# Design and Analysis of RC4-like Stream Ciphers 

by<br>Matthew E. McKague

A thesis<br>presented to the University of Waterloo<br>in fulfillment of the thesis requirement for the degree of Master of Mathematics<br>in<br>Combinatorics and Optimization

Waterloo, Ontario, Canada, 2005
© Matthew E. McKague 2005

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Matthew E. McKague


#### Abstract

RC4 is one of the most widely used ciphers in practical software applications. In this thesis we examine security and design aspects of RC4. First we describe the functioning of RC 4 and present previously published analyses. We then present a new cipher, Chameleon which uses a similar internal organization to RC4 but uses different methods. The remainder of the thesis uses ideas from both Chameleon and RC4 to develop design strategies for new ciphers. In particular, we develop a new cipher, RC4B, with the goal of greater security with an algorithm comparable in simplicity to RC4. We also present design strategies for ciphers and two new ciphers for 32-bit processors. Finally we present versions of Chameleon and RC4B that are implemented using playing-cards.


## Acknowledgements

This thesis was undertaken under the supervision of Alfred Menezes at the University of Waterloo. The cipher Chameleon was developed under the supervision of Allen Herman at the University of Regina. Financial support was provided by the University of Waterloo, the University of Regina and the National Science and Engineering Research Council of Canada (NSERC). I would like to thank my supervisors for their support and contributions and NSERC for providing funding. I would also like to thank Kishan Gupta for his comments and Iko for putting up with me when I was thinking of RC4 all the time.

## Contents

1 Introduction ..... 1
1.1 History of RC4 ..... 1
1.2 History of Chameleon ..... 2
1.3 Outline and Contributions ..... 2
2 Description of RC4 ..... 4
2.1 A Simple Algorithm ..... 4
2.2 General Description ..... 4
2.3 The Internal State ..... 5
2.4 State Change Function ..... 7
2.5 Output Function ..... 8
2.6 The Key Schedule ..... 8
2.7 Discussion ..... 8
3 Previous Attacks on RC4 ..... 10
3.1 Types of Attacks ..... 10
3.1.1 Distinguishers ..... 10
3.1.2 Shortcut Attacks ..... 11
3.1.3 Related and Weak Key Attacks ..... 11
3.2 Knudsen's Attack ..... 12
3.3 Predictive States ..... 14
3.3.1 Uses of Predictive States ..... 14
3.3.2 Fluhrer and McGrew's Distinguisher ..... 16
3.3.3 Mantin and Shamir's Broadcast Attack ..... 17
3.3.4 Predicting Attacks on RC4 ..... 18
3.4 Biases in the Key Schedule ..... 19
3.4.1 Roos's Class of Weak Keys ..... 19
3.4.2 Key Schedule Invariance ..... 20
3.4.3 Related Key Attack Based on Weak Keys ..... 22
3.5 IV Weaknesses ..... 23
3.5.1 Methods of Using IVs ..... 23
3.6 Other Attacks and Analysis ..... 25
3.6.1 Mironov's Analysis ..... 25
3.6.2 Finney's States ..... 27
3.6.3 Golic's Distinguisher ..... 28
3.6.4 Paul and Preneel's Distinguisher ..... 28
3.7 Conclusions ..... 29
4 Chameleon: A New Cipher and a New Cryptographic Prim- itive ..... 30
4.1 The Mutating S-box ..... 30
4.1.1 Feedback in Block Ciphers ..... 31
4.1.2 Modelling with Finite State Machines ..... 31
4.1.3 A Finite State Machine Cryptographic Primitive ..... 32
4.1.4 Security ..... 33
4.1.5 Attacks ..... 33
4.1.6 Uses in Protocols ..... 37
4.2 Chameleon: An Example Cipher ..... 38
4.3 Analysis ..... 42
4.4 Attacks on Chameleon Inspired by RC4 ..... 43
4.4.1 Similarities Between RC4 and Chameleon ..... 43
4.4.2 Branch and Bound Attacks ..... 43
4.4.3 IV Weakness ..... 44
4.4.4 Key Schedule Invariance Weakness ..... 45
4.4.5 Biases and Other Attacks ..... 46
4.5 Conclusions ..... 46
5 Modifying RC4 ..... 47
5.1 Comparisons ..... 47
5.2 A New Viewpoint ..... 48
5.3 RC4B ..... 51
5.3.1 Key Schedule ..... 51
5.3.2 RC4B Round Algorithm ..... 53
5.4 RC4 Attacks on RC4B ..... 53
5.4.1 Predictive States ..... 54
5.4.2 Key Schedule Attacks ..... 54
5.4.3 Early Keystream Attacks ..... 54
5.4.4 Knudsen's Attack ..... 55
5.4.5 Other Modified RC4 Ciphers ..... 55
5.5 Conclusion ..... 57
6 Expanding RC4 to 32 Bits ..... 59
6.0.1 Parallelizing RC4 ..... 59
6.0.2 32-Bit Versions of RC4 ..... 60
6.1 RC4 as a Pool of Entropy ..... 61
6.2 A First Try ..... 62
6.2.1 Description ..... 62
6.2.2 Usefulness ..... 63
6.2.3 A Weakness ..... 63
6.3 A More Secure Version ..... 64
6.4 Sheet Bend: A 32-Bit Cipher ..... 65
6.4.1 Description ..... 65
6.4.2 Security of the PRNG ..... 66
6.4.3 Efficiency ..... 68
6.4.4 Further Modifications ..... 69
6.4.5 Analysis ..... 70
6.5 Bowline: Another 32-Bit Cipher ..... 71
6.6 Comparing Sheet Bend and Bowline ..... 72
6.7 Conclusions ..... 72
7 Card Game Ciphers ..... 74
7.1 Card-Chameleon ..... 75
7.1.1 Description of Card-Chameleon ..... 75
7.1.2 Analysis of Card-Chameleon ..... 76
7.2 Pocket-RC4 ..... 79
7.2.1 Description ..... 79
7.2.2 Security ..... 80
7.3 Conclusions ..... 80
8 Future Work and Conclusions ..... 82

## List of Figures

2.1 RC4 Key Schedule Algorithm ..... 5
2.2 RC4 Round Algorithm ..... 5
2.3 Complete RC4 Algorithm ..... 6
3.1 Branch and Bound Attack for RC4 ..... 13
3.2 Weakened RC4 Key Schedule Algorithm ..... 20
4.1 Soft-Chameleon Encryption Algorithm ..... 40
4.2 Soft-Chameleon Decryption Algorithm ..... 41
4.3 Chameleon Branch and Bound Attack Algorithm ..... 44
5.1 RC4B Key Schedule Algorithm ..... 52
5.2 RC4B Round Algorithm ..... 53
5.3 VMPC Stream Cipher ..... 56
5.4 RC4A Round Algorithm ..... 57
6.1 Sheet Bend Algorithm ..... 67
6.2 Bowline Algorithm ..... 73
7.1 Card-Chameleon Encryption Algorithm ..... 77
7.2 Card-Chameleon Decryption Algorithm ..... 78
7.3 Pocket-RC4 Algorithm ..... 81

## Chapter 1

## Introduction

This thesis is the result of the author's interest in two similar ciphers of independent origin. The first is the well known and often used RC4, developed by Ron Rivest. The second is a cipher of the author's own design, Chameleon. The similarity of the design between these two ciphers prompted the question of whether the considerable amount of analysis conducted on RC 4 could be of benefit in the analysis of Chameleon and whether design elements from Chameleon, together with elements from RC4, could be used to design new ciphers with greater security. The goal of this thesis is to explore these questions.

### 1.1 History of RC4

At the time of the development of RC 4 much of the research in stream ciphers was focused on so-called linear feedback shift registers, or LFSRs (see [16], page 196). These devices are easy to study from a mathematical point of view, making the analysis of their security an attractive topic. Unfortunately LFSRs use many bit operations, making them slow in software implementations.

RC4 was developed in 1987 by Ron Rivest, co-developer of RSA. (RC stands for "Ron's Code".) In contrast to LFSRs, RC4 uses byte operations that are more friendly to software implementations, especially on the 8 -bit and 16 -bit machines commonly available at the time. In addition, the algorithm is quite simple, making implementations compact and less error-prone than other algorithms such as DES (see [16], Chapter 7). This led to the use of RC4 in many software packages in addition to its use in standards such as Wired Equivalent Privacy (WEP) used in WiFi (a wireless networking
protocol) and the Cellular Digital Packet Data specification [12].
Rivest originally developed RC4 for RSA Security Inc. and it was kept as a trade secret until 1994, when source code was anonymously leaked on the Cypherpunks mailing list. Since then much public research has been done and many attacks have been found.

### 1.2 History of Chameleon

Chameleon was designed by the author during the Summer of 2004 under the supervision of Allen Herman at the University of Regina with funding provided by an NSERC USRA ${ }^{1}$. Its development started with an interest in the idea of developing a hand cipher that uses a deck of cards as its only necessary equipment with a security level comparable to computerized ciphers. During the development of Chameleon the author was aware of existence of RC4, but not of any details. It is perhaps surprising, then, that the core functioning of these two ciphers designed for very different environments is so similar.

### 1.3 Outline and Contributions

The main contributions of this thesis are a survey of the published analysis of RC4, a description and analysis of Chameleon, and the extension of the design principles of RC4 and Chameleon.

Chapter 2 contains a general description of RC4, its algorithm, and some general observations on its design. Chapter 3 gives further analysis by presenting the findings of the various papers on RC4. These papers describe both practical attacks and theoretical analysis.

Chapter 4 describes Chameleon. The design of Chameleon rests on a new cryptographic primitive called a mutating S-box. It is this primitive that contributes the similarity between Chameleon and RC4. In this Chapter we first describe the mutating S-box. A description of Soft-Chameleon, a variant designed to work in software, follows. Finally, the analysis of RC4 described in the previous Chapter is applied to Chameleon.

After describing both Chameleon and RC4, we examine ways in which they can be improved. In Chapter 5 we begin this examination by breaking down RC4 and Chameleon into smaller elements. From this analysis we produce a new cipher, RC4B. The goal behind this cipher is to use ideas

[^0]from both Chameleon and RC4 while increasing security against the attacks described in Chapter 3. In addition, we describe some other attempts to improve RC4.

In Chapter 6 we look at further modifying RC4 with the aim of increasing throughput on 32 -bit processors. We describe some new techniques that could be used to develop such ciphers. We also develop two new ciphers using these techniques.

In Chapter 7 we examine the context for which Chameleon was originally developed, hand ciphers using playing-cards. Here we describe CardChameleon, the playing-card version of Chameleon, along with a playingcard version of RC 4 B which we call Pocket-RC4.

In the final Chapter, we examine some of the possibilities for future work in this area and give some concluding remarks.

## Chapter 2

## Description of RC4

The RC4 algorithm is fairly simple and yet leads to a cipher that, despite much research and many attacks, is still secure enough for many applications. In this Chapter we give a detailed description of the algorithm and discuss some of the structures used. We also give a general analysis of the functioning of these structures.

### 2.1 A Simple Algorithm

Before a more abstract consideration of RC4 we present the algorithm in its entirety. $i$ and $j$ are $n$-bit words and $S[]$ is an array of $N=2^{n} n$-bit words indexed by the values 0 to $N-1$. All additions are carried out modulo $N$. The initial set-up phase, called the Key Schedule Algorithm takes a key $k[]$ consisting of $l n$-bit words. It is described in Figure 2.1. After the set-up, the round algorithm (see Figure 2.2) is executed once for each word output.

In virtually all practical applications RC 4 is implemented with $n=8$ in which case all the entries of $S$ along with $i$ and $j$ are bytes.

To see the algorithm in its entirety, see Figure 2.3

### 2.2 General Description

RC4 is somewhat unique in the world of stream ciphers because of its peculiar internal structure, in particular its internal state. We can think of RC 4 as a finite state machine that contains some internal state information, a state change function that depends on the current state, and an output function that depends on the internal state. The machine is triggered externally causing it to first execute the state change function and then return

Input: Key $k$ of $l$ words

1. For $i=0$ to $N-1$

$$
S[i]=i
$$

2. $j=0$
3. For $i=0$ to $N-1$
(a) $j=j+S[i]+k[i \bmod l] \bmod N$
(b) Swap $S[i]$ and $S[j]$.
4. $i=j=0$ Output: $i, j, S$

Figure 2.1: RC4 Key Schedule Algorithm

$$
\begin{aligned}
& \text { Input: RC4 state } S, i, j \\
& \text { 1. } i=i+1 \bmod N \\
& \text { 2. } j=j+S[i] \bmod N \\
& \text { 3. Swap } S[i] \text { and } S[j] \\
& \text { 4. Output } S[S[i]+S[j] \bmod N]
\end{aligned}
$$

Figure 2.2: RC4 Round Algorithm
the result of the output function. This general framework can be used to describe almost any stream cipher, and in the case of RC4 provides a nice picture of how the various parts function with each other.

### 2.3 The Internal State

RC4 operates on binary words of length $n$. On each tick of the clock RC4 outputs one word which can take any of the $N=2^{n}$ possible values. The internal state of RC4 is built around a random permutation of these $N$ values. This permutation, which we denote $S$ can be thought of in several ways. Those implementing RC4 would typically think of $S$ as an array or look-up table while those who come from a background of block ciphers might think of $S$ as an S-box. We want to be able to do two things with $S$, namely apply $S$ to a word and multiply $S$ by a transposition. We more typically use the

## Input: Key $k$ of length $l$

1. For $i=0$ to $N-1$

$$
S[i]=i
$$

2. $j=0$
3. For $i=0$ to $N-1$
(a) $j=j+S[i]+k[i \bmod l] \bmod N$
(b) swap $S[i]$ and $S[j]$.
4. $i=j=0$
5. Repeat the following for each byte of output required
(a) $i=i+1 \bmod N$
(b) $j=j+S[i] \bmod N$
(c) Swap $S[i]$ and $S[j]$
(d) Output $S[S[i]+S[j] \bmod N]$

Figure 2.3: Complete RC4 Algorithm
look-up table terminology where these operations are equivalent to looking up a value in the table and swapping two entries. The entire internal state consists of $S$ and two additional variables, $i$ and $j$.

The size of the internal state is an important factor for a stream cipher since it provides an upper bound on the complexity of a brute force attack. In the case of RC4 the state size is very large. In terms of the number of possible states it is $2^{2 n}(N!)$, which makes for $\log _{2} N!+2 n$ bits of information. In the case when $n=8$ this is about 1700 bits. This makes guessing the internal state in a brute force manner completely impractical. Determining the 8 bits of $i$ is easy but the rest is not. Another important factor is that the size of the internal state puts an upper limit on the cycle length. In particular, the cycle length (i.e. the number of outputs before the whole pattern repeats) cannot be more than the number of possible states. In this respect having a large internal state is again an advantage.

In addition to $S$ the internal state of RC 4 contains variables $i$ and $j$ which can be thought of as pointers into the look-up table $S$. Their role is critical since they control both the state change and output functions.

### 2.4 State Change Function

The state change function modifies each of $i, j$ and $S$. It is accomplished by steps 1,2 and 3 in the round algorithm.

First $i$ is incremented. When considered over $N$ rounds this means that $i$ points to every element in $S$ exactly once. This means that the swap operation in step 3 modifies the whole of $S$ after $N$ steps, ensuring a fairly rapid rate of mixing. Note that the movement of $i$ is not dependent on $S$ and is set to a constant during the set-up phase. Thus $i$ can be considered public knowledge.
$j$ is not incremented, but has $S[i]$ added to it. Considering $S$ as a pseudorandom function, $j$ can be thought of as a pseudo-random variable that points to different elements of $S$ in an unpredictable manner. Importantly, this variable depends on $S$, making it private knowledge. It also depends on the previous value of $j$, meaning that partial knowledge of $S$ at some point does not necessarily expose $j$.

The swap operation in step 3 forms the core of the state change function. This can be thought of as multiplying the permutation $S$ by a transposition. Another way of thinking about it is to see $S$ as a deck of cards and the transposition as a small shuffle. Over time the deck becomes more and more randomized as individual cards are shuffled. Since the movement of $j$ and the output are both dependent on $S$, this randomization helps create unpredictable behaviour in the output as well as the internal evolution.

It should be noted that the state change function in RC 4 is invertible. That is to say, the current state and the state change function are enough to determine previous states. Invertability is desirable since contributes to long cycle lengths. Intuitively if the state change function is not invertible then information is lost with each application. The amount of information in the output stream is not more than what is contained in the state, so the entropy in the output must decrease over time leading to behaviour that is less random. More formally, a randomly chosen state change function results in an average cycle length of about $2^{m / 2}$ where the internal state contains $m$ bits of information. For a randomly chosen invertible state change function, the average cycle length is $2^{m-1}[7]$. Although this is no guarantee of security or even of cycle length, being invertible lends a little credit to the claim of security.

### 2.5 Output Function

The output function is quite simple. If we again consider $S$ as a pseudorandom function, we first find two pseudo-random numbers, $S[i]$ and $S[j]$, and then add them and wrap the whole thing in $S$ again. These multiple levels have several uses. First, $S$ is not actually random so it will have some biases which could in principle be detected, leading to exposure of information about the internal state. To smooth out these biases we apply $S$ several times. Also, we use both $i$ and $j$ to eliminate any information that could be gleaned about $S$ based on the known input $i$, or about $j$ given some partial information about $S$. Also, we perform an addition, a linear operation that mixes the whole words but is hard to analyse after the application of a non-linear operation.

### 2.6 The Key Schedule

The round algorithm of RC 4 is not dependent on any key and requires a special kind of state. The obvious solution to these inconveniences is to make the initial state dependent on the key and use an algorithm that prepares the necessary state. The key schedule algorithm does just this.

The algorithm begins by initializing $S$ to the identity permutation and setting $j$ to zero. From there it applies what is essentially a modified version of the state change function a total of $N$ times. The only difference is how $j$ is incremented. For this we add a key byte to $j$ at each round, cycling over the length of the key. This particular method allows for a wide range in key lengths. Also, using the same principles as in the state change function makes the key schedule easy to analyse.

It should be noted that the key schedule finishes by setting $i=j=0$. This is purely convention since any value between 0 and $N-1$ would suffice. In fact, one could leave $j$ as its final value after swapping occurs, which could possibly increase security by introducing more key dependancies. In particular, several attacks described below on the beginning of the RC4 output rely on the fact that $j$ is set to a constant. Their effectiveness is suspect if $j$ changes between keys.

### 2.7 Discussion

RC4 has attracted much attention because of its simple algorithm that invites analysis. The simplicity of the four line round algorithm also makes for
ease of implementation, reducing the likelihood of errors. As well, the number of operations required is quite small, making it efficient. Further, only byte operations are required making it efficient when implemented on small processors such as those found in smart cards, and the memory requirements are reasonable.

On the other hand, it is not easy or efficient to parallelize RC4 and the byte operations do not take advantage of the wider busses available on newer processors. As a result there has been interest recently in adapting the algorithm to make use of wider busses. For one proposal, see Chapter 6.

Finally, we note that although $N$ is almost universally specified in the literature as being a power of 2 , and most often 256 , this is not necessary, but more of a convenience. (If $N=2^{n}$ then reduction modulo $N$ can be accomplished by masking off all but the the lower $n$ bits. If $N$ is some other value this is not possible.) In fact, in Chapter 7 we describe a variant of RC4 that uses $N=27$.

## Chapter 3

## Previous Attacks on RC4

Because of its prominence in applications and its novel structure, RC 4 has been the subject of much research. In this Chapter we explore the literature on RC4, discussing the attacks that have been developed as well as some insights into RC4's structure.

