]$	$j_{d+(n-1)k-2} = S^{-1}_{d+(n-1)k-3}[d]$, $j_{d+(n-1)k-1} = d + (n-1)k - 1$	$j_{d+(n-2)k+1\sim d+(n-1)k-3} \neq$ $d + (n-1)k - 1$

The Round column presents the round number in the KSA steps in the Round Interval column. There are $n = \lfloor \frac{256+k-1-d}{k} \rfloor$ rounds, which is also the times that the key difference repeats during KSA. We separate the conditions into two categories, Class 1 and Class 2. From Table 1, you see that the conditions in Class 1 column are computational dominant compared with Class 2. This is because for j at some time to be some exact value, probability will only be 2^{-8} assuming random distribution, while not equal to some exact value in Class 2 has a relatively much higher probability. Also the main point for finding a colliding key pair is how to meet those low probability conditions in Class 1 column. In the rest of the paper, we focus on the Class 1 conditions. When we say a KSA procedure (a trial) under some key K passes round i and fails at round $i+1$, we indicate that all the Class 1 j conditions are satisfied in the previous i rounds and fails at the $i+1$-th round.

3 Known searching techniques

3.1 Brute force search

The most trivial method is to do the brute force search. The attacker simply generates a random secret key K with length k, and runs the KSA to test its random variable j's behavior. If the trial fails, then repeat the procedure until one colliding key pair is found. In [2], it has been demonstrated that for each trial, the successful probability is around $(\frac{1}{256})^{n+2}$. Thus the complexity for the brute force searching is $2^{8(n+2)}$. For 24-byte keys, the complexity is around 2^{96}, and for 22-byte keys which is actually found by us, it is around 2^{104}.

3.2 Matsui's searching algorithm

A searching algorithm is proposed in [2]. Here we make a short introduction on his searching technique which is described in Table 2. It defines a search function with two related keys as input, and output a colliding key pair or fail. When some trial fails to find the colliding key pair, the algorithm does not restart by trying another random related key pair, instead, it modifies the keys as $K_1[x] = K_1[x]+y, K_1[x+1] = K_1[x+1]-y$ for every x and y. Since $j_x = j_x+y$ and $j_{x+1} = j_x + S_x[x+1] + K_1[x]$, thus j_{x+1} after the modification will not be changed. This means that by modifying in this way, the next trial will have a relatively close relation with the previous trial, in other words, if the previous trial before the modification tends to achieve a collision, then the next trial after the modification will also have the tendency. The algorithm recursively calls the function Search(K_1, K_2) until it return a colliding key or fail.

4 New techniques to reduce the searching complexity

In this section, we propose several techniques to reduce the searching complexity so that we can find short colliding keys in practical time.

4.1 Bypassing the first round deterministically

Our first observation is that we can pass the first round. Recall that in the first round, there are two j conditions in Class 1 that we need to satisfy, namely

$$j_d = d \quad \text{and} \quad j_{d+1} = k + d$$

As in [2], the setting of $K[d+1] = k - d - 1$ always meets the condition $j_{d+1} = k+d$ since we have $j_{d+1} = j_d+d+1+K[d+1]$. But still we have another condition $j_d = d$ left in the first round. This condition can be easily satisfied by modifying

$$K[d] = 255 - j_{d-1}$$

at the time when KSA is proceeded at index $d-1$ after the swap. Since $j_d = j_{d-1} + d + K[d] = d$, and by modifying $K[d]$ dynamically when the previous value j_{d-1} is known, $j_d = d$ will always be satisfied. Then we can bypass the first round and reduce the necessary number of rounds to $n-1$.

Table 2. Matsui's Algorithm

Input: Key length k, $d = k - 1$

Output: colliding key pair K_1 and K_2 such that $K_2[d] = K_1[d] + 1$, $K_1[i] = K_2[i]$ if $i \neq d$, $KSA(K_1) = KSA(K_2)$.

1. Generate a random key pair K_1 and K_2 which differs at position d by one.

Set $K_1[d+1] = K_2[d+1] = k - d - 1$.

2. Call function Search(K_1, K_2), if Search(K_1, K_2)=1, collision is found, else goto 1.

Search(K_1, K_2) :

$s = MaxColStep(K_1, K_2)$

If $s = 255$, then return 1.

