Lecture Notes on Stream Ciphers and RC4

Rick Wash

rlw6@po.cwru.edu

Abstract. In these notes I explain symmetric key additive keystream ciphers, using as an example the cipher **RC4**. I discuss a number of attack models for this class of ciphers, using attacks on **RC4** as examples. I cover a number of attacks on **RC4**, some of which are effective against implementations of **RC4** used in the real world.

1 Introduction

Stream ciphers are a very important class of encryption algorithms. These notes explain what stream ciphers are, explain common subclasses of stream ciphers, and discuss the attack models relevant to stream ciphers. They also discuss the stream cipher **RC4** in detail, using it as an example for discussing a number of different attacks.

2 Stream Ciphers

Symmetric key cryptosystems are an important type of modern cryptosystem. Symmetric key systems are cryptosystems where the same key is used for both encryption and decryption. This class of cryptosystems is important in modern cryptography because, in general, symmetric key cryptosystems are much faster than public key cryptosystems.

2.1 Block vs. Stream Ciphers

The two major types of symmetric key systems are block ciphers and stream ciphers. Block ciphers in general process the plaintext in relatively large blocks at a time. The encryption function is the same for every block. A *block cipher* can be represented by a bijective function f which accepts as input a block of plaintext of a fixed size, and a key, and outputs a block of ciphertext. See Eq. 1.

$$f(p, k) = c \tag{1}$$

Stream ciphers, on the other hand, process plaintext in small blocks (sometimes as small as a single bit). In contrast to block ciphers, stream ciphers keep some sort of memory, or state, as it processes the plaintext and uses this state

as an input to the cipher algorithm. More specifically, a *stream cipher* is two functions, f and g, given in Eq. 2.

$$\begin{aligned} \sigma_{t+1} &= f(\sigma_t, p_t, k) \\ c_t &= g(\sigma_t, p_t, k) \end{aligned} \tag{2}$$

f is the next state function, which given the current state, the next block of plaintext, and the key produces a new state. g is the output function, which given the same three inputs produces a block of ciphertext as output. Note that then next time f and g is called (at time $t+1$), the state will be different.

2.2 Types of Stream Ciphers

In [1], an interesting distinction is made between two types of stream ciphers – synchronous stream ciphers and self-synchronizing stream ciphers. A *synchronous stream cipher* is a cipher where the a keystream is generated separately from the plaintext and is then combined with the plaintext later to form the ciphertext. More specifically, a synchronous stream cipher is

$$\begin{aligned} \sigma_{t+1} &= f(\sigma_t, k) \\ z_t &= g(\sigma_t, k) \\ c_t &= h(z_t, p_t) \end{aligned} \tag{3}$$

where f is the next state function, g is the keystream output function, and h is the function that combines the plaintext with the keystream to produce the ciphertext. Note that decryption only requires inverting the h function.

The two functions f and g in Eq. 3 are together known as a the *keystream generator*. The output of these two functions, the sequence of z_t values, is known as the keystream. As such, synchronous stream ciphers are also known as keystream generator ciphers. This class of ciphers has the advantage that the keystream can be precomputed without knowledge of the plaintext or ciphertext.

A particularly popular subclass of keystream generator stream ciphers is the *binary additive stream ciphers*. In this class of ciphers, the h function is the XOR function (represented by $\oplus$). This will be the primary model of stream cipher that will be analyzed.

3 Attack Models

Stream ciphers are generally studied with respect to a number of common attack models. These attack models are considered good models to study the security of stream ciphers in, but they do not cover all possible attacks. Some of these models provide for more practical attacks than others do.

One thing to remember is that stream ciphers (and most encryption algorithms in general) do not provide for message integrity. This must be done by some external algorithm, such as a MAC (Message Authentication Code). It is

possible to use the lack of message integrity checks to partially determine some of the plaintext, but that is beyond the scope of this paper.

A MAC is particularly important for binary additive stream ciphers. For this class of ciphers, flipping a bit of the ciphertext will flip the corresponding bit of the plaintext, and only affect that one bit. This can be used by an attacker to change messages. A MAC will prevent this type of attack. This property, that a change in the ciphertext will produce a known predictable change in the plaintext, is called *malleability*.

3.1 Brute Force Key Search

The basic attack against any symmetric key cryptosystem is the *brute force attack*. In this attack, the attacker keeps guessing what the key is until they guess correctly. In general, one known plaintext, or the ability to recognize a correct plaintext is all that is needed for this attack. However, all good cryptosystems should be designed such that this attack is impractical.

For a key of size n, a brute force search would consist of trying all 2^n keys to see which one works. If it is possible to recognize the correct plaintext, then on average the correct key will be found in 2^{n-1} guesses. Although this attacks works on all modern cryptosystems, its complexity grows exponentially with respect to the key size, to choosing an appropriately large key size will provide the needed security. Experts suggest having at least a 90 bit key for security in today's world.

Time/Memory Tradeoffs It is possible to speed up a brute force search by pre-computing some values and storing them in memory. This is trading off memory usage for time it takes to perform the attack.

Sampling Resistance In [2], an interesting property of stream ciphers is discussed in the context of time/memory tradeoffs of brute force attacks. This property, known as the *sampling resistance*, is how easy it is to find keys that generate keystreams with a certain property of the output.

3.2 Real or Random Distinguishers

In a theoretical model, the output of a "good" keystream generator should appear random. If the output was truely random, then the cipher would be a one-time-pad. However, since the output cannot be truely random, at some point in the keystream we should be able to distinguish between the real keystream generator and a truely random keystream generator.

