Evaluation of RC4 Stream Cipher

July 31, 2002

Information Security Research Centre
Queensland University of Technology
Ed Dawson, Helen Gustafson,
Matt Henricksen, Bill Millan

Evaluation of RC4 Algorithm

Executive Summary

This is a report on the analysis of the stream cipher, RC4. For this algorithm the evaluators have

(i) investigated the security of RC4 against various attacks;
(ii) evaluated basic cryptographic properties;
(iii) evaluated statistical properties;
(iv) summarised previous attacks;
(v) estimated practical security of RC4 in SSL;
(vi) surveyed the speed.

RC4 is a table based binary additive stream cipher which uses the output word of the keystream generator for its keystream. For most applications the word length is $n = 8$.

RC4 is a unique design for a keystream generator. The large internal memory of RC4 and the dynamic updating of tables imply that RC4 is secure from conventional attacks on keystream generators. Over the past eight years RC4 has been extensively evaluated in the open literature. Several interesting properties of RC4 have been discovered, and some weaknesses of the original initialisation process have been found. However, to date there has been no weakness discovered that is serious enough to conclude that RC4 is insecure for use as a stream cipher, provided the word size is sufficiently large and the initial part of the output sequence is suppressed. Our investigations reveal that using a word size of $n = 8$ and suppressing the first few output bytes prevents most attacks.

The keystream output from RC4 was analysed using standard statistic tests. The only weakness that was identified was a strong bias in the second byte of the keystream. This bias can be used to identify the use of RC4 and is a leakage of information in second byte of cryptogram. This statistical weakness can be prevented by suppressing the first two bytes of the keystream.

Previous attacks on RC4 were evaluated. The type of attack on RC4 found to be of most concern was the attack on RC4 in WEP. The original use of RC4 in WEP was demonstrated to be seriously flawed because of a “bad” re-keying process. To prevent such attacks on other uses of RC4 in communications systems requiring frequent resynchronisation a secure method to re-key RC4 is required. The use of a hash function such as SHA to mix the key and initialisation vector is recommended. However, a precise re-keying process needs to be defined in order for its security to be properly evaluated. For example, if a hash function is used to mix the key and IV, these details must be standardised. However, we note that using a hash function adds implementation complexity and reduces initialisation speed, which may not be suitable for some applications.

A second attack on RC4 which should be noted is a distinguisher attack. The goal of such an attack is to distinguish output keystream of RC4 from random keystream. For RC4, such distinguishers are practically impossible to avoid given a suitably large keystream. However, in most application this does not cause any security problems since the actual use of RC4 is not intended to be secret.

The use of RC4 in SSL appears secure provided the first two words of output are suppressed.

RC4 is very fast in software. However, in hardware RC4 does not operate faster. In certain applications requiring high speed dedicated hardware this may cause some problems.

In summary, the evaluators recommend RC4 as being suitable for most applications subject to the above recommendations. These recommendations in point form are:

- The word size used should be n=8
- The first two output words must be suppressed
- A specific mechanism for re-keying should be standardised

1 Introduction to RC4

The first known use of a dynamic permutation in stream cipher design was RC4, designed by Ron Rivest in 1987. The details of the algorithm remained proprietary until 1994 when it was posted to an Internet mailing list [Ano94]. The common use of this cipher in communications protocols such as SSL and WEP may make RC4 the world's most widely used stream cipher.

The RC4 algorithm has two parts (we let n=8 be the nominal word size):

- Initialisation mode: accepting a variable key as a sequence of bytes
- Output mode: producing a pseudo-random byte sequence

Both these modes update the table of 256 bytes. The complete algorithm is given below in Figure 1.

```
Initialisation Mode:
Input: n=8,
          key length (in bytes) = k;
          S[i]=i for i=0 to 255
          Key :  K[0…(k-1)]

j=0;
For i=0 to (n-1)
    j=j+S[i]+K[i mod  k]
    swap(S,i,j)
i=j=0;

Output Mode:
Repeat
   i=i+1
   j=j+S[i]
  swap(S,i,j)
  output=S[S[i]+S[j]]
Until done
```

Figure 1 - The RC4 Algorithm

Let the k-word key be given in the array K[0…(k-1)]. The parameter n indicates the bit-width of the permutation S, which occupies $n*2^n$ bits of memory. For real applications, n=8 is used as it is a good trade-off between security and memory requirements, and it is also natural and easy to implement. Two extra bytes of memory, i and j, are used as pointers into the table. One byte pointer is incremented mod 2^n and the other takes on pseudo-random values generated by table lookups (and also by the key in Initialisation mode). The function swap(S,a,b) swaps the values in S[a] and S[b], thus updating the table while conserving the permutation property. These are both already in temporary variables, so the swap operation takes 2 clock cycles.

The algorithm of RC4 can be seen to have 4 stages. The essential parts of this cipher are:

- A simple basic counter, modulo the table size
- Another pointer that is updated using a table value
- Output a value that depends on table value of both pointers
- Update the table

A simplistic model of software suggests that RC4 uses 2 operations for each of these stages, and so, given some system inefficiencies, should run at a little less than the underlying clock speed in software. This is confirmed by some optimised implementations of the evaluators (see Appendix D). The memory requirements are 256 bytes for the dynamic table and a few other bytes for the pointers and temporary values. This is just over ¼ Kbyte memory and so it is suited for small applications with restricted memory.

2 Known Weaknesses

Since it first appeared in the open literature in 1994, RC4 has been extensively examined. The published attacks on RC4 seem to follow one of the following approaches:

- Identify weak keys [Roo95], [Wag95], or related key attack [GW00].
- Distinguishing RC4 output from a random sequence [Goli97], [FM00].
- Backtracking Algorithms [MT98], [K+98], [Goli00].
- Attacks based on weakness in the Initialisation [MS01], [FMS01].

The first known weaknesses in RC4 were reported on the sci.crypt mailing list in 1995 by Roos [Roo95] and Wagner [Wag95]. They described several classes of keys that have specific weaknesses including predictable output or output that leaks key information. Later a related key attack was observed for long keys (2048 bits) [GW00].

Since the output an RC4 stream cipher is used to encrypt the plaintext by bit-wise XOR, any observable bias in the output can be used as the basis for an attack. A correlation was detected by Golic [Goli97] between bytes at time t and t+2. Many stronger correlations were later reported by Fluhrer and McGrew [FM00].

Algorithms that attempt to guess the internal state and then check for consistency with known output have been studied independently by several researchers and the results reported in [MT98], [K+98] and [Goli00]. The consensus is that these attacks are infeasible for n=8 due to very large computational complexity.

The most significant attacks on RC4 have been based on exploiting the simplicity of the initialisation algorithm to discover an observable bias in the first few bytes of the output sequence. A bias in the second output byte was reported in [MS01]. The value zero occurs with twice the expected probability for a random sequence. Some parity preserving properties of the initialisation process are used in [FMS01] to define a large class of weak keys. A bias in the first byte was recently reported in [Mir02]. This result stems from a new model for the analysis of RC4 style ciphers.

These problems with RC4 have seriously reduced the security of wireless LANs due to the failure of WEP, the link-layer security protocol for 802.11 networks, as demonstrated in [SIR01] which describes a practical attack.

We now examine the results of each of these weaknesses.

2.1 Correlations and Distinguishers

These attacks can be used to distinguish output of RC4 from random output. It should be noted that in general many encryption algorithms suffer from such weakness as was shown in the recent paper [CHJ02]. In most applications the actual use of RC4 is not kept secret so that leakage of this information is not significant.

It is known that RC4 output sequences are slightly biased [Jen94]. A gap length test has been discussed also in [MT98], where testing results on 2^{30} elements of RC4 keystream are reported. Define the gap at i for a sequence x to be the smallest integer t>0 such that S[i]=S[i-t-1], where S[] is the RC4 table. For a random sequence in which each element takes on one of 2^n values the probability that t=k is given by $((2^n-1)/2^n)^k * 1/2^n$. The gap test looks at the ratio of the actual to the expected gap probability, for a range of word size n=2 to n=6 and gap length from zero to six. For all values of n, gaps of length 0 are more likely than expected, and gaps of length 1 are less likely than expected. It has also been observed that after a gap length of zero, the probability that S[i]=0 is lower than expected and the probability that S[i]=2^n-1 is higher than expected.

Linear statistical weaknesses of RC4 were considered in [Goli97]. The main result shows that the LSB of Z_t XOR Z_{t+2} is correlated to 1 with coefficient close to 15 * 2^{-3n} where n is the word size of RC4-n. The length of known keystream required to detect this weakness is around $64^n/255$. For n=8 it is realistic to collect the 2^{40} bits required to detect this correlation. Once detected, this distinguished RC4 from random and can be used to find the word-size, were this to be kept secret.

The main point of the linear sequential circuit approximation method applied to RC4 is that the table at time t can be approximated by the table at time t-1. The correlation coefficient is calculated (or estimated) using some heuristic arguments including modelling unknown values as random. It is reported that computer simulations confirm the result.

The cycle structure of RC4 was examined in detail in [MT98]. Results of the success of some “tracking” algorithms on scaled down RC4 are provided. The results show that RC4 can be distinguished from random if only a fraction of a full period is available. However for n=8 this is still impractical. The tracking attack offers considerable speedup over exhaustive search. With the word size n=5, the state can be recovered using 2^{42} steps where the nominal keyspace of the system is 2^{160}.

A distinguisher requiring much less known keystream was reported in [FM00]. This paper focuses upon the calculation of digraph probabilities, where a digraph is considered to be a consecutive pair of n-bit output words. Given the assumption that all initial states have equal probability, the digraph probabilities are calculated by counting the number of internal states that are consistent with each digraph. This counting is feasible since only 4 words of the RC4 state influence the current output.

The full digraph probabilities for n=3,4,5 were counted in [FM00] with about 2^{40} operations and found to be consistently non-uniform. The largest bias found was that the (0,0) digraph occurs with probability $1+2^{-n+1}$ times greater than would be expected for a random sequence. There are ten other digraph types that have bias $1+2^{-n}$. These anomalies are studied in [FM00] where some examples are shown to explain how they occur. Many future directions of this line of research are briefly mentioned in [FM00]. These include trigraph counting and lagged digraphs.

The length of known keystreams required to detect these anomalies are estimated in [FM00] using some theoretical arguments. They estimate that $2^{30.6}$ bytes of output are required for RC4 with n=8. It is interesting to note that, using the same methods, they estimate the keystream required to detect the bias reported in [Goli97] at $2^{44.7}$, which is a bit more than the estimate of 2^{40} that appeared in [Goli97] following different heuristics. The evaluators are of the opinion that it may be difficult to know the exact complexity of some algorithms until they are much more closely examined. Another complication is the (often implicit or unstated) choice of probabilities for false positives and false negatives in hypothesis based statistical testing.

Mantin and Shamir discovered a major statistical weakness [MS01] in the probability distribution of the second output byte of a RC4 keystream. The value zero occurs with twice the expected probability for a random sequence. This bias results in an efficient distinguisher between RC4 and random. Experiments show that only 200 different output streams are required to detect that they are generated by RC4. The bias is demonstrated to be caused by the PRNG starting with j=0. The weakness only occurs at the start since j has that value. Later in the keystream j takes on uniformly random value so the bias towards zero vanishes.

The authors of [MS01] discuss specific structures for RC4 states that give rise to a class of “predictive states” that lead to distinguishers and they describe attack algorithms that are more efficient for a special subset called “fortuitous” states. (It is fortuitous for the attacker)

Recently, a statistical weakness in the first byte of RC4 output was claimed in [Mir02]. It is estimated that 1700 first bytes are required to detect this bias and hence distinguish RC4 from random. The evaluators further evaluate this weakness in Section 3.3. The results presented in this section do not agree with the previous results in [Mir02] and do not show any bias in the first byte.

2.2 Weak Keys and Related Key Attacks

In 1995, A. Roos [Roo95] described a class of weak keys for which the initial byte of the keystream is highly correlated with the first few key bytes. The weak keys have the first two n-bit words sum to zero mod 2n. The reduction in search effort from this attack is $2^{5.1}$, but if linearly related keys are used, the reduction increases to 2^{18}. This weakness can be avoided by discarding the first few output keystream bytes.

A related key weakness of RC4 was discussed in [GW00]. It is observed that when RC4 is initialised with a single 2048 bit key, then there exist related keys (that differ only in two consecutive bytes), which produce output streams that are initially well correlated.

Let K and K' be the two RC4 keys of length 2048 bits, and let them differ only in two consecutive bytes. This change is called a “twiddle” by [GW00], and it is parameterised by two values: t and d. We have K'(t)=K(t)+d and K'(t+1)=K(t+1)-d, where K'=K for all other bytes. It follows from the initialisation process that j'=j+d at time t and also j' is likely to be the same as j at time t+1. The two changes of +d and –d in consecutive key bytes cancels each others effect, except that a different swap for K' than for K will occur during iteration t. This results in a different table at the end of the initialisation process. When the PRNG mode begins the outputs will be the mostly the same until the table differences affect the j pointer. At that iteration the two versions of the algorithm have “derailed” in the terminology of [GW00]. The time until this event is called the derailment time and the probability distribution of this time is discussed in detail in [GW00]. Good ‘twiddles’ cause less change to the initial table and so have longer derailment time.

This attack is not applicable to the usual mode of initialising RC4 with a 128 bit key that is repeated eight times to provide the required 2048-bit value, as any single change to the key causes eight changes to the table, thus greatly reducing the expected derailment time. Another way to avoid the correlated output segment is to suppress the first 256 output bytes.

A property called the “Invariance Weakness” has been investigated in [FMS01] and more fully explained in a recent masters thesis by Mantin [Man01]. The Invariance Weakness, (defined below), identifies a large class of weak keys in which a small part of the secret key determines a large number of bits in the RC4 table at the end of the initialisation process. These patterns are translated into the initial part of the output keystream. Thus RC4 has the undesirable property that for these weak keys its initial outputs are greatly affected by a small number of key bits.

We now present the details of the Invariance Weakness. Let a permutation *S[]* of *{0,…,N-1}* have the property that *S[t]= t (mod b)* then the permutation *S[]* is said to *b-conserve t*. The permutation *S[]* is said to be *b-conserving* if all *N* elements are *b-conserved*. The permutation is defined as *almost b-conserving* if the number of *b-conserved* elements is not less than *N-2*.

