Protecting Sensitive Data on a PC by a Custom Algorithm

Fredrik Lillehagen Skarderud

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2005

The MSc programme in Information Security
is run in cooperation with the Royal Institute
of Technology (KTH) in Stockholm.

Institutt for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

To store secrets and sensitive data in secure ways is necessary in the modern life. Since most hardware technologies used to store sensitive data are expensive for a single user, it is very common to store this kind of data by the means of software or software implemented encryption procedures. If for example the secret to protect is the secret key in a Public Key Infrastructure environment it would be important to determine the best way of storing it on a hard drive. It is also important to determine how secure the existing storing applications or available encryption algorithms are. A brief overview of the most common methods and solutions for storing secrets, such as encryption algorithms are given, and a proposal of a new stream cipher, its implementation and analysis are presented in this thesis.

The stream cipher makes use of well known and understood techniques, such as linear feedback shift registers, irregular clocking, and truth tables, as elements and building blocks to achieve very high security and easy understanding of the fundamental structure of the cipher.

The cipher is implemented in C#, and both the implementation and design of the cipher is made to achieve high efficiency when running in software, especially on a standard 32 bit processor (CPU).

Cryptanalysis and thorough statistical testing are applied to the cipher to evaluate its cryptographic strength and security.

Both the cryptanalysis and the statistical testing conducted on the cipher indicate that the cipher is secure and has good statistical properties. Efficiency testing shows that the cipher design is very fast in software.

Sammendrag (Abstract in Norwegian)

Å kunne lagre hemmeligheter og sensitive data på en trygg måte er en nødvendighet i dagens samfunn. Siden en stor menge av hardwareteknologi som tilbyr "sikker " lagring av sensitiv data er dyrt for den enkelte bruker, er det veldig vanlig å benytte seg av programvare eller software-implementerte krypteringsalgoritmer for å løse dette problemet.

Hvis for eksempel den sensitive informasjonen vi vil beskytte er den hemmelige nøkkelen i et privat/offentlig nøkkelpar, er det viktig å finne den beste måten å lagre nøkkelen på, hvis mediet som brukes er en PC med harddisk. Det er også viktig å vite noe om hvor sikre og hvor bra eksisterende applikasjoner og krypteringsalgoritmer implementert i software er for å få en viss trygghetsfølelse når det gjelder lagringen av sensitiv data.

En gjennomgang av noen av de mest kjente og brukte krypteringsalgoritmene i software og andre metoder for lagring av sensitive data, samt forslag til et nytt stream cipher for lagring av sensetive data på en PC, implementasjon og testanalyse er hovedpunktene i denne masteroppgaven.

Ved å anvende godt kjente og analyserte teknikker, som for eksempel lineære shift registre, iregulær klokking og sannhetstabeller som bygningsblokker for krypteringsalgoritmen, er målet å oppnå høy sikkerhet og god forståelse for algoritmens struktur.

Algoritmen er implementert i C#. Både algoritmens implementasjon og struktur er gjort med hensyn på å oppnå høy effektivitet i software. Spesielt for standard 32 bits prosessorer som er mest vanlig i dagens standard datamaskiner.

Svakhetsanalyse og grundig statistisk testing er utført på algoritmen for å vurdere algoritmens sikkerhet og styrke.

Både svakhetsanalysen og den statistiske analysen av algoritmen indikerer at den virker sikker og har gode statistiske egenskaper. Praktisk utførte efektivitetstester viser at algoritmen eksekveres raskt i software.

Acknowledgments

I would like to thank my supervisor Professor Slobodan Petrovic for all his ideas and help throughout the master's thesis period. Without his guidance, this thesis would not be possible.

A big thanks also to my friends and mates Ole Martin Dahl, Anders Wiehe, Ole Kasper Olsen and Torkjel Søndrol at room A032 in Gjøvik University College, for humorous times and inspiring talks.

A special thanks goes also to all my friends and family for supporting me and being patient when days have been long and the stress factor huge.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topics Covered by This Thesis

Keeping secrets away from others is one of many aspects of information security. This is essential in many scenarios to maintain good security and reliability, confidentiality and it might even effect integrity. The topic of this thesis is the technical aspect and security analysis of different methods used to protect secret or sensitive data on a PC. Different cryptographic algorithms are described, attacks on these are discussed, and the process of designing a stream cipher is presented. Security and efficiency metrics are applied to this new custom cipher to perform its relevant assessments. A smaller goal for this thesis is to determine if different kinds of software and methods designed to secure sensitive data are good enough and reliable enough to store secrets.

Keywords:

Technology, cryptography, stream cipher, data processing, security architecture, Operating systems security, Code obfuscation, reverse engineering.

1.2 Problem Description

Many software and application developers do not produce hardware. When processing sensitive data, many applications make use of solutions implemented only in software. Hardware is often expensive for a single user both financially and regarding complexity of use. Things also very often get more complex when combining software and hardware. Therefore, many software distributors and applications store secrets in software or on a hard drive to minimize costs and complexity for users and developers. Personal computers represent a hostile environment which raise questions about how good or how securly it is possible to maintain confidentiality and integrity of sensitive information and secrets such as passwords and private keys. This again raise questions on how responsible it is to rely on software-based technology to store secrets and sensitive information.

There are many publicly known encryption algorithms, which can be used to store sensitive data on a hard drive. However, getting hold of software implementations of the algorithm can often be a problem. Many COTS[1] pieces of software claim to have implemented the different algorithms, however they are often expensive, and since it mostly is not open source software, one cannot be sure of what is really implemented. This kind of software mostly stores the keys used for encryption on the same medium as the stored secrets encrypted with the key, so if the key can be found on the same medium, applying the key on the encrypted secret would decrypt the secret with hardly any problems. Such applications provide latency, not security.

Publicly known algorithms are available to everyone. One cannot be sure that algorithms used are not broken by any groups or potential adversaries. The only thing one can be sure of, or reasonable to assume, is that an adversary is at least at the same level of breaking a cipher as the academic literature written about it. Therefore, maybe a cus-

[1]Commercial-off-the-shelf

tom cipher, which is not publicly known, if well designed and tested can be appropriate for storing a single user's sensitve data on a hard drive.

1.3 Justification, Motivation and Benefits

Storing secrets and keeping them secret are essential in our information society. Secrets play an important role in the world of information security and information management in general. Keeping things secret is a must in many different contexts. If for example a secret such as a private key is given away or revealed to others, the private-public key pair would be useless in the future. If the private key is lost somehow or damaged, it would be impossible to repair or generate a new one corresponding to a already existing public key. How to store and where to store the private key is important for the PKI technology to function the way it is intended. Most secrets today are stored on a hard drive using a software system to protect them. The quality of such a protection must be tested very seriously.

Not knowing the capabilities of adversaries, if a public cipher is broken in secret, or if it is possible to calculate the minimum strength of a known cipher is among factors that justify and motivate for this thesis.

Many stream cipher designs in use are either secret or proprietary designs [1]. To design a custom cipher specially adopted to software implementation would lead to a greater variety of choices when selecting a stream cipher for encrypting sensitive and confidential data.