It follows from this logic that the more keystream output that is needed to distinguish between the real cipher and random output, the closer that output is to being random, and the better the cipher. Therefore, a common academic attack model for stream ciphers is the *real or random distinguisher*. In this model, the goal is to come up with a distinguisher.

A distinguisher is a probabilistic polynomial-time algorithm A. A takes as input N bits of data, which are either from the real stream cipher or are completely random data. A then has to, in polynomial time, output either "Real" or "Random". If this is correct with a non-negligible probability, then this is a good distinguisher.This N indicates how good the distinguisher is. In most cases, larger N will produce a more accurate distinguisher.

In [3], a distinction is made between two different type of distinguishers. The first type of distinguisher is what has already been discussed, and is called a *polynomial-time distinguisher*. The other type is what is known as a *polynomial-space distinguisher*. For this type of distinguisher, the attacker is given a block box which is either the cipher or a true random generator. This black box can be reset and rerun with a random key a polynomial number of times. The goal is to distinguish between the two black boxes. According to [3], these two notions are equivalent from the information-theoretical viewpoint, though the difference in bias can be significant.

3.3 Key Weaknesses

There are also some more specific models that should be discussed, because they are directly relevant for the discussion on the security of **RC4**. These are related to the key. Specifically, the key completely determines the output sequence from a keystream generator. In a good keystream generator, each bit of the output will depend on the entire key for its value, and the relationship between the key and a given bit (or set of bits) should be extremely complicated.

Key-Biased Output The first condition listed in §3.3 is every bit of the output is dependent on the entire key for its value. This means that changing any single bit of a key should have a $\frac{1}{2}$ probability of affecting each bit of the output.

When this property holds, then in order to brute force a key, every possible key must be tried, and there will be no relationship between bits of the key and the output. This means that uncertainties of the individual bits of the key multiply when calculating the total number of possible keys.

Let us see what happens when this property does not hold. Assume that the 8 bits of the output is dependent only on the first 8 bits of the key. This means that for a given value $q = k_1 \dots k_8$ of the first 8 bits, all keys that have the value q for the first 8 bits will have the same first 8 bits output.

How can this be used to decrease the brute force search time? If you first guessed all possible values of q, you could check if they are correct by comparing the output to the first 8 bits of the output. Once you determine q, then you can brute force the other 2^{n-8} bits of the key. This total time required to brute force this key would be $2^8 + 2^{n-8} \approx 2^{n-8}$. This caused a factor of 256 reduction in the amount of work necessary to brute force the key.

Thus, any way in which the output is baised by a subset of the bits of the key, this information can be used to mount an attack on the cipher.

Related Key Attacks The second condition listed in §3.3 is that the relationship between the key and each bit of the output should be complex. What this means is that a given known relationship between two keys should not produce a known relationship in the output of the keystream generator. This kind of information can also be used to provide an attack by reducing the effective brute keysearch time. These kind of attacks are known as *related key attacks*.

For this purpose an example also serves well. Assume we have a keystream generator that has the property that the bitwise compliment of the key produces the bitwise compliment of the keystream. This information can be used to (slightly) speed up a brute force search. For each keystream generated by a guess of the key, compliment the keystream and see if that one generates the correct plaintext. If it does, then the compliment of the key is correct. In this way, you never have to check the compliment of any key you have already checked. This reduces the search space by a factor of two.

Use of Initialization Vectors One problem with binary additive keystream ciphers is that the same key will always produce the same keystream. This means that repeatedly using the same key is just as bad as reusing a one-time-pad. To solve this problem, the concept of initialization vectors is useful.

An *initialization vector* (IV) is a random value that changes with every instance of the cipher that is used to add some randomness to the output of the cipher. Since this value is random and unique, it makes the output of the stream cipher different than other outputs, even if the same key is used. This is useful when key exchange is expensive.

In ciphers with a variable size key, a common method of adding an IV to a cipher is to combine it with the real key in some fashion. For example, prepending a small IV to the real key to form a larger session key is quite common. Another way of using an IV is to encrypt the IV with the real key, or encrypt the real key with the IV, and use this encrypted value as a session key.

It is important to recognize the proper role of the IV. In this kind of usage, the initialization vector is not part of the secret key, and does not need to be kept secret. This means that it can be transmitted in the clear to the recipient. However, making sure it is random can help prevent precomputation-based time/space tradeoffs for brute force attacks.

However, the use of IV's can be security weakness, depending on how the IV is used. A good example is any cipher in which partial knownledge of the key together with some ciphertext can be used to find more of the key. If this is true, then just prepending the IV to the key to form a session key can leak information about the real key. **RC4** suffers from this problem.

4 RC4

RC4 is a very popular cipher from RSA Data Security, Inc. It was designed by Ron Rivest of M.I.T. It is a trade secret of RSA, but was leaked to a number

of mailing lists and newsgroups in the early-mid 90's. People with access to real implementations of RC4 have confirmed its authenticity. It was described in [4].

RC4 is used in a number of applications currently. One of its most important uses is in SSL (also known as TLS), which is used to secure most of the worlds electronic commerce over the world wide web. It is also used in WEP, the IEEE 802.11 wireless networking security standard. It can also be found in a number of other applications including email encryption products.

4.1 Description

RC4 is a binary additive stream cipher. It uses a variable sized key that can range between 8 and 2048 bits in multiples of 8 bits (1 byte).

This means that the core of the algorithm consists of a keystream generator function. This function generates a sequence of bits that are then combined with the plaintext with XOR. Decryption consists of re-generating this keystream and XOR'ing it to the ciphertext, undo'ing it.

The other major part of the algorithm is the initialization function,n which accepts a key of variable size and uses it to create the initial state of the keystream generator. This is also known as the key schedule algorithm.