### 3.1 Types of Attacks

As a stream cipher, RC4 is supposed to mimic the behaviour of random strings. This means that any attack which distinguishes RC4 outputs from random strings can be considered a vulnerability. In addition, any attack which reveals partial or complete key information is a vulnerability. From an information theoretical point of view, any cipher with a small key is vulnerable to both these types of attacks, but only careful analysis has produced attacks on RC4 that are computationally feasible.

### 3.1.1 Distinguishers

Consider the following situation: An attacker is given a black box that outputs data. The attacker is told that it either implements RC4 with a fixed secret key or is a random source. The attacker "wins" if he can guess the nature of the black box with probability better than half. The amount of computation required as well as the number of output words provide metrics on the difficulty of mounting an attack. An attack that aims to win in this scenario is called a weak distinguisher. There have been many attacks of this type. The most important is that of Fluhrer and McGrew[6].

Another class of distinguishers, call strong distinguishers has a slightly
different set-up. The attacker is given a black box as above except that it has one external control. If the black box implements RC4 then this control causes a new key to be chosen at random. Subsequent outputs are then the keystream generated by the new key. If the box is a random source then the control does nothing. Again the attacker's goal is to guess whether the box implements RC 4 or a random source with probability better than one half. The difficulty of mounting such an attack can be measured by the number of new keys necessary in addition to the amount of computation and data required. This type of scenario allows us to check for biases introduced as a result of a weak key schedule or other peculiarities that affect the beginning of a keystream. Mantin and Shamir [14] developed the first and most important of these attacks.

### 3.1.2 Shortcut Attacks

Distinguishing a keystream from a random string has theoretical interest, but most often does not lead to a complete break of the cipher. For this it is most often necessary to find the key or the internal state of the cipher. A brute force attack on the key is always an option but there exists the possibility that some peculiarity of the algorithm may allow recovery of internal state information in a way that speeds up the search. We call such attacks shortcut attacks. The branch and bound attack often referred to as Knudsen's attack is the most successful shortcut attack on RC4. In addition, this attack can be augmented by using the biases that are exploited in distinguisher attacks.

### 3.1.3 Related and Weak Key Attacks

It is often the case that an otherwise secure cipher can be successfully attacked by analysing its key schedule. Such attacks fall into several categories. Weak key attacks identify a class of keys that cause a detectable difference in the output of the cipher. Related key attacks identify a relationship between keys that causes a detectable similarity in the two outputs. These attacks can be used to gather information about the secret key by examining the output and either identifying the characteristic output of a class of weak keys (hence reduce the number of keys to search) or comparing the output with known keys, looking for a relationship with the secret key. Andrew Roos [22] developed one of the first such attacks on RC4: a set of weak keys causing a bias in the initial outputs.

Another means of attacking the key schedule of a cipher is to use partial
information about a key to derive information about the output or additional information about the key. The famous attack on WEP, developed by Mantin, Fluhrer and Shamir [5], is of this type.

### 3.2 Knudsen's Attack

When considering the possibility of a brute force attack to determine the internal state of RC4, there are several considerations which are helpful. First, the internal state at the beginning of the algorithm has $i$ and $j$ fixed at zero and there are $N$ bytes contained in $S$. There are in fact fewer than $N$ bytes of information contained in $S$ since not all possible assignments are allowed (only permutations). Thus we expect that $N$ bytes of output should contain enough information to reconstruct the state. If this is not the case then there must be some bias in the output that accounts for the lack of information. This is probably not the case since $i$ points to every element of $S$ in the first $N$ outputs, causing these outputs to be dependent on every element of $S$. Based on this reasoning we expect to be able to construct an algorithm that determines the state given $N$ words of output.

Another important observation is that the round algorithm does not require complete knowledge of $S$. In fact, only three elements of $S$, namely $S[i], S[j]$ and $S[S[i]+S[j]]$, are affected or used by any particular round of $S$. Thus we don't have to guess the complete state at once in order to test against some given output.

The above two observations give rise to an algorithm that dramatically reduces the time necessary to guess the internal state of RC 4 given some output. This algorithm was independently developed by Knudsen et al [10] and Mister and Tavares [18]. It is a branch and bound type algorithm. It can begin with some partial information about $S$ or with nothing and will guess the remaining elements of $S$ until a state that is consistent with the output is found. Figure 3.1 presents the algorithm found in [18]. The input to the algorithm is an output stream $c_{t}$.

The basic idea behind the algorithm is to guess information as late as possible. If some partial state proves to be inconsistent with the given output stream, then we can eliminate all states that contain this partial state. We can think of a tree whose nodes represent partial state information at times where a guess is required, edges as guesses, and leaves as complete states. Each time a contradiction is found, a whole subtree and all associated states are eliminated from consideration. Also, the output stream provides some information about the state directly, if the correct index is known or guessed.

Input: Portion of RC4 stream $c_{0}, c_{1}, c_{2}, \ldots$

1. Mark all entries of $S$ as unassigned.
2. Set $i=0, j=0$ and $z=0$
3. Repeat the following
(a) $i=i+1$
(b) If $S[i]$ is unassigned, branch over all possible values of $S[i]$
(c) $j=j+S[i]$
(d) If $S[j]$ is unassigned, branch over all possible values of $S[j]$
(e) Swap $S[i]$ and $S[j]$
(f) Set $t=S[i]+S[j]$
(g) If $S[t]$ is unassigned and $c_{z}$ is not assigned in $S$, set $S[t]=c_{z}$
(h) If $S[t] \neq c_{z}$ then there is a contradiction. Close this branch.
(i) $z=z+1$
(j) If $z$ equals the length of the output, terminate and output $S$.

Figure 3.1: Branch and Bound Attack for RC4

There are many implementation details required to make this algorithm practical. In particular, schemes for keeping track of branches and possible values for $S$ are required. The author implemented this attack using several recursive functions and a table indicating whether or not each value was already assigned in $S$.

Knudsen et al, in [10] provide some minor improvements to the algorithm and estimate the complexity of the attack. They estimate that the number of steps required is typically less than $\sqrt{N!}$. The actual number given for $n=8$ is $2^{779}$, as compared to $2^{1684}$ required for a brute force attack.

Knudsen's attack, although impractical due to the high complexity, is a useful theoretical tool since it can be modified to take partial state information and some output bytes and provide a complete state. Thus if some other attack exposes some state information, it can be used in conjunction with this attack to provide a complete break. Also, if probabilistic information about the state is known, this can be used to choose the order in which the algorithm branches, choosing the most likely branches first and accelerating the algorithm.

### 3.3 Predictive States

The observation was made by Mantin and Shamir [14] that, because only a few entries of $S$ are used in each round, a relatively small amount of information about the state can be used to predict output information. In certain circumstances the amount of information predicted is larger than expected because the elements of $S$ can be used more than once over several rounds. There have been several results that make use of these quirks in order to attack RC4. Mantin and Shamir [14] develop a general description of attacks that use these quirks. Previously, Fluhrer and McGrew [6] discussed the same ideas as Mantin and Shamir in a limited context. Before describing these results we present some terminology introduced by Mantin and Shamir.

Definition 1. An a-state is a partially specified RC4 state consisting of values for $i, j$ and a entries of $S$.

Definition 1 quantifies the amount of information that is specified in some partial state. Note that there are typically many states that are consistent with a particular $a$-state. A particular $a$-state may have the property that the RC 4 algorithm, when run with any state consistent with the $a$-state, always has certain outputs. In this case we say that the $a$-state predicts these outputs. The amount of output information predicted is quantified in the following two definitions:

Definition 2. If all states consistent with a given a-state cause the same output to be produced in the r-th position then the a-state is said to predict the $r$-th output.

Definition 3. If an a-state predicts outputs $r_{1} \ldots r_{b}$ where $r_{1}, \ldots, r_{b} \leq 2 N$ then we say that the $a$-state is $b$-predictive.

In general we are interested in $b$-predictive $a$-states where $b$ is close to $a$. In this situation the maximum amount of output information can be determined by the information contained in the $a$-state. It was conjectured by Mantin and Shamir [14] that $a \geq b$ for any $b$-predictive $a$-state. Paul and Preneel [19] gave a formal proof.

### 3.3.1 Uses of Predictive States

Predictive state, in particular those where $a$ is close to $b$, are useful because they cause biases in the output. First, we expect that our $b$ outputs will
occur with probability $N^{-b}$ at any particular point in a random stream. If RC 4 is unbiased, then this means that $N^{-b}$ of the states will produce our $b$ outputs in the necessary positions. However, we know that every state compatible with our $a$-state produces this output and, since there are $a+2$ constraints in the $a$-state, these states account for about $N^{-a-2}$ of the total states (for small $a$ ). In addition, we expect about $N^{-b}$ of the remaining states to have the predicted output. This means that the total fraction of states that cause the predicted output is about

$$
N^{-a-2}+\left(1-N^{-a-2}\right) N^{-b}=N^{-b}\left(1+N^{b-a-2}-N^{-a-2}\right) .
$$

This bias in the states translates directly to a bias in the output. Given enough output, the output bias can be detected using a statistical test.

Another use is to extract state information from the output. We have that the probability of the predicted output is $N^{-b}\left(1+N^{b-a-2}-N^{-a-2}\right)$, the probability of the $a$-state occurring is $N^{-a-2}$, and the probability of the predicted output given the $a$-state is 1 . Using Bayes rule we calculate the probability of the $a$-state occurring, given that the predicted output has occurred as

$$
\frac{N^{b-a-2}}{\left(1+N^{b-a-2}-N^{-a-2}\right)} .
$$

The denominator can be bounded by $1+N^{-2}$ since $b \leq a$. Thus the probability is close to $N^{b-a-2}$ as opposed to the trivial $N^{-a-2}$. This information can be used to improve the branch and bound algorithm. The idea is to keep a database of $a$-states and their predicted outputs, comparing outputs at each time with these predictions. If a predicted output occurs, then any branches that occur at that time choose information from the corresponding $a$-state first, since it has a high probability of being correct.

In their original description of this attack, Mantin and Shamir [14] describe an attack using a subclass of predictive states known as fortuitous states. These are $a$-predictive $a$-states where the first $a$ outputs are predicted. They used these states since they are easier to enumerate and find, allowing for an analysis of the complexity of the attack. They estimate that for $n=8$ the time required is about $2^{755.2}$ rather than $2^{779}$ for the original attack as estimated by Knudsen et al (see [10]).

Paul and Preneel [19] further analysed the attack, developing methods for discovering general predictive states. Their analysis shows that fortuitous states in general account for the bulk of predictive states. As an example, there are 2973 -predictive 3 -states for $n=8,290$ of which are fortuitous
states. Thus widening the attack to the general case of predictive states is not expected to cause a significant improvement in running time. The estimated decrease in running time when considering the additional 73 states is about 2 percent.

### 3.3.2 Fluhrer and McGrew's Distinguisher

Fluher and McGrew [6] developed a weak distinguisher based on digraph frequencies in the output of RC4. They discovered the a priori distributions of pairs of consecutive output words (digraphs) and found biases. This was done by calculating the number of states that are consistent with a particular digraph. The calculations are only feasible for small $N$, but the results proved to be sufficient to discover some biases that are independent of $N$ and uncover their mechanisms. The results were then extended to larger $N$.

Fluhrer and McGrew identified twelve digraphs ( 00 is one example) whose probabilities are different from the trivial probability and which occur regardless of the value of $n$. Two of these digraphs were predicted by specific partial states that depend on the values of $i$ and $j$. Each of these states specified three values in $S$ and can be seen as a 2 -predictive 3 -state. In addition, eight more digraphs that occurred with a positive bias were attributed to classes of 2 -predictive 2 -states (again, dependent on $i$ and $j$ ). Finally, two digraphs occur with less than expected probability. These are not explained in terms of predictive states, but in terms of states that fit a general pattern expected to produce a particular output that fails because of the particular values used.

The biases in the frequency of the twelve digraphs were used to construct a weak distinguisher. It is estimated that the size of the output required to reliably distinguish random output from RC 4 output is $2^{30.6}$ for $n=8$. This could potentially be improved by examining all digraph frequencies, but experiments on smaller $n$ revealed that the improvement would be small. For example, for $n=5$ about $2^{18.76}$ words of output using only the twelve digraphs while using all digraphs improved this to $2^{18.62}$. It was also suggested that trigraph frequencies could improve this number, however the computations required to discover the a priori probabilities is prohibitive.

The terminology of fortuitous states was developed by Fluhrer and McGrew in this same paper. Their approach, looking at digraph probabilities, led to an examination of the states that immediately predict output values instead of predicting later outputs. Interestingly, most of the observations made by Mantin and Shamir [14] about predictive states were also made by Fluhrer and McGrew about fortuitous states. In particular, they observed
that these states cause biases that can be used to create a distinguisher (the main result in their paper), that predicted outputs allow one to guess internal state information with non-trivial probability, and that this could be used to speed up Knudsen's attack. In addition, they developed a means of enumerating the number of fortuitous states of various lengths.

### 3.3.3 Mantin and Shamir's Broadcast Attack

Mantin and Shamir's discussion of predictive states [14] stems from an examination of the main result of the paper, a strong bias in the second output word. Interestingly enough, this doesn't result from a predictive state in their definition, but does satisfy a relaxed definition where it is allowed that the predicted output does not occur with some low probability. The bias is explained by the following result.

Theorem 1 (Bias in second output word). If the initial state of $R C 4$ satisfies $S[2]=0$ and $S[1] \neq 2$, then the second output word will be 0 .

Theorem 1 can be easily checked by the reader by applying the state change algorithm. Note that this bias is not really a predictive state since it only specifies what $S[1]$ cannot be. We can modify the theorem to specify only $S[2]=0$ and say that with high probability the output is 0 , in which case the conditions satisfy a relaxed definition of predictive state.

The bias in the output of the second word can be estimated by observing that there is one constraint on the state $(S[1] \neq 2$ occurs with probability close to 1 , so we are only concerned with $S[2]$.) Thus for about $\frac{1}{N}$ of the states the output will be 0 . For the other $1-\frac{1}{N}$ states the output will be 0 with probability about $\frac{1}{N}$ assuming a uniform distribution of outputs. The resulting probability is about $\frac{2}{N}$, which is double the trivial probability.

This bias can be used in several ways. The first and perhaps least useful way is to extract information about the initial state. The probability of being correct when the predictive state is guessed is about one half, and about $n$ bits of information are gathered, making this attack fairly weak, although it can be used to speed up Knudsen's attack. A more interesting use is in a strong distinguisher. Mantin and Shamir claim that by examining the second word output in about 200 streams (from different keys) RC4 can be reliably distinguished from random sources ( RC 4 will show more 0 s in these samples.) This is in sharp contrast with the $2^{30.6}$ required for Fluhrer and McGrew's distinguisher.

The main result of the paper is that if the same message is encrypted with many keys, then the bias can be used to identify one byte of plaintext.

The broadcast attack works by finding the frequencies of all second output words and guessing the most frequently occurring word as the plaintext. (XORing with 0 doesn't change the plaintext.) It should be noted that there are practical applications where the requirements for this attack are met. In particular, broadcast email and groupware applications could potentially encrypt the same message under many keys.

### 3.3.4 Predicting Attacks on RC4

In a recent paper, Mantin [13] extended his earlier work on predictive states. The first of these extensions is a distinguisher that examines the probability that a certain digraph occurs twice interspersed with a small number of outputs. The result is a weak distinguisher that correctly identifies RC4 streams with 0.9 probability using about $2^{29}$ output. This distinguisher can also be modified to correctly identify RC4A (described in Section 5.4.5) using a similar amount of output.

A more interesting extension of predictive states described in the new paper is recyclable states. These are predictive states that, once they occur, have a high probability of occurring again $N$ rounds later. Using these states an attacker can predict future outputs of the RC 4 stream when the outputs corresponding to a recyclable predictive state occur. This is the first attack that aims to predict future outputs with only partial information about the internal state.

In addition, it is possible to use recyclable states, as with normal predictive states, to speed up Knudsen's attack by providing known state data at particular points in the output. If about 100 elements are known then the attack is very fast. One way of learning so many elements is to wait for the predicted output of a 100 element predictive state. However, this is a low probability event, requiring about $2^{800}$ data before it is observed. If, instead, a recyclable state of 10 elements occurs and the predicted output occurs 10 times in the output then 100 elements of state information are learned (although at different times). Although the probability that the output repeats 10 times is low, it is still higher than the probability of a 100 element predictive state occurring. Mantin estimates that the required event happens about once in $2^{290}$ data, giving a significant speedup over the attack using predictive states.

### 3.4 Biases in the Key Schedule

In addition to insecurities in the round algorithm, there are some attacks that concentrate on the key schedule. In this Section we consider these attacks.

### 3.4.1 Roos's Class of Weak Keys

One of the earliest public results on RC 4 was discovered by Andrew Roos [22]. By examining the details of the key schedule algorithm, Roos was able to find a strong bias in the initial state which in turn causes a bias in the first few output words. This bias is in effect for about $\frac{1}{256}$ of keys (the analysis was done for $n=8$ ), producing a class of weak keys.

The bias stems from the fact that a particular state element is indexed by $j$ some time during the key schedule with probability about $1-\left(\frac{255}{256}\right)^{256}=$ 0.631 , assuming a uniform distribution on the values of $j$. This means that a particular element is swapped only once (when indexed by $i$ ) with probability about 0.37 . The value with which it is swapped depends of the value of $j$ and $S[j]$ at the time of the swap. Note that, early in the algorithm, the value of $S[j]$ probably has not been swapped out, so $S[j]=j$. Also, note that the $S[i]$, when added to $j$, has not previously been swapped by $i$ and thus probably has not been swapped and satisfies $S[i]=i$. If this holds for all the necessary elements of $S$ then we get $j=1+2+\ldots+i$ and $S[j]=j$. In this case, after the swap the value of $S[i]$ is

$$
\frac{i(i+1)}{2}+\sum_{m=0}^{i} k[m \bmod l] .
$$

With probability about 0.37 this value is retained to the end of the algorithm.
For small $i$ the probability that all the necessary events happen is quite high. Roos conducted experiments that show a steadily decreasing probability from 0.37 for $i=0$ to 0.06 by $i=46$. These probabilities show a very large bias when compared to the uniform probability of 0.004 .

In order to take advantage of the bias in the initial state, Roos constructs what are essentially predictive states: conditions on a small number of state elements that cause a particular output. In particular, we are interested in conditions on the first few elements of the state, since these have the highest bias. Roos describes one such condition, $S[1]=1$, in which case we get $i=j=1$ immediately before the swap. The swap does nothing and the output word is $S[2]$. The bias predicts that $S[1]$ will be $k[0]+k[1]+1$, so to
get $S[1]=1$ we want $k[0]+k[1] \equiv 0(\bmod N)$. In this case, the most likely value of $S[2]$, the first word output, is $k[2]+3$. Numerical experiments show that this happens in about 0.14 of keys satisfying the conditions, as opposed to the 0.004 that would be expected in a uniform distribution.

The bias caused in the first output byte is somewhat unique in that in provides information directly about the key rather than the initial state. This information can be used to speed up brute force attacks in cases where the biased output occurs.

### 3.4.2 Key Schedule Invariance

Fluhrer, Mantin, and Shamir [5] found a more sophisticated class of weak keys. This weakness depends on how $j$ is changed in the key schedule. The actual line in the algorithm is

$$
j=j+S[i]+k[i \bmod l] \bmod N
$$

Suppose that $i-1 \equiv j(\bmod b)$ for some $b, S[i] \equiv i(\bmod b)$, and $k[i \bmod$ $l] \equiv-i-1(\bmod b)$. Applying first two lines of the round algorithm we then get $j \equiv i(\bmod b)$. Now consider the slightly modified key schedule algorithm given in Figure 3.2. The only difference is that $i$ is updated at the beginning of the repetition instead of at the end.

Input: Key $k$ of length $l$

1. $i=j=0$
2. Repeat $N$ times
(a) $i=i+1 \bmod N$.
(b) $j=j+S[i]+k[i \bmod l] \bmod N$
(c) Swap $S[i]$ and $S[j]$.