1.4 Research Questions

Research questions to be answered arise from the process of developing the custom cipher. The cipher needs to be fast executed in software, and it needs to be secure. Four essential research question are identified as key questions to be answered throughout this thesis:

- What are advantages or disadvantages of using a custom cipher compared to publicly known ciphers?
- How well can efficiency/security trade off be achieved in cryptographic algorithms for PC protection?
- How secure is the new stream cipher?
- How efficient is the new generator?

1.5 Method

To get an overview over existing cryptographic algorithms, a literature study was used as the method to achieve it. Literature study was also the main used method for learning how different stream ciphers are designed. To get an wide understanding of popular techniques and components used as building blocks for stream ciphers, was very time demanding. In addition to a thorough literature study, dialogs with the supervisor have helped me to learn how to build a customly designed stream cipher.

For the statistical testing, empirical testing on different bit sequences produced by the stream cipher generator were conducted. Hypothesis testing methodology were applied for the testing, the statement H_0 was that the bit sequences produced is random, and the alternate hypothesis H_1 was that the sequences is not random.

1.6 Outline of Chapters

Chapter 2 provides a overview of existing techniques and methods used to protect sensitive data on a PC/in software. Further, chapter 2 gives a short introduction to cryptography, and a presentation of some existing encryption algorithms and known attacks on the specific ciphers.

In chapter 3, the new custom designed stream cipher is presented, as well as a theoretical cryptanalysis of the cipher.

In chapter 4, a thorough statistical testing is applied to the new cipher, and the results are discussed. Chapter 4 also provides efficiency comparisons between the new custom designed cipher and some widely used ciphers.

Chapter 5 discuss some of the research questions presented in Section 1.4. In chapter 6 we summarize the conclusions of the study, and in chapter 7, further work and new questions which have arisen throughout working with the thesis is presented.

2 Review of Existing Methods

2.1 Literature Survey

2.1.1 Existing Theories and Algorithms for Storing Secrets

Data can be stored on any medium which can store information over time. The medium should be persistent, so losing power would not mean losing the data stored on it. When talking about using software to store sensitive data, it most often means storing the software used and the protected data on a hard drive. One method used to store secrets or sensitive data on a hard drive is to encrypt them in some way so the sensitive information would appear encrypted on it. One popular way of doing this is to base the encryption on some kind of password. RFC 2898 [2] discusses methods and proposes a way of using passwords to generate keys for such encryption/decryption methods. [2] is the base for many algorithms and methods for encrypting sensitive data such as a private key in Public Key Infrastructure schemes [3], and for exchanging sensitive data [4].

Other methods of storing secrets on a hard drive could be to hide the secrets in some way. A software tool called Partition Manager[5] uses a technique that makes a hidden partition (or password protected if desired) so only those who know of it would find it. In [6] Eric Cole discusses different aspects of hiding data and use of techniques such as steganography.

Another method to store secrets could be to split the secrets into smaller pieces and then "hide" them individually, making it a puzzle for an intruder. This method is mainly a manual method in the meaning that the secret holder splits the secret by himself, then deploys the secrets in different places on the hard drive or within the file system or operating system. The security of this method relies on the fact that an adversary does not really know what he is looking for when the secret is split into pieces. The separate pieces might also be disguised as something quite different from what it really is. For example, a fake system file or a fake program file.

Another approach is to set rights on the secret data, so only persons authenticated in some way get to access them. This is used in operating systems such as Microsoft Windows and Linux [7, 8] authentication and access control tables. Techniques for authentication could be biometry, password or whichever attribute found good enough for authenticating an individual.

If the sensitive data to be stored are something that one has memorized, such as a password, and this password is used in some kind of authentication process, it is not necessary to use an encryption/decryption algorithm. In such cases a hash algorithm or a HMAC algorithm could be used. This kind of algorithms such as SHA1 [9] and MD5 [10] take some data as input, and generate an output of a given length. Sending the secret into this algorithm would result in the same hash value each time. On the other hand, it would be very difficult to find two different messages that produce the same hash value. In [11] Viega, McGraw claims this to be a very good method of storing passwords on hard drives.

The newest thing regarding storing of sensitive data are virtual storage Farms [12].

This kind of technology provides "outsourcing" for sensitive data. The main principle is to let others handle the data storage, and allow the rightful users to access the data over The Internet at any time preferred. This way of handling data storage derives several new aspects, such as network security, protocol security and IP security.

Many well known cryptographic algorithms and usage areas can be found in [13], and detailed cryptographic theory can be found in [14]. Some of these algorithms will be described more in detail in Section 2.2.

2.1.2 Existing Methods, Solutions and Products

Many applications need to store sensitive data locally on the hard drives to offer desired functionality. For instance, Internet browsers like Microsoft Internet Explorer[1], Opera[2] and Mozilla[3] need to store keys and certificates in a PKI structure to ensure encrypted data flow between client and servers. Mozilla uses the guidelines described in [2] to store such sensitive data [15] while Microsoft Internet Explorer is semi-integrated with the Microsoft operating system, and uses parts of the OS to handle sensitive data. Pretty Good Privacy (PGP) described by Garfinkel in [16] is another popular software, which needs to store secret data. This software let users send encrypted and authenticated messages over the Internet. OpenPGP [17] is a free standard of PGP, which does not make use of the patented IDEA algorithm. In OpenPGP implementations, the secret part of private keys are stored encrypted in a key-ring, where a pass-phrase and an optional salt[4] is used in a "“string to key”" algorithm to generate the key which the private key is encrypted with.

In addition to all the applications that need to store secret data, there are several others whose function is to help users store their sensitive and secret data. Applications such as [18, 19, 20] all offer an of-the-shelf software to help the users manage their confidential data. Most of this software uses a password protected or an encrypted database or file for security. A password is also often all that is used to encrypt the data. The security level achieved in these applications is questionable.

2.1.3 Security of Various Methods, Solutions and Products

Several documents are written, which look at the security and strength of different cryptographic algorithms and protocols. Work such as in [21, 22, 23] are few examples. However, it is not easy to perform a total security analysis of these algorithms [24]. Less information is to be gathered about work done to measure strength and compare the different ways or methods to store sensitive data.

Encryption seems to be the solution for many applications providing secure storing of secrets. Encryption is also often the advise given by others regarding storing of secrets. As en example; Guttman, L. Leong and G. Malkin [25] recommend that we encrypt all private data wherever we store them.

It seems that in most cases of software encryption and products offering this kind of service, the security of the encryption all comes down to the security of a password or something similar. This because the encryption keys are often derived from passwords, pass-phrases or something only the rightful owner should know.

[1]http://www.microsoft.com/windows/ie/default.mspx
[2]http://www.opera.com
[3]http://www.mozilla.org/
[4]Salt: Some arbitrary data combined with the pass-phrase to prevent dictionary attacks.

2.1.4 Practically Achievable Security

In most cases is it possible to rate "confidentiality level" of sensitive data. In some cases, stronger or better methods to protect the secret data are required than in other cases. Having confidentiality and integrity in mind, it would be desirable to develop some techniques to be able to make an estimate of how secure software protection can ever be.

