Efficient Reconstruction of RC4 Keys from Internal States[⋆]

Eli Biham and Yaniv Carmeli

Computer Science Department
Technion – Israel Institute of Technology
Haifa 3200, Israel
{biham,yanivca}@cs.technion.ac.il
http://www.cs.technion.ac.il/∼biham/
http://www.cs.technion.ac.il/∼yanivca/

Abstract. In this paper we present an efficient algorithm for the retrieval of the RC4 secret key, given an internal state. This algorithm is several orders of magnitude faster than previously published algorithms. In the case of a 40-bit key, it takes only about 0.02 seconds to retrieve the key, with success probability of 86.4%. Even if the algorithm cannot retrieve the entire key, it can retrieve partial information about the key. The key can also be retrieved if some of the bytes of the initial permutation are incorrect or missing.

Keywords: Cryptanalysis, Initial Permutation, Key Scheduling, RC4, Stream Cipher.

1 Introduction

The stream cipher RC4 was designed by Ron Rivest, and was first introduced in 1987 as a proprietary software of RSA DSI. The details remained secret until 1994, when they were anonymously published on an internet newsgroup [1]. RSA DSI did not confirm that the published algorithm is in fact the RC4 algorithm, but experimental tests showed that it produces the same outputs as the RC4 software.

More than twenty years after its release, RC4 is still the most widely used software stream cipher in the world. Among other uses, it is used to protect internet traffic as part of the SSL (Secure Socket Layer) and TLS (Transport Layer Security [3]) protocols, and to protect wireless networks as part of the WEP (Wired Equivalent Privacy) and WPA (Wi-Fi Protected Access) protocols.

The state of RC4 consists of a permutation S of the numbers $0, \ldots, N-1$, and two indices $i, j \in \{0, \ldots, N-1\}$, where $N = 256$. RC4 is comprised of two algorithms: the Key Scheduling Algorithm (KSA), which uses the secret key to create a pseudo-random initial state, and the Pseudo Random Generation Algorithm (PRGA), which generates the pseudo-random stream.

[⋆] This work was supported in part by the Israel MOD Research and Technology Unit.

1.1 Previous Attacks

Most attacks on RC4 can be categorized as distinguishing attacks or key-retrieval attacks. Distinguishing attacks try to distinguish between an output stream of RC4 and a random stream, and are usually based on weaknesses of the PRGA. Key recovery attacks recover the secret key, and are usually based on weaknesses of the KSA.

In 1994, immediately after the RC4 algorithm was leaked, Finney [4] showed a class of states that RC4 can never enter. This class consists of states satisfying $j = i+1$ and $S[i+1] = 1$. RC4 preserves the class of Finney states by transferring Finney states to Finney states, and non-Finney states to non-Finney states. Since the initial state (the output of the KSA) is not a Finney state (in the initial state $i = j = 0$) then RC4 can never enter these states. Biham et. al. [2] show how to use Finney states with fault analysis in order to attack RC4.

Knudsen et al. [11] use a backtracking algorithm to mount a known plaintext attack on RC4. They guess the values of the internal state, and simulate the generation process. Whenever the output doesn't agree with the real output, they backtrack and guess another value.

Golić [7] describes a linear statistical weakness of RC4 caused by a positive correlation between the second binary derivative of the least significant bit and 1, and uses it to mount a distinguishing attack.

Fluhrer and McGrew [6] show a correlation between consecutive output bytes, and introduce the notion of k-fortuitous states (classes of states defined by the values of i, j, and only k permutation values, which can predict the outputs of the next k iterations of the PRGA), and build a distinguisher based on that correlation.

Mantin and Shamir generalize the notion of fortuitous states and define b-predictive k-states (states with k known permutation values which predict only b output words, for $b \leq k$) and k-profitable states, which are classes of states in which the index j behaves in the same way for k steps of the PRGA. The predictive states cause certain output sequences to appear more often than expected in a random sequence, thus they are helpful in mounting a distinguishing attack on RC4.

Mantin and Shamir [14] also show that the second word of the output is slightly more probable to be 0 than any other value. Using this bias they are able to build a prefix distinguisher for RC4, based on only about N short streams.

In 2005 Mantin [13] observed that some fortuitous states return to their initial state after the index i leaves them. These states have a chance to remain the same even after a full cycle of N steps, and the same output of the state may be observed again. Mantin uses these states to predict, with high probability, future output bytes of the stream.

In practical applications, stream ciphers are used with a session key which is derived from a shared secret key and an Initial Value (IV, which is transmitted unencrypted). The derivation of the session key can be done in various ways such as concatenating, XORing, or hashing (in WEP, for instance, the secret key is concatenated after the IV).

Many works try to exploit weaknesses of a specific method for deriving the session key. Fluhrer, Mantin, and Shamir [5] have shown a chosen IV attack on the case where the IV precedes the secret key. Using the first output bytes of $60l$ chosen IVs (l is the length of the secret key), they recover the secret key with high probability. They also describe an attack on the case where the IV follows the secret key, which reveals significant information about the internal state just after l steps of the KSA, thus reducing the cost of exhaustive search significantly.

In March 2007, Klein [10] (followed by Tews et. al. [17]) showed a statistical correlation between any output byte and the value of $S[j]$ at the time of the output generation. They use this correlation to retrieve the entire secret key using the first bytes of the output streams of about 40,000 known IVs (for the cases the IV is concatenated either before or after the secret key).

Vaudenay and Vuagnoux [18] improve the attacks of [5,10] on the case of WEP (where the IV is concatenated before the secret key). They present the VX attack, which uses the sum of the the key bytes to reduce the dependency between the other bytes of the key, such that the attack may work even if the data is insufficient to retrieve one of the bytes.

Paul and Maitra [15] use biases in the first entries of the initial permutation to recover the secret key from the initial permutation. They use the first entries of the permutation to create equations which hold with certain probability. They guess some of the bytes of the secret key, and use the equations to retrieve the rest of the bytes. The success of their algorithm relies on the existence of sufficiently many correct equations.