RC4 is actuall a class of algorithms parameterized on the size of its block. This parameter, n, is the word size for the algorithm. This is recommended $n = 8$, but for analysis purposes it can be convenient to reduce this. Also, for extra security it is possible to increase this value.

The internal state of **RC4** consists of a table of size 2^n words and two word-sized counters. The table is known as the S-box, and will be known as S. It always contains a permutation of the possible 2^n values of a word. The two counters are known as i and j.

The Key Schedule Algorithm of **RC4** is shown in Figure 1. It accepts as input the key stored in K, and is l *bytes* long. It starts with the identity permutation in S and, using the key, continually swapping value to produce a new unknown key-dependent permutation. Since the only action on S is to swap two value, the fact that S contains a permutation is always maintained.

Initialization:
    For $i = 0$ to $2^n - 1$
        $S[i] = i$
    $j = 0$
Scrambling:
    For $i = 0$ to $2^n - 1$
        $j = j + S[i] + K[i \bmod l]$
        $Swap(S[i], S[j])$

Fig. 1. RC4 Key Schedule Algorithm

The **RC4** keystream generator is shown in Figure 2. It works by continually shuffling the permutation stored in S as time goes on, each time picking a different value from the S permutation as output. One round of **RC4** outputs an n bit word as keystream, which can then be XOR'ed with the plaintext to produce the ciphertext.

Initialization:
$i = 0$
$j = 0$
Generation Loop:
$i = i + 1$
$j = j + S[i]$
$Swap(S[i], S[j])$
Output $z = S[S[i] + S[j]]$

Fig. 2. RC4 Pseudo Random Generation Algorithm

4.2 Observations

- S starts as a permutation (the identity permutation) of the 2^n possibly words. Since the only operation on S is *Swap*, S remains a permutation. However, this permutation changes with time. This is where the strength of **RC4** comes from.
- The internal state is stored in $M = n2^n + 2n$ bits. However, since S is a permutation, this state holds effectively $\log_2(2^n!) + 2n \approx 1700$ bits of information.
- Knowing the entire M bits of the state at a given time is enough to predict all of the keystream bits in the future.
  - Knowing the entire initial state is enough to predict all of the keystream bits, effectively breaking the cipher
- The initial state is dependent solely on the key K. Therefore knowing the key is sufficient to break **RC4**.
- Although the KSA is reversable, it is difficult to determine the key K from the initial state of S. However, this is unnecessary to break the cipher.
- The key completely and uniquely determines the keystream.
- The period of **RC4** is difficult to predict, and dependent on the key. However, empirical evidence from [5] suggests that the period is normally very long.

5 State Progression Attacks

This is the first of two sections discussing the analysis of **RC4**. This section discusses attacks based solely on the pseudo-random generator algorithm. This include mainly distinguishers based on observations about the progression of

states as the algorithm runs. The next section deals with attacks based on the key schedule.

This section has four parts. The first is the state guessing attack developed in [5] and [6]. This forms a basis for attacks, and is used occasionally in the other attacks. The second section discusses a very interesting sampling bias in the second output word, discussed in [3]. The final section discusses a generalization of the *fortuitous states* defined in [7] and defines predictive states, discussed in [3].

5.1 State Guessing

In [6], a sophisticated attack on **RC4** is developed. This idea behind this attack is to guess some of the initial state of the **RC4** keystream generator, and by looking for contradictions in the keystream it is possible to detect incorrect guesses and to discover the rest of the initial state.

In this discussion, let i_t and j_t be the values of the two counters at time t. Also, let S_t be the state at time t. Therefore, the output of **RC4** at time t is

$$z_t = S_t[S_t[i_t] + S_t[j_t]] \tag{4}$$

This notation simplifies talking about the state as it progresses through time.

Let us begin our analysis by looking at simplified versions of **RC4**. Consider **RC4** without any swap operation. (This only applies to the PRGA). This means that the state S_t is the same for all time values. Denote this S. This leads to a quick theorem:

Theorem 1. *If the swap operation of **RC4** is omitted, the keystream becomes cyclic with a period of* 2^{n+1}

Proof. By Equation 4, $z_{t+2^{n+1}} = S[S[i_{t+2^{n+1}}]+S[j_{t+2^{n+1}}]]$. Because of the modular addition, $i_{t+2^{n+1}} = i_t$. Since S is constant, the step $j_t = j_{t-1}+S[i_t]$ can be applied repeatedly. This gives us $z_{t+2^{n+1}} = S[S[i_t]+S[j_t+\sum_{u=0}^{2^{n+1}-1} S[u]]]$. Since S is a permutation, the sum of all its elements is $\frac{(2^n-1)(2^n)}{2} = 2^{n-1} \pmod{2^n}$. Therefore, $\sum_{u=0}^{2^{n+1}-1} S[u] = 2 \cdot 2^{n-1} = 0 \pmod{2^n}$. So, $z_{t+2^{n+1}} = z_t = S[S[i_t] + S[j_t]]$. □

Looking at Equation 4, there are up to four unknowns. At any time t, i_t is always known. For a known plaintext attack, z_t is also known. The four variables that are possibly unknown are j_t, $S[i_t]$, $S[j_t]$, and $S^{-1}[z_t]$. From any three of these variables, the fourth can be calculated:

$$j_t = S^{-1}[S^{-1}[z_t] - S[i_t]] \tag{5}$$

$$S[i_t] = S^{-1}[z_t] - S[j_t] \tag{6}$$

$$S[j_t] = S^{-1}[z_t] - S[i_t] \tag{7}$$

$$S^{-1}[z_t] = S[i_t] + S[j_t] \tag{8}$$

The algorithm to recover S works as follows. Initially, guess a small subset of the values of S. Use equations 5-8 as time t progresses to derive additional values of S. If a contradiction arises, then the initial guess was incorrect. Repeat this process for all possible guesses.