To be able to give an answer to this question, one of the aspects needed to be investigated is how secure the software protecting the sensitive data is. Methods to secure software such as code obfuscation [26], software obfuscation, software diversity [27], white-box cryptography [28], software tamper resistance [29] and security by obscurity [30] are known techniques to protect software against attacks. Code obfuscation/software obfuscation and code transformations are techniques to prevent reverse engineering attacks [31]. Reverse engineering means generating source code from already compiled machine code. White-box cryptography may be applied for protecting secret keys in untrusted host environments. Software tamper resistance is applied for protection against program integrity threats, and software diversity is an approach for protection against automated attack scripts and widespread malicious software.

To compare these techniques and to be able to say something about the security strength in these methods, we must be able to measure and compare them in some way. Little work have been done to develop analysis techniques and metrics for evaluating and comparing the strength of various software protection techniques [32, 33].

Software does not run in a stable or constant environment. Many different operating systems exist and are under constant evolution, and new operating systems are also developing. The *Palladium* project [34] is a work toward a new Microsoft operating system where security is the priority number one. Operating systems are also a factor to be considered when giving an answer to this question.

2.1.5 Information, Attributes and Aspects, Which Are Relevant When Comparing Methods and Solutions Used to Protect Sensitive Data

It is important to identify resistance against any form of attacks directed to the methods described above. Attributes such as quality of software implementation, the easiness of detecting which methods used to protect the secrets and attack statistics (for example the results of penetration testing etc.) are parts of a complete picture to form the strength regarding the security of the method protecting the secrets.

Very little work is done to identify attributes to help develop some method or identify metrics to compare and measure the security strength of different techniques and protection methods [32].

2.2 Cryptographic Algorithms

2.2.1 Introduction

To be able to store or transfer secrets or confidential information have been a demand for centuries. Methods needed to be developed to achieve this goal, and encryption were early introduced as a solution to this problem. An example of such a solution or encryption algorithm is the Ceasar cipher which Julius Ceasar (100-44 BC) used in his "government" communications. However, modern encryption theory and algorithms has its roots from two essential works; "La cryptographie militaire" by Auguste Kerckhoffs [35] and "“Communication theory of secrecy systems”" by Claude Shannon [36] the latter being based upon his previous work in [37]. Kerchoffs is known for his "rules" regarding the use of encryption, whereas Shannon is specially known for his communication model.

Encryption algorithms can be divided into two types of technology, symmetric and asymmetric[5]. The main difference between these two types of encryption is that asymmetric algorithms make use of key pairs where one key is private or secret (only known to the key pair owner), and one key is publicly known. If for example some data are encrypted with the public key, only the holder of the private key can decrypt the data. A widely used asymmetric algorithm is the RSA algorithm [38], which takes advantage of the high complexity of prime factorization in mathematics.

In symmetric encryption algorithms, the same key is used for encryption and decryption. Symmetric algorithms can then again be classified as a block cipher or a stream cipher. The difference of ciphers in these classes is that stream ciphers operate on the entire data to be encrypted "on the fly", while block ciphers divide the data which are to be encrypted into blocks of a given size, then encrypt the data block by block.

Generally, symmetric encryption algorithms are much faster than asymmetric algorithms. So when it comes to encryping larger amount of data, symmetric algorithms are to be used for efficiency. Many different protocols and services uses public key cryptography for authentication, signatures and encryption of a symmetric key which then might be used later in a protocol for efficiency and safety.

Computational complexity is a widely used concept in cryptography. As a basic principle, we can say that the computational complexity for specific cryptographic algorithms is two to the power of the keylength k used by the algorithm. 2^k. This means that also the computational complexity for breaking a cryptographic cipher is also 2^k. However, this is the ideal case. If some method applied on the cipher reduces the effort needed to break the cipher, the computational complexity of braking the cipher is reduced.

2.2.2 Block Ciphers

Block ciphers are the most common and widely used ciphers. Many block ciphers are publicly known ciphers, and they have been studied for decades. The basic structure of most block ciphers is to divide the information which is to be encrypted or decrypted into blocks of a given size, then iterate through an algorithm n times, before one block of ciphertext/plaintext is generated.

2.2.2.1 Designs Principles

Feistel design [39], is a common design principle in many block ciphers. A basic Feistel cipher takes $2l$ plaintext bits, and is a permutation, F, which uses m round permutations, F_i, so that,

$$F = F_0 \circ F_1 \circ ... \circ F_{m-1}$$

where $\circ$ denotes composition of functions.

Round i acts on half the input bits, the r bits, *R*, by means of the keyed function, f_i, and XORs the result with the other half of the bits, the l bits, *L*. It then swaps the left and right halves. Thus we have,

$$[L', R'] = F_i[L, R] = [R, L \oplus f_i(R)$$

where $[L', R']$ becomes the new input $[L, R]$ to round $i + 1$. Although F and F_i must be

[5]Asymmetric cryptography is often referred to as public key cryptography

permutations, the f_i need not be. After two rounds in the Feistel twister, all plaintext bits have been acted on in a non-linear way.

The Feistel structure allows decryption to be accomplished using the same process as described above, using the sub-keys in reverse order. Ciphers like DES, MISTY1 and Camellia are all based on the Feistel structure.

Another design principle is the Substitution-permutation network (SPN). An SPN network separates the role of confusion (substitution) and diffusion (permutation) in the cipher. As with most block ciphers, the cipher is decomposed into iterative rounds where each round comprises a layer of S-boxes (substitution boxes), followed by a permutation or diffusion layer. The S-box layer provides the non-linearity or the confusion, and the permutation layer provides the rapid diffusion. To separate the tasks of confusion and diffusion allows the designer to maximise non-linearity of the S-box(es), and maximise information spread for the diffusion layer. Among ciphers which use the SPN design principles we find ciphers like Rijndael, Khazad, Hierocrypt, SAFER++ and IDEA.

2.2.3 Popular Block Cipher Designs

Beneath follows a short presentation of some popular and widely used block ciphers. Two of the ciphers, SAFER++, and SHACAL, are presentated not because they are widely used, but because they both have a special design.

2.2.3.1 Triple-DES

One variant of triple-DES[6] which is much in use is called two-key Triple-DES. Like all variants of DES, it occurs as a natural extension of the old standard DES algorithm [40]. Two-key Triple-DES operates on 64 bit plaintext blocks, and takes a $2 \times 56 = 112$ bit key as input. The security in Two-key Triple DES is enhanced in particular by repeating the cipher three times and the key twice. The three runs of the encryption algorithm are encrypt, decrypt, then encrypt again. This order is chosen to make the cipher easily backwards compatible with single DES. However, the form encrypt, encrypt, encrypt is also backwards compatible by using the all zero key in the first two encryptions, then a single-DES key on the last encryption.

A double-DES variant of the DES algorithm is not an option due to the meet-in-the-middle attack which renders double-DES with no greater security than single DES [41], [42].

A much better variant of DES is the three-key Triple DES variant. This version of Triple DES operates on 64 bit plaintext blocks, and uses a 168 bit key for encryption. This version of the cipher is also widespread in use, and is considered to be a lot more secure than two-key Triple DES.

Another version of the DES algorithm is the DESX [43] (see also Section 2.2.6). This version of DES also takes three keys as input, but requires only one single DES encryption preceded with XOR with another key, and completed by XOR with a third key. Because DESX only requires to run through the DES encryption procedure once, the efficiency of DESX is much higher than Triple DES variants.