Let a key, *K*, have *w* words of *n*-bits each, where w is a multiple of some integer *b*. Then K is called a *b-exact* key for any index r that satisfies

$$K[r \bmod w]=1-r \pmod{b}.$$

In the case K[0]=1 and MSB(K[1])=1 then K is called a special b-exact key.

In [FMS01] the properties of a slightly modified version of the initialisation process of RC4 is considered. The only change made is the sequence of operations. It is shown that moving the increment of the basic modular counter index *i* to the beginning of the loop (rather than the end as in standard RC4), causes the invariance weakness to occur with greater effect than for standard RC4. Theorem 1 of [FMS01] states that, (for the modified version) all b-exact keys produce initial tables that are *b-conserving*. A modified proof of how the weakness occurs about half the time in real RC4 appears in [Man01]. The fraction of determined permutation bits is proportional to the fraction of fixed key bits. It states in [FMS01] that for modified RC4 with n=8 and w=6, it is so that 6 of the 48 key bits completely determine 252 of the 1684 permutation bits.

When some key bits are known, a related key attack is possible using the invariance weakness. Section 8.1 of [FMS01] discussed this attack for RC4 with n=8 and a 32 byte key. They state the attack has complexity of only 2^{40}, compared with exhaustive search complexity 2^{256}! Note that a special pattern of known key bits is required to initiate this attack, which supposes also that the attacker has adaptive “chosen key difference” access to the encryption device. In most applications this is not practical so that the invariance weakness is mainly of theoretical interest.

2.3 Backtracking Attacks

Several algorithms are given in [K+98] for the initial state reconstruction of RC4 from short sequences of output. The attacks are independent of the initialisation process as they recover the initial table itself.

The importance of the swapping operation in RC4 is examined by a detailed series of experimental attacks on modified versions of RC4 where the swapping is done less frequently, for example once every x iterations. The results of the heuristic algorithms used show that the security of the RC4 variants increased as the swapping was done more often. The attacks involve guessing some values in the table and using the known output sequence to check for contradictions. The number of initial correct guesses (required to allow the table to be recovered successfully with 50% probability) increases from 40 to 240 as the frequency of swapping increases from 1/128 to 1/2.

To attack the real RC4, the guesses are made not at the start but as needed. The actual heuristics used are not revealed in [K+98], however they do state that their best attack consistently has complexity less than searching through the square root of the initial state space. They have also identified that output sequences that have some bytes the same yield more quickly to this attack. Another attack is presented using probability distributions, but it works only if a certain number of table entries are already known. For n=8 about 160 entries are needed. The time complexity is 2^{6n} steps each consisting of computing the product of appropriate probabilities.

These attacks are infeasible for n>=5, so the usual instance of n=8 is secure from this method.

Iterative Probabilistic Cryptanalysis of RC4 was considered in [Goli00]. This method uses a short segment of keystream and a probabilistic model that improves upon the one from [K+98]. Both forward and backward recursive expressions for state value probabilities are developed under two versions of the independence assumption. The basic iterative algorithm is composed of a number of rounds, where each round uses a forward recursion and a backward recursion. These two recursions take as input the others previous output. The initial probability distribution for each of the state values is uniform. The recursions calculate state value probabilities given the known output sequence.

The paper [Goli00] gives results of computer experiments on these iterative algorithms for RC4 with n=3 and n=4. It turns out that a number of table entries have to be guessed correctly before this algorithm can be successful. It is interesting to note that the algorithm using the simplified independence assumption is more efficient than those using the more theoretically accurate independence assumption. The total number of computational steps remains the same, but the complexity of most is reduced when using the simplified assumption. The success of the algorithm does not seem to be affected by the inaccuracy in the independence assumption! Although more effective than [K+98], (they require fewer table entries to be guessed beforehand) the attacks of [Goli00] are still expected to remain infeasible for RC4 used with word size n=8.

2.4 Weaknesses in the Key Initialisation

Several weaknesses in the key initialisation process of RC4 were described in [FMS01]. They identify a large class of weak keys, that lack diffusion in the key to state/output mappings, This means that some state and initial output bits depend on a reduced subset of the secret key bits. These weak keys lead to distinguishers for RC4 and allow related key attacks with practical complexity. Also, the “public IV” mode of operation used in the WEP (part of the 802.11 standard) is shown to be insecure for RC4. In [FMS01] they show a new passive ciphertext only attack that recovers an arbitrarily long key in time that grows only linearly with its size.

When the RC4 key is a secret part concatenated with a public IV part, and the same key is used with numerous IVs, then a related key attack can use knowledge of the first output word from these different streams to reconstruct the secret key! The success probability, effort required and number of IV streams required depends on the order of the concatenation, the size of the IV and sometimes on the value of the key. Many details of these attacks are examined in [FMS01]. Although these attacks are applicable to some deployed encryptions systems, it is not effective against SSL as hash functions are used to securely combine the key and IV before encryption.

[FMS01] makes some other security relevant observations.

- They point out that the LSB biases combine in a natural way with the ASCII bias in the LSB of English texts. This results in ciphertext only distinguisher using only the knowledge that the plaintext is in English.
- The security measure introduced in [BSW00] called sampling resistance is low for RC4 due to the many known biases. This results in improved efficiency for time/memory/data trade-off attacks.

2.4.1 Practical Attacks on WEP

The attacks of [FMS01] are applicable to WEP (the link layer security protocol in the wireless 802.11 standard) since the first byte of each plaintext payload is a known constant, so the first byte of keystream is always known. WEP uses a 3 byte IV and uses the concatenation of the IV and Key, expressed as IV|Key, as the RC4 key. Then it transmits the IV and the ciphertext together. By examining enough packets the secret key can be recovered. With 60 different IV streams, the attacker can re-derive the key bytes in order one at a time. The attack recovers each byte separately with the same effort, so the increase in security is linear for increasing the key length.

Consideration of practical issues led to the estimate in [FMS01] of a few million packets being required to break WEP by this method. Soon afterwards real network traffic was used in experiments to recover 128-bit WEP keys in [SIR01]. They report that capturing the WEP encrypted packets off the wireless network proved to be the most time consuming part of the attack. They also report that hand decrypting packets after the initial attack failed showed that an additional 802.2 encapsulation header is added for both ARPO and IP traffic (this is due to RFC1042). This made the attack even easier as all IP and ARP packets would now have the same first plaintext byte.

They find in [SIR01] that 5 or 6 million intercepted packets was enough data to recover the entire 128-bit WEP key. It took a few hours to collect this amount of data on their partially loaded network. Improvements to the algorithm from [FMS01] lead to a reduction in the number of required texts down to around 1,000,000.

As software tools for these attacks have now appeared on the internet under names such as Airsnort and WEPcracker, is seems clear that 802.11 based wireless networks are practically insecure.

2.5 Summary

In this section we summarise previous attacks on RC4 and describe methods to prevent such attacks where applicable.

2.5.1 Correlations and Distinguishers

These attacks are used to distinguish the use of RC4 from a random keystream. There are two different types of distinguishers.

- Single key output distinguisher

  In a single key distinguisher the attacker knows a single output keystream. The length of the keystream required to distinguish RC4 from a random keystream is 2^{40} using methods from [Goli97] and $2^{33.6}$ using methods from [FM00].

  There is no simple method to avoid the single key output distinguisher weakness in RC4. However in most applications this is not considered to be a serious weakness. So far these distinguishers have only been used to identify the use of RC4 not as a method to recover the key. In most applications the interoperability requirements ensure the fact that RC4 is being used can be considered as already publicly known.

- Multiple key distinguisher

  In a multiple key distinguisher the attacker knows the byte in a fixed position from multiple keys. For first byte position [Mir02] claims 1700 first bytes are sufficient to distinguish RC4 random. The evaluators show that this claim is incorrect in Section 3.3. For second byte position [MS01] claim that only 200 different second bytes are required to distinguish RC4. The results in Section 3.3 agree with this claim.

  The multiple key distinguisher can be avoided by suppressing the first two bytes of the keystream. The evaluators strongly recommend that this method be used since the bias in the second byte provides leakage of information about the corresponding ciphertext byte.

2.5.2 Weak Keys and Related Key Attacks

The probability of having a weak key as described in [Roo95] is small, 2^{-16}. As well such a weakness can be avoided by discarding the first few bytes.

The related key attacks in Section 2.2 are in general not practical and only of theoretical interest. These attacks pose no threat to RC4 as used in practical applications.

2.5.3 Backtracking Attacks

The backtracking attacks reviewed in Section 2.3 are only practical for RC4 when the word size n is small. For RC4 with n = 8 which is used in most applications such attacks are not feasible.

2.5.4 Weakness in the Key Initialisation

The practical attacks on the use of RC4 in WEP, reviewed in Section 2.4, are caused by a bad rekeying process. Methods to prevent such attacks are presented in Section 4.2.

3 Keystream Properties

For keystream sequences to be used in stream ciphers that provide cryptographic security, the keystream must possess certain basic properties. These include a large period, large linear complexity and white-noise statistics.

Experimental results, which are included below for linear complexity and statistical analysis, were obtained by the evaluators using the CRYPT-X package. This is a statistical package, which was designed previously by the evaluators for analysing encryption algorithms. The relevant pages from the CRYPT-X manual have been included as Appendix C.

3.1 Period

The period of RC4 was analysed in [M98]. We provide a summary of this research below highlighting any significant results.

It was shown that for all keys the period T of RC4 is (2n)(z) for some integer z where n denotes the word length. To gain a better understanding of the expected period all possible keys for n = 2 and 3 were tried. It was shown that for these lengths that RC4 does resemble a random permutation, but a slightly larger number than expected of elements are in shorter cycles than expected. For n = 4, experimental attempts at determining the cycle length were unsuccessful, indicating that short periods are very rare.

In summary the research in [M98] indicates that, with high probability, the period of RC4 should be sufficiently large especially for the recommended word length of n = 8.

3.2 Linear Complexity

The linear complexity checks for the minimum amount of knowledge required to reconstruct the whole stream using a linear feed back shift register. It is difficult to determine the linear complexity of a sequence from RC4. In order to obtain empirical evidence for the linear complexity and linear complexity profile the tests from the CRYPT-X package were applied.

Linear complexity tests from CRYPT-X package were applied to 100 RC4 keystreams of length 10^5 bits, and the results showed linear complexity values close to that expected for random data (i.e. half the bit-stream length). The linear complexity test was also applied to five RC4 keystreams of length 819,200 bits (the number limited by time constraints of the test) and the results support those obtained for the shorter keystreams. For more detailed results see Appendix C. The results for the linear complexity profile indicate that, as the bit-stream increases in length, the changes in linear complexity maintain the expected value of half the stream length. These results support the randomness of the keystream output from RC4, based on linear complexity, such that the whole bit-stream is required to re-construct the stream itself, thus giving an attacker no advantage in being able to create the bit-stream with a smaller number of output bits.

3.3 Statistical Analysis

Statistical analysis was conducted by the evaluators on RC4 in three areas.

3.3.1 Gaps Test

As was mentioned in Section 2.1 [Jen94] noted that for n = 2 to 6 more gaps of length zero than were expected between words were found. The evaluators examined keystreams of RC4 for word size of n = 4 to verify [Jen94] results and 8 to see if similar behaviour exists.

The number of gaps of length zero, for each different word pattern, in words of size n = 8 were investigated on 100 different RC4 keystreams of 1 Mb (2^{20} words), and for n=4 on 100 different RC4 keystreams of 500,000 bytes (10^6 words). For n = 4 there are $2^4 = 16$ possible patterns, and for n = 8 there are $2^8 = 256$ possible patterns. The expected number of gaps of length zero for each pattern is $10^6/2^4 - 1 = 62{,}499$ (for n = 4), and $2^{20}/2^8 - 1 = 4095$ (for n = 8).

A goodness-of-fit test to a uniform distribution, with expected count as given above, was applied to the 100 results. Refer to Appendix A for a line graph illustrating the count of the number of gaps of length zero for each RC4 keystream.

The results for n = 4 show a bias towards a higher count than expected, confirming results from [Jen94]. The results for n = 8 show no apparent deviation from randomness. These results further support the use of word size of n = 8.

3.3.2 Byte Bias

As was mentioned in Section 2.1, [MS01] demonstrated a major statistical weakness in the second output byte of a RC4 keystream since the value zero occurs with twice the expected probability for a random sequence. This allows leakage of information about the second byte of any cryptogram formed with RC4 and allows the use of RC4 algorithm to be easily identified. Section 2.1 also mentioned that in a recent paper, [Mir02], it is claimed that a statistical bias also occurs in the first byte of RC4 where it is estimated that 1700 first bytes are required to detect this bias and distinguish RC4 from random.

In order to examine further the possible bias in the output bytes of RC4 a frequency test was applied to the first ten byte positions in the output stream of 100,000 different RC4 keystreams. Table 1 summarises the statistic (chi-square with 255 degrees of freedom) and p-value for each byte position.

Uniformity Test on Byte Position		
Position	**Statistic**	***p-value***
1	217	0.9589
2	644	0.0000
3	216	0.9628
4	304	0.0179
5	288	0.0780
6	300	0.0283
7	287	0.0834
8	274	0.1997
9	235	0.8075
10	251	0.5615

Table 1

These results support a non-uniformity of the byte patterns in byte position 2. It should be noted that the figure of 0.0000 for the p-value of byte position 2 indicates the extreme low probability for this distribution that the output was from a random file.

Since there are 100,000 keystreams and 256 possible byte patterns, then the expected number of each byte pattern is 1000,000/256 = 390.625 ($\approx$ 391). Refer to Appendix B for a line graph illustrating the count of each byte pattern (expressed as its equivalent decimal value 0 - 255) around the expected count. For all but Byte 2 the counts appear to fluctuate close to a horizontal line through 391. The count for the pattern equivalent to the decimal number "0" is just over double the expected count.

These results do not support the results in [Mir02] on the claim of a statistical bias in the first byte of RC4. All byte positions except position 2 appear random.