i_0 and j_0 are both known (0) when the algorithm starts. If the guesses of S guess the first x values of S, then j_0 through j_{x-1} is known. For each of these, equations 7 and 8 can be used to determine more values of S. Once i_t goes past x, if $S[x+1]$ is not known, we lose the knowledge of j_t. We discard the next values of z_t until we can use Eq. 5 to discover j_t again. We can do this when $S[i_t]$ is known, the value z_t appears in what is known of S, and the value $S^{-1}[z_t] - S[i_t]$ also appears in what is known of S. Once j_t is recovered, we can then work backwards using 6 to recover more entries of S. Once we have finished working backworks, we continue as we started, using Eq. 7 and Eq. 8 to discover values of S until we lose the value of j_t.

Using this algorithm, the entire state of this variant of **RC4** can be determined. This is a fairly simple attack and does not require much time at all. Others variants studied in [6] include having a reduced swap frequency, where the swap operation is only executed once every 2^n iterations. From the results in [6], a swap frequency of 2^{-7} requires 40 correctly guessed values, and a swap frequency of 2^{-1} requires 240 correctly guessed values of S. (These numbers are for a success ration of 50%)

Full RC4 A modification of this attack can be used to successfully attack the full **RC4**. In this modification, no values of S are guessed initially, but are only guessed as needed. Since S is a permutation, this is useful to limit the number of possibilities for each guess, because the value cannot be one that is already in the table at the time. This can be implemented efficiently by a recursive function. The running time of this algorithm is calculated exactly in [6] and is $O(\sqrt{2^n!})$.

5.2 Second Byte Bias

There has been a good bit of analysis of the probabilities of any given value being output by **RC4**. Most of these analyses have approached **RC4** by looking at a given output. Mantin and Shamir, in [3], approach this differently. They looked for a polynomial-space distinguisher. In this search, they came up with a startling finding.

Theorem 2. *If* $S_0[2] = 0$ *and* $S_0[1] \neq 2$*, then* $Z_1 = 0$ *with probability* 1.

Proof. Initially, i and j are 0. i is incremented and now points to $S[1]$. Denote $S_0[1] = X$. Then j is incremented to $0 + S_0[1] = X$. So now j points to X. Denote the value $S_0[X] = Y$. X and Y are swapped in the permutation, and the value $S[X+Y]$ is output as the first output. Then i is incremented to 2. j is incremented to $X + S[2]$ which by assumption is $X + 0 = X$. The the values of X and 0 (the value in position 2) are swapped. The second output is then $S[X+0] = S[X] = 0$. The second assumption is needed to insure that the zero is not swapped out of its position before it is output. □

Theorem 3. *Given a random initial state, the value of the second output of* ***RC4*** *is* 0 *with probability* $\frac{2}{2^n}$.

Proof. When $S_0[2] = 0$, the probability that a zero is output is approximately 1. When $S_0[2] \neq 0$, then the probability that a zero is output is essentially random, meaning $\frac{1}{2^n}$. Thus, the total probability that the second output is zero is

$$\begin{aligned} P[z_1 = 0] &= P[z_1 = 0 | S_0[2] = 0] \cdot P[S_0[2] = 0] + P[z_1 = 0 | S_0 \neq 0] \cdot P[S_0 \neq 0] \\ &\approx 1 \cdot \frac{1}{2^n} + \frac{1}{2^n} \cdot \left(1 - \frac{1}{2^n}\right) \\ &= \left(1 + 1 - \frac{1}{2^n}\right) \cdot \frac{1}{2^n} \\ &\approx \frac{2}{2^n} \end{aligned} \tag{9}$$

which is twice the expected probability. □

An interesting (but not yet useful) observation is that by applying Bayes rule, we can predict the value of a single word in the state with a non-negligible probability.

$$\begin{aligned} P[S_0[2] = 0 | z_1 = 0] &= \frac{P[S_0[2] = 0]}{P[z_1 = 0]} \cdot P[z_1 = 0 | S_0[2] = 0] \\ &= \frac{\frac{1}{2^n}}{\frac{2}{2^n}} \\ &= \frac{1}{2} \end{aligned} \tag{10}$$

This means that when the second output is 0, we can guess the value of $S_0[2]$ with probability $\frac{1}{2}$. Unfortunately, no one seems to know how to use this to speed up an attack.

A Polynomial-Space Distinguisher A polynomial-space distinguisher can be constructed from this information. However, a theorem should be proven first.

Theorem 4. *Let* X,Y *be probability distributions. Supposed event* e *happens in* X *with probability* p*, and happens in* Y *with probability* $p(1 + q)$*. Then, for small* p *and* q*, the number of samples needed to distinguish between* X *and* Y *is* $O\left(\frac{1}{pq^2}\right)$.