DES is a Feistel cipher, and the key is input linearly via the XOR function, and there are 8 6-bit in, 4-bit out S-boxes applied in parallel to the 48-bit input to give 32 bit output. The DES standard recommends to run 16 rounds of the algorithm to be secure.

Three-key Triple DES is a concatenation of three instances of DES, where a different

[6]Data Encryption Standard

key is input for each instance of DES, to give a total key input length of $3 \times 56 = 168$ bits.

The security of Triple DES is a lot weaker than what 128 bits ciphers should be, and the two-key Triple DES is generally considered weaker than three-key Triple DES [1]. The efficiency performance in software or on a standard workstation PC is rather bad [1]. In [44], Merkle and Hellmann showed that two-key triple encryption can be broken using 2^{56} chosen plaintexts, and 2^{112} single encryptions. The standard way to attack Triple DES is to use the *meet in the middle attack* [13]. This kind of attack requires three plaintext/ciphertext pairs, and 2^{112} encryptions. An *advanced meet in the middle attack* against two-key Triple DES is proposed in [42].

Since Triple DES uses more or less the DES encryption/decryption scheme, all attacks on DES are relevant to Triple DES. In fact, the most successful attacks on reduced round of Triple DES, are the attacks on standard DES. It has been shown that differential cryptanalysis can cover as many as 18 rounds of DES, and it is suspected that this may also be for linear cryptanalysis.

Two of the best known attacks on three-key Triple DES are a *related-key attack* [45] by Kelsey *et al.*, and a *meet in the middle* attack [46] by Lucks. The meet in the middle attack can break three-key triple DES with about 1.3×2^{104} single encryption steps, and 2^{32} known plaintext/ciphertext pairs.

2.2.3.2 IDEA

IDEA [47] operates on 64 bit blocks of plaintext and ciphertext and is controlled by a 128 bit key. To achieve an acceptable security margin, the designers operate it with a requirement of 8.5 encryption rounds. The designers claim to achieve high security by concatenated use of three arithmetic operations from two dissimilar algebraic groups; Addition mod 2^{16}, Multiplication mod $2^{16} + 1$, and bitwise exclusive OR. The combined use of these three operations is used to achieve high non-linearity and to completely replace the more conventional use of S-boxes. The use of these three operations can often result in efficient implementations in software because many processors have special-purpose multiplication operators.

The key schedule of IDEA takes a 128 bit key and turns it into 52 16 bit key sub blocks which are then used throughout the 8.5 encryption rounds.

IDEA has been studied for over a decade and few security flaws have been found. There is no known attack against 5 or more rounds of IDEA. One of the best known attacks on IDEA is found in [48] where an attack on 4.5 out of 8.5 rounds of IDEA is presented. However, it has been found that IDEA has a large collection of weak-key classes. In [49] *Biryukov et al.* present weak-key classes that contain $2^{53} - 2^{64}$ weak keys.

2.2.3.3 AES, Rijndael

Rijndael [50] has recently been selected as the Advanced Encryption Standard AES and has therefore been subject to intensive study in the last few years. Rijndael is a non-Feistel cipher and is a variant of the Square block cipher [51], which employs a combination of optimal diffusion and optimal non-linearity of the S-box. The key is linearly added into the cipher via XOR, and can be either 128, 192, 256 bits long over 10, 12, or 14 rounds, respectively. A round in Rijndael can be written as follows:

```
Round(State,RoundKey)
```

```
{
ByteSub(State)
ShiftRow(State)
MixColumn(State)
AddRoundKey(State,RoundKey)
}
```

ByteSub is the optimally non-linear 8×8 bit S-box operation, which is x^{-1} over $GF^{7}(2^8)$, followed by an affine transformation. ShiftRow is a bytewise permutation over $GF(2^8)^4$, MixColumn is a bytewise affine transform over $GF(2)^{32}$, and AddRoundKey is the XOR of the key onto the output of the round. The diffusion layer is a linear transformation and comprises ShiftRow and MixColumn.

The key schedule of 128 bit Rijndael over 10 rounds takes in a 128 bit key and generates $128 \times 11 = 1408$ round key bits in the form of 11 128 bit subkeys for each round, and one for the beginning.

Some of the most successful attacks against Rijndael are the *Square attack* [50] [52] and the *Collision attack* by Gilbert and Minier [53]. However no attack is known on more than 7-8 rounds of Rijndael being more efficient than exhaustive keysearch. A bigger version of Rijndael also exists which works over a blocksize of 256 bits.

2.2.3.4 SAFER++

The SAFER++ cipher [54] exists in two variants, one which operates on 64 bit plaintext blocks, and one which operates on 128 bit plaintext blocks. Both variants are based on the previous SAFER and SAFER+ ciphers. The 128 bit plaintext version is adapted for use in the authentication scheme in the wireless communication protocol Bluetooth. Recent results show that using algebraic optimizations, the most common Bluetooth PIN can be cracked within less than 0.06-0.3 seconds by exploiting properties in the SAFER implementation [55].

The version which operates on 64 bit blocks takes a 128 bit key as input and generates 64 bit ciphertext blocks. The designers recommend to use 8 encryption rounds to achieve sufficient security for this version of the cipher. SAFER++ uses a *4-point Pseudo-Hadamard Transformation* to achieve fast, rapid diffusion at low complexity. One 16 byte subkey is used with each round, along with one post-cipher *output transformation* which is a final 8 byte *addition*. The cipher uses two incompatible group additions to achieve key addition, namely bitwise XOR (uses the 1, 4, 5, 8 subkey bytes) and bytewise addition modulo 256 (uses the 2, 3, 6, 7 subkey bytes). The S-boxes used in the cipher are exponential and logarithmic functions modulo 257.

The key schedule for the 64 blocks SAFER++ uses 9 16 byte bias words in order to randomize the produced subkeys so as to help avoid weak keys. These bias words, B_j, are determined by,

$$B_{i,j} = 45^{45^{(17i+j) \bmod 257}} \bmod 257$$

where $B_{i,j}$ is the i-th byte of B_j.

It is worth noticing that the spreading of 64 bits to 128 bits, and the subsequent dropping of 128 bits down to 64 bits is an unusual feature of SAFER++ which distinguishes it from many other block ciphers.

[7]Galois Field [14]

No security flaws have been found with SAFER++ (64 bit version) [54]. The designers conclude that SAFER++ with six rounds or more is secure against differential cryptanalysis, and with two and a half round or more is secure against linear cryptanalysis. However, one of the best attacks known on SAFER++ [56] shows that three and a half rounds of SAFER++ (64 bit version) can be attacked requiring 2^{33} known plaintexts.

2.2.3.5 SHACAL Family

SHACAL [57] is a 160 bit block cipher using a 512 bit key based on the FIPS[8] hash function standard. There are three variants of the SHACAL family of ciphers, all based on different versions of the Sha hash algorithms. SHACAL-1, which is based on the SHA-1 hash algorithm [9], is presented beneath.

SHACAL-1 places the 160 bit plaintext in 5 concatenated 32 bit variables, A, B, C, D, E, and updates these five variables on each of 80 consecutive steps, so that the final ciphertext is contained in the variables after 80 steps. In this process, the 512 bit key is expanded to 2560 bits. In each round of the algorithm the main elements used are addition modulo 2^{32}, AND, OR, and data rotation. Since the encryption algorithm is based on the hash function SHA-1, it is considered to have very fast implementations.