1.2 Outline of Our Contribution

In this paper we present methods that allow us to obtain significantly better results than the algorithm of [15]. A major observation considers the difference between pairs of equations instead of analyzing each equation separately. We show that the probability that the difference of a pair of equations is correct is much higher in most cases than the probabilities of each of the individual equations. Therefore, our algorithm can rely on many more equations and apply more thorough statistical techniques than the algorithm of [15]. We also show two filtering methods that allow us to identify that some of the individual equations (used in [15]) are probably incorrect by a simple comparison, and therefore, to discard these equations and all the differences derived from them. Similarly, we show filtering techniques that discard difference equations, and even correct some of these equations. We also show how to create alternative equations, which can replace the original equations in some of the cases and allow us to receive better statistical information when either the original equations are discarded or they lead to incorrect values. We combine these observations (and other observations that we discuss in this paper) into a statistical algorithm that recovers the key with a much higher success rate than the one of [15]. Our Algorithm also works if some of the bytes of the initial permutation are missing or contain errors. Such scenarios are likely results of side channel attacks, as in [9]. In these cases, our algorithm can even be used to reconstruct the full correct initial permutation by finding the correct key and then using it to compute the correct values. Details of an efficient implementation of the data structures and internals of the algorithm are also discussed.

The algorithm we propose retrieves one linear combination of the key bytes at a time. In each step, the algorithm applies statistical considerations to choose the subset of key bytes participating in the linear combination and the value which have the highest probability to be correct. If this choice turns out to be incorrect, other probable choices may be considered. We propose ways to discover incorrect choices even before the entire key is recovered (i.e., before it can be tested by running the KSA), and thus we are able to save valuable computation time that does not lead to the correct key.

Our algorithm is much faster than the algorithm of [15], and has much better success rates for the same computation time. For example, for 40-bit keys and 86% success rate, our algorithm is about 10000 times faster than the algorithm of [15]. Additionally, even if the algorithm fails to retrieve the full key, it can retrieve partial information about the key. For example, for 128-bit keys it can give a suggestion for the sum of all the key bytes which has a probability of 23.09% to be correct, or give four suggestions such that with a probability of 41.44% the correct value of the sum of all the key bytes is one of the four.

1.3 Organization of the Paper

This paper is organized as follows: Section 2 describes the RC4 algorithms, gives several observation about the keys of RC4, and defines notations which will be used throughout this paper. Section 3 presents the bias of the first bytes of the initial permutation, and describes the attack of [15], which uses these biases to retrieve the secret key. Section 4 gathers several observations which are the building blocks of our key retrieval algorithm, and have enabled us to improve the result of [15]. Section 5 takes these building blocks and uses them together to describe the detailed algorithm. In Section 6 we give some comments and observations about an efficient implementation to our algorithm. Finally, Section 7 summarizes the paper, presents the performance of our algorithm and discusses its advantages over the algorithm of [15].

2 The RC4 Stream Cipher

The internal state of RC4 consists of a permutation S of the numbers $0, \ldots, N-1$, and two indices $i, j \in \{0, \ldots, N-1\}$. The permutation S and the index j form the secret part of the state, while the index i is public and its value at any stage of the stream generation is widely known. In RC4 $N = 256$, and thus the secret internal state has $\log_2 \left(2^8 \cdot 256!\right) \approx 1692$ bits of information. Together with the public value of i there are about 1700 bits of information in the internal state. Variants with other values of N have also been analyzed in the cryptographic literature.

RC4 consists of two algorithms: The Key Scheduling Algorithm (KSA), and the Pseudo Random Generation Algorithm (PRGA), both algorithms are presented in

```
KSA(K)
Initialization:
  For $i = 0$ to $N - 1$
    $S[i] = i$
  $j \leftarrow 0$
Scrambling:
  For $i = 0$ to $N - 1$
    $j \leftarrow j + S[i] + K[i \bmod l]$
    Swap($S[i], S[j]$)
```

```
PRGA(S)
Initialization:
  $i \leftarrow 0$
  $j \leftarrow 0$
Generation loop:
  $i \leftarrow i + 1$
  $j \leftarrow j + S[i]$
  Swap($S[i], S[j]$)
  Output $S[S[i] + S[j]]$
```

Fig. 1. The RC4 Algorithms

Figure 1. All additions in RC4 are performed modulo N. Therefore, in this paper, additions are performed modulo 256, unless explicitly stated otherwise.

The KSA takes an l-byte secret key, K, and generates a pseudo-random initial permutation S. The key size l is bounded by N bytes, but is usually in the range of 5–16 bytes (40–128 bits). The bytes of the secret key are denoted by $K[0], \ldots, K[l-1]$. If $l < N$ the key is repeated to form a N-byte key. The KSA initializes S to be the identity permutation, and then performs N swaps between the elements of S, which are determined by the secret key and the content of S. Note that because i is incremented by one at each step, each element of S is swapped at least once (possibly with itself). On average each element of S is swapped twice.

The PRGA generates the pseudo-random stream, and updates the internal state of the cipher. In each iteration of the PRGA, the values of the indices are updated, two elements of S are swapped, and a byte of output is generated. During the generation of N consecutive output bytes, each element of S is swapped at least once (possibly with itself), and twice on average.

2.1 Properties of RC4 Keys

There are $2^{8 \cdot 256} = 2^{2048}$ possible keys (every key shorter than 256 bytes has an equivalent 256-byte key) but only about 2^{1684} possible initial states of RC4. Therefore, every initial permutation has on average about 2^{364} 256-byte keys which create it. Each initial permutation $\hat{S}$ has at least one, easy to find, 256-byte key: Since every byte of the key is used only once during the KSA, the key bytes are chosen one by one, where $K[i]$ is chosen to set j to be the current location of $\hat{S}[i]$ (which satifies, by this construction $j > i$). Thus, the Swap($S[i]$,$S[j]$) operation on iteration i swaps the value $\hat{S}[i] = S[j]$ with $S[i]$. The value $\hat{S}[i]$ does not participate in later swaps, and thus remains there until the end of the KSA.

The number of initial permutations which can be created by short keys, however, is much smaller. For example, the number of 16-byte keys is only 2^{128}, and the total number of keys bounded by 210 bytes is about $2^{8 \cdot 210} = 2^{1680}$, which is smaller than the total number of permutations.