Figure 3.2: Weakened RC4 Key Schedule Algorithm
Further, suppose that we have $k[x \bmod l] \equiv-x-1(\bmod b)$ for all $x$. We also have $S[x] \equiv x(\bmod b)$ at the beginning of the algorithm for all $x$ since $S$ begins as the identity permutation. After incrementing $i$ we have $i-1 \equiv j$ $(\bmod b)$. Thus $j \equiv i(\bmod b)$ after changing $j$. The two elements $S[i]=i$ and $S[j]=j$ will be swapped. After the swap we have $S[i]=j \equiv i(\bmod b)$ and $S[j]=i \equiv j(\bmod b)$. All of the preconditions mentioned above still
hold, so we can repeat this process $N$ times. After each repetition and, in particular, after the last repetition, we have $S[i] \equiv i(\bmod b)$.

Examining the original key schedule algorithm, the essential difference, from the point of view of the above property, is that $i \neq j$ initially. However, this can be repaired by using a slightly different key, described in the following definition, from [5].

Definition 4. Let $l$ and $b$ be integers and let $k$ be a key of length $l$. If $k[0]=1$, the most significant bit of $k[1]$ is 1 , and $k[i] \equiv i(\bmod b)$ for all $i \neq 0$, then $k$ is called a special b-exact key.

Note that in order to have a special $b$-exact key of length $l$ it is necessary that $b$ divides $l$.

If we have such a key and are using the original key schedule algorithm, then it is easy to see that the initial problem with $i$ and $j$ is corrected for the second repetition. However, the first swap affects $S[0]$ and $S[k[1]]=S[j]$, causing these two entries to violate the condition $S[i] \equiv i(\bmod b)$. We can argue, however, that with good probability this event will not cause problems. The likelihood of this problem being repaired is not good, but a significant portion of the time no more problems will be introduced.

First note that swap operations will not introduce more problems after $i$ and $j$ are syncronized. The only way that new problems will be introduced is if the update of $j$ uses an entry in $S$ that is problematic; such an event occurs when $i$ points to the value in question. $i$ will never point to $S[0]$ again, so unless $j=0$ at some point, it is unlikely to cause problems. For $S[k[1]], i$ will eventually reach it, but it is possible that $j$ will point to $S[k[1]]$ sometime before $i$ reaches $S[k[1]]$. In this case the problematic entry will be swapped out with the $i$ th value. After that, unless $j$ again points to it, it is 'behind' $i$ and will no longer cause problems. Since $k[1]$ is at least half way along (it has 1 as its most significant bit), the probability that $j$ will point to it sometime before $i$ reaches that position is better than half. Other unlucky events, like $j$ pointing to the bad entry again, reduce the probability of success somewhat. Fluhrer et al. calculate this probability as $\frac{2}{5}$.

In order to see how to use this special state, we first look at a modified round algorithm that doesn't include the swap operation. (We also ignore the two bad entries.) Examining the round operation we see that $i=j=0$ initially. After the first round we have $i=1$ and $j \equiv 1(\bmod b)$. After two rounds we have $i=2$ and $j \equiv 1+2(\bmod b)$. After $m$ rounds we have $i \equiv m$ $(\bmod N)$ and $j$ equivalent to the sum of the first $m$ integers, modulo $b$. The output value for the $m$-th round is $S[i+j]$ where $S[i+j] \equiv i+j \equiv m+\frac{m(m+1)}{2}$
$(\bmod b)$. This causes an obvious and predictable output (modulo $b$ ) with a short cycle length.

Reintroducing the swap operation and the bad entries, the predicted output is mostly destroyed. However, much of the state remains the same from round to round. Thus the predicted output will have a high correlation with the actual output, at least for the first few output words. Fluhrer et al. performed experiments to determine how well the predicted output fit with the actual output. Using $b=2$ and a special $b$-exact key, they report that 20 bits of output (the least significant bits of the first 20 outputs) were predicted with probability $2^{-4.2}$ instead of the trivial $2^{-20}$. Using $b=16,40$ bits (the lower 4 bits of the first 10 outputs) were predicted with probability $2^{-2.3}$ instead of $2^{-40}$.

### 3.4.3 Related Key Attack Based on Weak Keys

Suppose that we are given a black box implementing RC4. This box contains a secret key $k_{0}$ and allows two external operations. The first requests the next word in the output stream. The second resets the machine with a new key $k_{1}$ such that $k_{1}=k_{0} \oplus \Delta$, where $\Delta$ is input to the machine. Our goal is to devise an attack that discovers information about the secret key.

Fluhrer et. al. [5] devised such an attack, which works using key lengths of $2^{l}$ and special $2^{q}$-exact keys. It uses two subroutines. The first, called CheckKey takes as its input a parameter $q$ and a black box described above and analyses the first few words output from the box. It then decides whether the key was special $2^{q}$-exact or not and returns its decision as its output. This is a probabilistic operation but, as mentioned in the previous Section, the differences between random output and the output when using special exact keys is quite large, giving a good probability of success.

The second subroutine takes three inputs, an integer $q \leq n$, an RC4 black box as described above with secret key $k_{0}$, and a correction factor $\Delta$ such that $k_{0} \oplus \Delta$ is special $2^{q-1}$-exact. It operates by iterating over all possible values for the $q$-th bit in each key byte, forming a new $\Delta^{\prime}$ from $\Delta$, and checking whether the resulting key is special $2^{q}$-exact by giving $\Delta^{\prime}$ to the black box and invoking CheckKey. One such a key is found, the new $\Delta^{\prime}$ is output.

The above subroutine is modified for $q=1$. In this case it completely guesses $k[0]$ in order to force it to 1 . It also guesses the most significant bit of $k[1]$ as well as the least significant bit. These additional guesses are required in order to meet the requirements of a special exact key. Also, when $q=n$ there is only one special $2^{q}$-exact key, which can be easily determined. Thus
we can determine the output it causes and compare this directly with the output from the black box.

We use the above subroutine by invoking it for each $q$ from 1 to $n$. At each stage we determine the $\Delta$ that fixes the bottom $q$ bits of each word in the key in order to produce a special $2^{q}$-exact key. Note that there is only one possible value for these bottom bits that produces such a key. By the time we reach $q=n$ we have determined $\Delta$ such that $k_{0} \oplus \Delta$ gives the unique special $2^{n}$-exact key. Thus we have determined the secret key.

The complexity of this attack can be determined by the number of times that CheckKey is called. For $q=1$ it is called on average $2^{n+l}$ times. This happens once. For the remaining $n-1$ stages it is called on average $2^{l-1}$ times, for a total of $2^{n+l}+(n-1) 2^{l-1}$ times. For a particular $n$ and desired probability of success, CheckKey takes a constant amount of time. Thus the complexity is $O\left(2^{n+l}\right)$. For $n=8$ and $l=5$, giving a 256 -bit key this requires on the order of $2^{40}$ operations as opposed to $2^{256}$ required for a brute force attack.

It should be noted that $2^{40}$ operations is considered to be very practical given even modest computing power. Thus, from a computational standpoint, this attack poses a threat to the security of $R C 4$. However, the special situation required is unlikely to occur in practice. Moreover, the threat can be mostly eliminated by dropping the first $N$ output words for a new key, reducing the effectiveness of CheckKey.

### 3.5 IV Weaknesses

IV weaknesses constitute an interesting class of key schedule weaknesses that, in the case of RC4, lead to practical attacks. IVs, or initialization vectors, are public, message specific information strings that are used to obtain a session key from a secret key. This is done so that the same keystream is not used on two messages and yet only one secret key is needed. There are many different ways of obtaining a session key. There is no method specified by RC4, so these are perhaps not strictly speaking weaknesses in RC4. However they lead to the most practical attacks on actual implementations of RC4 and the only practical attacks that expose the secret key for $n=8$.

### 3.5.1 Methods of Using IVs

There are, of course, numerous ways of combining two data strings to obtain a third. Some of them lead to obvious relationships between the session and secret keys. A common procedure used with RC 4 is to concatenate the

IV with the secret key. Another is to XOR the two strings together. The simple relationships require a robust key schedule to destroy any information that could be gleaned about the secret key from the resulting keystreams. Unfortunately, the key schedule used in RC4 leaks information when these methods are used, resulting in attacks, described by Fluhrer et al. [5] that expose the secret key. A better method is to use a cryptographic hash function in order to obscure these relationships. (For example, concatenating the key with the IV and then hashing with SHA-1 to produce a session key.) There are no known attacks when such a method is used.

Before examining the attacks described in [5] we discuss some commonalities. Let $S_{i}$ represent $S$ at the $i$ th step. The first output word depends on exactly three variables: $X=S[0], Y=S[X]$, and $S[X+Y]$. Examining the key schedule, suppose that the algorithm reaches step $i$ and that $S_{i}[1]=X, S_{i}[1]+S_{i}\left[S_{i}[1]\right]=X+Y$. If none of the variables get swapped out in the remaining steps of the key schedule, then the first output word will be $S[X+Y]$. Fluhrer et al. call this a resolved condition and estimate that none of the critical variables are affected with probability about 0.05 . In the other cases one of the variables is set to a more or less random variable, causing a uniform distribution on the values of the output. The three attacks described in their paper use known values of certain of these variables to find information about the key.

## IV Concatenated with Key, IV Precedes Key

Suppose we are using a protocol wherein the IV is simply concatenated with the key, the IV coming first. This corresponds to knowing the first say $I$ bytes of the key input to the key schedule. Using this information, we run the key schedule up to step $I$ and examine the resulting state. Examining the state at this moment we see that in the next step

$$
S_{I}\left[j_{I+1}\right]=S_{I}\left[j_{I}+k[I]+S_{I}[I]\right]
$$

will be swapped out with $S_{I}[I+1]$. Thus after the swap we have

$$
S_{I+1}[I+1]=S_{I}\left[j_{I}+k[I]+S_{I}[I]\right] .
$$

$S_{I}$ and $j_{I}$ can be determined by running the key schedule with the known words of the key. We use the first output byte to determine the actual value of $S_{I}[I+1]$ which will provide enough information to deduce $k[I]$.

Suppose that after $I$ steps of the key schedule we have $S_{I}[1]<I, S_{I}[1]+$ $S_{I}\left[S_{I}[1]\right]=I+1$. Note that there is a chance (about 0.05 according to

Fluhrer et al.) that none of these values will be swapped out. This is because $i$ will not point to $S[1]$ or $S[S[1]]$ again since $S_{I}[1]<I$. They will only be swapped out if $j$ points to one of them. Assuming that they are not swapped out, the first word output will be $S[I+1]$. Again, assuming that $S[I+1]$ was not swapped out since step $I$, the first word output will be $S_{I+1}[I+1]$. This provides enough information to determine $k[I]$.

Using the above ideas an attack can be launched in the following way: First, collect several streams using different IVs and the first word output from each. Take the IV and, using it, run the key schedule algorithm forward until step $I$. Check to see if the necessary conditions hold. If they do, use the first word output along with the key schedule information to determine the first word of the key. Examining several such cases, there will be disagreement on what the value should be, but one will occur with higher frequency (since if a swap disturbing the conditions occurs then the first output should be essentially random). This, with high probability, will be the first key word. After this has been determined, we can add it to the end of the IVs and repeat, attempting to determine the next key word.

## IV Concatenated With Key, Key Precedes IV

In the case where the IV appears on the end of the session key, the analysis is similar, but more detailed. We refer the reader to [5] for a discussion.

### 3.6 Other Attacks and Analysis

In this Section we present some additional material that deals with specifics of the RC4 algorithm.

### 3.6.1 Mironov's Analysis

Much of the analysis of RC4 take one of two directions: assuming that the initial state is uniformly random, or carefully analysing the initial state for weaknesses. Taking a new approach, Mironov [17] creates a model of the shuffling mechanism used in the key schedule algorithm and analyses it to discover biases inherent in the method. As well, it has become a common recommendation to drop the first few output words in order to avoid Mantin and Shamir's broadcast attack. Mironov attempts to make recommendations on exactly how much output should be dropped in order to protect against possible attacks that take advantage of the nonuniform
initial state. (Note that Mantin and Shamir's attack does not depend on the key schedule.)

Mironov's analysis uses an idealized model of the key schedule algorithm which consists of an exchange shuffle. This is one of many possible algorithms for creating a random permutation given a source of random information. The idea is the same as in RC4's key schedule. One makes a pass through all elements in $S$, swapping each element with a randomly selected element. This differs from the key schedule algorithm only in that the second element which is swapped is not selected at random, but rather is determined by a pseudo-random function that depends on key material and the current state. Noting that the state changes in a similar way (dropping the key material) during the round algorithm, Mironov models the state changes after initialization as simply an extension of the exchange shuffle used in the key schedule.

Let $P_{t}$ be the exchange shuffle with $t$ swaps. Mironov summarizes several combinatorial results from the literature that show the bias of this shuffle with $t=N$ (the number of swaps in the key schedule). Among these results is the fact that there is an exponentially large gap (as $N \rightarrow \infty$ ) between the probabilities of the most and least likely resulting permutations. Also, the expected number of fixed points in a permutation resulting from the shuffle is lower than that of a random permutation. These results show the biases that are present in the shuffle, but are not very useful in analysing RC 4 since they are global properties and the outputs from RC4 depend on particular values in the permutation.

Mironov describes two new distinguishers for permutations resulting from the exchange shuffle. The first one makes use of the concept of the sign of a permutation. The sign of a permutation can be defined, among other ways, as the parity of the number of transpositions in a representation of the permutation as a product of transpositions. Note that in the exchange shuffle every swap is a transposition unless the two indices are the same, resulting in a no-op. The number of times that the two indices are the same is biased towards zero, resulting in a bias in the sign of the resulting permutation towards even rather than odd.

This bias is significant (about 0.05 for $N=256$ ), but it is difficult to use since calculating the sign requires knowledge of the entire permutation. It is, of course, very difficult to obtain the entire permutation from just the output of RC4. As an example of a more useful bias, Mironov establishes that the probability that a given value ends up in a particular position is not constant for the exchange shuffle, as it should be for a truly random shuffle. The details are beyond the scope of our explorations, however it should be
noted that the resulting distinguisher is $60 \%$ reliable for $P_{256}$ but becomes unreliable for $P_{1024}$.

The positional bias can be used to show that there is a bias in the first word output from RC4. The probability that a particular word is output depends on the probability that $S[S[1]+S[S[1]]]$ equals that word. The bias in the position of elements in the initial state makes this value biased, resulting in a distinguisher. This distinguisher can differentiate a random source from RC4 using about 1600 first output words (for $N=256$ ). Based on this, Mironov suggests that the first $3 N$ output words be dropped, making the position bias disappear and this distinguisher unusable.

In order to provide a better guarantee about the non-existence of distinguishers on the initial state, Mironov analyses the exchange shuffle in order to determine how many swaps are required to produce a state that is uniformly random. In order to do this he uses a result that bounds the probability of distinguishing two distributions based on their variance distance, which measures the differences in probabilities for collection of events. Mironov finds that, asymptotically, we need $t$ swaps, where $t$ satisfies

$$
\frac{t}{N}>\frac{1}{2} \ln \frac{1}{2 \epsilon}
$$

and $\epsilon$ is the desired upper bound on the probability of distinguishing the distribution of results of the $P_{t}$ shuffle from a uniform distribution. A second result shows that, for some $c$, the time required for $P_{t}$ to achieve an approximately uniformly random output can be bounded by $c N \ln N$. Experimentation approximates this value to be about $11.6 N$ for $N=256$.

Based on the above results Mironov claims that dropping about 12 N words from the beginning of the output of RC 4 likely eliminates the possibility of a strong distinguisher. As a practical precaution, he recommends dropping $2 N$ or $3 N$ outputs.

### 3.6.2 Finney's States

Soon after the RC4 algorithm was leaked, Finney [4] described an interesting set of states. These are small partial states with the property that they are linked by a set of short cycles. This is interesting since the large size of the internal state suggests that the cycle length should be very large. Fortunately, from a security perspective, these cycles do not occur in normal RC4 operation. They thus remain a theoretical curiosity.

Consider the partial state given by $i=a, j=a+1$ and $S[a+1]=1$ for some $a$. In this case, applying the round algorithm results in the following
conditions on the next state: $i=a+1, j=a+2, S[a+2]=1$. Thus next state is an example of this class of partial states, for a different $a$. It is easy to see, then, that all subsequent states will be of this form. In fact, a short cycle will form. The swap operation moves the 1 forward one step each round, also moving the entry $S[a+1]$ to $S[a]$. Thus the entries other than 1 migrate down in position as the 1 moves up in position. After $N$ rounds the 1 has returned to its original position, however the rest of the entries have shifted one position down. After this happens $N-1$ times all entries will have returned to their original positions. Thus the original state reoccurs after $N(N-1)$ rounds, forming a cycle.

Note that all of the states in all possible cycles of the form above match the partial state originally given for some $a$. Since each one lies in a cycle where all the states are of this form, and the round operation is reversible, it is not possible to go from a state not in this form to one that is. Note that in the initialization of RC 4 we set $i=j=0$, which is not in the required form. Thus these cycles do not happen as part of the normal RC4 operation.

### 3.6.3 Golic's Distinguisher

One of the early distinguishing attacks on RC4 was published by Golic [7]. This attack is quite different from other attacks on RC 4 in that it uses techniques that were developed for attacking LFSRs and block ciphers. In particular, it uses linear analysis. These techniques, first described in [15], use linear approximations to non-linear functions. Golic's attack makes use of the fact that the permutation $S$ in RC 4 evolves slowly. At each step $S$ is closely approximated by $S$ immediately before the swap. Golic also approximates the least significant bit of $S$ with a linear equation. These two techniques result in a linear equation relating the least significant output bits of an output byte and the second byte after. Golic argues that these two bits are different more often than they are the same.

The bias Golic uses is quite small, requiring about $2^{40}$ bytes of output to reliably distinguish RC4 from random sources. This is significantly more than required for other attacks such as Fluhrer and McGrew's distinguisher. However, Golic's attack is interesting in that the methods used are significantly different from the other attacks discussed.

### 3.6.4 Paul and Preneel's Distinguisher

After Mantin and Shamir [14] discovered the bias in the second output word of RC4 others became more interested in the beginning of the stream. Paul
and Preneel [20] analysed the digraph frequencies for the first two output bytes, finding that there is a bias against equal bytes. This bias arises from the fact that if $S[1]=2$ initially then the first two output bytes are always different. (We refer the reader to [20] for a proof of this fact.) For about $1 / N$ of the time, then, the first two output bytes cannot be equal (when $S[1]=2$ ). The rest of the time (when $S[1] \neq 2$ ) they are equal about $1 / N$ of the time. This results in a probability of about $(1-1 / N) / N$, or slightly less than the trivial probability of $1 / N$.

The bias discovered by Paul and Preneel can be easily used in a strong distinguisher by observing the distribution of the first two output bytes from several streams and comparing with the predicted bias. The authors report that about $2^{24}$ streams (with different keys) are sufficient to detect the bias. In addition a similar but smaller bias affects digraphs that occur whenever $i=0$. (The mechanism is the same, but the bias is smaller since $j=0$ only with probability $1 / N$.) This bias can be used in a weak distinguisher, observing digraphs occurring when $i=0$. The authors report that $2^{32}$ output data is sufficient to detect the bias.

### 3.7 Conclusions

RC 4 can still be considered secure in that there are no practical attacks that reveal the key or internal state for arbitrary implementations. However, there are attacks that uncover partial information about the stream and distinguish it from random data. In addition the IV weaknesses provide practical attacks in implementations using weak methods of combining IVs with keys. Care must be taken, therefore, when implementing RC4. In particular, at least $N$ words of output should be dropped from the beginning of each stream. This protects against the broadcast attack and the IV weaknesses. Also, it is prudent to hash the IV with the key instead of concatenating.