The key schedule for SHACAL-1 is linear, and it expands the 512 bit master key to 2560 bits. The master key is a concatenation of 16 32 bit words, and if the master key is shorter than 512 bit, padding is used to generate the 512 bit master key.

No security flaws have been found in SHACAL-1. However, recently (February 2005) Xiaoyun Wang *et al* [58] found collisions for the SHA-1 algorithm in 2^{69} calculations, about 2,000 times faster than brute force (2^{80}). This finding might also have consequences for the SHACAL-1 encryption scheme, since the structure of SHACAL-1 is very similar to the SHA-1 hash algorithm. One of the best known attacks on SHACAL-1 is the Rectangle attack described in [59]. This attack works for 49 steps of the compression function with a data complexity of $2^{151.9}$ chosen plaintexts and a time complexity of $2^{508.5}$.

2.2.4 Stream ciphers

A stream cipher is an algorithm for encrypting a sequence of elements or characters from a plaintext alphabet, usually the binary alphabet which consists of only zeroes and ones. Stream ciphers are commonly classified as being synchronous or self-synchronising. In a synchronous stream cipher the keystream is generated independently of the plaintext and ciphertext, so the keystream depends only on the key. In contrast, the keystream of a self-synchronising stream cipher depends on the key and a fixed amount of the previously generated ciphertext. Most stream ciphers can be classified as additive stream ciphers. An additive stream cipher is a synchronous cipher in which the ciphertext is the XOR of the plaintext and the keystream. In specific applications, stream ciphers are more appropriate than block ciphers:

- Stream ciphers are generally faster than block ciphers.
- Stream ciphers have less hardware complexity.
- Stream ciphers process the plaintext character by character, so no buffering is required to accumulate a full plaintext block (unlike block ciphers).

[8]Federal Information Processing Standard

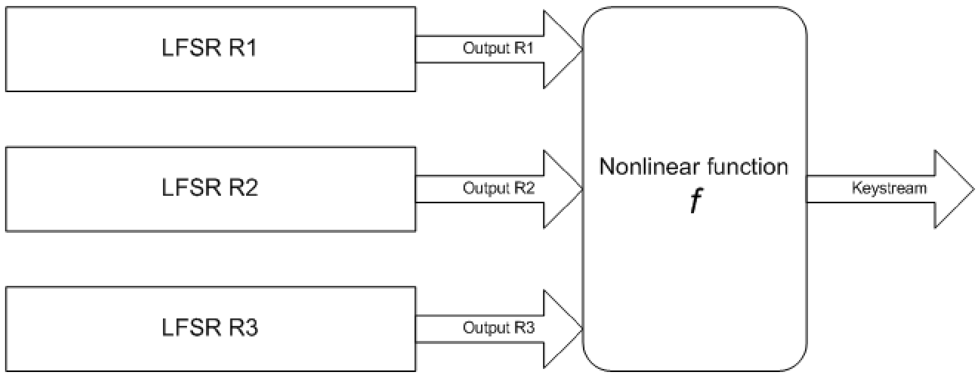

Figure 1: Non-linear Combination Generator.

- Synchronous stream ciphers have no error propagation.

Basically there is a belief that stream ciphers offer advantages over block ciphers in situations when low power consumption is required, low hardware copmplexity is required or when extreme software efficiency is needed [60]. Most stream ciphers are based on simple devices that are easy to implement and run efficiently. A common example of such a device is the linear feedback shift register (LFSR) [61]. Such simple devices produce predictable output given some previous output. Thus, the output of such devices is typically used as the input to a function that produces the keystream (an LFSR cannot be directly used as a keystream generator since it is totally linear). Keystreams can also be produced by using certain modes of operation of a block cipher.

There is no dedicated standard for stream ciphers such as the AES standard for block ciphers (see section 2.2.2). One probable reason is that most stream ciphers in use are either secret or proprietary designs [1].

2.2.4.1 Designs Priciples

Stream Ciphers Based on Feedback Shift Registers

Linear feedback shift registers are widely used as building blocks for stream ciphers. If designed properly, LFSRs have large periods and good statistical properties, and there exist mathematical techniques to analyse them. Since the output from a LFSR is linear, the output is easily predictable. So when using LFSRs as elements in a stream cipher, it is very important that the output sequence from the cipher does not inherit linearity properties from the output produced by the linear LFSRs. Beneath follow descriptions of three common techniques used to achieve non-linearity in stream ciphers built up with one or several LFSRs.

Non-linear Combination Generators: A non-linear combination generator uses several LFSRs. The output of these LFSRs are taken as input to a non-linear function f, which then produces a non-linear keystream.

Figure 1 illustrates this principle.

Non-linear Filter Generators: Non-linear filter generators use the stages of a single LFSR as input to a non-linear function f, which then produces the final keystream. A non-linear filter can also be applied at the end of any cipher to give the output keystream even higher non-linearity. Figure 2 illustrates a basic non-linear filter

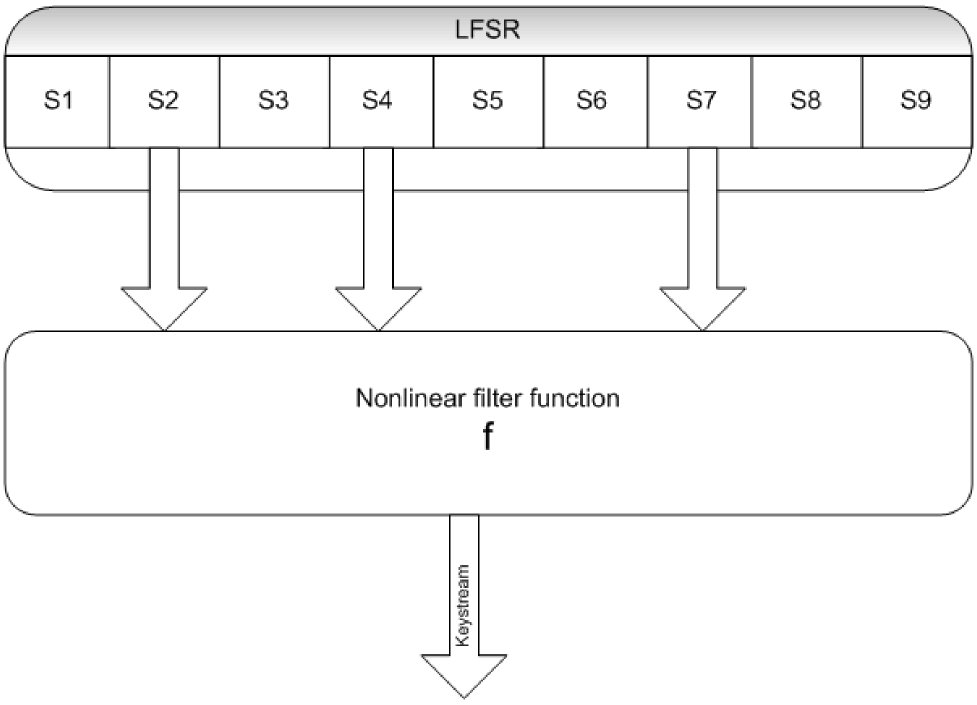

Figure 2: Non-linear Filter Generator.

generator.