Proof. Let X_e and Y_e be random variables specifying the number of occurances of e in t samples. The expectations, variances, and standard deviations are (assuming small values of p and q):

$$\begin{aligned} E[X_e] &= tp & E[Y_e] &= tp(1+q) \\ V(X_e) &= tp(1-p) \approx tp & V(Y_e) &= tp(1+q)(1-p(1+q)) \approx tp(1+q) \\ \sigma(X_e) &= \sqrt{V(X_e)} \approx \sqrt{tp} & \sigma(Y_e) &= \sqrt{V(Y_e)} \approx \sqrt{tp(1+q)} \approx \sqrt{tp} \end{aligned}$$

Our goal is to find the number t of samples required to have one standard deviation between the two expectations:

$$\begin{aligned} E[Y_e] - E[X_e]. &\geq \sigma(X_e) \\ tp(1+q) - tp &\geq \sqrt{tp} \\ tpq &\geq \sqrt{tp} \\ t^2p^2q^2 &\geq tp \\ t &\geq \frac{1}{pq^2} \end{aligned}$$

Therefore, $O\left(\frac{1}{pq^2}\right)$ values suffice for distinguishing betwen X and Y. □

Let X be the probability distribution of the second word in randomly generated output, and ley Y be the same probility for **RC4**. Let the event e be that the second word output is 0. This happens in X with probability $\frac{1}{2^n}$, and in Y with probability $\frac{2}{2^n}$. By using Theorem 4 with $p = \frac{1}{2^n}$ and $q = 1$, we can conclude that we need $\frac{1}{pq^2} = 2^n$ outputs to reliably distinguish between **RC4** and random.

5.3 Predictive States

Fluhrer and McGrew in [7] and Mantin and Shamir in [3] define a class of states in which a non-negligable bias appears in the keystream. These collectively are known as *predictive states*. But first, some definitions.

Definition 1. *An a-state is a partially specified state that includes values for i, j, and a values in S. Note that the values in S are not necessarily consecutive.*

Definition 2. *Let A be an a-state. If all **RC4** states that are compatible with A have the same output word after r rounds, then A is said to predict its r^{th} output.*

Definition 3. *Let A be an a-state. If there exists some $r_1, \ldots, r_b$ such that A predicts the outputs of rounds $r_1, \ldots, r_b$, then A is called b-predictive.*

Therefore, a b-predictive a-state is a state where b outputs are predicted accurately by just the a values in the state. The interesting part is that the b values are not necessarily consecutive, and to not necessarily follow directly after a. It is hypothesized that b-predictive a-states only exist when $b \leq a$.

Distinguishers Any b-predictive a-state can introduce some bias in the output of **RC4**. Denote the event that the current state is compatible with a given a-state A as E_A. Denote the event that the outputs in rounds $r_1, \ldots, r_b$ are those predicted by A as E_B. E_A contains $a + 2$ values (i, i, and a values of S) that thus has a probability of approximately $(2^n)^{-(a+2)}$. Whenever E_A occurs,

so does E_B. If E_A does not occur, then E_B occurs with probability $(2^n)^{-(b+1)}$. (based on the choice of i and the b outputs) Thus, the probability of E_B is:

$$\begin{aligned} P[E_B] &= P[E_B|E_A] \cdot P[E_A] + P[E_B| \neq E_A] \cdot P[\neq E_A] \\ &\approx 1 \cdot (2^n)^{-(a+2)} + (2^n)^{-(b+1)} \left(1 - (2^n)^{-(a+2)}\right) \\ &= (2^n)^{-(b+1)} \left(1 - (2^n)^{-(a+2)} + (2^n)^{b+1-(a+2)}\right) \\ &\approx (2^n)^{-(b+1)} \left(1 + (2^n)^{b-a+1}\right) \end{aligned} \tag{11}$$

In this case, a new theorem is:

Theorem 5. *For an b-predictive a-state, there exists a distinguisher that distinguishes **RC4** from random that requires $O\left((2^n)^{2a-b+3}\right)$ output words.*

Proof. By applying Theorem 4 with $p = (2^n)^{-(b+1)}$ and $q = (2^n)^{b-a+1}$, the number of output words required to use this distinguisher to distinguish from random is $O\left(\frac{1}{pq^2}\right) = O\left((2^n)^{2a-b+3}\right)$ output words. □

Note that the second byte bias in §5.2 is a 1-predictive 1 state.

Attacks with Predictive States Applying Bayes rule like we did in §5.2, we achieve another interesting result:

$$\begin{aligned} P[E_A|E_B] &= \frac{P[E_A]}{P[E_B]} \cdot P[E_B|E_A] \\ &\approx \frac{(2^n)^{-(a+2)}}{(2^n)^{-(b+1)}} \\ &= (2^n)^{b-a-1} \end{aligned} \tag{12}$$

In this case, E_A is the internal event (that the state is a-predictive), and E_B is the external event (that the outputs were as predicted). Thus, when we observe E_B, we can conclude with probability $(2^n)^{b-a-1}$ that the internal state is compatible with E_A. If we see $(2^n)^{-(b-a-1)}$ occurrences of E_B, then at least one is likely to be an occurance of E_A.

Let us try to construct an attack based on this. We continually observe the output stream for b-predictive a-state (of which we have a list somehow). Whenever we find the predicted b outputs of one of these states, we will in the appropriate a values in the state and run the Branch and Bound attack from §5 with this partial information to try and determine the rest of the state. This can provide a reduction in the amount of searching the Branch and Bound attack needs to do. With enough occurances of the appropriate predicted outputs, one of the b-predicted output sequences is likely to correspond to the correct a-state.

Intuitively, the value a determines the quantity of information that is revealed, but it also decreases the probability of it occuring. As such, the attack

is limited to small values of a. Also, the value $b - a$ determines the number of false positive that have to be examined. Thus, reducing this reduces the time complexity of the attack. Working with the hypothesis that $b \leq a$, the best predictive state are a-predictive a-state, for appropriately small values of a.

6 Key Schedule Weaknesses

6.1 Related Key Analysis

In [8], Grosul and Wallach analyze **RC4** in a related key attack model. In this model, they determinet that keys related in a certain way cause very similar outputs for the first few output bytes.