2.2 Notations

We use the notation $K[a,b]$ to denote the sum of the key bytes $K[a]$ and $K[b]$, i.e.,

$$K[a,b] \triangleq K[a \bmod l] + K[b \bmod l] \bmod N.$$

Similarly, $K[a,b,c]$, $K[a,b,c,d]$, etc., are the sums of the corresponding key bytes for any number of comma-separated arguments. We use the notation $K[a \ldots b]$ to denote the sum of the key bytes in the range $a, a+1, \ldots, b$, i.e.,

$$K[a \ldots b] \triangleq \sum_{r=a}^{b} K[r \bmod l] \bmod N.$$

We also use combinations of the above, for instance:

$$K[a, b \ldots c] \triangleq K[a \bmod l] + \sum_{r=b}^{c} K[r \bmod l],$$

$$K[a \ldots b, c \ldots d] \triangleq \sum_{r=a}^{b} K[r \bmod l] + \sum_{r=c}^{d} K[r \bmod l].$$

We use the notations S_r and j_r to denote the values of the permutation S and the index j after r iterations of the loop of the KSA have been executed. The initial value of j is $j_0 = 0$ and its value at the end of the KSA is j_N. S_0 is the identity permutation, and S_N is the result of the KSA (i.e., the initial permutation that we study in this paper). For clarity, from now on the notation S (without an index) denotes the initial permutation S_N.

3 Previous Techniques

In 1995 Roos [16] noticed that some of the bytes of the initial permutation have a bias towards a linear combination of the secret key bytes. Theorem 1 describes this bias (the theorem is taken from [16], but is adapted to our notations).

Theorem 1. *The most likely value for $S[i]$ at the end of the KSA is:*

$$S[i] = K[0 \ldots i] + \frac{i(i+1)}{2} \mod N. \tag{1}$$

Only experimental results for the probabilities of the biases in Theorem 1 are provided in [16]. Recently, Paul and Maitra [15] supplied an analytic formula for this probability, which has corroborated the results given by [16]. Theorem 2 presents their result.

Theorem 2 (Corollary 2 of [15]). *Assume that during the KSA the index j takes its values uniformly at random from $\{0, 1, \ldots, N-1\}$. Then,*

$$P\left(S[i] = K[0 \ldots i] + \frac{i(i+1)}{2}\right) \geq \left(\frac{N-i}{N}\right) \cdot \left(\frac{N-1}{N}\right)^{\frac{i(i+1)}{2}+N} + \frac{1}{N}.$$

For any fixed value of i, the bias described by (1) is the result of a combination of three events that occur with high probability:

1. $S_r[r] = r$ for $r \in \{0, \dots, i\}$ (i.e., the value of $S[r]$ was not swapped before the r-th iteration).
2. $S_i[j_{i+1}] = j_{i+1}$.
3. $j_r \neq i$ for $r \in \{i+1, \dots, N-1\}$.

If the first event occurs then the value j_{i+1} is affected only by the key bytes and constant values:

$$j_{i+1} = \sum_{r=0}^{i} (K[r] + S_r[r]) = \sum_{r=0}^{i} (K[r] + r) = K[0 \dots i] + \frac{i\,(i+1)}{2}.$$

If the second event occurs, then after $i+1$ iteration of the KSA $S_{i+1}[i] = j_{i+1}$. The third event ensures that the index j does not point to $S[i]$ again, and therefore $S[i]$ is not swapped again in later iterations of the KSA. If all three events occur then (1) holds since

$$S_N[i] \underset{\underset{3}{\uparrow}}{=} S_{i+1}[i] \underset{\underset{2}{\uparrow}}{=} j_{i+1} = \sum_{r=0}^{i} (K[r] + S_r[r]) \underset{\underset{1}{\uparrow}}{=} K[0 \dots i] + \frac{i\,(i+1)}{2}.$$

The probabilities derived from Theorem 2 for the biases of the first 48 entries of S ($S[0] \dots S[47]$) are given in Table 1 (also taken from [15]). It can be seen that this probability is about 0.371 for $i = 0$, and it decreases as the value of i increases. For $i = 47$ this probability is only 0.008, and for further entries it becomes too low to be used by the algorithm (the a-priori probability that an entry equals any random value is $1/256 \approx 0.0039$). The cause for such a decrease in the bias is that the first of the aforementioned events is less likely to occur for high values of i, as there are more constraints on entries in S.

Given an initial permutation S (the result of the KSA), each of its entries can be used to derive a linear equation of the key bytes, which holds with the probability given by Theorem 2. Let C_i be defined as

$$C_i = S[i] - \frac{i \cdot (i+1)}{2}.$$

Table 1. The Probabilities Given by Theorem 2

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Prob.	.371	.368	.364	.358	.351	.343	.334	.324	.313	.301	.288	.275	.262	.248	.234	.220
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Prob.	.206	.192	.179	.165	.153	.140	.129	.117	.107	.097	.087	.079	.071	.063	.056	.050
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Prob.	.045	.039	.035	.031	.027	.024	.021	.019	.016	.015	.013	.011	.010	.009	.008	.008

Using (1) the i'th equation (derived from the entry $S[i]$) becomes:

$$K[0 \ldots i] = C_i. \tag{2}$$

The RecoverKey algorithm of [15] uses these equations in order to retrieve the secret key of RC4. Let n and m be parameters of the algorithm, and recall that l is the length of the secret key in bytes. For each combination of m independent equations out of the first n equations of (2), the algorithm exhaustively guesses the value of $l - m$ key bytes, and solves the m equations to retrieve the rest of the key bytes. The success of the RecoverKey algorithm relies on the existence of m correct and linearly independent equations among the first n equations. The success probabilities and the running time of the RecoverKey algorithm for different key sizes and parameters, as given by [15], are presented in Table 2.[1]

4 Our Observations

Several important observations allow us to suggest an improved algorithm for retrieving the key from the initial permutation.