Irregularly Clocked Stream Ciphers: Most irregularly clocked stream ciphers include a combiner from which the output is decimated in some way. A combiner consists of a linear part (often one or several LFSRs) and a Boolean function (typically a nonlinear Boolean function). To create the keystream, some positions are taken from the internal state of the linear part and fed into the Boolean function. It is basicly the Boolean function which determines which produced keystream bits are used, and which bits are discarded. The output of the Boolean function is then combined with the message by a ciphering transformation, typically the XOR operation. An irregularly clocked stream cipher is often designed with one or several LFSRs which control the keystream data generator. The keystream data generator is often another LFSR which is irregularly clocked based on the clocking LFSRs and some Boolean function. The output from the data LFSR will then be non-linear because of the irregular clocking. Figure 3 shows a basic irregularly clocked generator.

Stream Ciphers Based on Block Ciphers

Block ciphers can be run in some modes of operation, which make the ciphers produce a keystrem sequence. The most common modes are the Output Feedback mode (OFB), the Cipher Feedback mode (CFB) and the Counter mode (CTR). Since stream cipers based on block ciphers have the underlying structure of a block cipher, these can potentially be attacked by cryptanalysis of the underlying block cipher. For block ciphers in either OFB or CTR mode there exist generic distinguishing attacks. Given a block cipher with the blocksize b, $2^{b/2}$ blocks of keystream are sufficient to distinguish the keystream from a truly random sequence. This is achieved by looking for repeated occurrences of blocks,

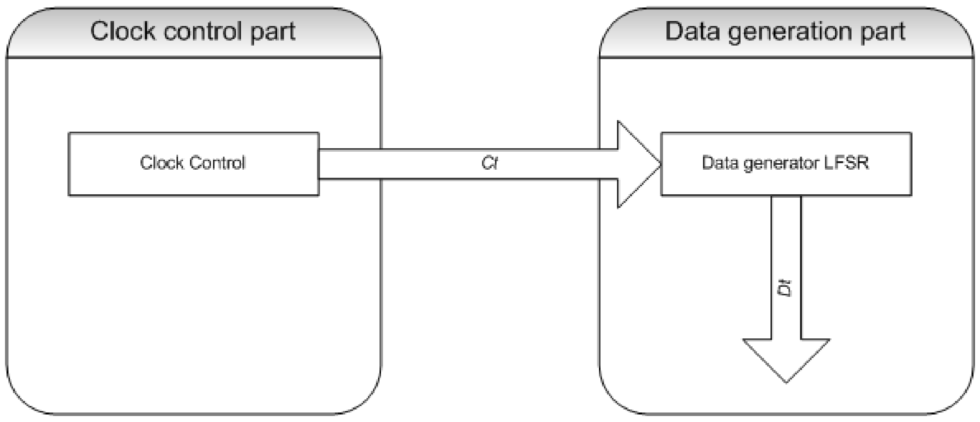

Figure 3: Irregular Clocking.

which are not possible when the keystream is generated by a block cipher in CFB mode, unless the sequence starts repeating itself.

Modular Arithmetic Generators

This class of generators is based on the presumed intractability of an underlying number theoretic problem. The RSA generator and the Blum-Blum-Shub generator [62] are examples of such generators. Since this class of generators is based on modular arithmetic, which is very resource-demanding, this kind of generators is extremely slow compared to other generators. Thus this kind of generators is primarily used as pseudorandom number oracles, not as building blocks for encryption algorithms.

2.2.4.2 Attacks Against Stream Ciphers

When considering attacks on stream ciphers, known plaintext attack is the most common. In other words we assume that a large amount of the keystream is known for a potential adversary. The statistical deviations from the keystreams are then exlploited in one of three ways as an attack on the stream cipher; Distinguishing Attacks, Prediction and Key Recovery attacks.

Distinguishing Attack A distinguishing attack is simply a method for distinguishing output from the keystream generator (the cipher) from a random bit-sequence of the same length. If we are able to do that, we can for instance find out which generator is used, and a given non-random pattern in the keystream generator is proven.

Prediction Attack Prediction Attacks find a method to predict some output from the keystream generator in a more accurate way then by guessing. To be able to predict some output from the cipher, will lead to less complexity of recovering the original plaintext.

Key Recovery Attack If someone has successfully mounted a key recovery attack, that someone has also enabled both a Distinguishing attack and a prediction attack. A key Recovery attack is the far most effective attack if succeeded. If the key used to generate the keystream is recovered, all the future and past ciphertexts generated with this key can easily be found. Thus frequent rekeying or key reinitialization is needed in most common designs and uses of stream ciphers.

Above, we have classified different types of attacks. We will now give a brief overview of the most used attack techniques against stream ciphers that we know today.

Exhaustive key search Being the most genearal attack, this attack technique can be applied to any stream cipher. The idea behind this attack is, given a keystream generated by an unknown key, an attacker simply tries all possible keys and compares the newly generated keystreams against the existing one. If he gets a match, he has recovered the originally used key. Exhaustive key search can be applied against all symmetric encryption algorithms, and is often reffered to as *brute force attack*. For stream ciphers there exist very efficient techniques to conduct an exhaustive key search attack, namely the time-memory tradeoff techniques [63]. The time complexity for exhaustive key search is split into a time and a memory complexity, and the name "time-memory tradeoff" results from this idea.

Periodic and Statistical Attacks If the period of a keystream generator is smaller than the amount of data which is to be encrypted (the keystream starts to repeat itself before the encryption process is finished), a prediction attack would be easy to mount. Thus, large period is essential in any keystream generator.

Exploiting Linear Complexity The linear complexity is the length of the shortest LFSR that can produce a certain sequence. If the linear complexity is too small, then an attacker can reproduce the sequence of an LFSR. The main approach to achieve this, is to make use of the Berlekamp and Massey algorithm described in [64].

Correlation Attacks Correlation attacks are the most general attack on LFSR based stream ciphers, and this kind of attacks are important to be aware of. In a correlation attack, the output from a keystream generator is correlated in some manner with the output from a much simpler device, such as a simple component LFSR of the complete keystream generator. This correlation can be used in prediction attack, and even sometimes be exploited to determine the key used for the specific keystream analysed. The first ideas of conducting a correlation attack were described by Thomas Siegenthaler in [65]. Later other authors have improved these ideas by developing *Fast Correlation Attacks* [66]. In a fast correlation attack one first tries to find a low weight parity check polynomial of the LFSR, then some iterative decoding procedures are applied.

Higher Order Correlation Attacks As many stream ciphers are built up with some linear sequence generator (often LFSRs) and some non-linear output function f to generate the keystream, correlation attacks try to find a linear approximation of the function f. A higher order correlation attack [67] tries to calculate an equivalent higher order approximation of the function f.

Divide and Conquer Attacks In Divide and Conquer Attacks a portion of the key, or sometimes the internal state of the cipher is guessed. Now the remaining bits of the the key might be found with less time complexity than with exhaustive key search.

Rekeying Attacks In many applications which make use of stream ciphers in some way, the cipher is frequently rekeyed. If someone manages to exploit this rekeying schedule to find the key, a Rekeying attack has been successfully mounted.