$MaxS = max_{x,y} MaxColStep(K_1\langle x, y\rangle, K_2\langle x, y\rangle)$

If $Maxs \leq s$, then return 0.

C=0

For all x and y, do the following:

If $MaxColStep(K_1\langle x, y\rangle, K_2\langle x, y\rangle) = MaxS$, call $Search(K_1, K_2)$

$C = C + 1$

If $C = MaxC$, then return 0.

Notations:

$MaxColStep(K_1, K_2)$: The maximal number of S-Box elements that S_1 differs from S_2.

$K\langle x, y\rangle : K[x] = K[x] + y, K[x+1] = K[x+1] - y$, $K[i] = K[i]$ if $i \neq x, x+1$.

4.2 Bypassing the second round with high probability

If we choose the differential key index carefully, we find that the second round can also be skipped with very high probability compared with the uniform distribution. Generally speaking, we would like to choose $d = k-1$ so that in the KSA procedure, the key differential index will be repeated as few times as possible. Actually choosing the d at the indices close to $k-1$ will have the same affect as the last index $k-1$. For example, for key with length 20-24 bytes, setting the key differential at indices $k-1, k-2, k-3, k-4$ will cause the key differential index to be repeated the same times during the KSA. Thus instead of setting $d = k - 1$, let's set

$$d = k - 3$$

so that after d, we have another two key bytes. For the first round and second round, the following two j conditions are necessary to meet:

$$j_{d+1} = j_{k-2} = 2k - 3$$

$$j_{d+k} = j_{2k-3} = 3k - 3$$

and we have

$$j_{2k-3} = j_{k-2} + K[k-1] + \sum_{i=0}^{k-3} K[i] + \sum_{i=k-1}^{2k-3} S_{i-1}[i] \tag{1}$$

$$\overset{P_{2nd}}{=} j_{k-2} + K[k-1] + \sum_{i=0}^{k-3} K[i] + \sum_{i=k-1}^{2k-3} S_{k-2}[i] \tag{2}$$

Thus by modifying

$$K[d+2] = K[k-1] = j_{2k-3} - j_{k-2} - \sum_{i=0}^{k-3} K[i] - \sum_{i=k-1}^{2k-3} S_{k-2}[i]$$

at the time $i = k-2$ after the swap, with probability

$$P_{2nd} = \frac{256-(k-2)}{256} \times \frac{256-(k-3)}{256} \times \cdots \times \frac{256-1}{256} = \prod_{i=1}^{k-2} \frac{256-i}{256}$$

we can pass the second round.

This can be explained as follows. For two fixed j values j_{k-2} in the first round, and j_{2k-3} in the second round, we have equation (1). At the time $i = k-2$ after the swap, we don't know $\sum_{i=k-1}^{2k-3} S_{i-1}[i]$, but we can approximate it by using $\sum_{i=k-1}^{2k-3} S_{k-2}[i]$. The conditions on this approximation is that for $i \in [k-1, 2k-4]$, j does not touch any indices $[i+1, 2k-3]$, which gives us the probability P_{2nd}. Then if we set the $K[d+2]$ as before, with P_{2nd} we can pass the second round. Notice that the reason why we can modify $K[d+2]$ is related to the choice of d. When modifying $K[d+2]$, we don't wish the modification will affect the previous execution, which has been successfully passed. When modifying $K[d+2]$ trying to meet the second round condition, this key byte is used for the first time during KSA, thus we won't have the previous concern. For short keys such as $k = 24$, $P_{2nd} = 0.36$, and for $k = 22$, $P_{2nd} = 0.43$. The successful probability is thus much bigger compared with the uniform probability $2^{-8} = 0.0039$.

4.3 Reducing the complexity in the last round

In the last round, there are two j conditions need to be satisfied, namely,

$$j_{(n-1)k+d-2} = r \text{ such that } S_{(n-1)k+d-3}[r] = d$$

$$j_{(n-1)k+d-1} = d + (n-1)k - 1$$

And from $j_{(n-1)k+d-1} = j_{(n-1)k+d-2} + S_{(n-1)k+d-2}[(n-1)k+d-1] + K[d-1]$, $K[d-1]$ can be decided if $j_{(n-1)k+d-2}$ is fixed to some value. During the KSA

procedure, $j_{(n-1)k+d-2}$ could be touching any indices, but with overwhelming probability, it will touch index d. This is because after step $i = d$, one of the two S-Box differentials will be staying at index d till step $i = (n-1)k + d - 2$ unless it is touched by any j during the steps $[d+1, (n-1)k + d - 3]$. Thus we can assume that

$$j_{(n-1)k+d-2} = d$$

and we can thus modify $K[d-1]$ at step $i = d-1$ before the swap as follows:

$K[d-1] = j_{(n-1)k+d-1} - j_{(n-1)k+d-2} - S_{(n-1)k+d-2}[(n-1)k + d - 1] =$
$(n-1)k + d - 1 - d - (d+1) = (n-1)k - d - 2$

This modification indicates that if some trial meets the $j_{(n-1)k+d-2} = d$ condition in the last round, then with probability 1, the other condition in this round on $j_{(n-1)k+d-1}$ will be satisfied. Simply speaking, 2^{16} computation cost is required to pass the final round, while we reduce it to

$$P_{last} = 2^8 \times (\frac{255}{256})^{-((n-1)k-3)}$$

For a 24-byte key, the computation cost can be reduced to around $2^{9.2}$, which is a significant improvement. The overall cost will be covered in the next section, here we just demonstrate to give a intuition.

4.4 Multi-key modification

In the area of finding hash collisions, multi-message modification is a widely used technique that first proposed by [8]. MD5 and some other hash functions are broken by using this technique. The idea is that when modifying the message block at some later round i to satisfy the i-th round conditions, leaving the previous rounds conditions satisfied (In hash functions, a message block is usually processed for many rounds in different orders). Since finding the key collision of RC4, to some degree, is related to finding hash collisions, we are motivated by the multi-message modification technique and find that we can also do such efficient modifications in finding RC4 colliding key pairs. Thus we call it multi-key modification.

After adapting previous proposed techniques, we may easily bypass the previous two rounds. Start from the third round, however, all the key bytes have been used more than once. This means that modifying any key bytes will definitely affect the previous rounds, which could make the previous round conditions become unsatisfied. In case of RC4, due to its property, we can to some degree maintain the previous round conditions while modifying the key in any later round. Let's assume for some round $2 < t < n-1$ for the easy demonstration, the t-th round conditions are not satisfied, namely, $j_{(t-1)k+d} \neq tk + d$, and all the previous rounds conditions are satisfied. The following equations should all

be satisfied in order to pass the first t rounds.

$$j_{2k+d} = j_{k+d} + \sum_{j=0}^{k-1} K[j] + \sum_{j=k+d+1}^{2k+d} S_{j-1}[j] \tag{3}$$

$$j_{3k+d} = j_{2k+d} + \sum_{j=0}^{k-1} K[j] + \sum_{j=2k+d+1}^{3k+d} S_{j-1}[j] \tag{4}$$

....

$$j_{(t-1)k+d} = j_{(t-2)k+d} + \sum_{j=0}^{k-1} K[j] + \sum_{j=(t-2)k+d+1}^{(t-1)k+d} S_{j-1}[j] \tag{5}$$

There are four parts in each of these equations, and when the trial fails to pass the round t, (5) does not hold while all the previous equations hold. From the satisfied equations, the sum of the secret key is fixed, and when modifying the secret key in round t, we should not change the sum $\sum_{j=0}^{k-1} K[j]$, otherwise the previous equations will not be satisfied anymore. Then our problem now becomes how to modify K to satisfy condition on $j_{(t-1)k+d}$. There are many ways to modify the secret key without changing the sum. Matsui's algorithm actually uses one of the ways, namely, $K[x] = K[x] + y$ and $K[x+1] = K[x+1] - y$. Setting the modification targets next to each other reduce the steps that different j values will change the previous correct S-Box sum. Matsui's algorithm tries this modification for every x and y ($x \in [0, k-2]$, $y \in [0, 255]$) one by one, hoping that for some x and y, the S-Box sum $\sum_{j=(t-2)k+d+1}^{(t-1)k+d} S_{j-1}[j]$ will be the correct one so that condition on $j_{(t-1)k+d}$ is satisfied, while leaving the all the previous S-Box sum satisfied. We point out that modifying the secret key in this way have some drawbacks. First, only some specific x and y values will satisfy the condition on $j_{(t-1)k+d}$ leaving the previous conditions satisfied, while most of the other modifications will fail. In other words, for passing round t, this modification can be seen as brute force search (but its effect on the previous rounds is less than brute force search, we will cover it in the complexity evaluation). Second, as also mentioned in [2], such modification will generate many duplicated searching paths. Especially, since it is a recursive algorithm, one duplication in the small depth of the tree will cause a considerable amount of computation waste.