3.3.3 CRYPT-X Statistical Tests

The statistical analysis applied to RC4 are the tests explained in the CRYPT-X package (see Appendix C), namely the frequency, binary derivative, change point, subblock, runs distribution, sequence complexity and linear complexity tests. The tests are based on the hypothesis that the measure obtained from the output stream supports randomness. The p-values obtained from the tests represent the probability that such sample result (or a less random one) would be obtained if the algorithm produces a random stream. Very small p-values would support non-randomness.

The first five tests were applied to one hundred different RC4 keystreams of length 1 Megabyte (2^{23} bits) each, and the two complexity tests were applied to the first 10^5 bits of these keystreams (due to the amount of time required for these tests).

The subblock test was applied to the RC4 keystreams by dividing the bit-stream into non-overlapping subblocks of length 4, 8, 16 and 30 bits. The maximum subblock length of 30 was determined by the length of the keystream and the limitations of the test applied.

The frequency test was also applied to bit positions 1 to 64 in non-overlapping subblocks of 64 bits on the one hundred RC4 keystreams. The number of bits in each test was $2^{23}/2^6$ = 131,072 bits.

Results of Statistical Analysis

The results of the tests applied give 31 of the resulting p-values from the 2,015 tests falling below 0.01. This is sufficiently close to the expected value of 0.01, and satisfies the requirements for randomness. There are no strong indications of statistical weaknesses in the RC4 keystreams resulting from the CRYPT-X tests applied. For more details on these results see Appendix C.

The sequence complexity test provides an effective method of detecting periodicity or periodic patterns in the bit-stream. In the bit-streams tested all sequence complexity values exceeded both the threshold value for randomness, and the average sequence complexity for bit-streams of length 10^5 bits. These results supported that the period of the keystreams exceeded the length of the stream tested, and that there was no detection of patterns in the RC4 keystreams.

4 Cryptanalytic Techniques

RC4 is a different type of stream cipher than the standard linear feedback shift register (LFSR) based stream cipher. RC4 is a table based stream cipher with a very large internal memory. The tables are being updated dynamically during each encryption.

The standard divide and conquer attacks on stream ciphers require a known division. The dynamic updating of Tables in RC4 prevent such attacks. If a less dynamic updating occurs then attacks are possible against RC4 as discussed in Section 2.3.

The fast correlation attacks which have been extensively applied to LFSR based stream ciphers generally require parity checks based on the feedback function from the LFSR. There is no parallel concept for table based stream ciphers such as RC4, especially as in the case of word size n = 8 which defines a very large internal memory state.

The analysis of RC4 required the design of novel methods for attacking the algorithm. Section 2 reviewed various methods proposed over the last eight years. In Section 2.5 the significance of these possible attacks is summarised.

5 Application of RC4

There are several issues related to applying RC4 for encryption on actual communication systems that are discussed in this section.

- Use in SSL

  RC4 is one of the recommended encryption algorithms for SSL protocol.

- Rekeying

  For many applications of RC4 it is important to have secure and efficient rekeying mechanisms.

- Efficient Implementation

  Many applications of RC4 require efficient speed in either software or hardware.

5.1 SSL

SSL (Secure Sockets Layer) [Net96] is an application-layer protocol that provides security on top of reliable transport layers such as TCP. SSL abstracts from Berkeley-type sockets using a client-server model. The security services that it offers include confidentiality, privacy and optionally authentication of the server and client. SSL uses RC4, along with other symmetric ciphers, as a primitive for ensuring confidentiality.

SSL contains two sub-protocols: the handshake protocol, which the client and server use to negotiate security services, and the record protocol, which provides a secure channel between the two parties.

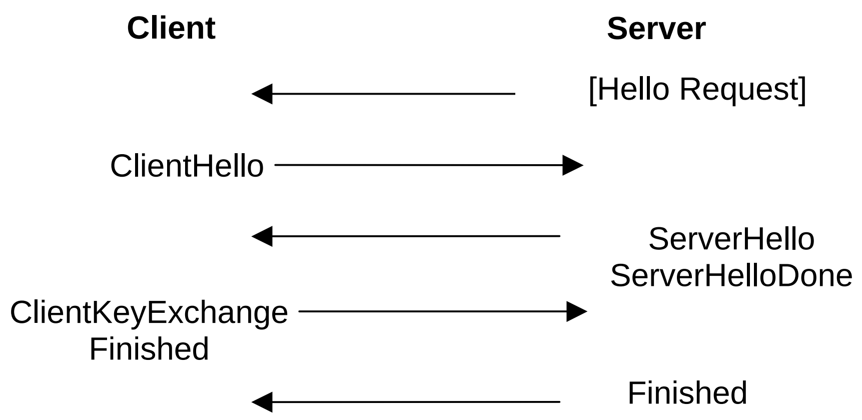


Figure 2 - Simple SSLv3 Handshake

Figure 2 shows the steps in a run of the SSLv3 Handshake protocol. The server starts the session with an optional HelloRequest message. The client and server negotiate parameters through the ClientHello and ServerHello messages, including the symmetric key algorithm used to provide confidentiality. The client generates a “pre-master secret” using a public-key algorithm and provides it (encrypted) to the server through the ClientKeyExchange message. The last stage of a successful run is the exchange of Finished messages, following which the client and server adopt the negotiated cipher suites to secure their channel.

Keying material for RC4 is derived from 32-byte random values passed in the clear in ClientHello and ServerHello messages, and from the client’s pre-master secret. To create its 32-byte random value, the client generates 28 bytes from a secure PRNG and concatenates it with a four-byte timestamp. The server does the same. The client generates the 48-byte pre-master secret by encrypting random bytes under the server’s RSA or Fortezza key, or by performing the Diffie-Hellman key exchange algorithm, using values supplied by the server.

Both parties generate the master secret from the pre-master secret using MD5 and SHA-1 HMACS – refer to Figure 3.

```
master_secret =
  MD5(pre_master_secret + SHA('A' + pre_master_secret +
         ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('BB' + pre_master_secret +
         ClientHello.random + ServerHello.random)) +
  MD5(pre_master_secret + SHA('CCC' + pre_master_secret +
         ClientHello.random + ServerHello.random));
```

Figure 3 - Generating the SSLv3 master secret [Net96]

The master secret generates the key material, again using MD5 and SHA-1, iterated until the sufficient material is available. Refer to Figure 4.

```
key_block =
   MD5(master_secret + SHA(`A' + master_secret +
         ServerHello.random + ClientHello.random)) +
   MD5(master_secret + SHA(`BB' + master_secret +
         ServerHello.random + ClientHello.random)) +
   MD5(master_secret + SHA(`CCC' + master_secret +
         ServerHello.random + ClientHello.random));
```

Figure 4 - Generating the SSLv3 key material [Net96]

SSL partitions this material for use by MAC keys, cipher keys and IVs. The cipher algorithm negotiated in the handshake protocol uses the cipher key material. SSL offers a number of block ciphers (DES, Triple-DES, IDEA, etc), but its only stream cipher selection is RC4. While RC4 is parameterised, SSL uses RC4 only with a word-size of 8 bits and a key size of 16 bytes (128 bits) or, less frequently, 5 bytes (40 bits) to cater for now-defunct U.S. export laws.

Since RC4 does not require a synchronization vector, the 128 bits of cipher key material is used directly as key K in the initialisation phase of Figure 1 (with $n = 8$, $k = 16$).

SSL does not provide a re-keying algorithm for RC4. Because SSL always operates on top of a reliable channel, entities are guaranteed that they will receive all packets in correct order. The state of RC4 following the processing of one packet is used for the processing of the next.

The server can explicitly attempt re-keying by requesting that the session is restarted for example in the case of loss of synchronisation. It does this by sending a HelloRequest message to the client. The client can choose to ignore this message, following which the server may abort the session. If the client agrees to re-start the session, both parties generate new keys using the same master-secret, but new client and server random material passed in the Hello messages. The overheads involved in this process are very high and equate to encrypting many blocks using RC4. This process is not practical if frequent rekeying is required.

The SSL procedure outlined above in Figures 2, 3 and 4 defines a method to create a key for input into the RC4 algorithm. This method is secure from the attacks on the keys described in Section 2.4.1 on the application of RC4 to WEP. The key information, which is the premaster secret and the known random information is mixed in a secure manner, using one-way hash functions MD5 and SHA-1. This procedure prevents leakage of any information about the premaster secret.

However the use of RC4 in SSL should be amended to include a recommendation to suppress the first few output bytes of the keystream, otherwise the statistical weakness in the second byte position described in Section 3.3.2 will still be present in the keystream.

5.2 Rekeying

Many communication systems require frequent resynchronisation. For a message encrypted on such a network using a stream cipher such as RC4 each resynchronisation requires rekeying the algorithm. In many cases for efficiency this rekeying is conducted with a base key and an initialisation vector which is sent in the clear. The base key and initialisation vector are used to form a session key to encrypt part of the message until resynchronisation is required.

Two session keys should be unrelated, otherwise attacks such as that on RC4 in WEP outlined in Section 2.4 may be possible. There are numerous methods that can be used. Methods for RC4 are described in [Riv01] where it is stated "RSA Security has discouraged such key derivation methods, recommending instead that users consider strengthening the key scheduling algorithm by pre-processing the base key and any counter or initialisation vector by passing them through a hash function such as MD5. Alternatively, weaknesses in the key scheduling algorithm can be prevented by discarding the first 256 output bytes of the pseudo-random generator before beginning encryption. Either or both of these techniques suffice to defeat the new attacks on WEP and WEP2".

The evaluators agree with RSA Security that the use of MD5 may prevent the attacks on WEP and WEP 2. However a more precise description for this process is required in order to determine its effect in preventing attacks. The evaluators strongly disagree with the alternative method described by RSA to only discard the first 256 output bytes. This will still leave session keys exposed to the attacks from [FMS01] and [SIR01] on WEP and WEP2.

Both methods recommended by RSA should be used. The use of MD5 to form session keys will prevent related key attacks. MD5 is a specialized hash-function that runs approximately 60% faster than RC4 on a standard Intel platform [Dai00]. The use of MD5 should not present too great a decrease in efficiency for the increase in security. Discarding some output bytes is needed to prevent the leakage of information in the first few output bytes. At a minimum the evaluators strongly recommend suppressing the first two output bytes, no matter how the key and IV are combined before initialisation.

The evaluators recommend that the re-keying algorithm be included in any stream cipher standardisation process, otherwise ad-hoc solutions can introduce new insecurities. Note that the New European Schemes for Signatures, Integrity and Encryption (NESSIE) has included a call for their stream cipher submissions to have the re-keying process clearly defined and this is included as part of the overall algorithm evaluation. The evaluators recommend all aspects of a cipher, including the intended context, should be included in the standardisation process for RC4.

5.3 Implementation

RC4 is an unusual symmetric cipher in that it is inherently serial and does not lend itself to parallel application in hardware. For this reason, RC4 does not run faster in dedicated hardware than in software.

There are a number of dedicated hardware implementations of ARC4 (Alleged-RC4) that produces the same keystream as RC4 when given the same inputs. Motorola claims that its MPC190 coprocessor produces a bulk encryption throughput using ARC4, of 630 Mbits/second (compared to DES at 1139 Mbits/second) [Mot02].

In software, RC4 appears to be one of the fastest mainstream ciphers on a single processor. Appendix D contains the implementation of RC4 in the C language on a linux 2.4 kernel. The test machine was a 1 Gigahertz Intel Pentium 3 with 32Kb L1 cache. For a gcc implementation requiring 1 kilobyte of memory, the peak throughput was 963 Mbits/second. Implementation in assembly language is expected to yield slightly higher results, in the order of 25%. A smaller implementation, requiring around 260 bytes of memory runs at approximately 330 Mbits/second.

6 References

[Ano94]
An0nYm0Us UsEr. "*RC4?*" posted to sci.crypt, 1994.

[BSW00]
A. Biryikov, A. Shamir and D. Wagner "*Real time cryptanalysis of A5/1 on a PC*", In proceedings of FSE2000, LNCS vol 1978, pp. 1-18, 2001.

[CHJ01]
D. Coppersmith, S. Halevi and C. Jutla, "Cryptanalysis of stream ciphers with Linear Masking" available at the iacr eprint archive, 2002.

[Dai00]
Dai, W. *Crypto++ 4.0 Benchmarks*, available at http://www.eskimo.com/~weidai/benchmarks.html, 2000.

[FM00]
S.R. Fluhrer and D.A. McGrew, "*Statistical Analysis of the Alleged RC4 Keystream Generator*", Proceedings of Fast Software Encryption 2000 FSE'00, LNCS vol. 1978, pp. 19-30, Springer-Verlag, 2001.

[FMS01]
S. Fluhrer, I. Mantin, A. Shamir, ``*Weaknesses in the Key Scheduling Algorithm of RC4",* Proceedings of Selected Areas in Cryptography 2001, SAC'01, LNCS vol. 2259, pp. 1-24, Springer-Verlag, 2001.

[Goli97]
J. Dj. Golic, ``*Linear Statistical Weakness of alleged RC4 keystream generator",* Advances in Cryptology - Eurocrypt'97, LNCS vol. 1233, pp. 226-238, Springer-Verlag, 1997.

[Goli00]
J. Dj. Golic, ``*Iterative Probabilistic Cryptanalysis of RC4 Keystream Generator",* Proceedings of ACISP'2000, LNCS vol. 1841, pp. 220-233, Springer-Verlag, 2000.

[GW00]
A. I. Grosul and D. S. Wallach, ``*A Related Key Cryptanalysis of RC4",* Manuscript from Dep. Computer Science, Rice University, 6 June 2000.

[Jen94]
R. J. Jenkins, "*Re: RC4?*", posting to sci.crypt, Sept 1994.

[K+98]
L. Knudsen, W. Meier, B. Preneel, V. Rijmen and S. Verdoolaege, ``*Analysis methods for (alleged) RC4",* Advances in Cryptology - AsiaCrypt'98, LNCS vol. 1514, pp. 327-341, Springer-Verlag, 1998.

[Man01]
I. Mantin, "Analysis of the Stream Cipher RC4". Masters Thesis, Weizmann Institute of Science, Israel. Nov 27 2001.

[MS01]
I. Mantin and A. Shamir, ``*A Practical Attack on Broadcast RC4",* Proceedings of Fast Software Encryption, FSE 2001, to appear in LNCS, Springer-Verlag, 2002.