Although the weaknesses discovered in RC 4 make it a less than ideal choice for new applications, it still enjoys widespread use. This, together with its novel internal structure, continue to make it an interesting topic of research.

## Chapter 4

## Chameleon: A New Cipher and a New Cryptographic Primitive

Before studying RC4, the author developed a cipher, called Chameleon, which turned out to resemble RC4 to a great extent, although it has several important differences. Chameleon is built from a new cryptographic primitive, called a Mutating S-box, that shares some basic internal structures with RC4. One key difference, however, is that Chameleon is not a random number generator. Instead it is an autokey cipher - the plaintext influences its internal functioning.

In this Chapter we present one version of Chameleon, called Soft-Chameleon. The strong resemblance between Soft-Chameleon and RC4 invites comparisons. After describing Soft-Chameleon, therefore, we attempt to analyse it in light of the previous work done on RC4. Although many of the attacks against RC4 are simply not applicable to Soft-Chameleon, some provide new insights into Chameleon. The results are mixed; in some ways Chameleon is stronger than RC 4 while in others it is weaker.

Later, in chapter 7, we develop Card-Chameleon, which functions essentially identically to Soft-Chameleon but operates using a deck of cards rather than in software.

### 4.1 The Mutating S-box

In this Section we develop a new cryptographic primitive, called a mutating S-box. This primitive will serve as the basis for Card-Chameleon and Soft-

Chameleon.

### 4.1.1 Feedback in Block Ciphers

When using block ciphers in building protocols, feedback modes are often used to increase security. CBC, or cipher block chaining (see [16], Chapter 6 ), is one such mode. In this mode the ciphertext from the previous block is combined with the incoming plaintext block through some group operation, usually XOR, before the cipher is applied. This mode offers several advantages over protocols without feedback because information about changes in the plaintext is obscured. Notably, a message that differs in one block will result in a ciphertext that is different from that block onward. In addition, feedback obscures patterns that may be preserved by a block by block encryption. This mode of feedback does not protect against attacks on the underlying cipher, however, since an attacker with plaintext/ciphertext pairs with feedback can undo the group operation, producing plaintext/ciphertext pairs in the underlying cipher.

### 4.1.2 Modelling with Finite State Machines

There is more than one way to model CBC mode using finite state machines. For one example, let $A$ be a block cipher alphabet and let $f(x, k)$ represent a block cipher where $x \in A$ is the input plaintext and $k$ is a key. Further, let $s$ be an element from $A$, and $\cdot$ be a group operation on the alphabet. A finite state machine is constructed with state $s$, output function $g(x)=f(x \cdot s, k)$ and state change function $s=g(x)$. The initial state, often known as the initialization vector, is any valid block in $A$.

A more useful model, from our point of view, uses a much larger state. Let $A$ be the alphabet underlying the block cipher $f(x, k)$. The state consists of elements $\{S[a] \mid a \in A\}$. It is initialized to $S[a]=f(a \cdot v, k)$ where $v \in A$ is equivalent to the initialization vector mentioned above. The output function is $g(x)=S[x]$ and the state change function assigns $S[a]=f(a \cdot g(x), k)$ for each $a \in A$.

From a computational standpoint this second model is useless, especially since modern block ciphers often have alphabets with $2^{128}$ or more elements; the state change function would be impractical to evaluate. However, the structure reveals some interesting properties. There are two aspects of the state change function that appear to be somewhat limiting.

The first immediately obvious limiting aspect of the above machine is that the state information is not used in the state change function; all in-
formation is simply replaced. The state, however, contains a great deal of entropy which could be put to good use. One method of taking advantage of this information is to incorporate $S[a]$ into the state change function using the group operation. For example, $S[a]=f(S[a] \cdot a \cdot g(x), k)$.

From a cryptographic standpoint there doesn't appear to be any benefit to changing any state information other than that which is revealed by the output function. Since the key is secret an attacker has no information about the value of any $S[a]$ except $S[x]$ which is output by $g(x)$. (Actually, if the attacker can get $k$ from $x$ and $f(x, k)$ then all state information can be calculated.) There is no reason, therefore, to change any state information except for that which is revealed, which is $S[x]$. The bijectivity of $g$ must be preserved, however. One way to accomplish this is to exchange $S[x]$ with some $S[y]$. This approach preserves the entropy associated with the rest of the state information and creates a much simpler state change function.

### 4.1.3 A Finite State Machine Cryptographic Primitive

Using the ideas above, a simple cipher can be designed. Let $A$, the input and output alphabet, be any finite set. Define a finite state machine with the following elements:

1. A state $S$ consisting of $|A|$ values $\{S[a] \mid a \in A\}$.
2. An output function $g: A \rightarrow A$ defined by $g(x)=S[x]$.
3. A state change function defined by the following steps
(a) Let $j=K(k, n) \in A$.
(b) Exchange $S[x]$ and $S[j]$.
where $K$ is a function with values in $A$ and parameters $k$, a key, and $n$, the number of blocks encrypted. Alternatively, $n$ may be replaced by some other parameters including added state information. From now on we simply use $K()$ to denote the value of the function with the appropriate parameters.

The initial state is chosen so that each element of $A$ appears in exactly one position $S[x]$ with $j$ any element of $A$. Thus $S[x]$ is simply a permutation on the elements of $A$. In fact, the cipher is basically a substitution, an S-box, except that the substitution table changes with each block encrypted. For this reason we refer to this structure as a mutating $S$-box.

Decryption using a mutating S-box uses a similar finite state machine. The state is the same and the state change function uses the same $K$ function. The output function takes element $x$ and outputs the value $a$ for which
$S[a]=x$. The state change function is the same but uses this value of $a$ instead of the input value $x$.

### 4.1.4 Security

Modern substitution ciphers - that is, block ciphers - use large block sizes in order to defeat attacks such as frequency analysis and databases of ciphertext/plaintext pairs. Perhaps surprisingly, the security of a mutating S-box is not dependent on having large alphabets. To see this, consider a mutating S-box on one bit with $K(t)$ a pseudo-random number generator (PRNG) with $t$ the number of bits encrypted. Suppose an attacker has a ciphertext along with the first $n$ bits of the corresponding plaintext. At each bit the attacker can determine the state that arrived at the ciphertext by comparing with the corresponding plaintext. Thus the attacker has complete knowledge of the $n$ states in a row. However, unless the attacker knows the value of $K(n)$ the state after encryption is not known. That is, the attacker can gain no knowledge about the next bit of plaintext without knowing the next bit from the PRNG. However, the attacker can deduce $n$ bits of the pseudo-random sequence. Thus the security of this system is dependent on the security of the PRNG.

The above case is extreme, not only in terms of the number of alphabet elements, but also the amount of information known to the attacker. Each state can be determined solely based on the plaintext/ciphertext pair. With larger alphabets this is not the case. Consider an alphabet consisting of 256 elements. There are 256 ! possible values for the initial state. An attacker, gaining a ciphertext/plaintext pair, knows exactly $1 / 256$ of the information contained in that state. To guess the next state (which will allow him to decrypt the next ciphertext block) the attacker must know, not only the value of $K$, which specifies how the state changes, but also the current state. As well, for one bit the attacker simply has to examine the subsequent plaintext/ciphertext pair to discover the value of $j$. However, this does not work for larger alphabets since it takes 255 values to determine the whole state. Thus larger block sizes protect information about $K$ against attackers.

### 4.1.5 Attacks

In this Section we consider attacks on the mutating S-box. We consider two approaches: attempting to first discover the State or attempting to first discover the keystream used in the state change function.

## Attacks on the State

Assume that the function $K$ and its key are known, along with the plaintext and ciphertext for a message in which each ciphertext element occurs at least once. Under these conditions there exists an attack that can recover the complete state information at a single moment. The algorithm for this attack is as follows

1. Initialize state elements $\{S[a] \mid a \in A\}$ with a dummy element $z$ not in A
2. For each plaintext/ciphertext pair $(p, c)$ in the sequence do the following:
(a) Let $j=K()$
(b) Let $S[p]=S[j]$
(c) Let $S[j]=c$
3. When $S[a] \neq z$ for all $a \in A$ then done.

In order to reconstruct the state we first notice that, regardless of the plaintext, $S[j]$ will be assigned the ciphertext element after the substitution is made. Thus knowing the value of $j$ allows a partial reconstruction of the state. A sequence of values for $j$ and the ciphertext allows more state information to be recovered. The assignments made for $S[p]$, where $p$ is the plaintext, must be taken into account or the recovered state will not remain in synchronization with the state of the encrypting machine. For this reason the plaintext is necessary.

The amount of plaintext required for this attack is fairly small since each ciphertext element need occur only once. To see this note that each alphabet element occurs exactly once in the state. When the element occurs in the ciphertext it is assigned to a position in the state. After its position in the state is known it is always known since if moved its new position is given by the plaintext and $K$.

Once a state has been recovered there are two ways an attacker can use the information. First of all the attacker can run the machine backwards to recover the initial state. Depending on the exact details of the protocol this may enable the decryption of subsequent ciphertexts. Another option is to continue decryption of the ciphertext using the standard decryption algorithm. This allows an attacker to decrypt a ciphertext where only an initial part of the plaintext is known.

This general method can be extended to discover the key $k$ for the $K$ function along with the initial state. The attacker simply executes the above algorithm for all possible values of $k$. After determining the state, the attacker decrypts some more of the ciphertext. Candidate keys will produce text that matches the actual plaintext. The complexity of this attack is dependent on the keyspace for $k$.

Before developing a second attack, consider the state of the mutating S-box. The actual values of the state are not used except in the output function. In particular, the state change function does not depend on the actual values of the state, it merely exchanges values. Thus a permutation on the values of the state is equivalent to applying the permutation after the output function. If $K$ and the key are known encrypting with an arbitrary state will produce a ciphertext which is equivalent to the correct ciphertext up to a permutation on the alphabet. This equivalence easily produces the permutation which, applied to the guess for the initial state, reveals the correct initial state. The condition of equivalence of ciphertexts up to a permutation on the alphabet can thus be used in a brute force attack to detect the correct key.

Another, simpler, attack attempts to build the initial state from a large number of ciphertexts for which plaintexts are available. If the initial state is the same for all messages then an attacker needs simply to note the first plaintext/ciphertext pair from each message. Since these come directly from the initial state they each reveal a small amount of information that can be used directly to build a copy of the state. The attacker simply builds a table of initial plaintext/ciphertext pairs. Each plaintext need only occur once (at the beginning of a message) for an attacker to recover all initial state information. Note that this does not necessarily reveal information about the key $k$ used for $K$.

The information revealed by the above attacks can be limited by using an initialization vector (IV) to create a unique initial state for each message. The IV cannot simply be copied into the state, but must be used to modify some other, secret, initial state. If a key schedule is in use then the IV can be used as additional key information. Another option is to use a sequence of alphabet elements as the IV and apply the state change function using this sequence as a plaintext, ignoring the output. Upon completing this operation the machine will be in a new state, dependent on the key and IV used. The same sequence can be prepended to the ciphertext allowing the receiver to recreate the initial state by applying the state change function.

## Attacks on the State Change Function

Given a known initial state, it is easy to determine the sequence of values for $j$ by using a chosen plaintext. The attacker creates a message consisting of one alphabet element, say $z$, repeated many times. The first element in the ciphertext, $c_{1}$ will simply be $S[z]$. If the second ciphertext element, $c_{2}$, is the same, then the first value for $j$ is $z$. If not, then the value for $j$ is $x$ where $S[x]=c_{2}$ in the initial state. After recovering the first value for $j$ the state change function can be applied to update the state. A similar examination of $c_{2}$ and $c_{3}$ will produce the second value for $j$. This can be repeated as many times as necessary. The attacker can then launch a direct attack on $K$ with the hopes of replicating its values or discovering $k$.

The above method can be extended to a known plaintext attack by taking advantage of the uniform distribution of values of $K$. The attacker, knowing the initial state, examines the plaintext sequence $p_{1}, p_{2}, \ldots$ looking for the second occurrence of the element $p_{1}$. If the second occurrence, say at $p_{i}$, is close enough to the beginning of the message then there is a good probability that $S\left[p_{1}\right]$ has not changed since the first state change. In that case $c_{i}=S[j]$ where $S[j]$ and $j$ are the first values. Examining several messages, the attacker will notice a higher probability of one value occurring for $c_{i}$ than others, which will reveal the correct guess. The reasoning used in the previously described attack can then be used to discover $j$. After this has been done the same messages can be used to discover the second value for $j$ by applying the above procedure to $p_{2}$.

Both of the above attacks can be extended by performing an exhaustive search of all possible initial states. The correct guess will allow successful decryption of all messages. For this reason the alphabet size should be large if there are potential weaknesses in the $K$ function.

The security of a mutating S-box against such attacks is dependent on one of two things: secrecy of the initial state or a $K$ function that is resistant to attacks. The first option can only be accomplished by using an initialization vector since attacks described in the previous Section can recover an initial state without IVs. If an IV is used, however, the security of the $K$ function can be relaxed.

## Further Observations

The previous attacks depended on either knowing or guessing the internal state or $K$. If neither is known then there is no known general attack. However, there are two possible weaknesses that could be exploited.

The first weakness is that an attacker can observe values of $S$ directly. In particular, the attacker knows $S[p]$ at every point (if the plaintext is known). Of course this value is immediately swapped out. After the swap, however, the attacker knows $S[j]$. This is not immediately useful since the attacker does not know $j$. With a little more information it can become useful.

Suppose that the attacker knows the plaintext and that at some point two equal outputs occur for inputs $p_{t}$ and $p_{t+1}$. After the swap we have $S[K(t)]=c_{t}$. The next output is $S\left[p_{t+1}\right]$. Since $c_{t}=c_{t+1}$ and no swaps have occurred then we have $S[K(t)]=S\left[p_{t+1}\right]$. Since $S$ is a permutation this means that $K(t)=p_{t+1}$. Thus the attacker has learned one value of $K$.

The above observation depends on the attacker knowing both the input and output for the mutating S-box. If either the input or output is modified outside the mutating S-box this attack is prevented.

Another observation worth mentioning is that the plaintext is not guaranteed to have any particular distribution. Although there are no known attacks based on this fact, it does have an effect on the internal functioning of the cipher.

### 4.1.6 Uses in Protocols

Effective and secure use of mutating S-boxes requires attention be paid to the attacks outlined above. In particular

1. Initialization vectors should be used. The number of possible initialization vectors should be large to minimize the probability of collisions. Several collisions on one IV are necessary to launch an attack and this number is proportional to the size of the alphabet.
2. Smaller alphabets require more secure $K$ functions.
3. The number of keys for $K$ indicates an upper bound on the complexity of a successful attack.
4. The size of the alphabet indicates an upper bound on the complexity of an attack that reveals a sequence of values of $K()$.
5. A protocol that uses a bare mutating S-box without modifying either the input or output will have an attack that discovers individual values of $K$.

From a computational perspective, the mutating S-box can be very efficient, as evident in the cipher described below. The memory requirements,
however, can be large. For this reason only the use of small alphabets is practical. Also, large states will require a large time to fill. Even in applications where memory is not limited, the state must be filled during initialization by a key schedule. Large states thus imply low key agility.

### 4.2 Chameleon: An Example Cipher

The mutating S-box was originally conceived as a means of developing a hand cipher that could be used with a deck of cards as the only necessary equipment. The desire was to have a fast cipher that had security comparable to computerized ciphers. The result of this development is CardChameleon, whose description can be found in Section 7.1. Interestingly, the algorithm is easily adaptable to software. The resulting cipher is simple, efficient and resembles RC4.

Chameleon uses a mutating S-box with a simple $K$ function and an alphabet consisting of the numbers 0 to $N-1$ for some $N$, typically 256 . The output function $K$ is defined by an ordering of the elements of $\{0, \ldots N-1\}$, with values of $K$ simply cycling through the elements in this order.

In order to avoid the problems mentioned in Section 4.1.5 a modification is made to the original mutating S-box algorithm. The incoming plaintext is first modified by applying $S$. The result is then fed into the mutating S-box. This has two affects. First, since $S$ is private, an attacker no longer knows what is input to the mutating S-box. This means that the attack in Section 4.1.5 no longer applies. In addition, since $S$ is random and changes over time, the distribution of the data being input to the mutating S-box is smoothed out.

The encryption algorithm for this version, called Soft-Chameleon ("Soft" from "Software"), follows. It requires two look-up tables, $K$ and $S$ and one extra storage byte, $j$.

## Key Schedule

The key mechanism used in Soft-Chameleon is very similar to that of RC4, with the distinction that $K$ needs to be initialized as well. The shuffle mechanism is used to derive both the permutation needed for the initial state and that used for the $K$ function.

The shuffle mechanism simply exchanges pairs of elements where the position of one element is determined by a counter and the other by a simple pseudo-random function. The input is a sequence of numbers $k[]$ of length $l$ and a look-up table $S[]$.

1. Let $j=0$
2. For $i=0$ to $N-1$
(a) $j=(j+i+k[i \bmod l]) \bmod N$
(b) Swap $S[i]$ and $S[j]$

Generating the state is done in the following way. The inputs are the key $k\left[\right.$ ], and an initialization vector $v=\left\{v_{i} \mid 0 \leq i<16\right\}$.

1. Initialize state table $S[]$ so that $S[i]=i$ for $0 \leq i<N$
2. Shuffle $S[$ ] with key $k$
3. Shuffle $S[$ ] with key $v$ (i.e. replace $k$ with $v, l$ with 16 and run the shuffle algorithm)

This algorithm creates an initial state that is dependent on both key and initialization vector. The algorithm itself works for any key length between 1 and $N$ bytes. Not all keys are distinct. For example, the keys defined by the ASCII sequences "ab", "abab", and "ababab" are all equivalent since the algorithm cycles over the elements in the sequence.

An $N$-cycle is required for $K$. This is accomplished in the following way. The input is the key $k$.

1. Initialize table $S[]$ so that $S[i]=i$ for $0 \leq i<N$
2. Shuffle $S[$ ] with key $k$
3. Set $t=S[0]$
4. For $i=1$ to $N-1$
(a) Set $K[t]=S[i]$
(b) Set $t=S[i]$
5. $K[t]=S[0]$

The algorithm begins by creating a permutation of the $N$ elements. Only one key is used. The permutation created is equivalent to that created in step 1 of creating the initial state. This fact can be used in a combined algorithm to speed up the key set-up. Symmetry between the state and the cycle is not a factor since the initialization vector is used to modify the

## Input: Key $k$, plaintext $p$

1. Use the key schedule and key $k$ to derive a permutation on $N$ elements with cycle length $N$. Initialize look-up table $K[]$ with this permutation. ( $K[x]$ should follow $x$ in the cycle.)
2. Use the key schedule, key $k$, and initialization vector $v$ to derive a permutation on $N$ elements. Initialize look-up table $S[]$ with this permutation. (i.e. the permutation takes $x$ to $S[x]$.) Send $v$ as the first part of the message.
3. Set $j=0$
4. For each plaintext letter $p$ do the following:
(a) $j=K[j]$
(b) $t=S[p]$
(c) $c=S[t]$
(d) $S[t]=S[j]$
(e) $S[j]=c$
(f) Output $c$ as the next character in the message.

Figure 4.1: Soft-Chameleon Encryption Algorithm
state. The first permutation is used to create another consisting of a cycle length of $N$ by interpreting the entries in the look-up table as a cycle and creating entries in another look-up table ( $K[]$ ) appropriately.

Although keys can be as short as one byte, this is, of course, not recommended. 128 bit keys are currently considered to be large enough to protect against brute force attacks. Low security applications can, of course, use smaller keys. Implementations should enforce a suitable lower bound on key length.