We discover that by adding a strategy on x and y in the key modification instead of brute force search, we could overcome the previous two drawbacks. Let's again consider the trial that passes all the previous $t-1$ rounds and fails to pass the t-th round, where we assume $2 < t < n-2$. Let's run the KSA until step $i = (t-1)k + d - 1$ after the swap, then we check if the Class 1 j conditions on round t is satisfied or not, namely whether

$$S_{(t-1)k+d-1}[(t-1)k+d] = j_{(t-1)k+d} - j_{(t-2)k+d} - \sum_{j=(t-2)k+d+1}^{(t-1)k+d-1} S_{j-1}[j] - \sum_{j=0}^{k-1} K[j]$$

If the equation holds, we pass the t-th round and proceed the next round. Otherwise, let's denote

$$\Delta_{(t-1)k+d} = j_{(t-1)k+d} - j_{(t-2)k+d} - \sum_{j=(t-2)k+d+1}^{(t-1)k+d-1} S_{j-1}[j] - \sum_{j=0}^{k-1} K[j]$$

And we wish the value $\Delta_{(t-1)k+d}$ could be at index $(t-1)k+d$ before i touches it. We check if $\Delta_{(t-1)k+d} \leq (t-2)k+d$. If this is the case, it means that we have available S-Box value that can be swapped here. In other words, modify the key as follows:

$$K[\Delta_{(t-1)k+d}] = K[\Delta_{(t-1)k+d}] + (t-1)k + d - j_{\Delta_{(t-1)k+d}}$$
$$K[\Delta_{(t-1)k+d}+1] = K[\Delta_{(t-1)k+d}+1] - (t-1)k - d + j_{\Delta_{(t-1)k+d}}$$

We can store all the previous j values so that $j_{\Delta_{(t-1)k+d}}$ is available when we need it for the key modification. If $\Delta_{(t-1)k+d} > (t-2)k+d$, it means that no matter how we modify the key, we can not pass the i-th round by changing $S[(t-1)k+d]$. In this case, we go back one step to test if $S_{(t-1)k+d-2}[(t-1)k+d-1]$ is the correct one assuming $S_{(t-1)k+d-2}[(t-1)k+d] = S_{(t-1)k+d-1}[(t-1)k+d]$. Keep testing until $i = (t-2)k+d+1$. Now modifying the key becomes target oriented instead of brute searching all x and y, and thus duplicated searches can be greatly reduced. And another big advantage is that once the modification succeeds, we pass the t-th round, while in [2], after the modification assures the passing of the previous $t-1$ rounds, we need to pass the t-th round in a random way.

4.5 New Searching Algorithm

All the techniques described previously compose our new searching algorithm, which is summarized in Table 3. It is a recursive algorithm with recursive depth set to be n, which is the maximum rounds. If the newsearch function returns the maximum rounds, then it indicates that a collision is found. Note that when implementing, it can be further optimized by combining Matsui's algorithm and our new proposed one to proceed part of the rounds accordingly, so that a better performance could be achieved. For the simplicity, we just describe the most straightforward way in Table 3.

5 Complexity Evaluation

5.1 Complexity for our proposed algorithm

We will see from a theoretical point of view, how efficiently our proposed algorithm can perform. We start by giving the following theorem which is important to compute the complexity, and show the proof.