[Mir02]
I. Mironov, "*(Not so) Random Shuffles of RC4*", accepted for Crypto'02. Also available on the iacr e-print archive.

[Mot02]
Motorola Corporation, *MPC190 Security Co-Processor's User's Manual*, pages 1-13, 2002. Available at http://e-www.motorola.com/brdata/PDFDB/docs/MPC190UM.pdf.

[MT98]
S. Mister and S.E. Tavares, ``*Cryptanalysis of RC4-like Ciphers",* Proceedings of SAC'98, LNCS vol. 1556, pp. 131-143, Springer-Verlag, 1999.

[Net96]
Netscape Corporation, *"The SSL Protocol version 3.0"*, available at http://wp.netscape.com/eng/ssl3/draft302.txt, 1996.

[Riv01]
R. Rivest, "RSA Security Response to weaknesses in key scheduling algorithm of RC4", Technical note available from RSA Security Inc. site. http://www.rsasecurity.com/rsalabs/technotes/wep.html, 2001.

[Roo95]
A. Roos. "*Class of weak keys in the RC4 stream cipher*". Two posts in sci.crypt, 1995.

[SIR01]
A. Stubblefield, J. Ioannidis and A. D. Rubin, ``*Using the Fluhrer, Mantin and Shamir Attack to break WEP",* (TD-4ZCPZZ) AT&T Labs Technical Report, 2001.

[Wag95]
D. Wagner. “*My RC4 weak keys*”. Posting in sci.crypt, 1995.

Appendix A

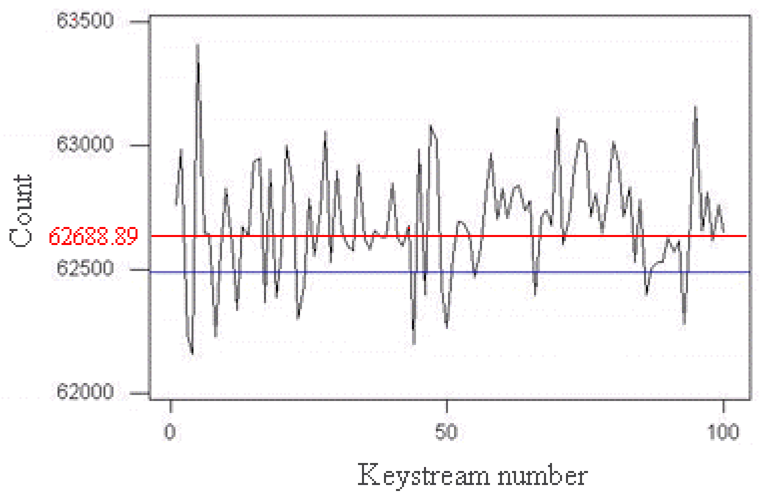
Gaps test

Gaps of length zero for RC4 keystreams of word length = 4:

100 RC4 keystreams of length 500,000 bytes (1,000,000 4-bit words) were generated using 100 different keys.

For random output, the expected number of gaps of length zero is 1,000,000/16 - 1 = 62,499. This is shown as a horizontal blue line in the graph below). The majority of the 100 keystreams have a count above this expected value, thus supporting a bias towards a higher count of gaps of length zero (i.e. pairs of 4-bit words in the keystream).

The average number of gaps obtained from the 100 keystreams was 62,688.89 (shown as a horizontal red line in the graph below. A test of goodness-of-fit to a uniform distribution gives a test statistic of 81.5. This is compared to a chi-square distribution with 99 degrees of freedom and gives a p-value of 0.8986. This large p-value supports uniformity of the number of gaps - about a mean (62,688.89) that is higher than that expected for randomness.

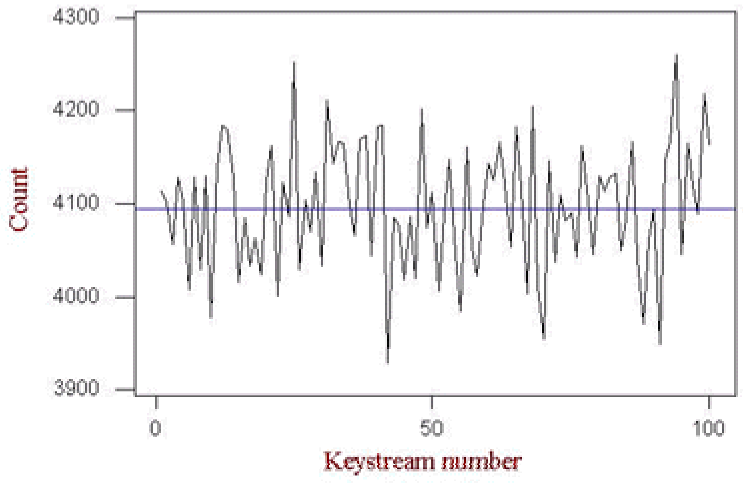


Gaps of length zero for RC4 keystreams of word length = 8 bits.

100 RC4 keystreams of length 2^{20} bytes (1,048,576 8-bit words) were generated using 100 different keys.

For random output, the expected number of gaps of length zero is $2^{20}/2^{8} - 1 =$ 4,095. This is shown as a horizontal blue line in the graph below). The 100 keystream counts appear to vary randomly about this horizontal, indicating randomness for the number of gaps of length zero (i.e. pairs of 8-bit words in the keystream).

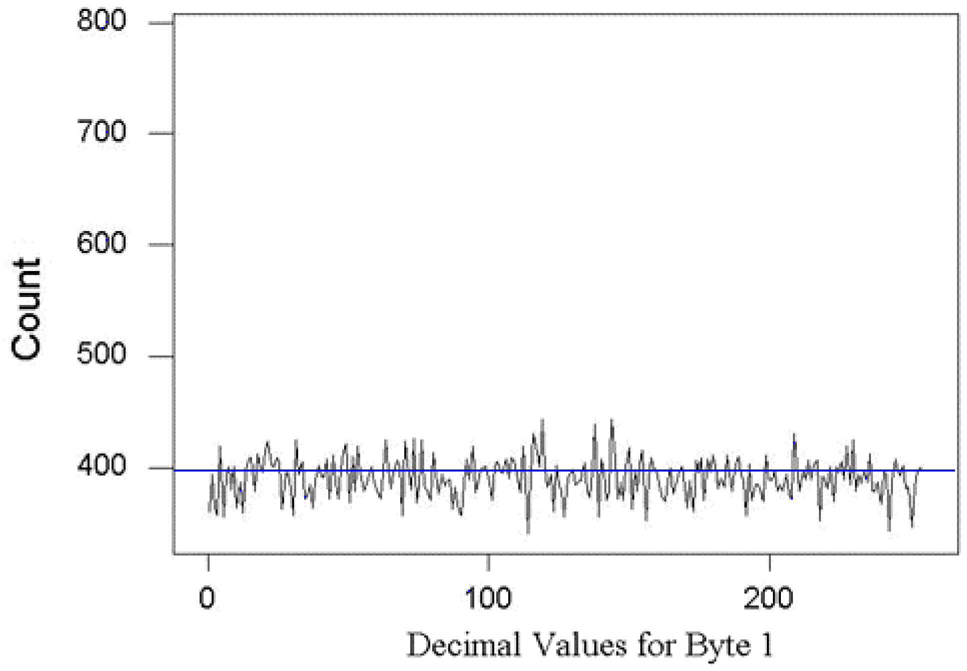
The average number of gaps obtained from the 100 keystreams was 4,097.66, which is close to the expected mean of 4,095. A test of goodness-of-fit to a uniform distribution gives a test statistic of 111.95. This is compared to a chi-square distribution with 99 degrees of freedom and gives a p-value of 0.1762. This p-value supports randomness in the number of gaps of length zero between words of length 8 bits (i.e. randomness in the occurrence of byte pairs.).



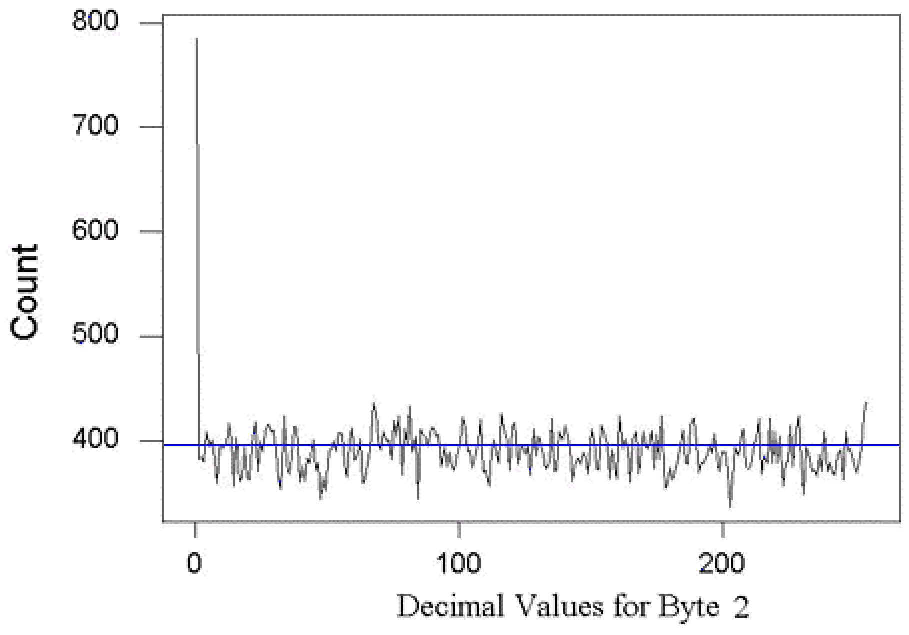
Appendix B

Byte Bias

Line graphs of the number of different byte pattern in bytes 1 to 10 over 100,000 files. Number of different byte patterns = 256.
Horizontal axis shows decimal representation of byte pattern from 0 to 255.
The expected number of each byte pattern = 390.625, and this is highlighted by a horizontal blue line.

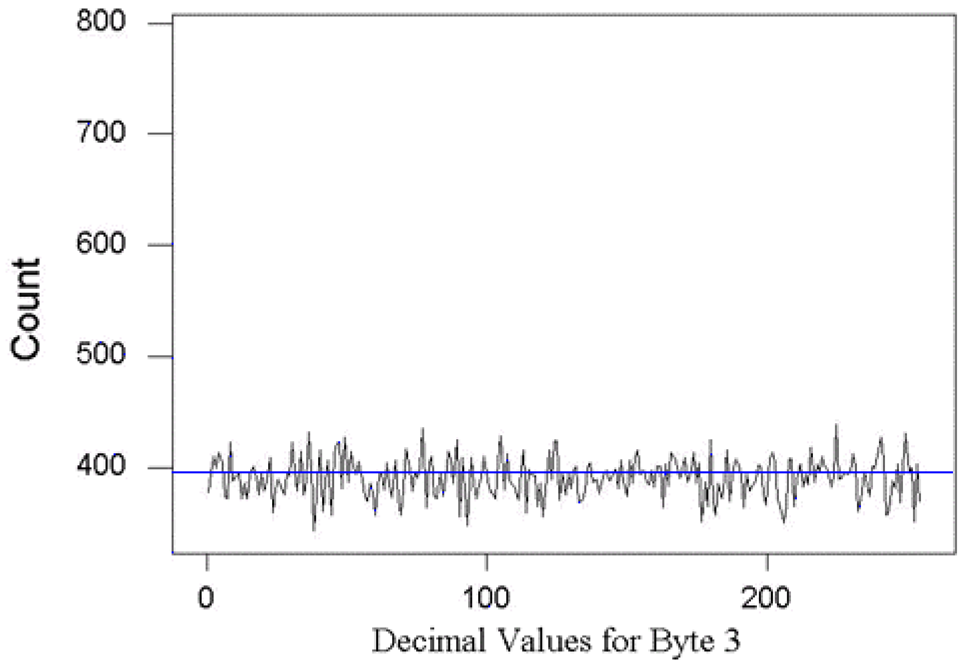


Byte 1 shows no strong deviation from the expected count.

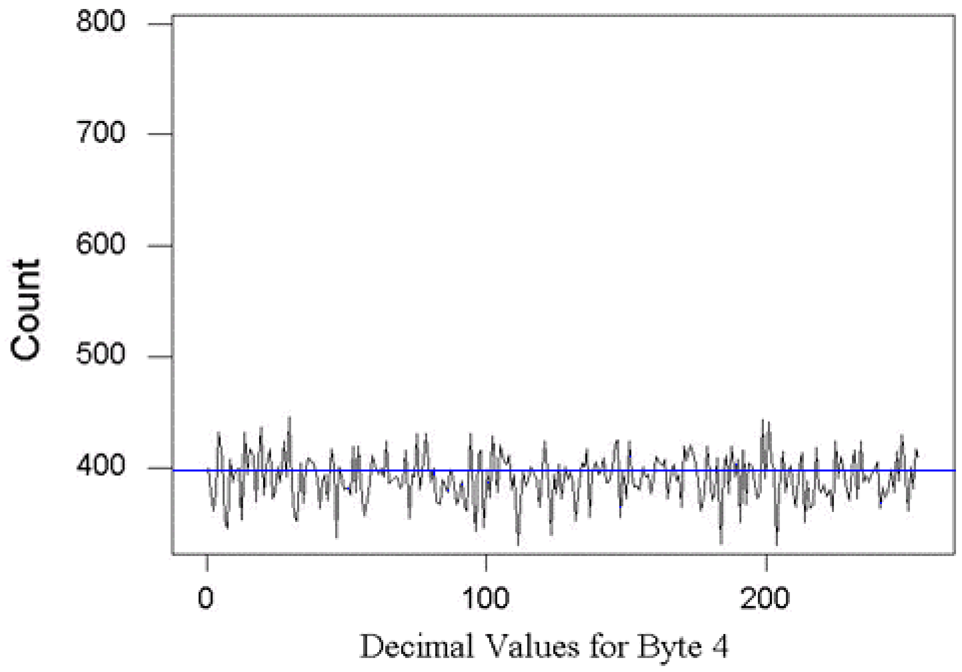


Byte 2 shows a large deviation from the expected count for pattern '0'.