Consider the difference in the initial state of a one byte difference in a full key. That is, $K'[i] = K[i]$ except where $i = t$, when $K'[i] \neq K[i]$. Up until the point t, the key schedule (and the state) will be the same, but at point t the value of $j = (j + S[i] + K[i])$ will be different for the two keys, and the remaining key schedule will be completely different. If t is small, then the resulting initial states for the two keys will be completely different. However, if t is near 2^n, then the initial states will be very similar.

For notation, let j_t be the value of j at the t^{th} step in the key schedule. Consider two complimentary changes in the key to produce a related key: $K'[i] = K[i] + \delta$ and $K'[i+1] = K[i+1] - \delta$. In this case, $j'_t = j_t + \delta$ and most likely $j'_{t+1} = j_{t+1}$. This is called *twiddling* the key at position t, and K and K' are related keys.

So, what happens if the swap on the t iteration affects future values of j or j'? This happens whenever $j_t > t$ or $j'_t > t$. The probability of this occuring is $1 - \frac{t}{2^n}$.

Unless $j_t = t + 1$ or $j'_t = t + 1$, immediately following the twiddle $j' + t + 1 = j_{t+1}$. This is a *good twiddle*, meaning only three values in S differ from those in S' after the t^{th} step. Following a good twiddle, the key schedule will be identical until $S[i] \neq S[i]$. At this point, the two key schedules will diverge.

6.2 Key-Biased Output

In [9], the first half of the paper describes a set of weak keys in which a certain subset of the key bits completely determine a subset of the output bits. First, a couple of definitions must be made. Then, the attack will be explained. For the first bit, we will work with a variant of the **RC4** Key Schedule Algorithm known as KSA^*. See Figure 3 for the definition. The primary difference is that i is incremented at the beginning of this loop instead of the end (or equivalently, i is initialized to 1).

Definition 4. *Let S be an RC4 State table, t be an index in S and b be some integer. If $S[t] \equiv t \pmod{b}$, then the state S is said to b-conserve the index t.*

Definition 5. *Let $I_b(S)$ be the indices that the state S b-conserves.*

Initialization:
For $i = 0$ to $2^n - 1$
$S[i] = i$
$i = 0$
$j = 0$
Scrambling:
Repeat N times
$i = i + 1$
$j = j + S[i] + K[i \bmod l]$
$Swap(S[i], S[j])$

Fig. 3. The Key Schedule Algorithm Variant KSA^*

Definition 6. *A state S is said to be b-conserving if $I_b(S) = 2^n$. A state S is said to be almost b-conserving if $I_b(S) = 2^n - 2$.*

Definition 7. *Let b, l be integers such that $b|l$. Let K be a key of l words. K is called b-exact if for all indices t, $K[t \mod l] \equiv (1 - t) \pmod b$. In the case where $K[0] = 1$ and the most significant bit of $K[1]$ is 1, K is called a special b-exact key.*

Notes on Key Schedule Now on to prove some facts about these states and keys.

Lemma 1. *If $i_{t+1} \equiv j_{t+1} \pmod b$, then $I_b(S_{t+1}) = I_b(S_t)$.*

Proof. Only operations that modify S can change I. Therefore, the only operation to consider is the swapping operation. When $i_{t+1} \equiv j_{t+1} \pmod b$, then S_t either b-conserves both indices i_t and j_t or neither. Therefore, the swap operation does not affect the number of indices that S b-conserves. □

Theorem 6. *Define $b = 2^q$ for some $q \leq n$. Suppose that $b|l$ and K is a b-exact key of l words. The the permutation $S = KSA^*(K)$ is b-conserving.*

Proof. The proof of Theorem 6 is by induction on t. We will prove that for any $0 \leq t \leq 2^n$, $I_b(S_t) = 2^n$ and $i_t \equiv j_t \pmod b$. This implies that $I_b(S_{2^n}) = 2^n$ and therefore S_{2^n} is b-conserving.

For $t = 0$, the basis case, the claim is trivial, since $i_0 = j_0 = 0$ and the identity permutation is b-conserving for every b. Now assume that $i_t \equiv j_t \pmod b$ and that S_t is b-conserving. Now:

$$i_{t+1} = i_t + 1 \tag{13}$$

$$j_{t+1} = j_t + S_t[i_{t+1}] + K[i_{t+1} \mod l] \equiv i_t + i_{t+1} + (1 - i_{t+1}) \pmod b \tag{14}$$

Therefore, $j_{t+1} \equiv i_{t+1} \pmod b$. Then by Lemma 1, $I_b(S_{t+1}) = I_b(S_t)$. □

This is a really interesting result. This means that a small subset of the bits of K (lq bits) completely determines a large number of bits in the initial state ($q2^n$ bits).

Difference between *KSA* and *KSA** The primary difference between KSA and KSA^* is that KSA does not even preserve the equivalence of i and j after the first round. However, if K is a special b-exact key, then $K[0] = 1$, and the value of j after the first round is forced to be 1. Therefore, the equivalence between i and j is preserved except for the first round. Therefore, for a special b-exact key, the resulting intial permutation is almost b-conserving much of the time. If the value of i reaches one of the non-b-conserving indices, then the value of j will diverge from $i \pmod b$. So, if the most significant bit of $K[1]$ is 1, then the non-b-conserving values will be put in the second half of the permutation, where j has plenty of chance to meet it and swap it to before i before i gets to it.

[9] claims that it can be proven that the probability that a special b-exact key produces an almost b-conserving initial permutation is greater than $\frac{2}{5}$, and in practice is ususally greater than $\frac{1}{2}$.