Table 3. Proposed Searching Algorithm

Input: Key length k, different index $d = k - 3$, $n = \lfloor \frac{256+k-1-d}{k} \rfloor$

Output: K_1 and K_2 such that $K_2[d] = K_1[d] + 1$, $K_1[i] = K_2[i]$ if $i \neq d$, $KSA(K_1) = KSA(K_2)$

1. Store the following j^* values in the table, which are the conditions needed to be satisfied. $j_d^* = d, j_{d+1}^* = k + d, j_i^* = i + k$ for $i \in \{d + k, ..., d + k(n - 2)\}$, $j_{d-2+k(n-1)}^* = d$, $j_{d-1+k(n-1)}^* = d - 1 + k(n - 1)$. (Class 1 j conditions)
2. Randomly generate a key K_1 with key length k. Modify
   $K_1[d - 1] = (n - 1)k - d - 2$, $K_1[d + 1] = k - d - 1$.
   Set $K_2 = K_1$ and $K_2[d] = K_1[d] + 1$.
3. Run the KSA until $i = d - 1$ after the swap. Modify $K_1[d] = 256 - j_{d-1}$, and $K_2[d] = K_1[d] + 1$.
4. Keep running the KSA until $i = d + 1$ after the swap. Modify
   $K_1[d + 2] = j_{2k-3}^* - j_{k-2}^* - \sum_{i=0}^{k-3} K_1[i] - \sum_{i=k-1}^{2k-3} S_{1,k-2}[i]$
   $K_2[d + 2] = j_{2k-3}^* - j_{k-2}^* - \sum_{i=0}^{k-3} K_2[i] - \sum_{i=k-1}^{2k-3} S_{2,k-2}[i]$
5. Set the recursive depth variable $R = 0$.
6. If newsearch(K_1, K_2)=n
   Colliding key pair found. Output K_1 and K_2.
   else goto 2.

newsearch(K_1, K_2):

If $Round(K_1, K_2) = n$
  then return n.
$MaxR = Round(K_1, K_2) = t - 1$, set $r = (t - 1)k + d$
while $r > (t - 2)k + d$
  set $\Delta_r = j_{(t-1)k+d} - j_{(t-2)k+d} - \sum_{j=r-k+1}^{r-1} S_{j-1}[j] - \sum_{j=0}^{k-1} K[j]$.
  If $\Delta_r \leq (t - 2)k + d$
    modify the key as follows:
    $K_1[\Delta_r] = K_1[\Delta_r] + r - j_{\Delta_r}$ $K_1[\Delta_r + 1] = K_1[\Delta_r + 1] - r + j_{\Delta_r}$
    $K_2[\Delta_r] = K_2[\Delta_r] + r - j_{\Delta_r}$ $K_2[\Delta_r + 1] = K_2[\Delta_r + 1] - r + j_{\Delta_r}$
    If $Round(K_1, K_2) \leq MaxR$ or $R = n$
      return $Round(K_1, K_2)$.
    Else $R = R + 1$, **newsearch(K_1, K_2)**
  $r = r - 1$

Notation

$Round(K_1, K_2)$: The number of rounds that a key pair K_1, K_2 can pass. In other words, key pair K_1 and K_2 satisfy all the j conditions in the first $Round(K_1, K_2)$ rounds.

Theorem 1. *Define $Pr_{t,(x,y)}$ be the probability for a trial that passes round t ($t > 2$) by modifying the secret key as $K[x] = K[x] + y, K[x+1] = K[x+1] - y$ according to the multi-key modification given the previous trial fails to pass the t-th round. Then*

$$Pr_{t,(x,y)} \approx \sum_{i=1}^{t} \Bigg((\frac{(t-1)k-2}{256}) \times \prod_{j=0}^{i-2} (\frac{256-(t-j)k+x+3}{256})^4 \times \frac{(t-i+1)k-x-3}{256} \times \sum_{j=0}^{3} (\frac{256-(t-i+1)k+x+3}{256}) \Bigg) + \frac{256-(t-1)k+2}{256}$$

Proof. Now let's consider some trial that passes all the first $t-1$ rounds and fails to pass the t-th round. Then we modify the secret key at indices x and $x+1$ with value difference y so that $K[x] = K[x] + y, K[x+1] = K[x+1] - y$. Let's denote $j'_{s,x}, j'_{s,x+1}$ and $j_{s,x}, j_{s,x+1}$ be the j values for the current trial and the trial after the key modification at the modified key indices at round s. It is easy to see that for each such key modification, the change of the 4 j values at each round will cause 4 S-Box values to be changed.