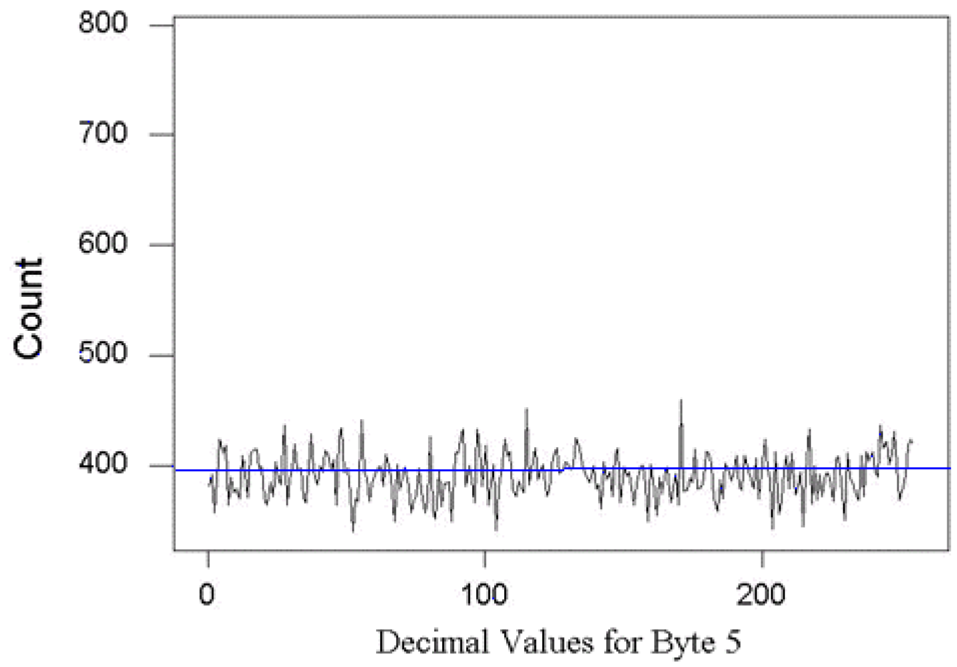
The number of occurrences of '0' is more than double that expected.



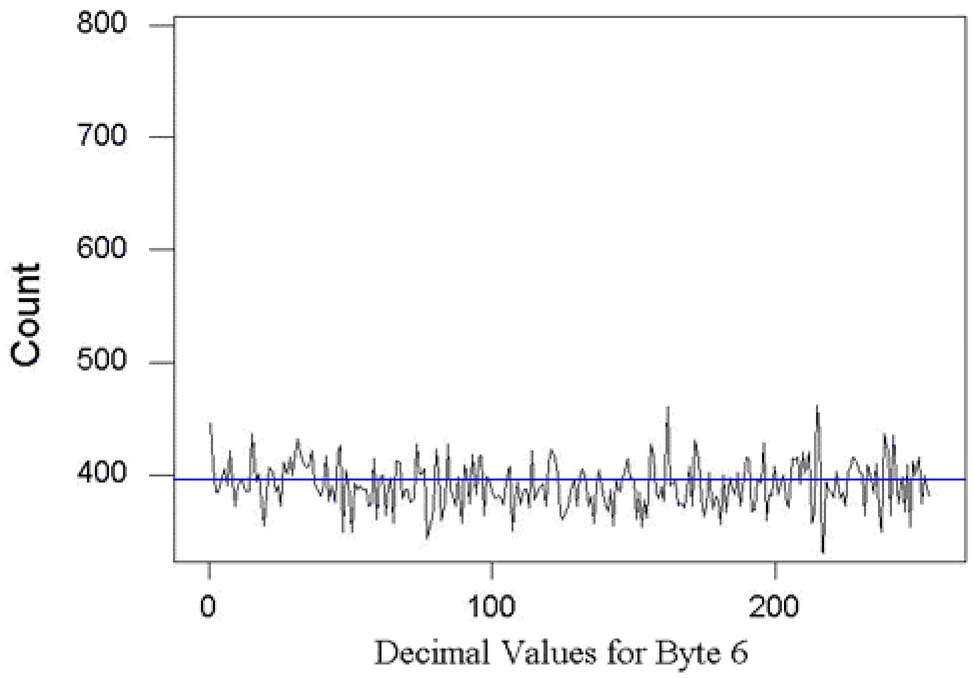
Byte 3 shows no strong deviation from the expected count.



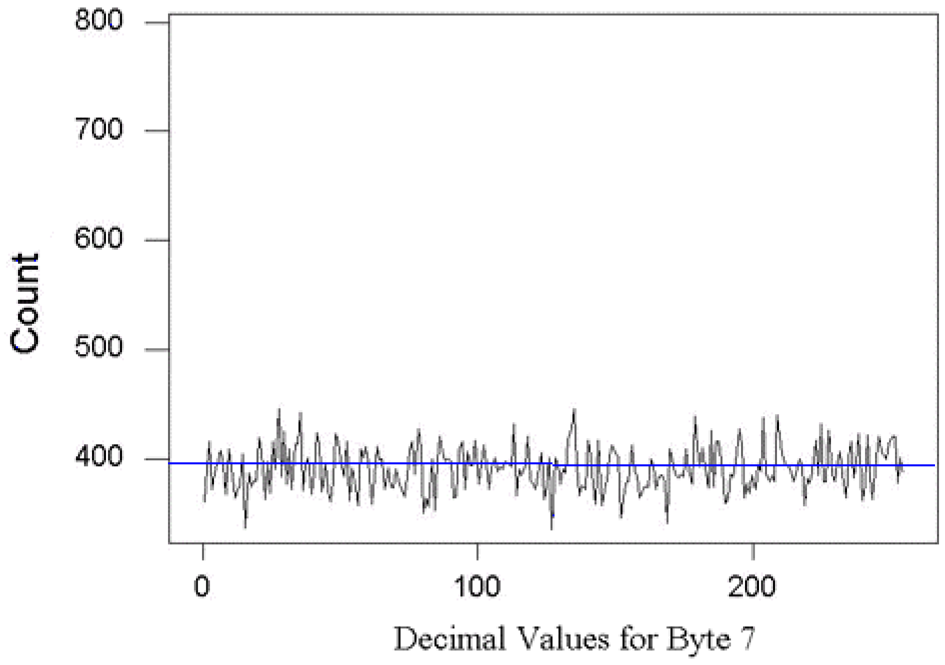
Byte 4 shows no strong deviation from the expected count.



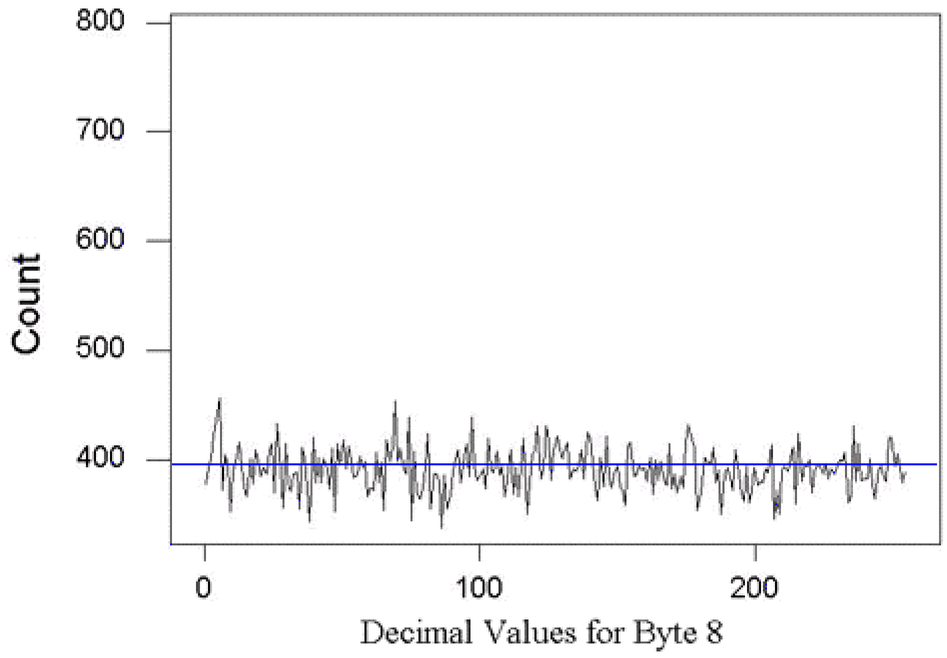
Byte 5 shows no strong deviation from the expected count.



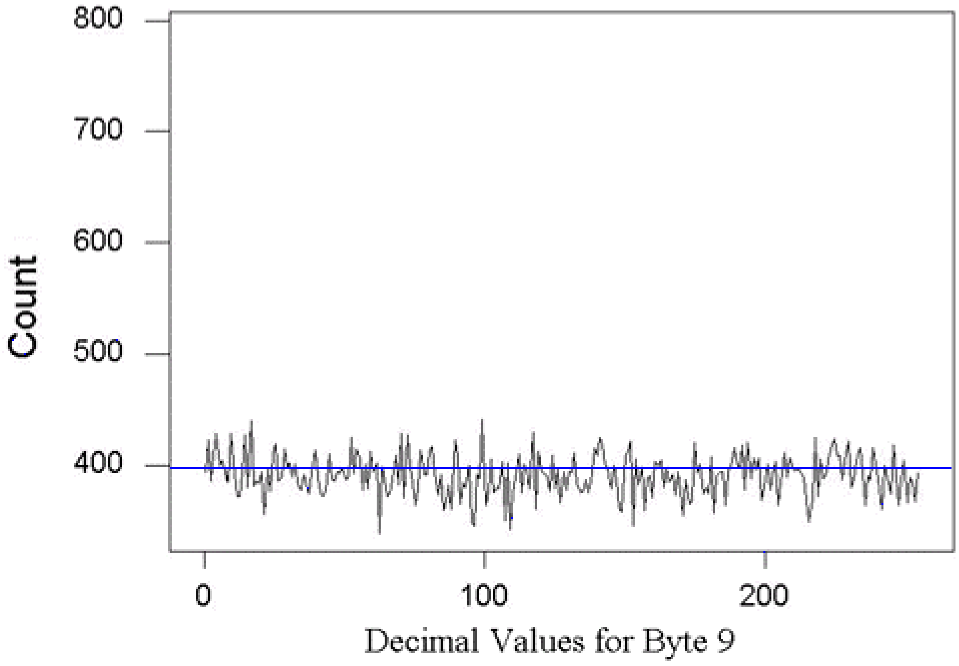
Byte 6 shows no strong deviation from the expected count.



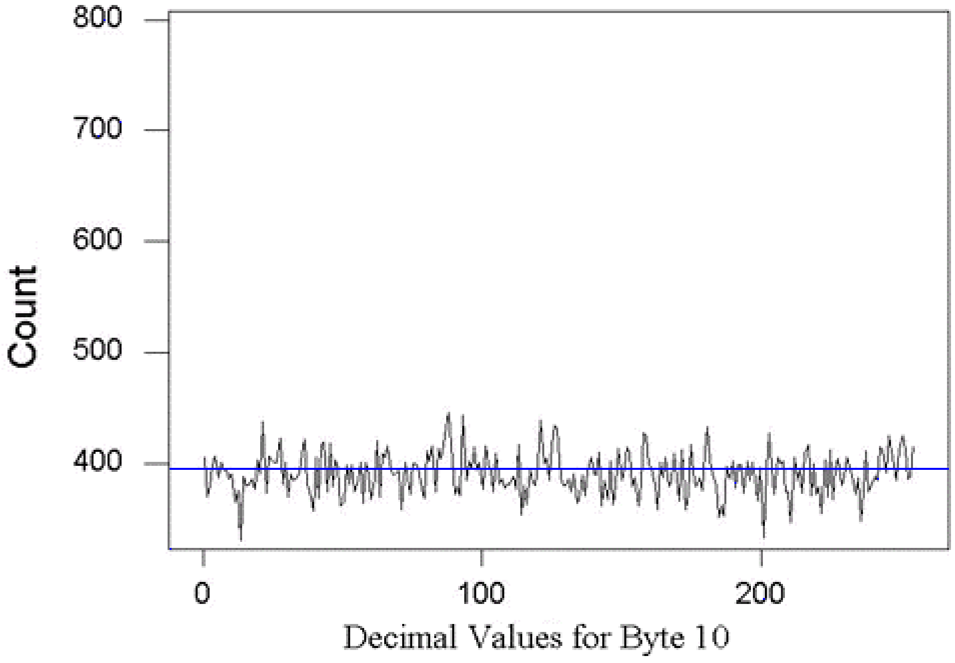
Byte 7 shows no strong deviation from the expected count.



Byte 8 shows no strong deviation from the expected count.



Byte 9 shows no strong deviation from the expected count.



Byte 10 shows no strong deviation from the expected count.

Appendix C

CRYPT-X Statistical Tests

This appendix gives a mathematical description of the statistical tests used from the CRYPT-X statistical package. In each case an example is given to illustrate a particular test. The first five tests examine the hypothesis that the bit stream was based on Bernoulli trials where the proportion of ones and zeros is $\frac{1}{2}$. The two complexity tests examine the knowledge that a small subsection of the bit stream can be used to produce the remainder of the stream. If this is possible the string would not be considered to be random, especially in relation to its use in a stream cipher.
The recommended size of a sample stream to test depends on the size of the average message which is being encrypted using the keystream. i.e. if an average cryptogram has size five million bits then one should use test samples of this length. It should be noted that not all of the tests can be applied to a string of this length due to computational limitations. For example, in the linear complexity test one would need to examine a smaller substring of the keystream. It is recommended that strings of length at least 100000 bits be used for testing.