4.1 Subtracting Equations

Let $i_2 > i_1$. As we expect that $K[0 \ldots i_1] = C_{i_1}$ and $K[0 \ldots i_2] = C_{i_2}$, we also expect that

$$K[0 \ldots i_2] - K[0 \ldots i_1] = K[i_1 + 1 \ldots i_2] = C_{i_2} - C_{i_1} \tag{3}$$

holds with the product of the probabilities of the two separate equations. However, we observe that this probability is in fact much higher. If the following three events occur then (3) holds (compare with the three events described in Section 3):

1. $S_r[r] = r$ for $r \in \{i_1 + 1, \ldots, i_2\}$ (i.e., the value of $S[r]$ was not swapped before the r-th iteration).
2. $S_{i_1}[j_{i_1+1}] = j_{i_1+1}$ and $S_{i_2}[j_{i_2+1}] = j_{i_2+1}$.
3. $j_r \neq i_1$ for $r \in \{i_1 + 1, \ldots, N - 1\}$, and $j_r \neq i_2$ for $r \in \{i_2 + 1, \ldots, N - 1\}$.

[1] We observe that the formula for the complexity given in [15] is mistaken, and the actual values should be considerably higher than the ones cited in Table 2. We expect that the correct values are between 2^5 and 2^8 times higher. The source for the mistake is two-fold: the KSA is considered as taking one unit of time, and the complexity analysis is based on an inefficient implementation of their algorithm. Given a set of l equations, their implementation solves the set of equations separately for every guess of the remaining $l - m$ variables, while a more efficient implementation would solve them only once, and only then guess the values of the remaining bytes. Our complexities are even lower than the complexities given in [15], and are much lower than the correct complexities.

We also observe that the complexities given by [15] for the case of 16-byte keys do not match the formula they publish (marked by * in Table 2). The values according to their formula should be 2^{82}, 2^{79}, 2^{73} and 2^{69} rather than 2^{60}, 2^{63}, 2^{64} and 2^{64}, respectively. Their mistake is possibly due to an overflow in 64-bit variables.

Table 2. Success Probabilities and Running Time of the RecoverKey Algorithm of [15]

l	n	m	Time	$P_{Success}$
5	16	5	2^{18}	0.250
5	24	5	2^{21}	0.385
8	16	6	2^{34}	0.273
8	20	7	2^{29}	0.158
8	40	8	2^{33}	0.092
10	16	7	2^{43}	0.166
10	24	8	2^{40}	0.162
10	48	9	2^{43}	0.107
12	24	8	2^{58}	0.241
12	24	9	2^{50}	0.116
16	24	9	2^{60} *	0.185
16	32	10	2^{63} *	0.160
16	32	11	2^{64} *	0.086
16	40	12	2^{64} *	0.050

* Incorrect entries — see footnote [1].

If the first event occurs then the index j is affected in iterations i_1+1 through i_2 only by the key bytes and constant values:

$$j_{i_2+1} - j_{i_1+1} = \sum_{r=i_1+1}^{i_2} (K[r] + S_r[r]) = K[i_1+1\ldots i_2] + \sum_{r=i_1+1}^{i_2} r$$

If the second event occurs, then after i_1+1 iteration of the KSA $S_{i_1+1}[i_1] = j_{i_1+1}$, and after i_2+1 iteration $S_{i_2+1}[i_2] = j_{i_2+1}$. The third event ensures that the index j does not point to $S[i_1]$ or $S[i_2]$ again, and therefore $S[i_1]$ and $S[i_2]$ are not swapped again in later iterations. If all three events occur then (3) holds since

$$\begin{aligned} S_N[i_2] - S_N[i_1] &\underset{\overset{\uparrow}{3}}{=} S_{i_2+1}[i_2] - S_{i_1+1}[i_1] \underset{\overset{\uparrow}{2}}{=} j_{i_2+1} - j_{i_1+1} = \\ &= \sum_{r=i_1+1}^{i_2} (K[r] + S_r[r]) \underset{\overset{\uparrow}{1}}{=} K[i_1+1\ldots i_2] + \sum_{r=i_1+1}^{i_2} r = \\ &= K[i_1+1\ldots i_2] + \frac{i_2(i_2+1)}{2} - \frac{i_1(i_1+1)}{2}, \end{aligned}$$

and therefore

$$K[i_1+1\ldots i_2] = C_{i_2} - C_{i_1} \ .$$

Theorem 3 states the exact bias of such differences.

Theorem 3. *Assume that during the KSA the index j takes its values uniformly at random from $\{0,1,\ldots,N-1\}$, and let $0 \le i_1 < i_2 < N$. Then,*
$P\left(C_{i_2} - C_{i_1} = K[i_1+1\ldots i_2]\right) \ge$
$$\left[\left(1-\tfrac{i_2}{N}\right)^2 \cdot \left(1-\tfrac{i_2-i_1+2}{N}\right)^{i_1} \cdot \left(1-\tfrac{2}{N}\right)^{N-i_2-1} \cdot \prod_{r=0}^{i_1-i_2-1}\left(1-\tfrac{r+2}{N}\right)\right] + \tfrac{1}{N}.$$

The proof of Theorem 3 is based on the discussion which precedes it, and is similar to the proof of Theorem 2 given in [15]. The proof is based on the analysis of the probabilities that the values of j throughout the KSA are such that the three events described earlier hold.

As a result of Theorem 3 our algorithm has many more equations to rely on. We are able to use the difference equations which have high enough probability, and furthermore, we can now use data which was unusable by the algorithm of [15]. For instance, according to Theorem 2, the probability that $K[0\ldots 50] = C_{50}$ is 0.0059, and the probability that $K[0\ldots 52] = C_{52}$ is 0.0052. Both equations are practically useless by themselves, but according to Theorem 3 the probability that $K[51\ldots 52] = C_{52} - C_{50}$ is 0.0624, which is more than ten times the probabilities of the individual equations.

Moreover, the biases given by Theorem 2 and used by the RecoverKey algorithm of [15] are dependent. If $S_r[r] \neq r$ for some r, then the first event described in Section 3 is not expected to hold for any $i > r$, and we do not expect equations of the form (2) to hold for such values of i. However, equations of the form (3) for $i_2 > i_1 > r$ may still hold under these conditions, allowing us to handle initial permutations which the RecoverKey algorithm cannot.