For the trial before the key modification, the successful pass of the first $t-1$ rounds indicates the correct S-Box sum for some fixed key sum $\sum_{i=0}^{k-1} K[i]$. Since our modification doesn't change the key sum, thus, after the key modification, the previous correct S-Box sum should still be satisfied in order to have a chance to pass the t-th round. Otherwise, the key modification will only cause a failure at an rather early round. For example if the previous trial passes the first $t-1$ rounds, for the key modification in round $s \leq t-1$ (assuming this key modification passes all the previous s-1 rounds), the S-Box sum $\sum_{i=x+(s-1)k}^{(t-1)k-1} S_{i-1}[i]$ should not be violated by the 4 changed j values $j'_{s,x}, j'_{s,x+1}$ and $j_{s,x}, j_{s,x+1}$.

First let's consider the probability that due to the key modification that the previous correct S-Box sum is violated. The modification is processed in the same order as the KSA procedure. And notice that in each round, due to the key modification, we have 4 changed j values, and they are checked in the sequence $j'_{s,x}, j_{s,x}, j'_{s,x+1}, j_{s,x+1}$ whether the failure conditions are satisfied. Notice that due to the use of the multi-key modification technique, the S-Box sum in the t-th round can not be touched since we have already precomputed the sum and are expecting the corresponding swap. The following events define the the S-Box intervals that once touched, the previous correct S-Box sums will be violated due to the modification in round s.

- $A_s : j'_{s,x} \in [x + (s-1)k, tk - 3]$ (the original $j'_{s,x}$ violates the S-Box sum $\sum_{i=x+(s-1)k}^{tk-3} S_{i-1}[i]$)
- $B_s : j_{s,x} \in [x + (s-1)k, tk - 3]$ (the newly modified $j'_{s,x}$ violates the S-Box sum $\sum_{i=x+(s-1)k}^{tk-3} S_{i-1}[i]$))

– $C_s : j'_{s,x+1} \in [x + (s-1)k + 1, tk - 3]$ (the original $j'_{s,x+1}$ violates the S-Box sum $\sum_{i=x+(s-1)k+1}^{tk-3} S_{i-1}[i]$)
– $D_s : j_{s,x+1} \in [x + (s-1)k + 1, tk - 3]$ (the newly modified $j_{s,x+1}$ violates the S-Box sum $\sum_{i=x+(s-1)k+1}^{tk-3} S_{i-1}[i]$)

Denote $Pr(S_s)$ to be the probability that the modification in round s will not break the Class 1 j conditions that have been satisfied in the previous trial.

$$\begin{aligned} Pr(S_s) &= (1 - Pr(A_s)) \cdot (1 - Pr(B_s)) \cdot (1 - Pr(C_s)) \cdot (1 - Pr(D_s)) \\ &= Pr(\bar{A}_s) \cdot Pr(\bar{B}_s) \cdot Pr(\bar{C}_s) \cdot Pr(\bar{D}_s) \end{aligned}$$

Denote $Pr(F_s)$ to be the probability that the modification in round s will break the Class 1 j conditions that have been satisfied in the previous trial so that the current trial fails to pass round t.

$$Pr(F_s) = 1 - Pr(S_s)$$

The exact values for the four events can be computed as follows for $s > 2$:

$$Pr(A_s) = Pr(B_s) = \frac{(t - s + 1) * k - x - 2}{256}$$
$$Pr(C_s) = Pr(D_s) = \frac{(t - s + 1) * k - x - 3}{256}$$

Recall that the multi-key modification may fail because no available S-Box element can be swapped to the corresponding location in the t-th round. We approximate this probability to be

$$Pr(F_{multi}) \approx \frac{256 - (t-1)k + 2}{256}$$

And the probability that we successfully find a candidate for the multi-key modification is

$$Pr(S_{multi}) \approx \frac{(t-1)k - 2}{256}$$

Then the total probability that after the key modification the trial fails to pass the t rounds can be computed as follows:

$$\begin{aligned} Pr(F) = Pr(F_{multi}) + Pr(S_{multi}) \cdot Pr(F_1) + Pr(S_{multi}) \cdot Pr(S_1) \cdot Pr(F_2) + \\ \cdots + Pr(S_{multi}) \prod_{i=1}^{t-1} Pr(S_i) \cdot Pr(F_t) \end{aligned}$$

Thus the probability that for some key modification succeeds to pass the t-th round while the trial before the modification passes the previous $t-1$ rounds is

$$Pr_{t,(x,y)} = 1 - Pr(F)$$

After replacing with detailed parameters we complete our proof.

Then the complexity can be derived by the Theorem 2.