Output correlation

Theorem 7. *Let **RC4*** be a weakened version of **RC4** with no swap operations. Let S_0 be a b-conserving initial permutation with $b = 2^q$, $q \leq n$. Define $\{X_t\}_{t=1}^{\infty}$ be the output sequence of **RC4*** using S_0 as the initial state. Also define $x_t = X_t \bmod b$. Then the sequence $\{x_t\}_{t=0}^{\infty}$ is predictable and periodic with period $2b$.*

Proof. The value of $S_0 \bmod b$ is known, as well as $i = j = 0$, the initial values of the counters. Since S never changes, and i, j, and S are all known $\bmod b$, the PRGA without swaps can be simulated $\bmod b$ accurately. With deeper analysis (see [9]), it can be shown that the period is $2b$. □

Since in each step of the **RC4** swaps at most two values of S, the S permutation remains very similar to the S permutation in **RC4***. Therefore, for a substantial number of outputs of **RC4**, the output will mirror that of **RC4***.

This fact can be used to form a distinguisher for **RC4** from random for a subset of the possible keys. For these keys, the outputs of **RC4** will be significantly similar to the outputs of **RC4***, much more so than in the random distribution. Since the number of predetermined bits is linear in l, the size of the bias is also dependent on l. The analysis in [9] suggests that a distinguisher for **RC4** can be built using this fact that can distinguish **RC4** from random in 2^{21} output words for 64 bit keys.

Another interesting observation is that the biases in the least significant bits of the output can be combined with the biases in the least significant bits of english text to provide a ciphertext-only distinguisher.

Related Key Attack This provides a good attack in a rather non-standard attack model. In this model, the attacker is given a black box that has **RC4** with a given key K in it. The attacker can give the black box a value, Δ, and the black box will generate output using $K' = K \oplus \Delta$ as the key. The goal of this model is to determine K.

This attack works in n stages, where in stage q the Q^{th} bit of every key word is exposed. This attack has two algorithms that is uses. The first is the *CheckKey* algorithm. This algorithm accepts the **RC4** black box, a value Δ and a parameter q, and determines whether the output of the black box is special 2^q-exact. This can be done in $O(1)$ time (see [9]). The second algorithm is *Expand*, which accepts a black box and a Δ which form a special 2^{q-1}-exact key and makes it special 2^q exact. This is accomplished by enumerating all possibilities of the q^{th} bits of each key word and invoking *CheckKey* to determine if it is a special 2^q-exact key. There are 2^{l-1} possibilities for these bits. For $q = 1$, this is a little different. *Expand* with $q = 1$ works by determining the entire $K[0]$ (by setting it to 1) and setting the most significant bit of $K[1]$ to 1. for $q = 1$, the time complexity of *Expand* is $O(2^{n+1})$, and for all other values of q it is $O(2^{l-1})$.

Since there is only one special 2^n-exact key, the value of K' is known for the special 2^n-exact key, and therefore K can be calculated. The time complexity of this attack is $O(2^{n+l} + n2^{l-1}) = O(2^{n+l})$, which is significantly faster than brute force ($O(2^{2^n!})$).

6.3 Initialization Vector Weaknesses

This section deals with attacks on **RC4** using an initialization vector. Specifically, for brevity, this section will focus on the case where the IV precedes the secret key. The case where the IV follows the secret key is covered in [9], as well as the attack presented here.

First, some notation is necessary. Let i_t and j_t be the values of the two counters after t steps of KSA, and also S_t be the state of the permutation after t steps of KSA.

The first word output from **RC4** is dependant on only three values of S.

Definition 8. *Let $X = S_i[1]$ and $Y = S[X]$. In KSA, if $i \geq 1$, $i \geq X$, and $i \geq X+Y$, and $S_i[1]$, $S_i[X]$, and $S_i[X+Y]$ are all known, then the situation is called resolved. In this case, the first word output from **RC4** will be $S[S[1] + S[S[1]]]$ with probability greater than $e^{-3} \approx 0.05$.*

This probability comes from the probability that none of the three relavent values will be disturbed in the rest of the key schedule. Since the value of i has already passed the three values in the resolved condition, only by j pointing to one of those values can they be disturbed. Modelling j as a random value for the rest of the key schedule, the probability is the probability that j never equals one of those three specific values.

Let us consider the case where a 3 word IV is prepended to a secret key to form the session key. If K' is the secret key and IV is the IV, then $K = IV||K'$ is the session key. Since the IV is 3 words long, and the secret key is l words long, the session key is always $3+l$ words long. The secret key values are $K[3] \ldots K[l+2]$, and the (known) IV values are $K[0] \ldots K[2]$.

Consider the case where we known the first A values of the secret key ($K[3] \ldots K[A+2]$). Examine a series of IV's of the form $(A+3, -1, X)$, where X is a random value.

In the first step, j is advanced by $S_0[0] = A + 3$. On the next step, i is advanced, and the advance on j is computed to be $j_1 = j_0 + S_0[1] + K[1] = j_0 + 1 + (-1)$. Therefore, j is not moved. Then $S[i]$ and $S[j]$ are swapped. On the next step, j is advanced by $X + 2$. From here on until $i = A + 3$, the key schedule can be computed. Since all the X's are different, each IV acts differently. At this point ($i_{A+2} = A + 2$), the value of j_{A+2} and of the entire permutation S_{A+2} are known. At this point, if either $S[0]$ or $S[1]$ have been disturbed, the attacker throws out this IV. We also know $S_{A+2}[A+3]$. The value that will be $S_{A+3}[A+3]$ depends on $j_{A+3} = j_{A+2} + S_{A+2}[A + 3] + K[A + 3]$, which is two knowns and one unknown. $S_{A+3}[A + 3] = S_{A+2}[j_{A+3}]$. This is a state in a (pseudo) resolved condition. Therefore, with probability approximately 0.05, none of the three important values will be touched by the remaining key schedule, and the first output word will be $S[A + 3]$.