## Encryption

The encryption algorithm for Soft-Chameleon is given in Figure 4.1

## Decryption

Decryption is a little more complicated, requiring another look-up table, $S^{-1}[]$. The algorithm is given in Figure 4.2.

1. Use the key schedule and key $k$ to derive a permutation on $N$ elements with cycle length $N$. Initialize look-up table $K[]$ so that $K[x]$ follows $x$ in this cycle.
2. Use the key schedule, key $k$ and initialization vector $v$ to derive a permutation on $N$ elements. Initialize look-up table $S$ [ ] so that $x$ goes to $S[x]$ in this permutation. Send $v$ as the first part of the message.
3. Initialize look-up table $S^{-1}[]$ so that $S^{-1}[S[x]]=x$.
4. Set $j=0$
5. For each ciphertext letter $c$ do the following:
(a) $j=K[j]$
(b) $t=S^{-1}[c]$
(c) $p=S^{-1}[t]$
(d) $S[t]=S[j]$
(e) $S^{-1}[S[j]]=t$
(f) $S[j]=c$
(g) $S^{-1}[c]=j$
(h) Output $p$

Figure 4.2: Soft-Chameleon Decryption Algorithm

### 4.3 Analysis

The attacks against the mutating S-box are potentially weak points in SoftChameleon. For $N=256$ there are $256!\sim 2^{1684}$ possible states and cycles. Obviously these attacks are much more difficult than a brute force attack on the key unless the key is quite large. The key schedule is flexible, allowing for long keys if such security is required.

One issue regarding the key schedule is the possibility of working the shuffling algorithm backwards, using a known key, from a known state to retrieve the initial state. This becomes an issue since the shuffle algorithm is used to modify the initial state using a public key, the initialization vector. If the attacker can determine the state after the initialization vector has been applied then it is easy to apply the shuffle algorithm backwards to retrieve the initial state, which will allow decryption of all subsequent messages.

The best method of determining the state after the IV has been applied (the state at the beginning of encryption) is using a known plaintext attack with at least $N-1$ pairs. These pairs must have been encrypted with the same IV. The probability of this occurring is quite small since it requires $N-1$ collisions on the same IV. However, there is a weakness in the way that the IV is used in Soft-Chameleon that defeats the use of IVs as a mechanism to protect the state before the IV is applied. This attack is described in Section 4.4.3.

From a programmer's perspective the Soft-Chameleon algorithm has some interesting properties. First of all, only 5 instructions are needed to encrypt a byte (ignoring instructions for input and output). This contrasts with even the fastest block ciphers which require about 18 instructions per byte on 32 bit processors or many more on 8 bit processors, and RC 4 which requires 8 (including the XOR with plaintext). As well, all instructions in Soft-Chameleon operate on 8 bit blocks so there are only 5 instructions required for any processor that is 8 or more bits wide. The decryption algorithm, with the addition state information, is less efficient than encryption. However, at 7 instructions per byte (again excluding input and output instructions) it is still very fast.

More interestingly, all instructions in both encryption and decryption are for data movement; only look-ups and assignments are used. This is in contrast to most modern ciphers which rely heavily on mathematical constructions such as group and logical operations.

The key schedule, by contrast, is slow compared to other ciphers. The shuffle algorithm uses nine instructions per pass through the loop (2 additions, 2 mod, 3 look-up and two assignments) and $N$ passes for a total of

2304 instructions when $N=256$. This is used twice. The result of the first shuffle can immediately be used to create $K$ [ ] before it is shuffled again. The creation of $K[]$ requires takes $2 N$ instructions. The total, then, is 5120 instructions, assuming all loops are unrolled, for $N=256$.

### 4.4 Attacks on Chameleon Inspired by RC4

In this Section we examine the similarities between RC4 and Chameleon and investigate the resistance of Chameleon to the attacks designed for RC4.

### 4.4.1 Similarities Between RC4 and Chameleon

There is a great deal of similarity between RC4 and Chameleon. To start, the state is quite similar. The permutation $S$ functions in an almost identical way, providing a pseudo-random function. Also, there is one private counter, $j$, that points to different elements of $S$ in a pseudo-random way, and one public counter, $i$. These parallel the $N$-cycle and the plaintext in Chameleon (in a known-plaintext attack). Finally, the swap operation is identical.

The key schedule is also very similar, which is primarily due to the fact that this algorithm is one of the simplest methods of generating a randomlike permutation. The only difference in the actual shuffle mechanism is the line where $j$ is incremented. Here $S[i]$ is replaced with $i$. Of course there is more set-up involved in Chameleon, due to the extra information required for $K$.

The similarities between these two ciphers invite deeper comparisons and an attempt to apply the attacks on RC4 to Chameleon. Some of these attacks are too specific to RC4 to be of use, but others can be modified in a way that exposes weaknesses.

### 4.4.2 Branch and Bound Attacks

It is possible to launch an attack that attempts to discover both the initial state and the state change function at the same time. Rather than a brute force attack on one of these, a branch and bound algorithm of the type used in Knudsen's attack on RC4 can be used to improve the speed of the search. The concept is analogous to Knudsen's attack, closing off branches of possible choices, but the exact details are different. The algorithm is given in Figure 4.3 .

Experiments on small $N$ have shown that our attack on Chameleon, despite its simpler nature, has a higher computational complexity than Knud-

Input: Some ciphertext with corresponding plaintext

1. $j=0$
2. Repeat the following for each plaintext/ciphertext pair $(p, c)$
(a) If $K[j]$ is not defined then branch over all unassigned values
(b) If $S[p]$ is not defined then branch over all unassigned values
(c) If $S[t]$ is not defined and $c$ is not assigned in $S$ then $S[t]=c$
(d) If $S[t] \neq c$ then a contradiction is found. Close this branch
(e) Swap $S[t]$ and $S[j]$

Figure 4.3: Chameleon Branch and Bound Attack Algorithm
sen's attack for the same $N$. The complexity is very close to $N$ !, instead of $\sqrt{N!}$ as in RC4. A closer examination shows that for our attack there is a branch for every element of $K$ since there are no equations that allow these values to be filled in from other information. In Knudsen's attack, by contrast, there are three places where values can be filled in, allowing potential branch points to be avoided. Also, both $S$ and $K$ must be guessed. These effects combine to give a much higher complexity. In fact, the complexity is so high that it is comparable to the best attacks described against the mutating S-box.

### 4.4.3 IV Weakness

Although the IV weaknesses in RC4 do not apply directly to Soft-Chameleon, the idea of using different IVs to gain information about the key turns out to be fruitful. To see this, we examine how the key schedule uses the IV.

The key schedule initializes the look-up table $S$ to the identity permutation. Afterwards $S$ is shuffled by the shuffling algorithm using the secret key. Finally, $S$ is shuffled using the IV. Examining the internals of the shuffling algorithm, we see that the essential feature is the swap. We can consider this operation as the multiplication of the permutation $S$ with a transposition that exchanges $i$ and $j$. From this it is easy to view the entire shuffling process as the multiplication of a sequence of transpositions. Taking a longer view, the two shufflings performed, using the key and using the IV, can be considered as the multiplication of one long sequence of transpositions. By associativity, we can consider this as two permutations multiplied together,
one formed by shuffling with the key and the other formed by shuffling with the IV.

Let $S_{k}$ be the permutation corresponding to shuffling with the secret key and $S_{I}$ the permutation corresponding to shuffling with the IV. Imagine an attacker that has collected a series of initial plaintext/ciphertext pairs from several streams, possibly with different IVs. The attacker can easily derive the permutation $S \circ S$ from which it is possible to find $S$ with some addition effort. Now for any particular pair $x, y$ with $S[x]=y$ we have

$$
y=\left(S_{I} \circ S_{k}\right)[x]
$$

since no swap has been performed at this time. Equivalently we have

$$
S_{I}^{-1}[y]=S_{k}[x] .
$$

Note that $S_{I}$ is completely determined by the IV, which is public knowledge. Thus the attacker can determine one element of $S_{k}$. If sufficiently many initial plaintext/ciphertext pairs are known the entire table can be determined.

Once $S_{k}$ has been determined, one can apply the shuffling algorithm with the IV from subsequent messages, obtaining the initial state for these streams. This information can be used to launch an attack attempting to discover $K$. Since $K$ does not change with different IVs information from several streams can be combined in these attacks.

It should be noted that this attack does not affect the IV mechanism used in Card-Chameleon since thehe mechanism in use there is key dependent. In particular, the incremented $i$ is replaced with the cycle $K$, and so $S_{I}$ can no longer be computed from public knowledge. Thus thehe attack fails. This suggests a means of strengthening the key schedule for Soft-Chameleon. One improvement is described in Section 5.3.1.

### 4.4.4 Key Schedule Invariance Weakness

Since the shuffling mechanism used in Soft-Chameleon is so similar to that in RC4, there exists the possibility that the weak keys described in Section 3.4.2 also affect Soft-Chameleon. In fact, a closer examination reveals that SoftChameleon is more susceptible because the use of $S[i]$ is replaced with $i$. Thus the condition $S[i] \equiv i(\bmod b)$ is no longer an issue. When considering the special exact keys the problematic elements in $S$ are no longer of concern since they can never influence $j$. For Soft-Chameleon, then, the number of problem entries in $S$ is always two when using a special exact key.

Of course in Soft-Chameleon two shufflings occur, one with the key and one with the IV. For the final permutation to have the invariance property, then, the IV must also be special-exact in the same manner as the key.

Using the special initial form of $S$ is a little different for Soft-Chameleon than for RC4. Since the output mechanism depends on the value of the plaintext it is a little harder to analyse. However, it can be argued in much the same way that if $S$ has the correct initial form then the first few outputs will, with high probability, obey $S[x] \equiv x(\bmod b)$. These events could easily be detected with known plaintext/ciphertext pairs. In conclusion, the key invariance attack on RC 4 can be modified to attack Soft-Chameleon.

### 4.4.5 Biases and Other Attacks

Although some of the attacks on RC 4 are quite successful when adapted to Soft-Chameleon, most of the bias attacks are not adaptable because they rely on particular details about the output mechanism in RC4. Predictive states, for example, will not be useful for Soft-Chameleon because one key variable, the plaintext, is not controlled as part of the cipher. In fact any output string can be created using a suitable choice of plaintext.

### 4.5 Conclusions

Although there are currently no practical attacks against its main algorithm, Soft-Chameleon should be regarded as experimental. With further development, however, we are confident that a secure, efficient cipher based on the mutating S-box can be constructed, perhaps with only little modification to Soft-Chameleon. In particular, the key schedule requires modification.

The mutating S-box shows promise as a tool for use in designing simple, efficient ciphers. Its main limitation is the memory requirements, which limit its use to small $N$. However, this makes it an excellent choice for designing ciphers for 8 -bit machines such as those used in smart cards. The example cipher, Soft-Chameleon, exemplifies these properties by providing a cipher that compares favourably with other 8-bit ciphers (namely RC4) for speed and simplicity.

In addition to being an interesting object of study itself, the mutating S-box bears a strong resemblance to RC4. By examining their common properties and how they differ we can better learn about both ciphers with the goal of developing ciphers that are more secure while retaining the simplicity and efficiency present in both RC4 and Soft-Chameleon.

## Chapter 5

## Modifying RC4

In the previous Chapter we explored Soft-Chameleon and applied the analyses of RC4 to it. In this Chapter we wish to use the insights and techniques used in Soft-Chameleon, combined with those of RC4 to explore new possibilities for cipher design. We present an abstraction of the two ciphers into a more general setting and explore some possible applications that this setting suggests. As well, we develop a new cipher, RC4B, that borrows from both RC4 and Soft-Chameleon. Finally, we present two other ciphers, RC4A and VMPC, that were developed as variants of RC4.

### 5.1 Comparisons

In the previous Chapter we examined the two ciphers, offering one comparison between the two. In that comparison, $j$ was equated in the two ciphers. We now argue that $j$ in Soft-Chameleon behaves more like $i$ in RC4. Although the former is private and the latter is public, they have two important similarities. First, they both repeat with cycle length $N$. This is in contrast to $j$ in RC 4 and the plaintext in Soft-Chameleon, which (potentially) have much less predictability. Also, we can view the plaintext in Soft-Chameleon as private information, making it seem closer to $j$ in RC4.

Consider $i$ in RC4. It seems to play an important role in that it does not depend on the state and addresses every element in the state in every $N$ steps. The latter fact means that the swap operation and the update of $j$ will affect and be affected by every element of the state within this number of steps, ensuring that the entire state plays a role in the output within a short span of time. Since $i$ does not depend on the state, this also means that a few carefully chosen values in the state cannot cause a short cycle;
the entire state must play a role.
Clearly $i$ contributes significantly to the security of RC4. However, there are disadvantages to its particular sequence of values. The fact that it is simply incremented means that its value can be known by anyone who can count the number of output words. In practice this means that $i$ must be considered public knowledge. Another disadvantage is that since $i$ does not depend on the state it also does not depend on the key. This means that the state change operation is not key dependent.

Now consider $j$ in Soft-Chameleon. It performs much the same role as $i$ does in RC4. Since the incoming plaintext can not be guaranteed to have any particular distribution or visit any particular state element, it is possible that it only affects a small amount of the state. (For example, if the plaintext in English text encoded in ASCII, which only uses a small subset of all 256 byte values.) It becomes a critical property of $j$ that it is guaranteed to visit every element of the state in a short time, ensuring that the entire state contributes to the output.

In contrast with $i$ in RC4, $j$ in Soft-Chameleon has one important property that affects security: it is key dependent. This means two things. First of all, $j$ is no longer public knowledge. This becomes quite important since if somebody knows $j$ and the plaintext then they know all information and can fairly easily discover the state information (as discussed in Section 4.1.5). Also, this means that the state change operation is key-dependent.

The natural conclusion from the above comparison is that if we replace the incrementing of $i$ with a mechanism similar to how $j$ in Soft-Chameleon behaves, we stand to increase the security of RC4.

### 5.2 A New Viewpoint

The comparison between Soft-Chameleon and RC4 also suggests a new means of viewing RC4. We first examine Soft-Chameleon, developing this new viewpoint and then apply it to RC4.

Suppose we have a finite state machine that is built as follows: The internal state consists of $S$, a permutation on $N$ values. The machine takes two input values, $a$ and $b$ and performs the following operations:

1. output $S[b]$
2. Swap $S[a]$ and $S[b]$

We can view this simple machine as taking two streams of values, the $a$ stream and the $b$ stream, and outputting a new stream that combines them.

This is, of course, just the mutating S-box as described in Section 4.1.3 (with $a$ as the plaintext stream and $b$ as the $j$ stream.) However, we take it in a slightly more general context. One feature to note is that if the $a$ stream and the output stream are known together with the initial state then the $b$ stream can be obtained. This corresponds to decryption in the mutating S-box.

Now consider a slightly more complicated machine. This second machine has the same internal state and performs the following operations:

1. output $S[S[a]+S[b]]$
2. Swap $S[a]$ and $S[b]$

This is a generalization of RC4. Like the previous machine, it takes two streams and combines them into another stream. Also like the previous machine, it is possible to discover $b$ from the output stream and $a$ by maintaining a reverse look-up table which gives $S^{-1}$. The value $b$ is given by $S^{-1}\left[S^{-1}[\right.$ output $\left.]-S[a]\right]$. The swap operation can then be performed, resulting in the correct state for the next values in the streams. It is interesting to note that this machine is symmetric with respect to its inputs.

Now that we have these two machines, we examine the different streams that are actually used in constructing Soft-Chameleon and RC4. First, consider $i$ in RC4. This is a very basic stream. It is publicly available, and has a uniform distribution of values but short cycle length. Plaintext streams are, of course, quite variable. There is no guarantee about the distribution of their values or any other statistical property. As well, they must be considered public knowledge when studying anything but the most basic security properties. The $j$ used in Soft-Chameleon has a short cycle length but has a uniform distribution of values and is not public. Finally, the $j$ used in RC4 has somewhat random values and is not public. However, it is dependent on the internal state of the machine, introducing another window for attacking $S$.

After separating the basic machines and input streams, several new possibilities become apparent. First of all, we can mix different streams and machines. Some of the combinations will not be secure (for example, replacing the $K[]$ cycle in Chameleon with the public $i$ cycle). Others appear to be more secure. For example, replacing the output function in Chameleon with that of RC4 appears to be more secure because values of $S$ are not immediately public and depend on the value of $a$ as well as $b$. As another example, we can use the key dependent $K[$ ] in place of $i$ in RC 4 , thus cre-
ating a key dependent round algorithm and reducing the amount of public information.

## Other Uses of the Basic Machines

Another possibility of using these basic machines is in place of other mechanisms that are traditionally used to combine two streams. For example, the output of RC4 is combined with plaintext using the XOR function. This has the advantages of being fast and easily reversible. However, it leads to several attacks. The well known idea of XORing two ciphertexts together to eliminate the keystream is one example. Also, any attack that determines a single bit of keystream can be used to determine a bit of plaintext directly. We can reduce the power of these two attacks by using one of the basic machines above to combine the keystream with the plaintext.

The mutating S-box, as originally described, is intended for exactly this application, if stated somewhat differently. Attacks have already been discussed. Although it appears to be secure against known plaintext attacks (that is, it is not feasible to determine the internal state or the keystream given a stream of plaintext and ciphertext) it is vulnerable to chosen plaintext attacks. In particular, the attack described in Section 4.1.5 could be modified to determine partial information about the keystream without knowing the internal state.

Using the basic RC4 machine in this same context appears to be more secure. However, most of the same attacks apply. If the internal state is known, for example, it is possible to reconstruct the keystream given some known plaintext. If we have plaintext word $p$ with ciphertext word $c$ then we calculate the keystream value $k$ at that moment as

$$
k=S^{-1}\left[S^{-1}[c]-S[p]\right]
$$

If the keystream and plaintext ( $k_{i}$ and $p_{i}$ ) are known the attack becomes more complex. However, all information necessary to do the swaps is known. An attacker can proceed by first assigning a variable to each initial value of $S$. Each plaintext/ciphertext word gives a restriction on the state. The plaintext and keystream inputs point to unknown elements of $S$, but they can be traced back to one of the initial values of $S$ by reversing the swap operations. The end result is a collection of conditions on the initial state of the form $S_{i}[x+y]=c_{i}$ where $c_{i}$ is a ciphertext word and $x$ and $y$ are initial variables (where again $S_{i}$ represents $S$ at the $i$ th step).

The above observations can be used to build a branch and bound algorithm that attempts to learn the initial state given the keystream, plaintext,
and ciphertext. Basically we guess values for the initial state as required and check for a contradiction (two assignments to the state that are not compatible.) If no contradiction is found we move forward, applying the algorithm until a contradiction is found or no data remains. It is expected that this algorithm would have a lower complexity than Knudsen's attack on RC4. This is because this new attack is equivalent to Knudsen's attack on RC4 with $j$ known as well as $i$. The amount of information has increased, so the attack complexity should decrease.

### 5.3 RC4B

With the ideas from the previous Section in mind, we now develop a variant of RC 4 , which we call RC 4 B , that borrows from Chameleon, resulting in a cipher that is arguably more secure. The basic idea has already been outlined: replace one of the input streams to the basic RC4 machine with something else. In this case we replace $i$ with an $N$-cycle as used in Chameleon. The result is a state change function that is key dependent and an overall algorithm that hides all information about the internal state.

### 5.3.1 Key Schedule

Since RC4B uses an $N$-cycle in addition to the state information used in RC4, a new key schedule is required. While modifying the algorithm to provide the necessary state information, we also modify it to strengthen it against some of the attacks mentioned in Chapter 2.