Theorem 2. *The complexity to find a colliding key pair for secret key with key length k is*

$$Comp_{new} \approx Pr^{-1}_{n,(\bar{x},\bar{y})}$$

where $Pr_{n,(\bar{x},\bar{y})}$ *is the average case on all possible x and y, and* $n = \lfloor \frac{256+k-1-d}{k} \rfloor$, $d = k - 3$.

To find 22-byte and 24-byte colliding key pairs, the complexity is around 2^{45} and 2^{40}.

5.2 Complexity for Matsui's Algorithm

In [2], a searching algorithm was proposed without giving the complexity evaluation. In order to compare the efficiency, we also give the complexity evaluation for algorithm proposed in [2]. Since Matsui's algorithm is also a recursive based algorithm, we can use a similar way as previous to analyze. We point out the different points here.

Without using the multi-key modification technique that chooses the target position to modify the key, it tries all the values for x and y, thus the S-Box in the t-th round can be touched. Also they set key difference at $d = k - 1$. We can redefine the following events that for the changed j value violating the S-Box sum.

- $A^M_s : j'_{s,x} \in [x + (s-1)k, (t-1)k - 1]$
- $B^M_s : j_{s,x} \in [x + (s-1)k, (t-1)k - 1]$
- $C^M_s : j'_{s,x+1} \in [x + (s-1)k + 1, (t-1)k - 1]$
- $D^M_s : j_{s,x+1} \in [x + (s-1)k + 1, (t-1)k - 1]$

Since there is no concern for the multi-key modification failure , $Pr(F^M)$ can be denoted as

$$Pr(F^M) = Pr(F^M_1) + Pr(S^M_1) \cdot Pr(F^M_2) + \cdots + \prod_{i=1}^{t-2} Pr(S^M_i) Pr(F^M_{t-1})$$

where

$$\begin{aligned} Pr(S^M_s) &= (1 - Pr(A^M_s)) \cdot (1 - Pr(B^M_s)) \cdot (1 - Pr(C^M_s)) \cdot (1 - Pr(D^M_s)) \\ &= Pr(\bar{A^M_s}) \cdot Pr(\bar{B^M_s}) \cdot Pr(\bar{C^M_s}) \cdot Pr(\bar{D^M_s}) \end{aligned}$$

and

$$Pr(F^M_s) = 1 - Pr(S^M_s)$$

Also another big difference is that the modification of the key cannot guarantee the passing of the t-th round. Thus we have to assume the t-th round Class 1 j conditions will be satisfied randomly, namely,

$$Pr^M_{t,(x,y)} = (1 - Pr(F^M)) \times 2^{-8 \cdot (1 + \lfloor \frac{t}{n} \rfloor)}$$

This is because for any rounds except the last round, we have one j condition to satisfy, and we have two in the last round. Then we have the following theorems.

Theorem 3. *Define $Pr^{M}_{t,(x,y)}$ be the probability for a trial that passes round t by modifying the secret key as $K[x] = K[x]+y, K[x+1] = K[x+1]-y$ according to the Matsui's algorithm given the previous trial fails to pass the t-th round. Then we have*

$$Pr^{M}_{t,(x,y)} \approx \left(1 - \frac{(t-1)k - x}{256} \times \sum_{i=0}^{3}(\frac{256-(t-1)k+x}{256})^i - \sum_{i=2}^{t-1}(\frac{(t-j)k-x}{256} \times \prod_{j=1}^{i-1}(\frac{256-(t-j)k+x}{256})^4 \times \sum_{j=0}^{3}(\frac{256-(t-i)k+x}{256})^j)\right) \times 2^{-8(1+\lfloor \frac{t}{n} \rfloor)}$$

Theorem 4. *The complexity of Matsui's algorithm to find a colliding key pair for secret key with key length k is*

$$Comp_{matsui} = (Pr^{M}_{n,(\bar{x},\bar{y})})^{-1}$$

where $Pr_{n,(\bar{x},\bar{y})}$ is the average case on all possible x and y, and $n = \lfloor \frac{256+k-1-d}{k} \rfloor$.

As a result, the complexity for finding 24-byte colliding key pair is around 2^{48} and 2^{53} for 22-byte keys. The following figure shows the complexity to search for different colliding key pairs using two different algorithms.