4.2 Using Counting Methods

Since every byte of the secret key is used more than once, we can obtain several equations for the same sum of key bytes. For example, all the following equations:

$$\begin{aligned} C_1 &= K[0\ldots 1] \\ C_{l+1} - C_{l-1} &= K[l\ldots l+1] = K[0\ldots 1] \\ C_{2l+1} - C_{2l-1} &= K[2l\ldots 2l+1] = K[0\ldots 1] \end{aligned}$$

suggest values for $K[0] + K[1]$. If we have sufficiently many suggestions for the same sum of key bytes, the correct value of the sum is expected to appear more frequently than other values. We can assign a weight to each suggestion, use counters to sum the weights for each possible candidate, and select the candidate which has the highest weight. We may assign the same weight to all the suggestions (majority vote) or a different weight for each suggestion (e.g., according to its probability to hold, as given by Theorems 2 and 3). We demonstrate the use of counters using the previous example. Assume that $C_1 = 178$, $C_{l+1} - C_{l-1} = 210$ and $C_{2l+1} - C_{2l-1} = 178$ are the only suggestions for the value of $K[0\ldots 1]$, and assume that all three suggestions have equal weights of one. Under these conditions the value of the counter of 178 will be two, the value of the counter of 210 will be one, and all other counters will have a value of zero. We guess that $K[0\ldots 1] = 178$, since it has the highest total weight of suggestions.

A simple algorithm to retrieve the full key would be to look at all the suggestions for each of the key bytes, and choose the most frequent value for each one. Unfortunately, some of the bytes retrieved by this sort of algorithm are expected to be incorrect. We can run the KSA with the retrieved key to test its correctness, but if the test fails (the KSA does not produce the expected initial permutation), we get no clue to where the mistake is.

However, we observe that we do not need to limit ourselves to a single key byte, but rather consider all candidates for all possible sums of key bytes suggested by the equations, and select the combination with the highest total weight. Once we fix the chosen value for the first sum, we can continue to another, ordered by the weight, until we have the entire key. There is no need to consider sequences which are linearly dependent in prior sums. For example, if we have already fixed the values of $K[0] + K[1]$ and $K[0]$, there is no need to consider suggestions for $K[1]$. Therefore, we need to set the values of exactly l sums in order to retrieve the full key. Moreover, each value we select allows us to substantially reduce the number of sums we need to consider for the next step, as it allows us to merge the counters of some of the sums (for example, if we know that $K[0]+K[1] = 50$ then we can treat suggestions for $K[0] = 20$ together with $K[1] = 30$).

A natural extension to this approach is trying also the value with the second highest counter, in cases where the highest counter is found wrong. More generally, once a value is found wrong, or a selection of a sequence is found unsatisfactory, backtracking is performed. We denote the number of attempts to be tried on the t-th guess by λ_t, for $0 \le t < l$. This method can be thought of as using a DFS algorithm to search an ordered tree of height $l + 1$, where the degree of vertices on the t-th level is λ_t and every leaf represents a key.

4.3 The Sum of the Key Bytes

Denote the sum of all l key bytes by s, i.e.,

$$s = K[0 \ldots l-1] = \sum_{r=0}^{l-1} K[r].$$

The value of s is especially useful. The linear equations derived from the initial permutation give sums of sequences of consecutive key bytes. If we know the value of s, all the suggestions for sequences longer than l bytes can be reduced to suggestions for sequences which are shorter than l bytes. For example, from the following equations:

$$\begin{aligned} C_1 &= K[0 \ldots 1] \\ C_{l+1} &= K[0 \ldots l+1] = s + K[0 \ldots 1] \\ C_{2l+1} &= K[0 \ldots 2l+1] = 2s + K[0 \ldots 1] \end{aligned}$$

we get three suggestions C_1, $C_{l+1} - s$, and $C_{2l+1} - 2s$ for the value of $K[0]+K[1]$.

After such a reduction is performed, all the remaining suggestions reduce to sums of fewer than l key bytes, of the form $K[i_1 \ldots i_2]$, where $0 \le i_1 < l$ and $i_1 \le i_2 < i_1+l-1$. Thus, there are only $l\cdot(l-1)$ possible sequences of key bytes to consider. Furthermore, the knowledge of s allows us to unify every two sequences which sum up to $K[0 \ldots l-1] = s$ (as described in Section 4.2), thus reducing the number of sequences to consider to only $l\cdot(l-1)/2$ (without loss of generality, the last byte of the key, $K[l-1]$, does not appear in the sequences we consider, so each sum we consider is of the form $K[i_1 \ldots i_2]$, for $0 \le i_1 \le i_2 < l-1$). In turn, there are more suggestions for each of those unified sequences than there were for each original sequence.

Table 3. Probabilities that s is Among the Four Highest Counters

Key Length	Highest Counter	Second Highest	Third Highest	Fourth Highest
5	0.8022	0.0618	0.0324	0.0195
8	0.5428	0.1373	0.0572	0.0325
10	0.4179	0.1604	0.0550	0.0332
12	0.3335	0.1618	0.0486	0.0287
16	0.2309	0.1224	0.0371	0.0240

Fortunately, besides being the most important key byte sequence, s is also the easiest sequence to retrieve, as it has the largest number of suggestions. Any sum of l consecutive bytes, of the form $K[i+1 \ldots i+l] = C_{i+l} - C_i$, for any i, yields a suggestion for s. In a similar way, we can consider sequences of $2l$ bytes for suggestions for $2s$, and we can continue to consider sequences of αl consecutive bytes, for any integer α. However, for common key lengths, the probability of a correct sum with $\alpha > 2$ is too low.

As discussed in Section 4.2, we may want to consider also the second highest counter and perform backtracking. Our experimental results for the success probabilities of retrieving s are presented in Table 3. For each of the key lengths in the table, we give the probability that the value of s is the value with the highest counter, second highest, third highest, or fourth highest. The data in the table was compiled by testing 1,000,000,000 random keys for each of the key lengths, and considering all suggestions with a probability higher than 0.01.

4.4 Adjusting Weights and Correcting Equations

During the run of the algorithm, we can improve the accuracy of our guesses based on previous guesses. Looking at all suggestions for sequences we have already established, we can identify exactly which of them are correct and which are not, and use this knowledge to gain information about intermediate values of j and S during the execution of the KSA. We assume that if a suggestion $C_{i_2} - C_{i_1}$ for $K[i_1 + 1 \ldots i_2]$ is correct, then all three events described in Section 4.1 occur with a relatively high probability. Namely, we assume that:

- $S_r[r] = r$ for $i_1 + 1 \leq r \leq i_2$ (follows from event 1 from Section 4.1).
- $S[i_1] = j_{i_1+1}$ and $S[i_2] = j_{i_2+1}$ (together, follow from events 2 and 3 from Section 4.1.