Side channel Attacks If for example running different primitives or different inner states in a cipher causes a higher power consumption of the cipher, this can be used to attack the cipher. If the cipher causes external elements not connected directly to the cipher to behave in some special or predictable way, this can be exploited in a side channel attack.

2.2.5 Some Publicly Known Stream Ciphers

2.2.5.1 SNOW

SNOW [68] is a synchronous stream cipher. It uses a 128-bit or a 256-bit key, and has an internal memory of 576 bits. SNOW consists of an LFSR of length 16, and a Finite State Machine. The state bits of the generator are words in $GF(2^{32})$. The LFSR used in SNOW is defined by the recurrence relation

$$s_{t+16} = \alpha(s_t \oplus s_{t+3} \oplus s_{t+9})$$

where $\alpha \in GF(2^{32}), \oplus$ is addition in $GF(2^{32})$, and $+$ is addition modulo 2^{32}, and s_t is the generated state output from the recurrence.

The Finite State Machine consists of two registers whose values at time t will be denoted by a_t, b_t as follows:

$$a_{t+1} = a_t \oplus R(\int_t + b_t)$$

$$b_{t+1} = S(a_t)$$

$$f_t = (s_{t+15} + a_t) \oplus b_t$$

$$z_t = f_t \oplus s_t$$

where R denotes a 7-bit left rotation, and S is a 32-bit to 32-bit S-box. The sequence z_t is then used as the keystream.

There exist two attacks on SNOW in particular, one distinguishing attack [69], and one guess and determine attack [70]. The distinguishing attack on SNOW requires 2^{95} observed bits of keystream, and a workload about 2^{100}. The guess and determine attack however, with the first approach needed 2^{64} observed bits of keystream and a workload 2^{256}, which is no better than exhaustive key search. With some assumptions and modifications, the second approach to attack the cipher required 2^{224} observed keystream bits, and a workload 2^{95}.

Due to this attacks on SNOW, the authors released a new version of the cipher in 2002 called SNOW 2.0 [71], as an improvement to the original design.

2.2.5.2 SOBER-t16/t32

The SOBER family of stream ciphers [72] are synchronous ciphers. The SOBER-t16 uses a 128 bit key for encryption, while the SOBER-t32 uses a 256 bit key for encryption. The SOBER-t16 has an internal memory of 272 bits, while the SOBER-t32 has an internal memory of 544 bits. Both versions of the cipher use an LFSR of length 17, the t16 version uses the LFSR over the Galois field $GF(2^{16})$, the t32 version over the Galois field $GF(2^{32})$.

For SOBER-t16, the elements of the Galois Field are represented by 16-bit binary vectors corresponding to polynomials modulo the irreducible polynomial

$$x^{16} + x^{14} + x^7 + x^6 + x^4 + x^2 + x + 1.$$

For SOBER-t32, the elements of the Galois Field are represented by 32-bit binary vectors. The irreducible polynomial for SOBER-t32 is

$$x^{32} + (x^{24} + x^{16} + x^8 + 1)(x^6 + x^5 + x^2 + 1).$$

If we denote $\oplus$ as addition in GF(2^{16}), and $+$ as addition modulo 2^{16}, the recurrence relation from the LFSR in SOBER-t16 can be written as

$$s_{t+17} = \alpha s_{t+15} \oplus s_{t+4} \oplus \beta s_t$$

where $\alpha = 0xE382$ and $\beta = 0x67C3$.

If we denote $\oplus$ as addition in GF(2^{32}), and $+$ as addition modulo 2^{32}, the recurrence relation from the LFSR in SOBER-t32 can be written as

$$s_{t+17} = s_{t+15} \oplus s_{t+4} \oplus \alpha s_t$$

where $\alpha = 0xC2DB2AA3$.

Both the SOBER-t16 and SOBER-t32 cipher uses a non-linear filter (see Section 2.2.4.1) and a stuttering procedure together with the LFSR to produce the keystream.

In [73] a distinguishing attack on both a simplified version without stuttering, and one on the complete cipher SOBER-t16 is described. The attack on the simplified version (without stuttering) needs 2^{92} keystream words to distinguish between keystream from the generator and a complete random sequence with a probability of error 2^{-32}. This means that the computational complexity of this attack is 2^{92}. The attack on the complete cipher scheme requires 2^{111} keystream words to distinguish the keystream from a complete random sequence with probability of error 2^{-32}. The computational complexity of this attack is 2^{111}.

[73] also describes a distinguishing attack on a version of SOBER-t32 without the stuttering. This attack needs in the worst case scenario $2^{86.5}$ keystream words to distinguish the keystream from a totally random sequence with the probability of error 2^{-32}. This leads to a computational complexity of $2^{86.5}$ for this attack. In [74] Preneel *et al.* enhanced the attack described in [73] and applied it on the complete SOBER-t32 cipher. The computational complexity for this distinguishing attack was 2^{153}.

Beside the attacks mentioned above, both SOBER-t16 and SOBER-t32 are vulnerable to timing attacks and power attack due to their irregular decimation[9] [72]. Guess-and-determine attacks has also been applied on the unstuttered version of SOBER-t32, the best of this attack had a computational complexity of 2^{244}.

2.2.5.3 RC4

RC4 was developed in 1987, and is the most widely used stream cipher in software applications. It is used to protect Internet traffic in the SSL (Secure Sockets Layer) protocol, it is integrated into Microsoft Windows, Lotus Notes, Apple AOCE, Oracle Secure SQL, and many other software applications. It was also chosen to be part of the Cellular Digital Packet Data specification. RC4 implementations in software are extremely compact and efficient, and its design was kept a trade secret until 1994, when it was reverse engineered and anonymously posted to the Cypherpunks mailing list[10].

RC4 has a secret internal state which is a permutation S of all the $N = 2^n$ possible n-bit values, where n is typically chosen as 8. The initial state is derived from a key,

[9]Irregular Clocking
[10]`https://www.cypherpunks.to/list/`, last visited 4/6 05

typically in the range of 40 to 256 bits long, by a Key-Scheduling Algorithm (KSA). A pseudo-random generator algorithm (PRGA) then alternately modifies the state by exchanging two out of the N values and produces an output by picking one of the N values in S. For a standard $n = 8$, this gives RC4 a huge state of about 1700 bits. A more detailed description of RC4 is to be found in [75].

Since RC4 is widely used, a lot of cryptanalysis and attacks have been applied on the cipher during the last two decades. Analysis of the PRGA in RC4 has proven no great security weaknesses in the algorithm. For $n = 8$ and sufficiently long keys, the best known attack has a time complexity larger than 2^{700} time to find its initial state. However, many interesting properties of RC4 were found over the years. In [76] a major bias in the distribution of RC4's second output word were found. The word is zero with twice the expected probability of $1/N$, and thus RC4 outputs can be distinguished from random strings by analyzing only about 2^8 words of output produced by unrelated and unknown keys. The Key-Scheduling Algorithm in RC4 is proven to have severe weaknesses [77].

Although RC4 is a widely used, and good algorithm, care must be taken when applying it in practice. As an example, RC4 is completely insecure in a natural mode of operation which is used in the widely developed Wired Equivalent Privacy protocol (WEP, which is part of the 802.11b Wi-Fi standard) [78]. The RC4 based WEP and WEP2 protocols are considered to be broken. In a busy network, an entire 128 bit WEP key can be derived by passively observing it for a few hours.