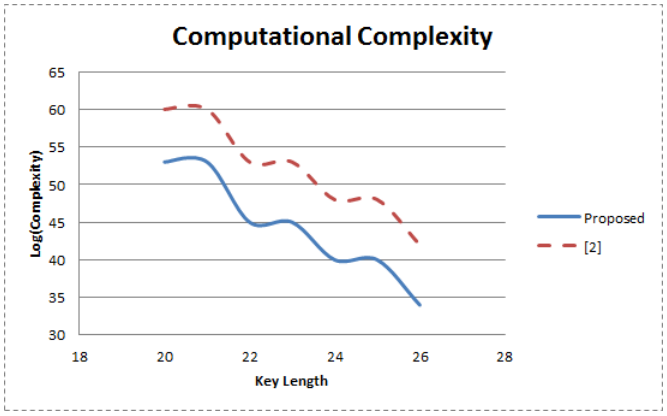

Fig. 1. Computational Complexity

We run the experiment under our proposed algorithm and successfully find by far the shortest 22-byte colliding key pair in about three days computational time by using parallel computer Cray XT5 (Quad-Core AMD Opteron 2.4GHz, 10 cores are used). In case of [2], around 10 days computational time and multiple cpus were used (the detailed information was not published) to find a 24-byte

colliding key pair. Also, our proposed algorithm has a better efficiency searching for other short colliding keys which seems difficult to find by using the algorithm in [2]. Here is the concrete 22-byte colliding key pair found by us in hexadecimal form:

$\mathbf{K_1}(\mathbf{K_2})$: A2 27 43 A7 03 94 2F 17 75 BB A7 27 8F DD 3E 7B C6 A1 C7 **81(82)** 02 5A

6 Conclusion

In this paper, we investigate how to find RC4 colliding key pairs efficiently. We propose several techniques that can be used to bypass several rounds faster than brute force search, and the multi-key modification technique allows us to further increase the searching efficiency without drawback of duplicate searching which is the problem in [2]. We also give the complexity evaluation for both our proposed algorithm as well as the one in [2]. And finally by showing by far the shortest 22-byte colliding key pair ever found, we confirm that our algorithm does work efficiently as expected.

Acknowledgements

This work is inspired by the previous work of Mitsuru Matsui, and the authors wish to thank him for his invaluable comments. Also, the authors wish to thank all the anonymous reviewers for their useful suggestions to help to improve this paper.

References

1. Anonymous: RC4 Source Code. CypherPunks mailing list (September 9, 1994), http://cypherpunks.venona.com/date/1994/09/msg00304.html, http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca0
2. Matsui, M.: Key Collisions of the RC4 Stream Cipher. In: Dunkelman, O., Preneel, B. (eds.) FSE 2009. LNCS, vol. 5665, pp. 1.24. Springer, Heidelberg (2009)
3. Chen, J., Miyaji, A.: Generalized RC4 Key Collisions and Hash Collisions. In: J.A.Garay., R.De Prisco (eds.): SCN 2010. LNCS, vol. 6280, pp.73-87, Springer, Heidelberg (2010)
4. Chen, J., Miyaji, A.: A New Practical Key Recovery Attack on the Stream Cipher RC4 Under Related-Key Model. In: et al. (eds.): Inscrypt 2010, LNCS, vol. 6584, pp.62-76, Springer, Heidelberg (2011).
5. Sepehrdad, P., Vaudenay, S., Vuagnoux, M: Statistical Attack on RC4. In: Paterson, K. (eds.): Eurocrypt 2011. LNCS, vol. 6632, pp.343-363, Springer, Heidelberg (2011)
6. Sepehrdad, P., Vaudenay, S., Vuagnoux, M: Discovery and Exploitation of New Biases in RC4. In: Biryukov, A., Gong, G., Stinson, D.(eds.): SAC2010. LNCS, vol. 6544, pp.74-91, Springer, Heidelberg (2011)
7. Subhamoy , M., Goutam , P., Sourav , S: Attack on Broadcast RC4 Revisited . In: .(eds.): FSE2011. LNCS, vol. 6733, pp. 199-217, Springer, Heidelberg (2011).

8. Wang, X., and Yu, H. How to break MD5 and other hash functions. In: Advances in Cryptology - EUROCRYPT 2005, LNCS, vol.3494, pp. 19-35, Springer, Heidelberg (2005)