The new algorithm is given in Figure 5.1. It is inspired by both the RC4 key schedule and that of Soft-Chameleon, with some important differences from both. It takes two inputs, the key $k$ of length $l_{1}$ and an initialization vector $v$ of length $l_{2}$. The first part of the algorithm (steps 1-3) sets up $S$ exactly as in RC4. $S$ is then used to derive the $N$-cycle $K$ (steps 4-6) instead of passing this as the internal state. Next the whole state is shuffled again, using the IV instead of the key. Also, instead of using $i$ to control $j$ and the swaps, the $N$-cycle is used, with variable $j^{\prime}$.

The RC4B key schedule has several properties that make it appear much more secure than either the RC4 or Soft-Chameleon key schedule. First, the state is shuffled twice. This helps smooth out the biases observed by Mironov, as described in Section 3.6.1. Since RC4 is now considered to be insecure without dropping at least the first $N$ bytes of output, the first $N$ rounds of keystream generation can effectively be considered to be the part of the key schedule. In the RC4B key schedule, this discarding is made

Input: Key $k$ of length $l_{1}$ and initialization vector $v$ of length $l_{2}=l_{1}-1$.

1. For $i=0$ to $N-1$, let $S[i]=i$.
2. $j=0$
3. For $i=0$ to $N-1$
(a) $j=j+S[i]+k\left[i \bmod l_{1}\right] \bmod N$
(b) Swap $S[i]$ and $S[j]$
4. Set $t=S[0]$
5. for $i=1$ to $N-1$
(a) Set $K[t]=S[i]$
(b) Set $t=S[i]$
6. $K[t]=S[0]$
7. $j^{\prime}=S[j]$
8. For $i=0$ to $N-1$
(a) $j^{\prime}=K\left[j^{\prime}\right]$
(b) $j=j+S\left[j^{\prime}\right]+v\left[i \bmod l_{2}\right] \bmod N$
(c) Swap $S\left[j^{\prime}\right]$ and $S[j]$
9. $i=S[j]$

Figure 5.1: RC4B Key Schedule Algorithm
explicit. Second, Roos' set of weak keys is eliminated. The fact that each element is swapped at least twice, the second time with an element that has already been swapped (unless $i=j$ at that moment) makes Roos' analysis inapplicable.

The invariance weakness in RC4 of Section 3.4.2 resulted from several coincidences that allowed a specially constructed key to move $j$ in such a way that the swap preserved the entries of $S$ modulo some $b$. One of the conditions that was noted about this is that in order to have a special exact key of length $l$ we must have $b \mid l$. In order to preserve the invariance over both shuffles we must have a special exact key $k$ and a special exact initialization vector $v$. However, by choosing different lengths that are coprime we make it impossible for this condition to be satisfied for any $b$ other than 1 , which
gives no information. In this way the invariance weakness is eliminated.
The most important weaknesses practical in the key schedule of both RC4 and Soft-Chameleon stems from a weak usage of the initialization vector. In the RC4B key schedule the attempt has been made to eliminate these weaknesses. First, the initialization vector does not interact directly with the key, as it does in RC4, making it much less likely that attacks based on weak IVs could expose key material. The IV is used in a shuffle, much like in Soft-Chameleon, but the shuffle is now key dependent, using $K$, eliminating the attack described in Section 4.4.3.

As one final note, $i$ and $j$ are not set to a constant at the end of the key schedule. $j$ is not modified, leaving it in an effectively random state, and $i$ is set to $S[j]$, which is also effectively random. This further reduces the amount of public knowledge about the state.

A careful analysis of Soft-Chameleon and RC4B show that they require an almost identical initialization (i.e. a permutation and an $N$-cycle). The one exception is that $i$ need not be set for Soft-Chameleon. Thus the key schedule for RC4B can be used as an alternative to the weak key schedule previously developed for Soft-Chameleon.

### 5.3.2 RC4B Round Algorithm

The round algorithm for RC 4 B is given in Figure 5.2. It is essentially the same as RC4, with one feature borrowed from Soft-Chameleon. The simple incrementing of $i$ is replaced with the $N$-cycle $K$.

1. $i=K[i]$
2. $j=j+S[i] \bmod N$
3. Swap $S[i]$ and $S[j]$
4. Output $S[S[i]+S[j] \bmod N]$

Figure 5.2: RC4B Round Algorithm

### 5.4 RC4 Attacks on RC4B

In this Section we consider some of the attacks on RC 4 previously described, giving an analysis of their effectiveness against RC4B.

### 5.4.1 Predictive States

Since the public knowledge of $i$ is lost in RC4B, a partial state must include information about how $i$ changes as well as the current values of $i, j$, and some of $S$. This, first of all, decreases the probability of a particular partial state occurring. Hence predictive states look like they should have reduced power. The actual picture is a little more complicated.

A careful examination of some previously known predictive states for RC4 shows that it is possible to convert many of these states into classes of predictive states for RC4B where there is one predictive state for each possible sequence of values for $i$. This has two implications. First of all, it means that many of the biases that appear in RC4 also appear in RC4B. Experimental results show that the digraph distributions have biases similar to that of RC4. A second implication is that, once a predicted output occurs, there is a whole class of possible states that could have caused that predicted output. Thus a considerable amount of guessing may be required to make use of a predicted output, increasing the complexity of attacks that use these states. If a predicted output occurs an attacker does not immediately learn information about the internal state because there are many possible partial states that would produce the predicted output.

### 5.4.2 Key Schedule Attacks

The key schedule of RC4B was designed with the attacks on RC4 in mind. In particular, it was desired that the new key schedule would defeat the invariance attack, and the attacks using IVs (both in Soft-Chameleon and in RC4). Both goals are attained using the second shuffling of $S$ with the initialization vector $v$. This shuffle uses $K$ instead of incrementing $i$ as in RC4. Since $K$ is private knowledge, it is not possible to construct a separate permutation based on the IV as in Soft-Chameleon. Also, since the IV is not involved directly with the key as in some implementations of RC4, it should not be possible to gain key information using IV information. Also, as previously mentioned, using coprime lengths for $k$ and $v$ defeats the invariance attack.

### 5.4.3 Early Keystream Attacks

There are several attacks which make use of information gained from the first few bytes of keystream generated. These attacks rely on the fact that $i$ and $j$ are both initialized to public constants. In RC4B these constants are replaced with key dependent variables, thereby limiting the amount of
information that can be gained from the first keystream bytes. In addition, more swapping occurs during the key schedule, decreasing the amount of bias present in the initial state.

### 5.4.4 Knudsen's Attack

Knudsen's attack can be implemented against RC4B with some changes. The big difference is that now $K$ must be guessed as well. The author has implemented this attack and done numerical experiments for small $N$. The complexity of the attack scales close to $N$ ! rather than $\sqrt{N!}$ as is the case for RC4. This means that it is possible to use RC4B securely for smaller $N$ than is possible for RC4. In Chapter 7 we take advantage of this to develop a card game cipher that uses RC4B with $N=27$.

### 5.4.5 Other Modified RC4 Ciphers

Previously other authors have modified RC 4 in an attempt to defeat the various attacks against it. In this Section we present two such ciphers.

## VMPC Stream Cipher

The VMPC stream cipher, described by Zoltak [27] is a straightforward modification of RC 4 that attempts to gain security by replacing the output function and the $j$ update. The function used is the so called VMPC one way function. VMPC is actually a family of functions, of which the one chosen for the output of the stream cipher is

$$
\begin{equation*}
S[S[S[x]+1]] . \tag{5.1}
\end{equation*}
$$

The goal of the VMPC is to limit the amount of information that can be gained about $S$. The key part is the +1 , which is intended to interrupt cycles in $S$, forcing an attacker to guess all entries of $S$. The algorithm is given in Figure 5.3. $K[]$ is a key of length $l$ and $V$ is an initialization vector of length $m$.

There are several differences between VMPC and RC4. First of all, the key schedule is modified, integrating initialization vectors in a way similar to RC4B and increasing the number of passes as recommended by Mironov. As well, the update of $j$ has been modified, adding one more use of $S$ to make $j$ more complex in its movement. In the round algorithm again the update of $j$ has been modified as has the output function, as mentioned

1. $j=0$
2. For $i=0$ to 255 , set $S[i]=i$.
3. For $n=0$ to 767
(a) $i=n \bmod 256$
(b) $j=S[j+S[i]+K[n \bmod l] \bmod 256]$
(c) Swap $S[j]$ and $S[i]$
4. For $n=0$ to 767
(a) $i=n \bmod 256$
(b) $j=S[j+S[i]+V[n \bmod m] \bmod 256]$
(c) Swap $S[j]$ and $S[i]$
5. $i=0$
6. Repeat for each byte output
(a) $j=S[j+S[i] \bmod 256]$
(b) Output $S[S[(S[j]+1) \bmod 256]]$
(c) Swap $S[i]$ and $S[j]$
(d) $i=i+1 \bmod 256$

## Figure 5.3: VMPC Stream Cipher

previously. Finally the order has been changed, placing the swap after the output function.

Many of the vulnerabilities that have been discovered for RC 4 have been avoided in VMPC, learning from the analysis done on RC4. Zoltak, through statistical analysis, found that the first byte output, element and digraph probabilities are all unbiased. As well, other more theoretical vulnerabilities such as those in Mironov's analysis and Knudsen's attack have been reduced. The cost, however, is a more complex algorithm with a lower efficiency.

## RC4A

After their analysis of RC4, Paul and Preneel [20] developed a variant of RC4, called RC4A, which attempts to increase security without decreasing efficiency. Their approach essentially takes two RC4 instances and crosses
information between them. They use a state consisting of two look-up tables, $S_{1}$ and $S_{2}$ and three counters, $j_{1}, j_{2}$, and $i$.

The key schedule of RC4A is the same as for RC4 except that it is used twice, once for $S_{1}$ and $S_{2}$. It is not specified how the keys for these two uses of the key schedule are found. They may be derived from a common key through the use of some pseudo-random number generator. After $S_{1}$ and $S_{2}$ are generated $i, j_{1}$ and $j_{2}$ are set to 0 .

1. $i=i+1 \bmod N$
2. $j_{1}=j_{1}+S_{1}[i] \bmod N$
3. Swap $S_{1}[i]$ and $S_{1}\left[j_{1}\right]$
4. Output $S_{2}\left[S_{1}[i]+S_{1}\left[j_{1}\right] \bmod N\right]$
5. $j_{2}=j_{2}+S_{2}[i] \bmod N$
6. Swap $S_{2}[i]$ and $S_{2}\left[j_{2}\right]$
7. Output $S_{1}\left[S_{2}[i]+S_{2}\left[j_{2}\right] \bmod N\right]$

Figure 5.4: RC4A Round Algorithm
The round algorithm for RC4A is given in Figure 5.4. The two instances of RC 4 basically operate independently except for the output, which depends on both instances. Note that each run through the round algorithm produces two bytes of output. The two instances share $i$, so there is a slight increase in efficiency.

The goal behind RC4A was to increase security primarily by increasing the internal complexity of the algorithm. By increasing the number of variables involved in each output the size of predictive states is increased, reducing biases. Also, the larger internal state size increases the complexity of attacks such as Knudsen's attack. The cost is a large increase in the memory requirements as well as set-up time.

### 5.5 Conclusion

Each of the three modified RC4 algorithms, RC4A, RC4B, and VMPC is modified from RC4 in a different way, making it difficult to directly compare them. However, we can make some general observations. First, all are
comparable in speed, owing largely to the great similarity with RC4. Also, each claims added security against various attacks against RC4. Without an in depth comparison, however, it is not possible to say which has best achieved this goal.

The RC4 algorithm is clearly quite flexible. There are many ways of modifying it, and we have presented three. In addition, there are possibilities for using portions of the RC4 algorithm in different contexts, making use of its simple and secure design in other applications.

## Chapter 6

## Expanding RC4 to 32 Bits

RC4, although very popular in applications, is becoming less desirable because of the increasing number of attacks against it. As well, at the time of its development it was very fast relative to other ciphers, especially among software ciphers, but its 8-bit nature does not fully take advantage of newer processors that are widely available. Newer ciphers are often designed with new processors in mind, making them more efficient. For these reasons in this Chapter we examine possibilities for new ciphers that aim to be more efficient and secure than RC 4 and yet borrow from its structure and preserve its simplicity.

There are many possible avenues for this exploration. First, one can attempt to parallelize RC4, taking advantage of the vector operations available on many processors. Second, one can modify RC4 in a way that preserves much of its efficiency on 8 -bit systems. Finally, one can modify RC4 without regards to efficiency on 8 -bit systems, allowing for much faster implementations on a more restricted set of processors.

### 6.0.1 Parallelizing RC4

A close examination of RC 4 reveals that it is very difficult to parallelize. Every instruction is dependent on the results of the immediately previous instruction. However, it is possible to slightly modify RC4 to gain additional throughput using the vector computation instructions that are available on many 32 -bit processors.

With the introduction of MMX, SSE, AltiVec and other instruction sets that implement vector computation, the cryptographer has a new set of tools to custom-design fast ciphers. Vector computation allows a processor to perform certain operations on a vector of values rather than on a single
value, for example, adding two vectors of four 8-bit values rather than two 32 -bit values. Also, vector computing implementations often use increased register sizes ( 64 or 128 bits). The number of instructions available for these registers is typically not as large as for regular registers, but instructions commonly used for ciphers such as addition, bit shifts, and XOR are all available.

Examining RC4 we see that three of the seven instructions are additions. Although we cannot perform these together, in general, since they are interdependent, we can take another approach. Instead of running one instance of RC 4 , we can run, say, eight instances, each operating independently. Since the additions of these different instances are independent, they can be performed simultaneously. The 64-bit registers available on processors equipped with MMX instructions would be suitable for such an operation since they can perform eight 8 -bit additions simultaneously. The outputs from these different instances could then be interlaced. The number of instructions per byte output would drop from 7 for RC4 to about 4 ( 3 for the additions plus $32=4(8)$ for the memory look-ups and assignments for each 8 bytes output.) Running 16 instances in parallel would be feasible when using 128 -bit registers like those available with AltiVec or SSE instruction sets, further increasing parallelism and hence efficiency.

Note that all the operations in this scheme are 8 -bit, although some are performed by the parallelism of vector computing. Thus 8 -bit machines would not be affected. The one drawback, however, is that the memory requirements have increased 8 -fold. This is, in general, not a concern for 32 -bit systems, but may be a factor for limited systems like smart cards. In such situations, however, the additional efficiency for the 32-bit systems is probably not a great advantage.

### 6.0.2 32-Bit Versions of RC4

The most natural way of extending RC4 to larger word sizes is to increase $N$. The larger words take a greater advantage of the processing capabilities of wider processors. However, the memory requirements of the state increase exponentially with the word size. Coupled with this is the fact that processors typically operate in 32,16 and 8 bit word sizes, but nothing in between, limiting options for a compromise. 8 -bit RC 4 has a small memory requirement and reasonable time to set up a key, but 16 -bit RC4 would be useful only for systems where key agility is not a factor and 32 -bit RC 4 , requiring 16 Gigabytes of memory, would be incredibly limiting (especially since 4 Gigabytes is the maximum amount of memory addressable by Pentium class
processors.)
One possible means of using 32 -bit operations without increasing $N$ or substantially changing the algorithm is to use a method similar to that described in the previous Section using vector operations. However, instead of preserving the isolation of the different instances of the cipher, the operations can be made fully 32 -bit (or 64 -bit for processors so capable). This is done on the additions only. Instead of adding, say $j$ and $S[i]$ for each instance separately or together in a vector, the values are packed into two 32 -bit registers and added together. This blurs the lines between the different instances since carries from lower order bytes can affect the higher order bytes. As well, the way that the bytes are packed can be modified each round (in some fixed pattern) so that the bytes cross over from one instance to another much like in RC4A (see Section 5.4.5). Unlike in RC4A, however, the additions are carried out together, making the parallelism more explicit.

Although the security of the above cipher would likely be increased, mostly due to the increased state size and interaction between instances, the efficiency is likely not going to be increased much. This is because the instructions for packing the bytes into the 32 -bit registers would probably take more instructions than the simple memory look-ups that would be done in plain RC4.

Clearly, in order to fully take advantage of the 32-bit operations available on larger processors, RC 4 must be modified in some fundamental way. In the remainder of this Chapter we develop some ideas that will lead to a 32bit version of RC 4 that has modest memory requirements and is many times more efficient that 8 -bit RC 4 on 32 -bit processors with vector processing.

### 6.1 RC4 as a Pool of Entropy

In the previous Chapters we discussed the idea of the internal state of RC 4 as a pseudo-random permutation. Later, when discussing Chameleon, we introduced the notion of the round-function of Chameleon as removing publicly known parts of the permutation and replacing it with new (pseudo) randomly selected information. We now wish to expand these notions, removing the restriction that the state contain a permutation.

In the permutation stored in the internal state of RC 4 and Chameleon the randomness is contained in the order that the values appear. The actual values contain no information, since every possible value for the alphabet used appears exactly once. If we remove the restriction that the state contains a permutation, then the actual values, in addition to their positions,
will contain information. We can then draw upon the entropy contained in the state by looking at some position, reading the value to obtain the information. The position chosen, as well as the value stored there, determine the output. We can then replenish the entropy of the state (since the output will presumably become public knowledge) by assigning a new value to that position.

This generalization of the Chameleon idea allows new possibilities when designing ciphers. In particular, since the restriction that the state be a permutation has been lifted, there is no particular reason to keep the state size as $N$. It can be modified, taking a smaller number of values. This is of particular interest to us in the design of 32-bit RC4-like ciphers since we don't want to have to store 16 Gigabytes of data.

### 6.2 A First Try

In this Section we attempt to use the above ideas in a cipher. However, the result is an insecure cipher.

### 6.2.1 Description

Suppose that we have two streams, $a$ and $b$ that take values in $\{0 \ldots N-1\}$ and $\{0 \ldots M-1\}$, respectively. Let $A[]$ be a list of $N$ values taken from the $M$ possible values for the $b$ stream. We construct a finite state machine that does the following;

1. Output $A[a]$
2. Assign $b$ to $A[a]$

This simple machine implements the ideas developed in the previous Section. It stores a pool of entropy, $A[$ ], from which outputs are selected and immediately replaced by new information. Basically, what this machine does is reorder the outputs of the $b$ stream.

Before the machine can function it requires $N$ initial values to populate $A$. These can simply be obtained from the first $N$ outputs of the $b$ stream.

In order to construct a cipher from this machine we need to provide the streams. To get our 32 -bit version we will use $N=2^{8}$ and $M=2^{32}$. Before specifying the exact streams, however, we will explore some security issues. First of all, what exactly do we gain by using this machine if it requires a 32-bit PRNG? In fact, we need two streams. The increase in security is a
consequence of the fact that outputs from the $b$ stream are not immediately public.

### 6.2.2 Usefulness

Suppose that $b$ is a PRNG that has some desirable properties, namely an approximately uniform distribution, a long cycle length, and high efficiency. Suppose further that this PRNG is cryptographically weak and cannot be further improved without substantially reducing efficiency. (Designing such ciphers that are cryptographically strong is, of course, a hard problem.) By adding some additional entropy, in the form of an additional stream, using the machine described above, we improve security in an important way: consecutive outputs of the PRNG are not necessarily consecutive in the output from the machine. In fact, it is not possible to guess when a particular output from the PRNG will occur in the output from the machine without knowing the behaviour of the $a$ stream.

As an example of how this is useful, consider two RC4 streams as the $a$ and $b$ streams. Many of the attacks developed against RC4 would be defeated because of the loss of position information on particular values from the $b$ stream. Fluhrer and McGrew's distinguisher, for example, relies on a bias in the digraph probabilities of RC4. After hiding the $b$ RC4 stream behind our machine, the digraphs are no longer preserved, eliminating this particular bias. ${ }^{1}$ The bias in the second output byte also becomes much less useful since the position of the second output byte in the machine's output is not known. The key schedule weaknesses which rely on first output bytes are similarly much less useful. In fact, since virtually all the attacks previously described depend on knowing the (possibly relative) position of outputs, they are either defeated or greatly reduced in strength. Of course, the efficiency is less than half since we need two instances of RC4 in addition to our machine.