This information can be used to better assess the probabilities of other suggestions. When considering a suggestion $C_{i_4} - C_{i_3}$ for a sum of key bytes $K[i_3 + 1 \ldots i_4]$ which is still unknown, if we have an indication that one of the three events described in Section 4.1 is more likely to have occurred than predicted by its a-priori probability, the weight associated with the suggestion can be increased. Example 1 demonstrates a case in which such information is helpful.

Example 1. Assume that the following three suggestions are correct:

1. $K[0 \ldots 9] = C_9$,
2. $K[12 \ldots 16] = C_{16} - C_{11}$,
3. $K[7 \ldots 14] = C_{14} - C_6$,

and assume that for each of them the three events described in Section 4.1 hold during the execution of the KSA. From the first suggestion we conclude that $j_{10} = S[9]$, from the second suggestions we learn that $j_{12} = S[11]$, and the third suggestion teaches us that $S_r[r] = r$ for $7 \leq r \leq 14$ (and in particular for r=10 and r=11). It can be inferred from the last three observations and according to the explanation in Section 4.1 that under these assumptions $K[10 \ldots 11] = C_{11} - C_9$. Since the probabilities that the assumptions related to $K[10 \ldots 11] = C_{11} - C_9$ hold are larger than the a-priory probability (due to the relation to the other suggestions, which are known to be correct), the probability that this suggestion for $K[10 \ldots 11]$ is correct is increased.

Similarly, we can gain further information from the knowledge that suggestions are incorrect. Consider r's for which there are many incorrect suggestions that involve C_r, either with preceding C_{i_1} ($C_r - C_{i_1}$, $i_1 < r$) or with succeeding C_{i_2} ($C_{i_2} - C_r$, $i_2 > r$). In such cases we may assume that $S_N[r]$ is not the correct value of j_{r+1}, and thus all other suggestions involving C_r are also incorrect.

Consider r's for which there are many incorrect suggestions that pass over r, i.e., of the form $C_{i_2} - C_{i_1}$ where $i_1 < r \leq i_2$. In this case, we may assume that during the KSA $S_r[r] \neq r$, and thus all other suggestions that pass over r are also incorrect. All suggestions that pass over r for which $S_r[r] \neq r$ is the only event (of the three events described in Section 4.1) that does not hold, must have the same error $\Delta = C_{i_2} - C_{i_1} - K[i_1 + 1 \ldots i_2]$ (which is expected to be $\Delta = S_r[r] - r$). Thus, if we find that for some r several suggestions that pass over r have the same error Δ, we can correct other suggestions by Δ.

4.5 Refining the Set of Equations

We observe that some of the equations can be discarded based on the values of the initial permutation, and some others have alternatives. This observation is also applicable to the equations used by the algorithm of [15], and could have improved its running time and success probabilities.

If $S[i'] < i'$ for some i', then the equation derived from $S[i']$ should be discarded, since $x = S[i']$ is not expected to satisfy (1). In this case, even if Event 1 and Event 3 (of the three events described in Section 3) hold, it is clear that Event 2 does not, as the number x has already been swapped in a previous iteration (when $i = x$), and is not likely to be in location $S[i']$ after i' iterations of the KSA.

If $S[i'] > i'$ for some i', then an alternative equation may be derived from $x = S[i']$, in addition to the equation derived by the algorithm of [15]. The equations used by [15] assume that the assignment $S[i'] \leftarrow x$ occurred with $i = i'$, and $j = S[j] = x$ (Figure 2(a)). However, in this case, another likely possibility is that the assignment $S[i'] \leftarrow x$ occurred with $i = S[i] = x$, and

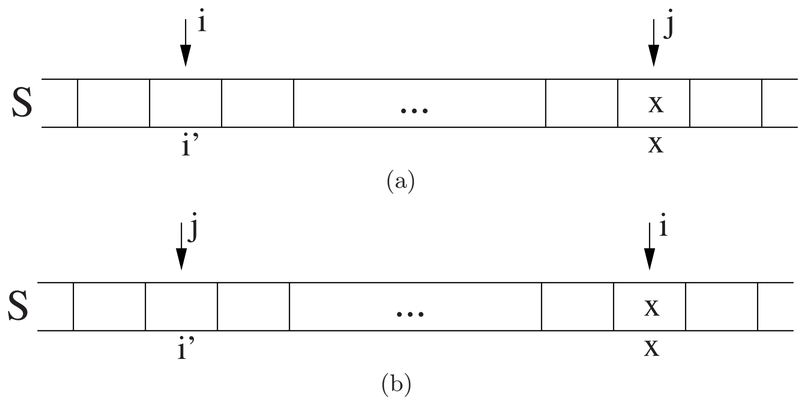

Fig. 2. Two Probable Alternatives to the Positions of the Indices i and j Right Before the Assignment $S[i'] \leftarrow x$ Occurred

$j = i'$ (Figure 2(b)). In the latter case, $j_{x+1} = i'$, and the following equation holds with a high probability:

$$i' = K[0 \ldots x] + \frac{x(x+1)}{2}.$$

It can be shown that this equation holds with a probability slightly higher than the probability given by Theorem 2 for $i = x$. We now have two likely possibilities for the value of j_{x+1}, i' and $S[x]$, which yield two alternative equations. Let $\bar{C}_x$ be defined as:

$$\bar{C}_x = S^{-1}[x] - \frac{x(x+1)}{2} \ .$$

Using this notation, the proposed alternative equation is

$$K[0 \ldots x] = \bar{C}_x \ .$$

Every time C_x is used to create a suggestion (by subtracting equations), the value $\bar{C}_x$ (if exists) can replace it to create an alternative suggestion for the same sum of key bytes. It can be shown that the probabilities that $\bar{C}_{x_2} - C_{x_1}$, $C_{x_2} - \bar{C}_{x_1}$ and $\bar{C}_{x_2} - \bar{C}_{x_1}$ hold are slightly higher than the probability that $C_{x_2} - C_{x_1}$ holds (for any $x_1 < x_2$). Note that we do not expect that many equations have such alternatives, because under the assumption that j takes its values uniformly at random, it is much more likely that $j_{i+1} > i$ for small values of i. Given the two alternatives it is possible to run the algorithm twice, while on each run consider only suggestions derived from the set of equations with one of the two alternatives. However, due to our use of counting methods, both equations can be added to the same set of equations, such that suggestions derived from both alternatives are counted together, in the same counting process.