C.1.1 Frequency Test

The *frequency test* checks that there is an equal proportion of ones and zeros in the bit stream. For randomness the proportion of ones and zeros in the bit stream should be approximately equal, since any substantial deviation from equality could result in a successful cryptanalytic attack on the cipher. For example, assume that a cryptanalyst attacking the stream cipher knows the type of plaintext being used, e.g. standard English text coded in 8-bit ASCII, and the keystream has $\frac{3}{4}$ of the bits zero. Under this assumption the cryptanalyst knows the frequency distribution of the plaintext in terms of single bits, digraphs and trigraphs. With this knowledge the cryptanalyst could recover a substantial amount of the plaintext, using ciphertext alone.
The number of ones in a random binary sequence follows a binomial distribution, with mean $\frac{n}{2}$ and variance $\frac{n}{4}$. This may be approximated using a normal distribution. The following notation is used:

$$n = \text{total number of bits;}$$

$$n_0 = \text{number of zeros;}$$

$$n_1 = \text{number of ones;}$$

$$\hat{p} = \frac{n_1}{n} = \text{proportion of ones in the sequence.}$$

The aim of the frequency test is to determine how the proportion of ones, $\hat{p}$, in the sample stream of length n bits, fits into the hypothesised distribution where the proportion of ones, $\pi = 0.5$ and the variance, $\sigma^2 = \frac{1}{4n}$. This is a two-tailed test [BHAT 77]. The standardised normal test statistic is:
$z = 2\sqrt{n}(\hat{p} - 0.5)$. The significance probability value, p, of the normal distribution is calculated for this statistic. This measures the probability of

obtaining a number of ones equal to or further from the mean of $\frac{n}{2}$ than this sample gives for the hypothesised (where $\pi = 0.5$ and $\sigma^2 = \frac{1}{4n}$).
A small significance probability indicates a significant result (i.e., the stream is considered to be non-random). For large values of *n* ($n > 100000$) a highly significant result (significance probability < 0.001) indicates a possible weakness in the cipher and it is recommended that no further tests be carried out on this sample as the imbalance of ones and zeros may effect their results.
It should be noted that passing the frequency test does not mean the stream is not patterned. The following highly patterned streams, where the number of ones and zeros are equal, will pass the frequency test:
11111111..........00000000.........
10101010101010.....................
Hence further testing is required to obtain knowledge of any patterns in the stream.
Example:
Test stream:
10100010000101110001011000111010101010101010000001
Calculations and results:

$$n = 50$$
$$\sigma^2 = \frac{1}{4\times 50}$$
$$n_1 = 21$$
$$\hat{p} = 0.42$$
$$z = \sqrt{50}(0.42 - 0.5) = -1.13137$$
$$p = 0.2579$$

Interpretation:

25.79 % of bit streams of length 50 will have a number of ones equal to or further from the mean of 25, for the hypothesised distribution, than this sample. This sample satisfies the frequency test.

C.1.2 Binary Derivative Test

The binary derivative is a new stream formed by the exclusive-or operation on successive bits in the stream. Successive binary derivative streams may be obtained from each new binary derivative, each one being of length one less than its predecessor [CARR 88].
The proportion of ones in the *i*-th binary derivative gives the proportion of overlapping (*i*+1)-tuples from the original stream in one of two known groupings of these (*i*+1)-tuples. This will be explained for $i = 1$ and $i = 2$.
When $i = 1$ (first binary derivative) we are looking at the overlapping two-tuples: 00, 01, 10, 11 (in the original stream).
The proportion of ones in the first binary derivative, $\hat{p}(1)$, gives the proportion of the total number of 01 and 10 patterns in the original stream.
$\hat{p}(1) >$ ½ means there is a larger proportion of the group of 01 and 10 two-tuples (in the original stream).
$\hat{p}(1) <$ ½ means there is a larger proportion of the group of 00 and 11 two-tuples (in the original stream).
A combination of the frequency test on the original stream and its first binary derivative is equivalent to testing that there is an equal number of these four

overlapping two-tuples in the original stream. This replaces the well-known Serial Test [DAWS 91].
When $i = 2$ (second binary derivative) we are looking at overlapping three-tuples: 000, 001, 010, 011, 100, 101, 110, 111 (in the original stream). The proportion of ones in the second binary derivative, $\hat{p}(2)$, gives the proportion of the total number of 001, 011, 100, 110 patterns in the original stream.
$\hat{p}(2)$ > ½ means there is a larger proportion of the group of 001, 100, 110, and 011 three-tuples.
$\hat{p}(2)$ < ½ means there is a larger proportion of the group of 000, 010, 101, and 111 three-tuples.
A combination of the frequency test on the original stream and a similar test on the first and second binary derivatives, tests that there is an equal number of the eight overlapping three-tuples in the original stream, for practically all cases. If a cipher gives a satisfactory result to these tests AND also the change point test, then it can be considered to generate equal numbers of the overlapping three-tuples.
Notation:

$$n_1(i) = \text{number of ones in the } i\text{-th derivative}$$

$$\hat{p}(i) = \frac{n_1(i)}{n-i} = \text{proportion of ones in the } i\text{-th derivative}$$

The frequency test is applied to each stream and the standardised normal variable is found for the proportion of ones in each of the first two binary derivatives: $z(i) = 2\sqrt{n-i}\,(\hat{p}(i) - 0.5)$, for $i = 1,2$.
The significance probability value, p_i, of the normal distribution is calculated for each statistic. A small significance probability indicates a significant result. For large *n* ($n > 100000$) a highly significant result (significance probability < 0.001) indicates a possible weakness in the cipher.
Example:
Test stream:
    10100010000101110001011000111010101010101010000001
Calculations and results:
D$_1$: 1110011000111001001110100100111111111111111000001
D$_2$: 001010100100101101001110110100000000000000100001

Frequency test on first binary derivative (D$_1$) :

$$\mathrm{n}_1(1) = 30$$

$$\tfrac{n-1}{2} = 24.5$$

$$\hat{p}(1) = \tfrac{n_1(1)}{n-1} = \tfrac{30}{50-1} = 0.61224$$

$$\mathrm{z}(1) = 2\sqrt{49}(0.61224 - 0.5) = 1.57143$$

$$p_1 = 0.1161$$

Interpretation:

11.61 % of bit streams of length 49 will have a number of ones equal to or further from the mean of 24.5, for the hypothesised distribution, than this sample. This sample satisfies the frequency test on the first binary derivative.

Since the frequency test is satisfied for the original stream and the first binary derivative then the cipher can be regarded as producing an equal number of overlapping two-tuples.

Frequency test on second binary derivative (D_2) :

$$\mathrm{n}_1(2) = 16$$

$$\tfrac{\mathrm{n}-2}{2} = 24$$

$$\hat{p}(2) = \tfrac{n_1(2)}{n-2} = \tfrac{16}{50-2} = 0.333$$

$$\mathrm{z}(2) = 2\sqrt{48}(0.333 - 0.5) = -2.3094$$

$$p_2 = 0.0209$$

Interpretation:

> *2.09 % of bit streams of length 48 will have a number of ones equal to or further from the mean of 24, for the hypothesised distribution, than this sample. This sample satisfies the frequency test on the second binary derivative.*

Even though the frequency tests on the original stream and the first and second binary derivatives were all satisfied, the cipher will still have to satisfy the change point test before regarding it as producing an equal number of overlapping three-tuples.

C.1.3 Change Point Test

At each bit position, t, in the stream the proportion of ones to that point is compared to the proportion of ones in the remaining stream.

The difference or *change* in these proportions is compared for all positions in the bit stream. The bit where the maximum change occurs is called the *change point*. The test applied determines whether this *change* is significant for a binomial distribution where the proportion of ones in the stream is expected to be 0.5.

This test is very useful for detecting patterned streams which have passed the frequency test on the stream and the first two binary derivatives. Even if $\pi = \tfrac{1}{2}$ and the stream has passed the frequency test it could be, for n = 10^6, that $\pi = \tfrac{1}{4}$ for the first 500000 bits and $\pi = \tfrac{3}{4}$ for the second 500000 bits. This is not considered to be a good pseudorandom sequence to be used as a keystream, and the change point test would detect such cases.

This test is also useful for checking that there is an equal number of overlapping three-tuples for streams which have passed the frequency test on the original stream and also on the first two binary derivatives.

The hypothesis to be tested is that there is no change in the proportion of ones throughout the whole stream. The statistic [PETT 79] used is

$\mathrm{U[t] = n \times S[t] - t \times S[n]}$ where

$$n = \text{total bits in stream}$$

$$S[n] = \text{total ones in stream}$$

$$S[t] = \text{number of ones to bit } t$$

The maximum absolute value of this statistic is found:

$$\text{Max} = \text{Maximum of ABS}(U[t]), \text{for } t = 1\ldots n$$

The significance probability, *p*, associated with this statistic is approximated by:

$$p = e^{-\frac{2Max^2}{nS[n](n-S[n])}}.$$

For small values of p the actual significance probability is smaller than that calculated. The smaller the value of p then the more significant the result. For large streams a highly significant result, p < 0.001, indicates a possible weakness in the algorithm.

Example:

Test stream:

s = 10100010000101110001011000111010101010101010000001

Calculations and results:

$n = 50$

$S[n] = 21$

$t = 43$

$S[t] = 20$

$\text{Max} = |50 \times 20 - 43 \times 21| = 97$

$p = e^{-\frac{2 \times 97^2}{50 \times 21(50-21)}} = 0.5390$

Interpretation:

The actual significance probability of the change in the proportion of ones is less than 53.9%. This result indicates there is no significant change in the proportion of ones in the bit stream. This sample satisfies the change point test.

C.1.4 Subblock Test

The stream is divided into S non-overlapping subblocks, each of length b. Any fractional subblocks remaining are ignored. For a stream of length *n*, the number of subblocks is the integral part of $\lfloor \frac{n}{b} \rfloor$, i.e. S = $\lfloor \frac{n}{b} \rfloor$.

For a subblock size of b ≤ 16 a test of uniformity is applied – i.e., there should be an equal number of each b bit pattern. The test compares the observed number of each b bit pattern with $S/2^b$.

The test statistic used is $\chi^2 = \frac{2^b}{S} \sum_{i=0}^{2^b-1} f_i^{\,2} - S$ [BEKE 82], where f_i is the frequency of subblock pattern whose equivalent decimal value is i. This statistic is compared with a chi-square distribution with degrees of freedom equal to $2^b - 1$. For values of b > 6 the normal distribution may be used to approximate the chi-square distribution. Limitations: The minimum length required for the stream to test for randomness using b-bit subblocks is $5b \times 2^b$ bits.

For a subblock size of b > 16 the repetition test [GUST 96], is applied. The repetition test measures the number of repeated patterns in a sample of S subblocks, each containing b bits. Given the binary stream is divided into S b-bit subblocks then, for a random stream, each of the $N = 2^b$ possible binary b-bit patterns is equally likely to occur. As the block length increases and $N \to \infty$, with a sample of size $S \to \infty$ where $\frac{S}{N} \to 0$, then the distribution of the number of subblock repetitions in the sample approaches a Poisson distribution with a mean of $\lambda = S - N(1 - e^{-\frac{S}{N}})$. When $S = 8\sqrt{N}$ the mean

converges to 32, for large values of b (say b > 16). The Poisson distribution is well approximated by the normal distribution for $\lambda = 32$.
The test requires a count of the number of subblock repetitions, r. (Note that if a particular pattern occurs three times, then this would add two to the number of repetitions).
The number of b-bit subblocks required for the test is $S = 8\sqrt{N}$, and gives $\lambda \approx 32$.
The procedure is to sort the subblocks and then determine the number of repetitions, *r*.
The test statistic is $z = \frac{r-\lambda}{\sqrt{\lambda}}$ (standard normal statistic for a Poisson distribution with a mean equal to λ), and is compared with the standard normal distribution. A two-tailed test applies since both too few or too many repetitions may indicate non-randomness of the stream.
The required stream length to apply the repetition test using b-bit subblocks is $b \times 2^{\frac{b}{2}+3}$ bits. This is considerably less than the length of stream required to apply the uniformity test for subblocks of the same size. Since the stream lengths required are very large, no sample stream will be shown. Instead, the following data will be used to illustrate a test calculation for the uniformity test:

$$b = 8 \text{ (hence the uniformity test is applied)}$$
$$n = 100000$$
$$S = \left\lfloor \frac{100000}{8} \right\rfloor = 12500$$
$$\text{Number of 8 bit patterns} = 2^8 = 256$$
$$\text{Assume } f_0 = 45,\ f_1 = 50, \ldots,\ f_{255} = 44, \text{ to give:}$$
$$\chi^2 = \frac{2^8}{12500}\sum_{i=0}^{255} f_i^2 - 12500 = 260 \text{ (say)}$$
$$\text{Degrees of freedom} = 255$$
$$z = \sqrt{2\times 260} - \sqrt{2\times 255 - 1} = 0.24248$$
$$p = 0.4042$$

Interpretation:

40.42% of all possible streams of length 100000 will have a distribution of 8-bit subblocks less uniform than this sample shows. This sample satisfies the subblock test for subblocks of length 8.

The following data is used to illustrate a test calculation for the repetition test:

$b = 18$ (hence the repetition test is applied)

$n = 100000$

$N = 2^{18} = 262144$

$S = 2^{\frac{18}{2}+3} = 4096$

Stream length tested $= 4096 \times 18 = 36864$ bits

$r = 38$ (say)

$$z = \frac{38 - 32}{\sqrt{32}} = 1.06066$$

$p = 0.1444$

Interpretation:

14.44% of all possible streams of length 36864 will have a 18-bit subblock repetition count further from the mean (32) than this sample shows. This sample satisfies the subblock test for subblocks of length 18.

C.1.5 Runs Test

The runs distribution test compares the distribution of the number of runs of ones (blocks) and zeros (gaps) with that expected under randomness. For a random binary stream where $\Pr(1) = \Pr(0) = \frac{1}{2}$ there should be an equal number of number of blocks and gaps of the same length. Based on Golomb's postulates, the expected number of runs of length i for a random binary stream should be $\frac{1}{2^i}$ of the number of runs, and for each length there should be an equal number of runs of ones and zeros, i.e. $E(r_{0i}) = E(r_{1i}) = \frac{Runs}{2^{i+1}}$, where *Runs* indicates the number of runs in the binary stream. The hypothesis to be tested is that the distribution of runs in the stream fits a binomial population for which $\Pr(1) = \Pr(0) = \frac{1}{2}$. The test applied is adapted from [MOOD40].

The long runs are added together to form new variables s_{0k} and s_{1k} corresponding to the number of gaps and blocks of length k or more, where

$s_{0k} = \sum_{i=k}^{n_0} r_{0i}$ and n_0 is the number of zeros in the stream.

By adding the long runs together a certain amount of information will be lost. In order to minimise the amount of information lost, it is recommended here that $k = \left\lfloor \log_2 \frac{n+1}{5} - 1 \right\rfloor$.