### 6.2.3 A Weakness

Suppose that the stream used for $b$ is weak in the following way: any two consecutive outputs determine the internal state and hence the entire stream. We can then attack the output from the machine described above in order

[^1]to determine the $b$ stream. We do this by first collecting $N+1$ outputs from the beginning of the output from the machine. Since only $N$ values are held in $A$, at least two of these outputs must have been adjacent in the output from $b$. It is then a simple matter to try all $N(N-1)$ pairs, determine a candidate $b$ stream, and compare the output. If the correct stream was chosen then, of the first $2 N+1$ outputs, $N+1$ will occur as the outputs from the machine. If the $a$ stream is constructed in such a way that it depends on $A$, then we can run the machine with a candidate machine to compare outputs directly. Note that there are $2 N$ possible places where the consecutive outputs could have occurred, meaning a further $2 N$ guesses.

The above attack requires about $2 N^{3}$ work. If the $b$ stream is stronger, say requiring $l$ words of output to determine the internal state, then the attack can be extended. First, in order to guarantee that a sequence of $l$ consecutive outputs appears, we need to consider $(l-1) N$ outputs from the machine. (A smaller number can be used if a small chance of this not happening is tolerable.) Then $((l-1) N)!/(l-1)(N-1)!\sim((l-$ 1) $N)^{l}$ guesses will guarantee the correct stream is found and a further $l N$ guesses for possible positions. The total complexity is thus $l(l-1) N^{l+1}$. For concreteness, this value is $2^{34.6}$ for $l=3$ and $N=256,2^{43.6}$ for $l=4$, and $2^{84.9}$ for $l=7$.

In addition to the above attack, it is also important to note that any bias in the frequency of output words from $b$ will appear in the output from the machine.

### 6.3 A More Secure Version

The previously described machine fails in essentially the same way that the mutating S-box fails to resist chosen plaintext attacks (see Section 4.1.5), the outputs directly reveal information about the internal state. For the above machine this means that the revealed information came directly from the $b$ stream. Although position information has been lost, the actual values still remain.

The most obvious way to secure the above machine is to isolate the internal state from the output. One simple way to do this is to take the sum (modulo $M$, of course) of two values in the state as the output. This dramatically reduces the amount of information that can be guessed about the internal state. The algorithm now requires three streams, $a, b$, and $c$, and is

1. output $A[a]+A[b] \bmod M$
2. Assign $c$ to $A[b]$

We now have $a$ and $b$ as streams that take $N$ values, and $c$ as the stream that takes $M$ values. There are, of course, many possible variations on this basic idea, like using other operations instead of summing, and updating $A[x]$, where $x$ depends on both $a$ and $b$.

### 6.4 Sheet Bend: A 32-Bit Cipher

Using the machine described in the previous Section, we propose a new cipher, which we call Sheet Bend ${ }^{2}$.

### 6.4.1 Description

Our new algorithm is remarkably similar to RC 4 in its construction. In particular, we use the $i$ and $j$ streams from RC 4 as the $N$ value streams (which we called $a$ and $b$ in the previous discussion). For the $M$ value stream we use a simple PRNG that is efficient on 32-bit machines and has a fairly uniform output. It is, by itself, insecure. However, it is believed that in this context it is adequate.

The complete round algorithm is

1. $i=i+1$
2. $j=j+A[i] \bmod N$
3. Output $A[i] \oplus A[j]$
4. $A[j]=c$

This should be quite familiar. The $i$ and $j$ are taken directly from RC4. The swap operation has been eliminated, however, in favour of the update in the final line. Also, the output function is simpler. This is to avoid outputting values that are contained in the current state, limiting the information that can be gained about the PRNG. Also, the order has been modified - the state is updated after the output. This is so that the output does not necessarily depend on the current value of the PRNG, further limiting the information that can be gained about this sensitive piece. Finally, note that the update uses $j$ instead of $i$. If $i$ were used, then an attacker would know that exactly

[^2]the most recent $N$ values from the PRNG are in the state. Since $j$ moves more randomly, this is much less certain.

The PRNG that we use has two secret values, $c_{1}$ and $c_{2}$ as well as two state values, $a$ and $b . a$ and $b$ take values between 0 and $M-1 . c_{2}$ will take odd values between 1 and $M-1$ while $c_{1}$ will be odd and between 1 and $\log _{2} M$. The PRNG algorithm is

1. $b=b+c_{2} \bmod M$
2. $a=\left(a<_{c} c_{1}\right)+b \bmod M$
3. Output $a$
where $<_{c}$ is a cyclic bit shift to the left. Note that the round operation is reversible, and hence it can be worked backwards to determine the stream from the beginning. It is recommended that $c_{1}$ and $c_{2}$ be odd in order to ensure that $b$ visits all possible values and the bits of $a$ are thoroughly mixed.

At this point it is prudent to note that the operation in the round algorithm uses an output function using XOR instead of addition. This is because addition is used in the PRNG. By using two non-commuting operations in this way the operation of the PRNG is more protected.

Of course, to get a complete cipher we have to specify $M, N$ and determine a key schedule. $A$ can be populated with the first $N$ values from the PRNG, so only $a, b, c_{1}$ and $c_{2}$ need come from the key. Since we are intending to use a 32 -bit processor, it makes sense to use $M=2^{32}$ while $N=256$ seems like a sensible choice from the perspective of memory requirements. We thus need $99=2(32)+31+4$ bits of information to start the PRNG. After $A$ is filled, we reset the secret values and initialize $i$ and $j$ with new values from the PRNG.

The complete algorithm is given in Figure 6.1. Note that $\gg$ is a bit shift (not cylic) and is equivalent to multiplying by the corresponding power of 2. Also note that || represents bitwise OR.

### 6.4.2 Security of the PRNG

It is fairly easy to discover that three consecutive outputs from the PRNG plus $\left(\log _{2} M\right)-1$ guesses is enough to determine all the secret information and hence the entire stream. First $c_{1}$ is guessed, after which two consecutive outputs, $z_{0}$ and $z_{1}$, can be used to give two conditions on the state at time 1. Before the state change $z_{0}=a_{0}$ and after the state change, $z_{1}$ is output, which is $z_{0}<_{c} c_{1}+b_{0}$, determining $a_{0}$ and $b_{0}$ with two consecutive

Input: 107-bit key $k$

1. Set $a=k \bmod 2^{32}, b=k \gg 32 \bmod 2^{32}$
2. Set $c_{1}=2\left(k \gg 64 \bmod 2^{31}\right)+1$
3. Set $c_{2}=2\left(k \gg 95 \bmod 2^{4}\right)+1, i=k \gg 99$
4. Repeat $N$ times
(a) $i=i+1 \bmod 2^{8}$
(b) $b=b+c_{1} \bmod 2^{32}$
(c) $a=a<_{c} c_{2}+b \bmod 2^{32}$
(d) $A[i]=a$
5. $b=b+c_{1} \bmod 2^{32}, a=a<_{c} c_{2}+b \bmod 2^{32}$
6. Set $c_{1}=a \| 1$
7. $b=b+c_{1} \bmod 2^{32}, a=a \ll{ }_{c} c_{2}+b \bmod 2^{32}$
8. Set $c_{2}=\left(a \bmod 2^{5}\right) \| 1, i=\left(a \gg 5 \bmod 2^{8}\right)$
9. Set $j=\left(a \gg 13 \bmod 2^{8}\right)$
10. Repeat once for each 32-bit word of output
(a) $i=i+1 \bmod N$
(b) $j=j+A[i] \bmod N$
(c) Output $A[i]+A[j] \bmod M$
(d) $b=b+c_{2}$
(e) $a=a<_{c} c_{3}+b$
(f) $A[j]=a$

Figure 6.1: Sheet Bend Algorithm
outputs. The third output gives $c_{2}$ since $z_{2}$, the third output, satisfies $z_{2}=z_{1}<_{c} c_{1}+b_{1}$, where the $b_{1}=b_{0}+c_{2}$. Thus $c_{2}$ is determined.

Note that there is no way to predict further complete outputs given only two consecutive outputs since each value of $c_{2}$ will give a different output. Thus almost all possible outputs will occur. The least significant bit of $c_{2}$ is known, however, since $c_{2}$ is always chosen to be odd. Thus the least significant bit of the output can be predicted.

Suppose that $a$ and $b$ have been determined at some time. Thus the least significant bit of $b$ can be determined for all times. At any point, then, a single known $a$ can produce a prediction on the least significant byte of the next output. Again, this does not extend to further time steps unless the rest of the output can be determined since otherwise an unknown bit will be shifted into the lowest position. However, only one bit of information is required.

Consider, now, the situation where $b$ has been determined at two different times, separated by some time. It is straightforward, then, to find $c_{2}$ from the difference between the $b$ values and the number of times steps between them. This done by dividing the difference by the number of time steps. If the number of time steps is even then there will be more than one possible solution (since there are no inverses of even numbers in $\mathbb{Z}_{M}$ for $M$ even) but these are few in number.

It is possible to eliminate some of the bit determining analysis mentioned above by allowing $c_{2}$ to be even. However, in these cases $b$ would always have the same least significant bit, possibly decreasing the cycle length and introducing biases into the output (since the least significant digit of $a$ will either always be the same or opposite the shifted bit from the previous output.)

Although this PRNG is weak, statistical analysis shows that it has a fairly uniform output. It is not certain how the weaknesses described above could be used to attack Sheet Bend, but the security is probably somewhat lower than if no output could be predicted without knowing three consecutive outputs. It is, of course, possible to modify the PRNG, possibly replacing it entirely. However, our proposed PRNG is efficient, making it a good candidate.

### 6.4.3 Efficiency

Sheet Bend is quite efficient on 32 -bit machines. The PRNG requires 3 operations for each 32 -bit value output. The rest of the algorithm requires three additions, two look-ups and one assignment, for a total of 9 instructions per

32 -bit word output. It thus requires only 2.25 operations per byte of output. The exact implementations details will affect how many clock cycles this will take, but it still compares favourably to other stream ciphers optimized for 32-bit machines, such as Rabbit [1] which requires 3.7 cycles per byte.

The efficiency of Sheet Bend can be improved much as RC4 can be by running two instances in parallel and utilizing vector computation. The look-ups cannot be parallelized, but all other instructions can be. The total number of instructions is then 6 instructions for the vector computations and 6 for the look-ups and assignments, for a total of 12 for each 8 bytes of output, or 1.5 instructions per byte. This assumes 64 -bit registers that can perform 8 -bit and 32 -bit additions plus 32 -bit rotations. These instructions have been available on Pentium processors since the introduction of the MMX instruction set. More recent processors have 128 -bit registers that allow vectors of four 32 -bit words to be manipulated. If rotations, additions are both available then four ciphers could operate in parallel, reducing the cost to 18 operations for each set of 432 -bit words output, or slightly less than one instruction per byte.

### 6.4.4 Further Modifications

Once the move has been made to vector addition and using several ciphers in parallel, the next step is to make them interdependent, further increasing security in much the same way that two interdependent RC4 ciphers contribute to higher security in RC4A (see Section 5.4.5.) For example, the same $A$ can be used for all ciphers (with different starting $i$ and $j$ to avoid collisions in the values used). This also reduces the memory requirements. A perhaps better way of combining the ciphers would be to exchange state information between the different PRNGs. For example, if two ciphers were used the $b$ values could be placed in a 64 -bit register and a 16 -bit shift on the whole register performed, mixing the information between the two values. Only one more instruction is required, but the PRNGs now operate with a much larger effective state size, increasing cycle length and attack complexity.

If 64 -bit registers are available, such as MMX registers, then the entire algorithm can be made 64 -bit. The only operations on 64 bits required are addition, bit rotations and memory movement, which are available for MMX registers. This strengthens the cipher by increasing the internal state size, increasing cycle lengths, and the amount of information required to guess or acquire for a successful attack, and increasing the key size. It can also, depending on the implementation details, increase the throughput by up
to a factor of 2 . The next logical step, using 128 -bit registers, would gain similar improvements.

### 6.4.5 Analysis

The most promising attack on the structure of Sheet Bend involves attempting to guess the internal state of the PRNG since it is known that only two consecutive values decrease the complexity of the attack to $2^{35}$ and three reduce it to 16 .

Suppose that $N / 2+2$ outputs have been gathered from the output. Since each output is the XOR of two words in $A$ and only one entry of $A$ is replaced per word output, some pairs of output words will be formed with one operand in common. In fact, $N / 2+1$ outputs are enough to guarantee that this happens. XORing the two outputs together will then produce a new value which is the XOR of the other two values used to form the two outputs. We can thus produce more values which are XORs of internal state values that were originally output.

Another way of using pairs of outputs with an operand in common is to take advantage of the commonality to guess three outputs together. By trying all $2^{32}$ possible values for the common value, three outputs from the PRNG can be guessed. Note that if they were consecutive, which is improbable, the PRNG is then compromised. The probability that they were consecutive is quite small, however; choosing three particular numbers from a list when picking four values happens with frequency $24 N^{-4}$, or about $2^{-27}$. Once the three values are found, a further 32 guesses are required to determine $c_{1}$ in the PRNG. A set of candidate values, producing a candidate stream, takes at least $N$ values to check. Finding the original pair of outputs required about $N^{2} / 4$ guesses. The total attack complexity is then about $2^{86}$. Note that 64 and 128 bit versions, as described in the previous Section, would result in a much higher attack complexity.

The predicted security level of Sheet Bend for the 32 -bit version is $2^{86}$ for a 99-bit key due to the attack described above. Longer keys would not give greater security. For the 64 -bit and 128 -bit versions the attack complexity would be greater than $2^{128}$ and thus the ciphers are predicted to have full strength for a 128 -bit key.

Another avenue for attacking Sheet Bend is timing attacks. In particular, Pentium class processors have different timings for rotations depending on the amount of rotation. Rotating by one bit takes less time than rotating by other amounts. Thus an attacker could possibly learn if $c_{1}$ is 1 or some other value. The amount of information gathered is relatively small, not
even one bit of state information on average. In those cases where $c_{1}$ can be determined, the previous attack is reduced in complexity to about $2^{81}$, which is still considered secure. This timing attack can be eliminated through careful implementation, for example using MMX registers where all rotations take the same amount of time.

### 6.5 Bowline: Another 32-Bit Cipher

In the development of the previous cipher we separated the function of the PRNG from the rest of the algorithm resulting in a separate, attackable component with a relatively small internal state. We can reintegrate the PRNG with the rest of the algorithm, however, to create a new algorithm that is faster and more cohesive, eliminating the weak isolated component. We call this new algorithm Bowline ${ }^{3}$.

The basic idea is to use the internal state $A$ as state information for the PRNG instead of maintaining a separate state. The algorithm is given in Figure 6.2. $A$ is a list of $N$ values between 0 and $M-1 . i$ and $j$ take values between 0 and $N-1$. $c_{1}$ is an odd key-dependent constant between 1 and $N . c_{2}$ is an odd key-dependent constant between 1 and $M . c_{3}$ is an odd key-dependent constant between 1 and $\log _{2} M$. Finally, $a$ and $b$ are variables between 0 and $M$. The algorithm is given for $N=2^{8}$ and $M=2^{32}$

Bowline is different from Sheet Bend in several respects. First, $i$ is not incremented by 1 , but by a key-dependent constant $c_{1}$. The idea here is simply to reduce the amount of public information and make the algorithm more key-dependent. The algorithm for the PRNG is incorporated, but changed slightly. $a$ and $b$ are eliminated, using elements of $A$ instead, making the whole algorithm more cohesive and possibly increasing the complexity of attack; it is no longer sufficient to determine the internal state of the PRNG, all of $A$ must be determined.

It should be noted that the round operation is reversible. This is often considered to be a feature of RC4 since it implies that information is not discarded from the state, potentially increasing the cycle length [7].

When choosing $c_{3}$ it should be considered that an odd value is likely better than an even value. This is because even values will not cause individual bits to visit all positions, as an odd value would. $c_{2}$ and $c_{1}$ should also be odd so that they are generators of $\mathbb{Z}_{M}$ (assuming $M$ is a power of 2 ).

The key schedule for Bowline was developed with speed as a primary goal. The key is copied into the state for the original PRNG used with

[^3]Sheet Bend which is then used to fill $A$. The constants $c_{1}, c_{2}$ and $c_{3}$ along with $i$ are initialized from further outputs of the PRNG. Finally, the round algorithm is executed $N$ times, ignoring the output, to reduce biases in the state.

### 6.6 Comparing Sheet Bend and Bowline

Although Bowline is derived from Sheet Bend they are quite different in operation due to the fact that Bowline operates as one unit while Sheet Bend has two separate components. From a security point of view this probably means that Sheet Bend is less secure since the PRNG can be attacked independently of the rest of the algorithm. From an analysis point of view, however, Sheet Bend is easier to deal with since it has two separate components, each of which is quite simple. From an efficiency point of view they are nearly equivalent and both can be easily expanded to use wider processors to increase the efficiency and security.

### 6.7 Conclusions

In this Chapter we explored the possibilities for expanding RC4 to take advantage of wider processors. The possibilities are quite wide. We presented a generalized concept of RC 4 in which the internal state no longer has to represent a permutation. This generalization greatly expands the possibilities of RC4-like algorithms. We also used this generalization to develop two example ciphers, Sheet Bend and Bowline.

## Input: 114-bit key $k$

1. Set $a=k \bmod 2^{32}, b=k \gg 32 \bmod 2^{32}$
2. Set $c_{1}=2\left(k \gg 64 \bmod 2^{7}\right)+1, c_{2}=2\left(k \gg 71 \bmod 2^{31}\right)+1$
3. Set $c_{3}=2\left(k \gg 102 \bmod 2^{4}\right)+1, i=k \gg 106$
4. Repeat $N$ times
(a) $i=i+c_{1} \bmod 2^{8}$
(b) $b=b+c_{2} \bmod 2^{32}$
(c) $a=a<_{c} c_{3}+b \bmod 2^{32}$
(d) $A[i]=a$
5. $b=b+c_{2} \bmod 2^{32}, a=a<_{c} c_{3}+b \bmod 2^{32}$
6. Set $c_{2}=a \| 1$
7. $b=b+c_{2} \bmod 2^{32}, a=a<_{c} c_{3}+b \bmod 2^{32}$
8. Set $c_{1}=a \bmod 2^{8} \| 1$
9. Set $c_{3}=\left(a \gg 8 \bmod 2^{5}\right)\left\|1, i=\left(a \gg 13 \bmod 2^{8}\right)\right\| 1$
10. Repeat 256 times, ignoring output, then once for each 32 -bit word of output
(a) $i=i+c_{1} \bmod N$
(b) $j=j+A[i] \bmod N$
(c) Output $A[i]+A[j] \bmod M$
(d) $A[j]=A[j]+c_{2}$
(e) $A[i]=A[i]<_{c} c_{3}+A[j]$

Figure 6.2: Bowline Algorithm

## Chapter 7

## Card Game Ciphers

In Neal Stephenson's novel Cryptonomicon [25] two of the characters exchange secret messages using a cipher called "Pontifex" whose mechanism employs a deck of playing-cards. Pontifex is a real cipher, developed by Bruce Schneier [24], who calls it Solitaire. The concept of using a deck of cards as a tool for building ciphers has several appealing qualities. First of all, there is the coincidence that 52 is twice 26 , meaning that such a cipher can easily use the Latin alphabet without any awkwardness. Also, any algorithm that is to be employed by hand using such a primitive tool has to be fairly simple itself. Designing a cipher that fits these constraints and is yet secure offers a significant challenge.