4.6 Heuristic Pruning of the Search

In Section 4.2 we have described the backtracking approach to finding the key as a DFS search on an ordered tree. Once a guessed value is found wrong (the

```
FIND_KEY(S)
1. Build the equations: Compute the values of $\{C_i\}$ and $\{\bar{C}_i\}$, for the indices $i$
   where they exist (described in Sections 3 and 4.5).
2. Sum the weights of suggestions for each of the $N$ candidates for $s$ (described
   in Section 4.3).
3. For $x = 1$ to $\lambda_0$ do:
   (a) Find a candidate for $s$ with the highest counter, $w_0$, which has not been
       checked yet, and set $s = K[0 \ldots l-1] = w_0$.
   (b) Mark the correct suggestions for $s = w_0$, adjust weights and correct
       remaining suggestions accordingly (described in Section 4.4).
   (c) Initialize $N$ counters for each sequence of key bytes $K[i_1 \ldots i_2]$ such that
       $0 \leq i_1 \leq i_2 < l-1$, and sum the weights of suggestions for each of them
       (described in Sections 4.1, 4.2 and 4.3).
   (d) Call REC_SUBROUTINE(1) to retrieve the rest of the key. If the correct
       key is found, return it.
4. Return FAIL.
```

Fig. 3. The FIND_KEY Algorithm

keys obtained from it fail to create the requested permutation) we go back and try the other likely guesses. Naturally, by trying more guesses we increase our chances to successfully retrieve the key, but we increase the computation time as well. If we can identify an internal nodes as representing a wrong guess, we can skip the search of the entire subtree rooted from it, and thus reduce the computation time.

Section 4.2 also describes the merging of counters of different sequences according to previous guesses, which allows us to consider fewer key sequences, with more suggestions for each. If the guesses that we have already made are correct, we expect that after such a merge the value of the highest counter is significantly higher than other counters. If the former guesses are incorrect, we don't expect to observe such behavior, as the counters of different sequences will be merged in a wrong way.

Let μ_t for $0 \leq t < l$ be a threshold for the t-th guess. When considering candidates for the t-th guess, we only consider the ones with a counter value of at least μ_t. The optimal values of the thresholds can be obtained empirically, and depend on the key length (l), the weights given to the suggestions, and the number of attempts for each guess (λ_t's). Even if the use of these thresholds may cause correct guesses to be aborted, the overall success probability may still increase, since the saved computation time can be used to test more guesses.

5 The Algorithm

The cornerstones of our method have been laid in Section 4. In this section we gather all our previous observations, and formulate them as an algorithm to retrieve the secret key from the initial permutation S. The FIND_KEY algorithm (presented in Figure 3) starts the search by finding s, and calls the recursive algorithm REC_SUBROUTINE (Figure 4). Each recursive call guesses another value for a sum of key bytes, as described in the previous section.

The optimal values of the parameters $\lambda_0, \ldots, \lambda_{l-1}, \mu_1, \ldots, \mu_{l-1}$ used by the algorithm and the weights it assigns to the different suggestions can by empirically found, so that the success probability of the algorithm and/or the average running time are within a desired range.

6 Efficient Implementation

Recall that on each iteration of the algorithm some of the sums of the key bytes are already known (or guessed). The suggestions for the unknown sums are counted using a set of N counters, one counter for each possible value of that sum. In Section 4.2 we stated that according to the prior guesses, the suggestions for several sums of key bytes may be counted together (i.e., after a new guess is made, some of the counters may be merged with counters of other sums). This section describes an efficient way to discover which counters should be merged, and how to merge them.

The known bytes induce an equivalence relation between the unknown sums of the key bytes. Two sums are in the same equivalence class if and only if the value of each of them can be computed from the value of the other and the values of known sums. We only need to keep a set of N counters for each equivalence class, as all

REC_SUBROUTINE(t)

```
1. If $t = l$, extract the key from all the $l$ guesses made so far, and verify it. If
   the key is correct, return it. Otherwise, return FAIL.
2. For $y = 1$ to $\lambda_t$ do:
   (a) Find a combination of key sequence and a candidate for its sum, with the
       highest counter among the sum of sequences that hasn't already been
       guessed yet. Denote them by $K[i_1 \ldots i_2]$ and $w_t$, respectively, and denote
       the value of that counter by $h$.
   (b) If $h < \mu_t$, return FAIL (described in Section 4.6).
   (c) Set $K[i_1 \ldots i_2] = w_t$.
   (d) Mark the correct suggestions for $K[i_1 \ldots i_2] = w_t$, adjust weights and
       correct remaining suggestions accordingly (described in Section 4.4).
   (e) Merge the counters which may be unified as a result of the guess from 2a
       (described in Section 4.2).
   (f) Call REC_SUBROUTINE($t + 1$). If the correct key is found, return it.
       Otherwise, cancel the most recent guess (revert any changes made during
       the current iteration, including the merging of the counters).
3. Return FAIL.
```

Fig. 4. The Recursive REC_SUBROUTINE Algorithm

suggestions for sums which are in the same equivalence class should be counted together. When we merge counters, we actually merge equivalence classes.