For a stream of length n = 10^6 this would give a maximum value of k = 16, and hence the number of gaps of length 16 or more would be added together to give $s_{0,16}$ and the number of blocks of length 16 or more would be added together to give $s_{1,16}$.

Explanation of terms:

n = number of bits in stream

n_1 = number of ones in the bit stream

r_{0i} = number of runs of 0 of length i

s_{0i} = number of runs of 0 of length i for i < k

s_{0k} = number of runs of 0 for lengths ≥ k

r_{1i} = number of runs of 1 of length i

s_{1i} = number of runs of 1 of length i for i < k

s_{1k} = number of runs of 1 for lengths ≥ k

The variables:

$$u_i = \frac{r_{1i} - n(\tfrac{1}{2})^{2+i}}{\sqrt{n}} \quad i = 1,\dots,k-1$$

$$u_k = x_k = \frac{s_{1k} - n(\tfrac{1}{2})^{k+1}}{\sqrt{n}}$$

$$u_{k+i} = y_i = \frac{r_{0i} - n(\tfrac{1}{2})^{2+i}}{\sqrt{n}} \quad i = 1,\dots,k-1$$

$$u_{2k} = z = \frac{n_1 - \tfrac{1}{2}n}{\sqrt{n}}$$

are asymptotically normally distributed with zero means and variances and covariances:

$$\sigma(x_i, x_i) = \sigma(y_i, y_i) = (\tfrac{1}{2})^{i+2} - (2i-1)(\tfrac{1}{2})^{2i+4}$$

$$\sigma(x_i, x_j) = \sigma(y_i, y_j) = (1-i-j)(\tfrac{1}{2})^{i+j+4}$$

$$\sigma(x_i, x_k) = -(i+k)(\tfrac{1}{2})^{i+k+3}$$

$$\sigma(x_k, x_k) = (\tfrac{1}{2})^{k+1} - (2k+1)(\tfrac{1}{2})^{2k+2}$$

$$\sigma(x_i, y_j) = (5-i-j)(\tfrac{1}{2})^{i+j+4}$$

$$\sigma(x_k, y_j) = (4-k-j)(\tfrac{1}{2})^{k+j+3}$$

$$\sigma(x_i, z) = \sigma(y_i, z) = (i-2)(\tfrac{1}{2})^{i+3}$$

$$\sigma(x_k, z) = (k-5)(\tfrac{1}{2})^{k+2}$$

$$\sigma(z, z) = \sigma_{zz} = \tfrac{1}{4}$$

Test procedure:

1. Determine k.
2. Take a sample stream of n bits from a stream cipher. Determine the number of runs of each length to give s_{1i} and s_{0i} for $i = 1,\dots,k$.
3. Calculate u_j for $j = 1,\dots,2k$ using above formulae.
3. Determine $S = [\sigma_{ij}]$ which is a $2k \times 2k$ matrix. Calculate $S^{-1} = [\sigma^{ij}] = [\sigma_{ij}]^{-1}$.

   This will require obtaining the inverse of a matrix of up to 32^2 (1024) elements for $n \le 10^6$ bits. Calculate $Q = \mathbf{u}^T S^{-1} \mathbf{u} = \sum \sigma^{ij} u_i u_j$ which follows a χ^2_{2k} distribution (chi-squared distribution with 2k degrees of freedom). There are $(2k)^2$ terms in this sum.

$$Q = \sum_{i=1}^{2k} \sigma^{ii} u_i^{\,2} + 2\sum_{i<j} \sigma^{ij} u_i u_j$$

The significance probability value, p, of the chi-squared distribution is calculated for this statistic. A small value of p indicates a significant result. For

large streams a highly significant result, $p < 0.1\%$, indicates a possible weakness in the algorithm.
The runs test can be used to support results from the previous tests. Failure of the runs test indicates that there is a bad distribution of run lengths or that there are no runs recorded above a certain length that are expected to occur for streams of the sample size. The zero frequencies recorded will result in a higher chi-squared statistic thus giving a smaller significance probability.
Example:

Test stream:

10100010000101110001011000111010101010101010000001

Calculations and results:

$$n = 50$$

$$\text{total runs} = 31$$

$$n_1 = 21$$

$$k = \lfloor \log_2 \tfrac{n+1}{5} - 1 \rfloor = \lfloor \log_2 \tfrac{51}{5} - 1 \rfloor = 2$$

$$s_{01} = 10, s_{02} = 5, s_{11} = 13, s_{12} = 3$$

$$u_1 = x_1 = \frac{13 - 50(\frac{1}{2})^3}{\sqrt{50}} = 0.9545941546018$$

$$u_2 = x_2 = \frac{3 - 50(\frac{1}{2})^3}{\sqrt{50}} = -0.4596194077713$$

$$u_3 = y_1 = \frac{10 - 50(\frac{1}{2})^3}{\sqrt{50}} = 0.5303300858899$$

$$u_4 = z = \frac{21 - \frac{1}{2}(50)}{\sqrt{50}} = -0.5656854249492$$

$$\sigma(u_1, u_1) = \sigma(u_3, u_3) = (\tfrac{1}{2})^3 - (2-1)(\tfrac{1}{2})^6 = 0.109375$$

$$\sigma(u_1, u_2) = -3(\tfrac{1}{2})^6 = -0.046875$$

$$\sigma(u_2, u_2) = (\tfrac{1}{2})^3 - 5(\tfrac{1}{2})^6 = 0.046875$$

$$\sigma(u_1, u_3) = 3(\tfrac{1}{2})^6 = 0.046875$$

$$\sigma(u_2, u_3) = 1(\tfrac{1}{2})^6 = 0.015625$$

$$\sigma(u_1, u_4) = \sigma(u_3, u_4) = -1(\tfrac{1}{2})^4 = -0.0625$$

$$\sigma(u_2, u_4) = -3(\tfrac{1}{2})^4 = -0.1875$$

$$\sigma(u_4, u_4) = \sigma_{44} = 0.25$$

Elements of the inverse matrix, S^{-1} :

$$\sigma^{11} = 6\tfrac{2}{3}, \sigma^{12} = -5\tfrac{1}{3}, \sigma^{13} = -4, \sigma^{14} = -3\tfrac{1}{3}, \sigma^{21} = -5\tfrac{1}{3}, \sigma^{22} = -5\tfrac{1}{3}, \sigma^{23} = 0, \sigma^{24} = -5\tfrac{1}{3},$$

$$\sigma^{31} = -4, \sigma^{32} = 0, \sigma^{33} = 12, \sigma^{34} = 2, \sigma^{41} = -3\tfrac{1}{3}, \sigma^{42} = -5\tfrac{1}{3}, \sigma^{43} = 2, \sigma^{44} = -\tfrac{1}{3}.$$

Q = 8.4733.. follows a χ_4^2 distribution.
p = 0.076
Interpretation:

7.6% of bit streams of length 50 will have a distribution of run lengths further from the expected distribution than this sample gives. This sample satisfies the runs distribution test.

The length of the longest run was also recorded.

Given a bit stream of length N, the expected number of runs = $\frac{\mathbf{N+1}}{\mathbf{2}}$.

Hence, for a bit stream of length 2^n, the expected number of runs ≈ 2^{n-1}.

Applying Golomb's Postulates, it is expected that $\frac{1}{2^i}$ of the runs have length i in an infinite random binary stream. So in a random bit stream of length 2^n, the expected number of runs of length k ≈ $2^{n - k - 1}$.

C.1.6 Sequence Complexity Test

The sequence complexity, c(s), is the number of different substrings encountered as the stream, s, is viewed from beginning to end [LEMP 76].
Example (n = 16) :
s = 1 0 0 1 1 1 1 0 1 1 0 0 0 0 1 0
Marking in different substrings :
s = 1/0/0 1/1 1 1 0/1 1 0 0/0 0 1 0/
Here the sequence complexity c(s) = 6
A *threshold value* of sequence complexity is used to measure the randomness of a sequence. This *threshold value* is $\frac{n}{\log_2 n}$ where n is the total bits in the stream. A stream with a sequence complexity measure below this *threshold value* would be considered to be patterned, ie not random. For the example given, the *threshold value* $= \frac{16}{4} = 4$. Hence the stream is not considered patterned.
An expected value for the sequence complexity of a random stream of the same length is calculated using the following algorithm [GUST 96]:

```
$i = 2;$
$c = 2;$
while $(i < n)$ do
begin
    $i = i + \left\lfloor \frac{\log(i-1)}{\log(2)} \right\rfloor + 2;$
    $c = c + 1;$
end;
if $(n < i)$ then $c = c - 1;$
```

It is expected that a good pseudo-random number sequence has a sequence complexity which is close to this value. It should be noted that the expected value of sequence complexity is always greater than the threshold value. However, a bit stream will only be considered to not satisfy the sequence complexity test if the value of c(s) is less than the threshold value.
The sequence complexity is used to replace the autocorrelation test which is commonly used to determine any periodicity in the pseudorandom number generator. Periodicity would greatly reduce the number of "different" substrings encountered. Hence c(s) would be low and fall below the threshold value. [DAWS 91]
Example
Test stream:

101000100001011100010110001110101010101010000001

Calculations and results:

$\text{n} = 50$

$\text{c(s)} = 10$

$\text{Threshold value} = 8.859191$

$\text{Expected value} = 13$

Interpretation:

This sample stream is considered random based on the sequence complexity test.

C.1.7 Linear Complexity Test

C.1.7.1 Linear Complexity

The linear complexity test checks for the minimum amount of knowledge (bits) needed to reconstruct the whole stream. Every finite stream, s, can be produced by a linear feedback shift register (LFSR). The length of the shortest LFSR which will produce the stream is said to be the linear complexity of the stream, which will be denoted by L(s).
If the value of L(s) is L then 2L consecutive terms can be used to reconstruct the whole sequence using the Berlekamp Massey algorithm. [MASS 69]
Hence, in order to avoid stream reconstruction, the value of L should be large.

Example:

01011001010100100111100000110111001100011101011111101101

This shortest recurrence relation which will create this sequence is:

$$u(t+6) = u(t+5) \oplus u(t+4) \oplus u(t+1) \oplus u(t)$$

where $\oplus$ is addition mod 2, and the first bit is *u*(0).
For example:

If $t = 0$ then $\begin{array}{c} u(6) = u(5) \oplus u(4) \oplus u(1) \oplus u(0) \\ 0 = 0 \oplus 1 \oplus 1 \oplus 0 \end{array}$.

If $t = 1$ then $\begin{array}{c} u(7) = u(6) \oplus u(5) \oplus u(2) \oplus u(1) \\ 1 = 0 \oplus 0 \oplus 0 \oplus 1 \end{array}$.

If $t = 2$ then $\begin{array}{c} u(8) = u(7) \oplus u(6) \oplus u(3) \oplus u(2) \\ 0 = 1 \oplus 0 \oplus 1 \oplus 0 \end{array}$.

This means that the linear complexity, L(s), of this sequence is six. If any twelve consecutive bits are known then the whole sequence can be reconstructed. [MASS 69]
It should be noted that some keystreams can pass all the previous tests yet possess a very small linear complexity. An example of this is an m-sequence (see [RUEP 84]). An m-sequence has a period of length $2^{\text{L}} - 1$ and a linear complexity of L. An m-sequence has the *best* possible distribution of zeros and ones for a sequence of period $2^{\text{L}} - 1$. In this fashion an m-sequence appears to be *statistically* random in terms of tests C.1.1 to C.1.6. In fact m-sequences are commonly used as *white noise* generators. However, in terms of their use in a stream cipher an m-sequence offers very low security. Knowledge of only 2L consecutive bits of the keystream is needed to derive the defining LFSR and hence determine the whole keystream.
For large n, L(s) is approximately normally distributed with $\mu = \frac{n}{2}, \sigma^2 = \frac{86}{81}$ [RUEP 84], [KREY 81]. Using the standardised normal statistic

$z = \sqrt{\frac{81}{86}}(L(s) - \frac{n}{2})$ the significance probability value, *p*, of the normal distribution is calculated.
Since only low values of L(s) signify a possible weakness to the cipher, only a one-tailed test (lower tail) need apply. A small value of p indicates a significant result. For large streams a highly significant result ($p < 0.1\%$) indicates a possible weakness in the algorithm.
The linear complexity test by itself can classify as random, streams which may be highly patterned, or contain large substrings which are highly patterned. Some of the previous test results should support this. e.g. a stream of $\frac{n}{2} - 1$ zeros followed by a one, and then followed by a repetition of these $\frac{n}{2}$ terms, has a linear complexity of $\frac{n}{2}$. This stream would be classified as being random using the linear complexity test. Clearly, such a stream is highly patterned and would not satisfy the previous tests. However, it is possible to construct a stream of length n which would pass all the previous statistical tests, and have a linear complexity of approximately $\frac{n}{2}$ yet would contain a large highly patterned substring. Hence the following linear complexity profile tests are carried out.

C.1.7.2 Linear Complexity Profile

Since some highly patterned streams can give a linear complexity measure close to $\frac{n}{2}$ a second test measures the change in the linear complexity profile of the stream as each bit is added. Let s(i) be the substring formed by taking the first i bits of s. If L(s(i)) for i = 1,...,n denotes the linear complexity of s(i) then the values of s(i) are defined to be the linear complexity profile of s and should follow approximately the $\frac{i}{2}$ line [MASS 69]. A failure in this test would highlight any large deviations from the $\frac{i}{2}$ line, which would appear for strings passing the linear complexity test and containing any large highly patterned substrings. A change in linear complexity signifies a *jump*.
There are two tests relating to the Linear Complexity Profile:

C.1.7.3 Linear Complexity Profile – Number of Jumps

Let the total number of jumps be F. For large n, F is approximately normally distributed with $\mu = \frac{n}{4}$ and $\sigma^2 = \frac{n}{8}$ [CART 87]. The standardised statistic for the number of jumps is $z = \sqrt{\frac{8}{n}}(F - \frac{n}{4})$. The significance probability, *p*, for this standardised statistic is calculated. Since a small number of jumps would indicate a sequence within which patterns may exist, a one-tailed test (lower tail) is applied. A small value of p ($p < 0.1\%$) indicates that the number of jumps in linear complexity is low, and there may be patterns in the stream which would indicate a possible weakness in the cipher.