Schneier's original aim was to design a cipher that would have security comparable to computerized ciphers. This goal was not met since Solitaire has a bias (see [3]), however it was successful in that it spawned an interest in playing-card ciphers. For example, Paul Crowley [2] has developed a cipher called Mirdek and John Savard [23] has developed another.

There are several operations that are possible on a deck of cards, limited mainly by imagination. Many useful operations can be found in card games while others are most directly suited to ciphers. The shuffling and dealing procedures used in various games can be useful. For example, cutting the deck is useful, changing the relative position of many cards at once. Cutting could be extended, as in Solitaire, to a double cut where the deck is divided into three Sections and two are exchanged. The deck can also be divided into piles or spread out on a table in various arrangements.

An important difference between card games and ciphers is that the latter need to be deterministic, at least to some extent, to allow decryption while games typically rely on randomness. Random operations can be con-
verted into deterministic ones in several ways. One way is to identify cards that control the position of a cut or mark out different Sections in the deck. This is done in Solitaire, where the position of the two jokers is used to identify how to cut the deck. Another way is to count through the deck to a certain position. These operations can be somewhat time consuming as compared to their random counterparts.

Another difference between card games and ciphers is that ciphers have to manipulate information in a way that can be applied to text. This can be done by identifying cards with letters (as in Mirdek) or with numbers (as in Solitaire). The cards can be assigned letters in a straightforward way, identifying two cards (excluding jokers) with each letter. The use of numbers is conducive to counting into a deck and other operations, such as modular arithmetic while letters are more suited to ciphers that manipulate the plaintext directly.

### 7.1 Card-Chameleon

As previously mentioned, the mutating S-box described in Chapter 4 was designed in an attempt to produce a card game cipher that was as secure as computerized ciphers. There are several features that make it efficient and convenient for use with playing-cards. First of all, a permutation is easy to encode in the order of cards. Second, the fact that there are no operations other than table look-ups and swaps reduces the amount of human computation that is necessary; swaps and look-ups can be easily accomplished as described below.

### 7.1.1 Description of Card-Chameleon

The card game implementation of Chameleon, which we will call CardChameleon to distinguish it from other implementations, uses a standard deck of 54 cards. Two jokers are optional, being used to add a space character. If jokers are used, they must be distinguishable. We will assume that one is red while the other is black. The deck is divided into two sets, the black cards and the red cards. Each set is mapped onto the cipher alphabet which consists of 26 letters and a space. The red cards are mapped by ordering the cards, ace of hearts to king of hearts followed by ace of diamonds to king of diamonds, and mapping each card to the letter of the alphabet in the corresponding position in the alphabetical order. Thus the ace of hearts is mapped to ' A ' and the king of diamonds is mapped to ' Z '. The red joker
is mapped to space. Similarly, the black cards are ordered, beginning with the ace of spades, and mapped to the letters.

To begin encrypting, the sending and receiving parties first agree on a key which consists of an ordering of a complete deck of cards. The sender encrypts using the algorithm given in Figure 7.1. Decryption uses a similar process, given in Figure 7.2.

Since the position of spaces is important it is recommended that underscores or hyphens be used to mark the positions of spaces to eliminate any ambiguities.

Conceptually there are two parts to this cipher. The first part is a set of 27 black/red pairs. Each pair describes the value of the S-box for one input. Moving from a black card to its paired red card is enciphering. Moving from a red card to its paired black card is the inverse. The second part is the ordering of the pairs, which is really the same as the ordering of the black cards. This encodes the cycles of values for $j$ with the current value being the top card.

### 7.1.2 Analysis of Card-Chameleon

The key used for Card-Chameleon is an ordered deck of 54 cards. However, many orderings are equivalent since it is only the ordering of the two subsets, the reds and the blacks, that matter. The number of non-equivalent keys is thus $(27!)^{2} \sim 2^{186}$. This is certainly large enough to protect against brute force attacks. However, there are attacks with lower complexity.

The use of an initialization vector limits the effectiveness of the attacks described in Section 4.1.5. The number of different initialization vectors is $27!\sim 2^{93}$ which is certainly large enough to avoid collisions, especially since messages are encrypted by hand. Note that to launch an attack on the initial state, 26 collisions on one IV are required (the 27th state element can be determined by elimination.) This number can be reduced by partially guessing the state. Given the low probability of one collision, however, it is not likely that any attack based on finding IV collisions would be feasible.

It should be noted that the attack described in Section 4.4.3 does not apply to Card-Chameleon. This is because, unlike the case of Soft-Chameleon, the IV does not induce a publicly known permutation on the state in CardChameleon. The IV does induce a permutation, but this permutation is dependent on the ordering of the black cards (the cycle $K[]$ ). Since this ordering is private, key dependent information, the induced cycle cannot be determined without guessing the ordering.

A closer look at the mechanical aspects of Card-Chameleon reveals some

1. Take the key deck face up. Deal out two piles face up, placing each red card on one pile and each black card on the other. Be sure to preserve the order exactly.
2. Make a new pile, face up, by interleaving the two piles. This is done starting with the first black card, then the first red card, the second black card, second red, and so on. The new pile should have a red card on top and alternate between red and black cards.
3. Create an initialization vector by shuffling the black cards from a spare deck. (Alternatively, create the initialization vector beforehand from the same deck, writing down the sequence.) Interpret each card as a letter and write the sequence down as the first part of the message. For each card in this sequence do the following steps.
(a) Find the same black card in the key deck.
(b) Look at the red card above this black card. Interpret this as a letter, $t$.
(c) Find the black card for the letter $t$.
(d) Exchange the red card above this black card with the top card.
(e) Move the black card and the next card (red) to the bottom.
(f) Put the top two cards (one red, one black) on the bottom.
4. Encrypt each letter in the plaintext with the following steps
(a) Find the black card in the key deck corresponding to the plaintext letter.
(b) Look at the red card above this black card. Interpret this as a letter.
(c) Find the black card in the deck corresponding to this new letter.
(d) Look at the red card above this black card. Interpret this as a letter and write it down as the ciphertext.
(e) Exchange this red card with the top card (also red).
(f) Move the top two cards (one red, one black) to the bottom.
5. Send the message, which has 27 letters more than the plaintext.

Figure 7.1: Card-Chameleon Encryption Algorithm

1. Take the key deck face up. Deal out two piles face up, placing each red card on one pile and each black card on the other. Be sure to preserve the order exactly.
2. Make a new pile, face up, by interleaving the two piles. This is done starting with the first black card, then the first red card, the second black card, second red, and so on. The new pile should have a red card on top and alternate between red and black cards.
3. The first 27 characters in the message are the initialization vector. For each letter in the sequence do the following.
(a) Find the black card in the key deck corresponding to the letter.
(b) Look at the red card above this black card. Interpret this as a letter, $t$.
(c) Find the black card for the letter $t$.
(d) Exchange the red card above this black card with the top card.
(e) Move the black card and the next card (red) to the bottom.
(f) Put the top two cards (one red, one black) on the bottom.
4. Decrypt each letter in the ciphertext ( 28 letters into the message) with the following steps
(a) Find the red card corresponding to the plaintext letter.
(b) Look at the black card below this red card. Interpret this as a letter.
(c) Find the red card corresponding to the new letter.
(d) Look at the black card below this red card. Interpret this as a letter and write it down as the plaintext.
(e) Exchange the red card with the top card (also red).
(f) Move the top two cards (one red, one black) to the bottom.

Figure 7.2: Card-Chameleon Decryption Algorithm
characteristics that differentiate it from other card based ciphers. In particular, there is no arithmetic involved. The only operations required are finding a card in the deck and moving cards. Encrypting or decrypting letters involves only a few steps, making it quicker.

Since the IV weakness found in Soft-Chameleon does not affect CardChameleon the best know attack is the branch and bound attack. As previously mentioned in Section 4.4.2 the complexity of this attack is estimated to be about $N!$. For Card-Chameleon this is $2^{93}$, which is more than enough to be considered secure.

As with any cipher, there is always the possibility of new attacks. CardChameleon, along with Soft-Chameleon, has not been examined by many people and for this reason further work is necessary before they can be considered for practical applications.

### 7.2 Pocket-RC4

In this Section we describe an adaptation of RC 4 for use with playing-cards. We call this new cipher Pocket-RC4.

### 7.2.1 Description

The permutation that is at the heart of RC 4 , which can be easily stored in the ordering of a deck of cards, suggests that a playing-card version would be feasible. However, Knudsen's attack is feasible for $N=32$, so the most likely scenarios, using $N=26$ or $N=27$, do not result in secure ciphers. For this reason we develop a new playing-card version of RC4, Pocket-RC4. It is similar to RC4B with $N=27$, with a modified key schedule, but uses several ideas from Card-Chameleon in its implementation. The encryption algorithm is given in Figure 7.3.

Although Card-Chameleon uses letters in its correspondence with cards, this is not useful for Pocket-RC4. Instead, the numbers 0 through 26 are used because the algorithm requires modular arithmetic. The outputs from the algorithm are numbers modulo 27 which are added to the plaintext, letter by letter (with a correspondence between letters and space, and numbers), to form the ciphertext.

In order to use RC4B we must have a means of implementing $S$. This is done in exactly the same way as for Card-Chameleon: the red and black cards are arranged in pairs. In the case of RC4B we don't have to do reverse look-ups, but looking up a value in $S$ is done by finding the index in the black cards and looking at its red partner.

The key schedule is somewhat different since the key is the deck of cards, as in Card-Chameleon, rather than a byte string. However, the essence is the same. A string of 27 letters (and spaces) is used for the initialization vector. The IV can be chosen in any random way. We suggest shuffling the deck and drawing cards with replacement.

Decryption is identical except where the initialization vector is concerned. The IV comes from the first 27 letters of the plaintext instead of being chosen at random. Also, the values are subtracted modulo 27 from the ciphertext instead of added as is done for the plaintext.

### 7.2.2 Security

The security of Pocket-RC4 is probably higher than RC4B for the same $N$ because shuffling a card deck can quickly produce a uniform permutation while the exchange shuffle is still biased after $2 N$ swaps. Although PocketRC4 has not been studied in any great detail, we can estimate its security by looking at the results from RC4. There are biases that are detectable, but it seems unlikely that the amount of output necessary to detect these biases would be produced by a human with a deck of cards, especially for any one key. Therefore the most likely threat comes from Knudsen-style branch and bound attacks.

As mentioned in Section 5.4.4 the complexity for this attack appears to be slightly lower than $N$ !. For $N=27$ this is about $2^{93}$. Provided that the estimate is close, this means that this attack is unlikely to be practical against Pocket-RC4.

### 7.3 Conclusions

Although playing-cards are not tools that are often used by cryptographers, the above ciphers show that this area has many possibilities. The peculiarities of playing-card decks, having a convenient number cards and easily storing a permutation, make them well suited to ciphers such as Chameleon and RC4. By looking at previously designed and studied ciphers, such as RC 4 , it is possible to expand the repertoire of tools for the design of playingcard ciphers.

1. Take the key deck face up. Deal out two piles face up, placing each red card on one pile and each black card on the other. Be sure to preserve the order exactly.
2. Make a new pile, face up, by interleaving the two piles. This is done starting with the first black card, then the first red card, the second black card, second red, and so on. The new pile should have a red card on top and alternate between red and black cards.
3. Create an initialization vector by shuffling the cards and drawing one a total of 27 times, producing a sequence of 27 letters. Send this as the first 27 letters of the ciphertext. (This could be done before ordering the deck with the key or using a second deck.)
4. For each card in the IV sequence do the following steps.
(a) Find the same black card in the key deck.
(b) Exchange the red card above this black card with the top card.
(c) Move the black card and the next card (red) to the bottom.
(d) Put the top two cards (one red, one black) on the bottom.
5. Set $j$ to the value of the bottom red card.
6. Encrypt each letter in the plaintext with the following steps
(a) Add the value of the top red card to $j$ modulo 27.
(b) Find the black card corresponding to $j$.
(c) Add the red card above the $j$ black card to the top red card modulo 27.
(d) Add the plainext letter to this number, modulo 27.
(e) Exchange the two red cards.
(f) Move the top black/red card pair to the bottom.
7. Send the message, which has 27 letters more than the plaintext.

Figure 7.3: Pocket-RC4 Algorithm

## Chapter 8

## Future Work and Conclusions

RC4 has several strengths. Probably the greatest of these is the simplicity of its round algorithm. It is also quite easy to modify. Compared to many ciphers, such as DES, it has high efficiency, especially in software.

Possibly the greatest weakness of RC 4 and similar ciphers is the time it takes to complete the key schedule. It is not easy to design a fast key schedule for these ciphers; the internal state must be filled or shuffled and the size of the internal state puts a lower bound on the number of operations required. As well, analysis has shown that the biases caused by insufficient shuffling in the key schedule can produce biases in the output, making it even more difficult to have an efficient key schedule.

To see the importance of this issue, consider the amount of time it takes to complete the key schedule in RC4. Creating the initial permutation requires about $3 * 256$ operations (one assignment, one increment and one compare for each element in $S$ ). The shuffling takes $10 * 256$ operations (three additions, one compare, three look-ups and three assignments.) The total is then about 3300 operations. Using this many operations we could encrypt about 400 bytes. In a context like Wired Equivalent Privacy (WEP) where the packets encrypted are small (less than 1500, and often much less) and a new initialization vector is required for every packet this is quite inefficient. Designing RC4-like stream ciphers with a very fast key schedule is an area for future research.

Although the number and strength of attacks against RC4 is increasing, it still draws interest because of its simplicity. In addition, there is much room for modification. For example, we provided a 32 -bit cipher with a
round algorithm based on RC4. There is much more work that can be done in this area. In addition, the possibilities of playing-card ciphers based on RC4 have only been lightly touched on.

In addition to the work done with RC 4 in this thesis we also introduced Chameleon. This cipher, like RC4, has a simple round algorithm that invites analysis. Also like RC4 there are many possibilities for modifications. Finally, there are many possibilities for using ideas from both Chameleon and RC4 in the development of new ciphers such as the methods used in designing Sheet-Bend and Bowline.

## Bibliography

[1] Martin Boesgaard, Thomas Pedersen, Mette Vesterager, and Erik Zenner, Rabbit: A new high-performance stream cipher, Fast Software Encryption 2003, Lecture Notes in Comput. Sci., vol. 2887, Springer, Berlin, 2003, pp. 307-329.
[2] Paul Crowley, Mirdek: A card cipher inspired by "solitaire", http: //www.ciphergoth.org/crypto/mirdek/.
[3] _ Problems with bruce schneier's "solitaire", http://www. ciphergoth.org/crypto/solitaire/.
[4] H. Finney, An RC4 cycle that can't happen, Post in sci.crypt, September 1994, 1994.
[5] Scott Fluhrer, Itsik Mantin, and Adi Shamir, Weaknesses in the key scheduling algorithm of RC4, Selected Areas in Cryptography, Lecture Notes in Comput. Sci., vol. 2259, Springer, Berlin, 2001, pp. 1-24. MR MR2054424 (2004m:94049)
[6] Scott R. Fluhrer and David A. McGrew, Statistical analysis of the alleged RC4 keystream generator, Fast Software Encryption 2000, Lecturen Notes in Comput. Sci., vol. 1978, Springer, Berlin, 2000, pp. 1930.
[7] Jovan Dj. Golić, Linear statistical weakness of alleged RC4 keystream generator, Advances in Cryptology-EUROCRYPT '97 (Konstanz), Lecture Notes in Comput. Sci., vol. 1233, Springer, Berlin, 1997, pp. 226-238. MR MR1603060
[8] A. Grosul and D. Wallach, A related key cryptanalysis of RC4, Tech. Report TR-00-358, Department of Computer Science, Rice University, June 2000.
[9] R. Jenkins, Isaac and RC4, http://burtleburtle.net/bob/rand/ isaac.html.
[10] Lars R. Ksnudsen, Willi Meier, Bart Preneel, Vincent Rijmen, and Sven Verdoolaege, Analysis methods for (alleged) RC4, Advances in Cryptography - ASIACRYPT '98, Lecture Notes in Comput. Sci., vol. 1514, Springer, Berlin, 1998, pp. 327-341.
[11] I. Mantin, Cryptanalysis of RC4 in different usage modes, Advances in Cryptography - ASIACRYPT 2005, to appear.
[12] Itsik Mantin, Analysis of the stream cipher RC4, Master's thesis, The Weizmann Institute of Science, 2001.
[13] , Predicting and distinguishing attacks on RC4 keystream generator, Advances in Cryptography - EUROCRYPT 2005, Lecture Notes in Comput. Sci., vol. 3494, Springer, Berlin, 2005, pp. 491-506.
[14] Itsik Mantin and Adi Shamir, A practical attack on broacast RC4, Fast Software Encryption 2001, Lecture Notes in Comput. Sci., vol. 2355, Springer, Berlin, 2001, pp. 152-164.
[15] Mitsuru Matsui, Linear cryptanalysis method for des cipher, Advances in Cryptography - EUROCRYPT 1993, Lecture Notes in Comput. Sci., vol. 765, Springer, Berlin, 1993, pp. 386-397.
[16] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC press, 2001.
[17] Ilya Mironov, (Not so) random shuffles of RC4, Advances in Cryptology - CRYPTO 2002, Lecture Notes in Comput. Sci., vol. 2442, Springer, Berlin, 2002, pp. 304-319. MR MR2054828 (2004m:94068)
[18] S. Mister and S. E. Tavares, Cryptanalysis of RC4-like ciphers, Selected Areas in Cryptography (Kingston, ON, 1998), Lecture Notes in Comput. Sci., vol. 1556, Springer, Berlin, 1999, pp. 131-143. MR MR1715807
[19] Souradyuti Paul and Bart Preneel, Analysis of non-fortuitous predictive states of the RC4 keystream generator, Progress in CryptologyINDOCRYPT 2003, Lecture Notes in Comput. Sci., vol. 2904, Springer, Berlin, 2003, pp. 52-67. MR MR2092291 (2005e:94194)
[20] Souradyuti Paul and Bart Preneel, A new weakness in the $R C 4$ keystream generator and an approach to improve the security of the cipher, Fast Software Encryption 2004, Lecture Notes in Comput. Sci., vol. 3017, Springer, Berlin, 2004, pp. 245-259.
[21] M. Pudovkina, Statistical weaknesses in the alleged rc4 keystream generator, Cryptology ePrint Archive 2002-171, IACR, 2002.
[22] Andrew Roos, A class of weak keys in the RC4 stream cipher, http: //marcel.wanda.ch/Archive/WeakKeys, Post in sci.crypt, September 2004.
[23] John Savard, Fun with playing cards, http://home.ecn.ab.ca/ ~jsavard/crypto/pp0105.htm.
[24] Bruce Schneier, The solitare encryption algorithm, http://www. schneier.com/solitaire.html.
[25] Neal Stephenson, Cryptonomicon, Avon Books, New York, 1999.
[26] A. Stubblefield, J. Ioannidis, and A. Rubin, Using the Fluhrer, Mantin and Shamir attack to break WEP, Proceedings of the 2002 Network and Distributed Systems Security Symposium (2002), 17-22.
[27] Bartosz Zoltak, Vmpc one way function and stream cipher, Fast Software Encryption 2004, Lecture Notes in Comput. Sci., vol. 3017, Springer, Berlin, 2004, pp. 210-225.


[^0]:    ${ }^{1}$ National Science and Engineering Research Council Undergraduate Student Research Award

[^1]:    ${ }^{1}$ Note that digraph biases in the $a$ stream may allow digraphs to be exposed in the $b$ stream. If the $a$ stream has a bias towards three consecutive equal values, digraphs in the $b$ stream will be output directly more often, passing $b$ 's bias into the output. However, the bias will be greatly reduced.

[^2]:    ${ }^{2}$ A Sheet Bend is a knot that is used to tie together two ropes of different diameters.

[^3]:    ${ }^{3}$ A Bowline is a simple knot that doesn't slip.