We represent our knowledge about the values of the sums as linearly independent equations of the key bytes. After r key sums are guessed, there are r linear equations of the form

$$\sum_{i=0}^{l-1} a_{i,j} K[j] = b_j,$$

for $1 \leq j \leq r$, where $0 \leq a_{i,j} < N$. The equations are represented as a triangular system of equations, in which the leading coefficient (the first non-zero coefficient) of each equation is one. These r equations form a basis of a linear subspace of all the sums we already know. In this representation the equivalence class of any sum of key bytes $K[i_1 \ldots i_2]$ can be found efficiently: We represent the sum as a linear equation of the key bytes, and apply the Gaussian elimination process, such that the system of equations is kept triangular, and the leading coefficient of each equation is one. Sums from the same equivalence class give the same result, as they all extend the space spanned by the r equations to *the same* larger space spanned by $r+1$ equations. The resulting unique equation can be used as an identifier of the equivalence class. When the counters are merged after a guess of a new value, the same process is applied — we apply Gaussian elimination to the equation representing the current equivalence class in order to discover the equivalence class it belongs to on the next level, and merge the counters. Note that as a result of the Gaussian elimination process we also learn the exact linear mapping between the counters of the current equivalence classes, and the counters of the classes of the next step.

7 Discussion

In this paper we presented an efficient algorithm for the recovery of the secret key from the initial state of RC4, using the first bytes of the permutation. The presented algorithm can also work if only some of the bytes of the initial permutation are known. In this case, suggestions are derived only from the known bytes, and the algorithm is only able to retrieve values of sums of key bytes for which suggestions exist. However, as a result of the reduced number of suggestions the success rates are expected to be lower. The algorithm can also work if some of the bytes contain errors, as the correct values of the sums of key bytes are still expected to appear more frequently than others.

Since changes to the internal state during the stream generation (PRGA) are reversible, our algorithm can also be applied given an internal state at any point of the stream generation phase. Like in [15], our algorithm is also applicable given an intermediate state during the KSA, i.e., S_i $(i < N)$, instead of S_N.

We tested the running times and success probabilities of our algorithm for different key lengths, as summarized in Table 4. The tests were performed on a Pentium IV 3GHz CPU. The running times presented in the table are averaged over 10000 random keys. We have assigned a weight of two to suggestions with

Table 4. Empirical Results of The Proposed Attack

Key Length	Time	$P_{Success}$	Time of Improved [15]* $_{[sec]}$
5	0.02	0.8640	366
8	0.60	0.4058	2900
10	1.46	0.0786	183
10	3.93	0.1290	2932
12	3.04	0.0124	100
12	7.43	0.0212	1000
16	278	0.0005	500

* Our rough estimation for the time it would take an improved version of the algorithm of [15] achieve the same $P_{Success}$ (see footnote [1]). The time of the algorithm of [15] is much slower.

probability higher than 0.05, a weight of one to suggestions with probability between 0.008 and 0.05 and a weight of zero to all other suggestions. The values of the parameters $\lambda_0, \ldots, \lambda_{l-1}, \mu_1, \ldots, \mu_{l-1}$ were chosen in an attempt to achieve the best possible success probability with a reasonable running time. As can be seen in the table, our algorithm is much faster than the one of [15] for the same success rate, and in particular in the case of 5-byte keys, it is about 10000 times faster. Note that with the same computation time, our algorithm achieves about four times the success rate compared to [15] in most presented cases.

Another important advantage of our algorithm over the algorithm of [15] is that when the algorithm of [15] fails to retrieve the key, there is no way to know which of the equations are correct, nor is it possible to retrieve partial information about the key. However, in our algorithm, even if the algorithm fails to retrieve the full key, its first guesses are still likely to be correct, as those guesses are made based on counters with high values. This difference can be exemplified by comparing the success rates of obtaining the sum of key bytes s (Table 3) with the success rates of obtaining the entire key (Table 4).

Acknowledgments

The authors would like to thank Adi Shamir for his comments.

References

1. Anonymous, RC4 Source Code, CypherPunks mailing list, September 9 (1994), `http://cypherpunks.venona.com/date/1994/09/msg00304.html`
2. Biham, E., Granboulan, L., Nguyễn, 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)

3. Dierks, T., Allen, C.: The TLS Protocol, Version 1.0, Internet Engineering Task Force (January 1999), `ftp://ftp.isi.edu/in-notes/rfc2246.txt`
4. Finney, H.: An RC4 Cycle That Can't Happen, Usenet newsgroup sci.crypt (September 1994)
5. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)
6. Fluhrer, S.R., McGrew, D.A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 19–30. Springer, Heidelberg (2001)
7. Golić, J.D.: Linear Statistical Weakness of Alleged RC4 Keystream Generator. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 226–238. Springer, Heidelberg (1997)
8. Grosul, A.L., Wallach, D.S.: A Related-Key Cryptanalysis of RC4, Technical Report TR-00-358, Department of Computer Science, Rice University (June 2000), `http://cohesion.rice.edu/engineering/computerscience/tr/TR_Download.cfm?SDID=126`
9. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys (February 2008), `http://citp.princeton.edu/pub/coldboot.pdf`
10. Klein, A.: Attacks on the RC4 Stream Cipher (2007), `http://cage.ugent.be/~klein/RC4/RC4-en.ps`
11. 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)
12. Mantin, I.: Analysis of the Stream Cipher RC4, Master Thesis, The Weizmann Institute of Science, Israel (2001), `http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Mantin1.zip`
13. Mantin, I.: Predicting and Distinguishing Attacks on RC4 Keystream Generator. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 491–506. Springer, Heidelberg (2005)
14. 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)
15. Paul, G., Maitra, S.: Permutation After RC4 Key Scheduling Reveals the Secret Key. In: Adams, C., Miri, A., Wiener, M. (eds.) LNCS, vol. 4876, pp. 260–377. Springer, Heidelberg (2007), `http://eprint.iacr.org/2007/208.pdf`
16. Roos, A.: A Class of Weak Keys in the RC4 Stream Cipher, Two posts in sci.crypt (1995), `http://marcel.wanda.ch/Archive/WeakKeys`
17. Tews, E., Weinmann, R.P., Pyshkin, A.: Breaking 104 Bit WEP in Less than 60 Seconds (2007), `http://eprint.iacr.org/2007/120.pdf`
18. Vaudenay, S., Vuagnoux, M.: Passive-only Key Recovery Attacks on RC4. In: Adams, C., Miri, A., Wiener, M. (eds.) LNCS, vol. 4876, pp. 344–359. Springer, Heidelberg (2007), `http://infoscience.epfl.ch/record/115086/files/VV07.pdf`