C.1.7.4 Linear Complexity Profile – Jump Size

If a stream passes the test on the number of jumps in linear complexity, then the distribution of jump heights may be investigated. The height of a jump is the difference in linear complexity when a change occurs. Let the total number of *jumps* in linear complexity be F, where f_i is the number of jumps of *height* i. For a random string based on Bernoulli trials where the probability of a one on each trial is one half, the probability, p_i that a given jump has height i is given by $p_i = (\frac{1}{2})^i$. Hence the expected number of jumps of height i, e_i, is given by $e_i = p_i \times F$.

The chi-squared statistic used is $\chi^2 = \sum_{i=1}^{m} \frac{(f_i - e_i)^2}{e_i}$ [CART 87]. The maximum value of $i = m$ is determined from the condition for the chi-squared test, that $e_i > 5$. The number of degrees of freedom, $m - 1$, is determined from the sample taken.

The significance probability value, p, of the chi-squared distribution is calculated for this statistic. A small significance probability indicates a significant result – i.e., the stream is considered to be non-random. For large samples a highly significant result, $p < 0.1\%$, indicates a possible weakness in the algorithm.

Example

Test stream:

10100010000101110001011000111010101010101010000001

Calculations and results:

Linear Complexity Test

$n = 50$

$\mu = \frac{n}{2}$

$\sigma^2 = \frac{86}{81}$

$L(s) = 25$

$z = \sqrt{\frac{81}{86}}(25 - \frac{50}{2}) = 0$

$p = 0.5$

Interpretation:

50 % of bit streams of length 50 will have a linear complexity less than this sample. This sample satisfies the linear complexity test.

Hence $2 \times L(s) = 50$ bits (the whole stream) is needed to reconstruct the stream using the Berlekamp-Massey algorithm.

Linear Complexity Profile - Number of jumps

$F = 15$

$z = \sqrt{\frac{8}{50}}(15 - \frac{50}{4}) = 1$

$p = 0.8413$

Interpretation:

84.13% of streams of length 50 will have a number of jumps in linear complexity less than this sample. This sample satisfies the test on the number of jumps in linear complexity.

Linear Complexity Profile – Jumps size

$f_1 = 11,\ f_2 = 2,\ f_3 = 0,\ f_4 = 1,\ f_5 = 1.$

$e_1 = 7.5$ Since $e_i < 5$ for $i > 2$, then these values are combined to give $e_{2+} = 7.5$. The corresponding values of f_i are combined to give $f_{2+} = 4$.

$$\chi^2 = \tfrac{(11-7.5)^2}{7.5} + \tfrac{(4-7.5)^2}{7.5} = 3.27$$

$$\text{Degrees of freedom} = m - 1 = 2 - 1 = 1$$

$$p = 0.0707$$

Interpretation:

Approximately 7.07% of bit streams of length 50 will have a sump size distribution further from the expected distribution than this sample gives. The sample satisfies the test on the distribution of the linear complexity jump size.

Results of CRYPT-X Tests

The p-value obtained from a test represents the probability of obtaining a result further than the test statistic lies from that expected, if the algorithm produces a random stream. Very small p-values would support non-randomness for the given measure.

Length of each RC4 keystream = 2^{20} bytes (8,388,608 bits).
Length of each RC4 keystream (Complexity tests) = 100,000 bits.
Number of keystreams for each test = 100

The table below gives the number of p-values falling below 0.1, 0.05 and 0.01 from the 100 RC4 keystreams tested. In a sample of 100 keystreams, the expected count for the number of p-values less than 0.1 is 10, for 0.05 is 5, and for 0.01 is 1.

	Number of p-values less than:		
Test	***.10***	***.05***	***.01***
Frequency	6	4	3
Binary Derivative (1)	8	6	2
Binary Derivative (2)	11	7	1
Change Point	24	13	5
Subblock (b = 4)	9	3	0
Subblock (b = 8)	9	6	1
Subblock (b = 16)	13	5	1
Subblock (b = 30)	9	4	0
Runs Distribution	5	3	2
Longest Run	Max = 33	Next = 28	
Linear Complexity	5	5	1
LC profile - Jumps	12	8	0
LC Profile - Jump Size	20	13	3
Sequence Complexity	Max = 6143	Min = 6110	

Further analysis was applied to the results when more than k of the p-values were greater than k(.01)%. An upper 99% limit for the count when k = 10 is calculated using:

$$100\left(0.1+2.326\sqrt{\frac{.1(1-.1)}{100}}\right) = 17$$

The corresponding values for k = 5 and k = 1 are: 10 and 3.
Counts falling above these values would be classified as significant.

The results for the Change Point test and the Linear Complexity Jumps Size test are the only ones that show this significance.

Length of Longest Run

For a random bit stream of length 2^{20}, the expected number of runs of length $33 \approx 2^{20 - 33 - 1} = 2^{-14}$. This implies that it is highly unlikely that a run of length 33 will appear in a bit stream of this length.

The expected number of runs of length $22 \approx 2^{20 - 22 - 1} = 2^{-3} \approx 0.125$, which supports that this is a more likely occurrence for the length of the longest run.

It would appear that the length of the longest run exceeds what would be expected in bit streams of this length. It should be noted that the length of the longest run may not exceed 33 for much longer streams. Hence we cannot conclude that this result shows any weakness in the RC4 algorithm.

Results of Linear Complexity Tests on Longer RC4 Keystreams

Length of output stream = 819,200 bits.
Number of streams = 5
Expected linear complexity for randomness = 409,600

Test	***Key 1***	***Key 2***	***Key 3***	***Key 4***	***Key 5***
Linear Complexity	409,599	409,598	409,600	409,600	409,600
LC p-value	0.1659	0.0261	0.5	0.5	0.5
Linear Complexity Profile:					
Jumps p-value	0.8517	0.6691	0.4938	0.7035	0.4751
Jump Size p-value	0.4088	0.5908	0.8016	0.1619	0.5926

Sequence Complexity

For streams of length 10^5 bits the sequence complexity threshold value = 6,021 and the expected value for randomness = 6,056.
Both the minimum and maximum values obtained are above these values, and hence there is no indication on non-randomness based on the sequence complexity test.

Results of Frequency Test Applied to Bit Positions in 8-bit Subblocks

Length of each RC4 keystream = 2^{20} bytes (8,388,608 bits).
Number of 8-bit subblocks = $2^{23}/2^{3} = 2^{20}$ bits (1,048,576 bits)
Number of keystreams for each test = 100

The table below gives the number of p-values falling below 0.1, 0.05 and 0.01 from the 100 RC4 keystreams tested. In a sample of 100 keystreams, the expected count for the number of p-values less than 0.1 is 10, for 0.05 is 5, and for 0.01 is 1.

Bit Position in 8-bit Subblocks	***Number of p-values less than:***		
	0.1	***0.05***	***0.01***
Position = 1	6	4	2
Position = 2	6	3	2
Position = 3	10	5	2
Position = 4	12	4	0
Position = 5	10	4	1
Position = 6	12	6	3
Position = 7	10	8	0
Position = 8	13	8	2

Further analysis was applied to the results when more than k of the p-values were greater than k(.01)%. An upper 99% limit for the count when k = 10 is 17, and the corresponding values for k = 5 and k = 1 are: 10 and 3. Counts falling above these values would be classified as significant.

The results of the frequency test on bit positions in 8-bit subblocks show that no bit positions give significant results.

Bibliography

[BEKE 82] H. Beker and F.Piper, **Cipher Systems: The Protection of Communications**, Northwood Books, London, 1982.

[BHAT 77] G. Bhattacharyya and R. Johnson, **Statistical Concepts and Methods**, John Wiley & Sons, 1977.

[CARR 88] J.M. Carroll and L.E. Robbins, “Using binary derivatives to test an enhancement of DES”, **Cryptologia**, Vo1 12 number 4, 1988, pp 193-208.

[CART 87] G. Carter, “A statistical test for randomness based on the linear complexity profile of a binary sequence”, **Technical Report** for Racal Comsec Ltd., 1987.

[DAWS 91] E.P. Dawson, **Design and Cryptanalysis of Symmetric Ciphers**, PhD Thesis, Queensland University of Technology, 1991.

[GUST 96] H.M. Gustafson, **Statistical Analysis of Symmetric Ciphers**, PhD Thesis, Queensland University of Technology, Brisbane Australia, 1996.

[KREY 81] E. Kreysig, **Introductory Mathematical Statistics**, John Wiley and Sons, 1981.

[LEMP 76] A. Lempel and J. Ziv, “On the complexity of finite sequences”, **IEEE Trans. on Information Theory**, Vol.IT-22, Jan.1976,pp 75-81.

[MASS 69] J.L. Massey, “Shift register sequences and BCH decoding”, **IEEE Transactions on Information Theory**, Vol. IT-15, Jan. 1969, pp 122-127.

[MOOD 40] A.M. Mood, “The distribution theory of runs”, **Ann. Math. Statist.**, Vol 11, 1940, pp 367-392.

[PETT 79] A.N. Pettitt, “A non-parametric approach to the change - point problem”, **Appl. Statist.**, Vol. 28 No. 2, 1979, pp 126-135.

[RUEP1 84] R.A. Rueppel, **New Approaches to Stream Ciphers**, PhD Thesis, Swiss Federal Institute of Technology, 1984.

[RUEP2 84] R.A. Reuppel, “**Analysis and Design of Stream Ciphers**”, Springer-Verlag, 1986.

Appendix D – An Implementation of RC4 in ANSI C

This appendix contains code for the implementation of RC4 that is described in section 4.3

D.1 The RC4.h header file

```
/**
 * @file rc4.h
 * Contains definitions for the RC4 stream cipher
 */
#ifndef _RC4_H_
#define _RC4_H_

#ifdef __cplusplus
extern "C" {
#endif

#include "defs.h"

/* Status codes */
#define RC4_OK 0

#define STATE_SIZE 256

#ifdef _OPT_
#define WORD word32
#else
#define WORD word8
#endif

typedef struct
{
    WORD state[STATE_SIZE];
    WORD i, j;

} RC4;

/**
 * Initialize RC4 keystream generator
 * @param  rc4   [In/Out]  RC4 keystream generator
 * @param  k     [In]      variable length key - byte 0 denotes size
in bytes
 * @returns RC4_OK
 */
int init_ks(RC4* rc4, const word8* key);

/**
 * Extract another word from the initialized RC4 keystream generator
 * @param  rc4  [In/Out]  RC4 keystream generator
 * @returns next word in key stream
 */
word8 update(RC4 *rc4);
/**
 * Encrypt text with the output of the RC4 keystream generator
 * @param  rc4    [In/Out]  RC4 keystream generator
 * @param  text   [In/Out]  plaintext/ciphertext
 * @param  length [In]      number of eight-byte blocks to encrypt
 * @returns RC4_OK
 * @note To encrypt texts of length not divisble by eight, pad
 * text buffer and discard unwanted texts
 */
int encrypt(RC4 *rc4, word8 *text, const int length);

#ifdef __cplusplus
}
#endif
#endif
```

D.2 The RC4.c Source File

```
/**
 * @file rc4.c
 * Contains implementation of the RC4 stream cipher
 */

#include "rc4.h"

#define SWAP(a, d, t) t = *a;  *a = *(a+d); *(a+d) = t;

/**
 * Initialize RC4 keystream generator
 * @param  rc4   [In/Out]  RC4 keystream generator
 * @param  k     [In]      variable length key - byte 0 denotes size
in bytes
 * @returns RC4_OK
 */
int init_ks(RC4* rc4, const word8* k)
{
    word8 key_size = k[0];
    word8 *key     = k+1;
    word8 t;

    word32 i, j = 0;

    for (i = 0; i < STATE_SIZE; i++) {
        rc4->state[i] =i;
    }
    for (i = 0; i < STATE_SIZE; i++) {
        j = (j + rc4->state[i] + key[(i % key_size)]) % 256;
        SWAP(&rc4->state[i], j-i,t );
    }
    rc4->i = rc4->j = 0;
    return RC4_OK;
}
```

```
#define UPDATE(state, state_i, state_j, val_i, val_j, result) \
     result ^= (state_i=(state_i+1) & 0xFF, \
                val_i=state[state_i], \
                state_j=(state_j+val_i) & 0xFF, \
                state[state_i]=val_j=state[state_j], \
                state[state_j] =val_i, \
                state[(val_i+val_j)&0xFF]);

/**
 * Extract another word from the initialized RC4 keystream generator
 * @param  rc4  [In/Out]  RC4 keystream generator
 * @returns next word in key stream
 */
word8 update(RC4 *rc4)
{
    word32 state_i, state_j;
    word8 result = 0;
    UPDATE(rc4->state, rc4->i, rc4->j, state_i, state_j, result);
    return result;
}

/**
 * Encrypt text with the output of the RC4 keystream generator
 * @param  rc4    [In/Out]  RC4 keystream generator
 * @param  text   [In/Out]  plaintext/ciphertext
 * @param  length [In]      number of eight-byte blocks to encrypt
 * @returns RC4_OK
 * @note To encrypt texts of length not divisble by eight, pad
 * text buffer and discard unwanted texts
 * @note Can get further optimization by passing 32 bit parameters...
 */
int encrypt(RC4          *rc4,
            word8        *text,
            const int    length)
{
    /* large speedup by proxying variables */
    word32 *state  = rc4->state;
    word32 state_i = rc4->i;
    word32 state_j = rc4->j;

    word32 val_i, val_j;
    word32 idx;

    word8 *t = text;

    /* Unrolling trick means user discards unwanted ciphertext */
    for (idx = 0; idx < length; idx++) {
        UPDATE(state, state_i, state_j, val_i, val_j, t[0])
        UPDATE(state, state_i, state_j, val_i, val_j, t[1])
        UPDATE(state, state_i, state_j, val_i, val_j, t[2])
        UPDATE(state, state_i, state_j, val_i, val_j, t[3])
        UPDATE(state, state_i, state_j, val_i, val_j, t[4])
        UPDATE(state, state_i, state_j, val_i, val_j, t[5])
        UPDATE(state, state_i, state_j, val_i, val_j, t[6])
        UPDATE(state, state_i, state_j, val_i, val_j, t[7])
        t += 8;
    }
    return RC4_OK;
}
```