In this case, it is possible to calculate $K[A + 3]$. Let's expand some formulas and see if we can derive it:

$$\begin{aligned} z_0 &= S[S[1] + S[S[1]]] \\ &= S[0 + S[0]] \\ &= S[A + 3] \\ &= S_{A+2}[j_{A+3}] \\ &= S_{A+2}[j_{A+2} + S_{A+2}[A + 3] + K[A + 3]] \end{aligned} \tag{15}$$

At this point, we know S_{A+2}, j_{A+2} and z_0 (the first output word). Therefore we can calculate $K[A + 3]$ as:

$$K[A + 3] = S^{-1}_{A+2}[z_0] - j_{A+2} - S_{A+2}[A + 3] \tag{16}$$

If we use approximately 60 IV's of the form above, we can calculate secret key byte A. If we iterate this attack over all the secret key bytes (l bytes), it takes approximately $60l$ chosen IV's and their corresponding first output byte to determine the entire secret key.

Generalization of the IV's First, let us see what conditions on the 3-byte IV's are necessary to make this attack work. The two most important conditions are:

1. We know the state S_{A+2} in its entirety.
2. The first output byte is $S_{A+3}[A + 3] = S[S[1] + S[S[1]]]$. This leads to the condition that $S_3[1] + S_3[S_3[1]] = A + 3$.

Also, for the resolved condition to hold, the values at time 3 must all be less than 3. Therefore, the additional condition of $S[1] < 3$ must also hold. Condition one always holds if we know the IV.

These three conditions are sufficient to carry out this attack. The number of IV's that fit these criteria is much larger than the number of IV's that fit our previous criteria. Therefore, a known IV attack, where the attack does NOT get to choose the IV's, is greated accelerated by using these criteria.

As a matter of fact, none of these criteria make significant use of the fact that the IV is 3 words long. This can be generalized to attack an IV of any size. There criteria for IV's of length I words are:

1. $S_I[1] + S_I[S_I[1]] = I + B$
2. $S_I[1] < I$

In this case, Equation 16 can be generalized to be:

$$K[A] = S^{-1}_{I+B-1}[z_0] - j_{I+B-1} - S_{I+B-1}[I + B] \tag{17}$$

where $K[A]$ is the A^{th} byte of the secret key (not including the IV).

WEP WEP, the Wireless Encryption Protocol, uses **RC4** with a prepended 3 byte IV for its encryption. Also, the first byte of the plaintext is always a known value. This attack has been successfully used to attack WEP encryption, requiring roughly 5,000,000 packets with the same key to determine that key. This can be gotten quite easily and is considered a major weakness in this system.

7 Summary

In this paper stream ciphers where explained and a number of attacks models were described. **RC4** was defined, and six attacks were developed against this algorithm: the Branch and Bound attack, the Second Word Bias, Predictive States, the Derailing Related Keys attack, the Special Exact Keys and the Initialization Vector Weakness. Overall, **RC4** is still considered secure if you use a hash function to form session keys from secret keys and IV's, and discard the first 2^n words of output before use.

References

1. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Chapter 6. In: Handbook of Applied Cryptography. CRC Press (1997) pp. 191–195 `http://www.cacr.math.uwaterloo.ca/hac/`.
2. Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of a5/1 on a pc. In: Fast Software Encryption (FSE 2000). LNCS, SpringerVerlag (2000) `http://www.cs.berkeley.edu/~daw/papers/a51-fse00.ps`.
3. Mantin, I., Shamir, A.: A practical attack on broadcast rc4. In: Fast Software Encryption (FSE 2001). LNCS, SpringerVerlag (2001) `http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/bc_rc4.ps`.
4. Schneier, B.: Chapter 17. In: Applied Cryptography. Second edn. John Wiley and Sons, Inc. (1996) pp. 397–398
5. Mister, S., Tavares, S.: Cryptanalysis of rc4-like ciphers. In Tavares, S., Meijer, H., eds.: Selected Areas of Cryptography (SAC '98). Volume 1556 of LNCS., SpringerVerlag (1999)

6. Knudsen, L., Meier, W., Preneel, B., Rijmen, V., Verdoolaege, S.: Analysis method for (alleged) rc4. In Ohta, K., Pei, D., eds.: Advances in Cryptology, Proc Asiacrypt '98. Volume 1514 of LNCS., SpringerVerlag (1998) http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Knudsen.ps.
7. Fluhrer, S., McGrew, D.: Statistical analysis of the alleged rc4 key stream generator. In Schneier, B., ed.: Fast Software Encryption (FSE 2000). LNCS, SpringerVerlag (2000) http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/FluhrerMcgrew.pdf.
8. Grosul, A.L., Wallach, D.S.: A related-key cryptanalysis of rc4. http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/GrosulWallach.ps (2000)
9. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the key scheduling algorithm of rc4. In: Selected Areas of Cryptography (SAC 2001). LNCS, SpringerVerlag (2001) http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Rc4_ksa.ps.
10. Golic, J.D.: Linear statistical weakness of alleged rc4 keystream generator. In Fumy, W., ed.: Advances in Cryptology, Proc. Eurocrypt '97. Volume 1233 of LNCS., SpringerVerlag (1997) pp. 226–238 http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Golic.PDF.

Index