2.2.5.4 LILI-128

LILI-128 [79] is a cipher developed from the LILI family of ciphers [80]. LILI-128 is a synchronous cipher and it uses a 128-bit key and an internal memory of 128 bits. LILI-128 can be viewed as a clock-controlled nonlinear filter generator. The cipher consists of two components, one used for clock control and one used for data generation. Each of the two components consists of an LFSR ($LFSR_c$ in the clock control component, and $LFSR_d$ in the data generation part), and a function f ($f_c and f_d$) which taps the LFSRs.

During the key schedule of LILI-128, the 128 key bits are loaded directly into the LFSRs. The first 39 bits are loaded into $LFSR_c$, and the last 89 bits are loaded into $LFSR_d$. Neither $LFSR_c$ nor $LFSR_d$ may be zero.

$LFSR_c$ in the clock control component is regularly clocked and has the length of 39. The feedback polynomial of this LFSR is

$$x^{39} + x^{35} + x^{33} + x^{31} + x^{17} + x^{15} + x^{14} + x^2 + 1$$

. This polynomial produces a maximum length sequence. Each time the $LFSR_c$ is clocked once, the function f_c takes the contents of stages 12 and 20 as input, and produces an output integer by

$$c = f_c(x_{12}, x_{20}) = 2x_{12} + x_{20} + 1.$$

$LFSR_d$ of length 89 in the data generation component also produces a maximum length sequence. Its feedback polynomial is

$$x^{89} + x^{83} + x^{80} + x^{55} + x^{53} + x^{42} + x^{39} + x + 1.$$

After the clock control component has produced the integer c, $LFSR_d$ is clocked c times. The nonlinear function f_d is defined by a truth table and takes the content of 10 stages of the LFSR_d as input and calculates an output bit z. This output bit is then used as a new keystream bit.

Many *time-memory tradeoff attacks* have proved that the 128-bit key for LILI-128 can be recovered faster than with exhaustive key search [81] [82]. A *fast correlation attack* [83] has proved that the key can be recovered with a computational complexity around 2^{71}. This attack assumes a received bit sequence of length around 2^{30} bits and a precomputation phase of complexity 2^{79} table lookups. Due to this successful attacks, LILI-128 was not selected for further study in the NESSIE project [1].

A new version of LILI-128, LILI-II [84], was later introduced due to the security findings in LILI-128. One of the most successful attack on this cipher is presentated in [60].

2.2.6 Encrypted File Systems

Another way to use encryption for protecting sensitive data and confidentiality is to have the file system handle the encryption. Different encrypted file systems such as those evaluated in [85] encrypts data on a lower level than the application or user level. Beneath follows a overview of the encrypted file system found in newer Windows environments.

2.2.6.1 Windows NTFS/EFS Encryption

With the release of Windows 2000, Microsoft announced their Encrypting File System(EFS). EFS security relies on Windows 2000 cryptography support, which Microsoft introduced in NT 4.0. This support is referred to as the CryptoAPI. EFS cooperates with the NTFS file system to provide encryption. By building encryption into the Operating System, Microsoft can make the encryption and decryption process transparent to both applications and users, which provides easy usability for the encryption tool.

Briefly, the encryption process is as follows: A 128 bit random number is generated and used as the File Encryption Key (FEK) to encrypt a file or an entire catalog structure. 56 bit of the FEK is used as input to a symmetric encryption algorithm (in US versions of Win2k the entire 128 bit FEK is used), DESX [43], which is an improved version of the DES algorithm regarding Exhaustive Key Search attacks. FEK is then stored together with the symmetric encrypted file(s), encrypted with a 1024 bit RSA public key assigned to the user profile. The private key is stored on the computer's hard drive by default (later versions of Windows are claimed to have features to export the private key to an external storage device, such as a smartcard). When stored on the hard drive, the private key is stored in "Windows Protected Store", which is specific to the currently logged on user. When a user is to access an encrypted file, the private key of the user is obtained, and the OS automatically decrypts the files without involving the logged on user.

The security of this system is questionable. One can never both encrypt and compress files, one attribute must be chosen, and encrypting system files is not possible or allowed, which can be exploited in several ways as shown in [86]. Another problem with this encryption process is that when a file is encrypted, a copy of the file is created which is then encrypted, the original file is still on the hard drive in a non encrypted form. Using disk editor tools like a DOS boot disk (or tools integrated in Windows NT) it can be easily accessed and read.

3 The Custom Software Cipher

3.1 Design

3.1.1 General Overview of the Design

The new custom cipher is built up with two primitives, each consisting of two different linear feedback shift registers (LFSRs) and two Boolean functions. Together, the primitives generate two 32 bit tuples, which are then combined in a combination algorithm, and a 32 bit tuple is finally generated as the key stream for one iteration of the key stream generator. Figure 4 shows the general design principles.

3.1.2 The Primitives

Both primitives in the design are based on two LFSRs and different lengths and feedback polynomials. Each primitive is divided into two parts, a clock control part and a data generation part. One of the LFSRs in each primitive works as a clock control register in the clock control part of the primitive; the other LFSR is found in the data generation part. The output from the clock control part at a given time Ct determines how many times we should shift the LFSR in the data generation part before we produce a new output Dt from the primitive. This principle is often referred to as irregular clocking (See figure 3 in Section 2.2.4.1).

All the feedback polynomials used in the cipher are irreducible. This is to ensure high periods in the LFSRs. The polynomials where calculated using a program specially designed for generating such feedback polynomials. The election of the four polynomials used in the cipher design was done based on their characteristics, to avoid bad feedback polynomials as described in [87].

The LFSR R1 in the clock control part of primitve 1 is of length 107, and contains 32-bit tuples in each state. The feedback polynomial for this LFSR is

$$x^{107} + x^{92} + x^{62} + x^{27} + 1.$$

This gives the LFSR the maximum period $2^{107} - 1$.

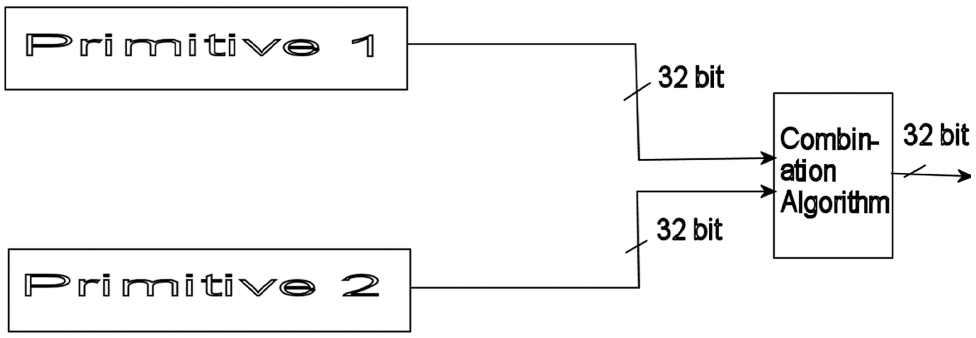

Figure 4: General Overview of the Cipher.

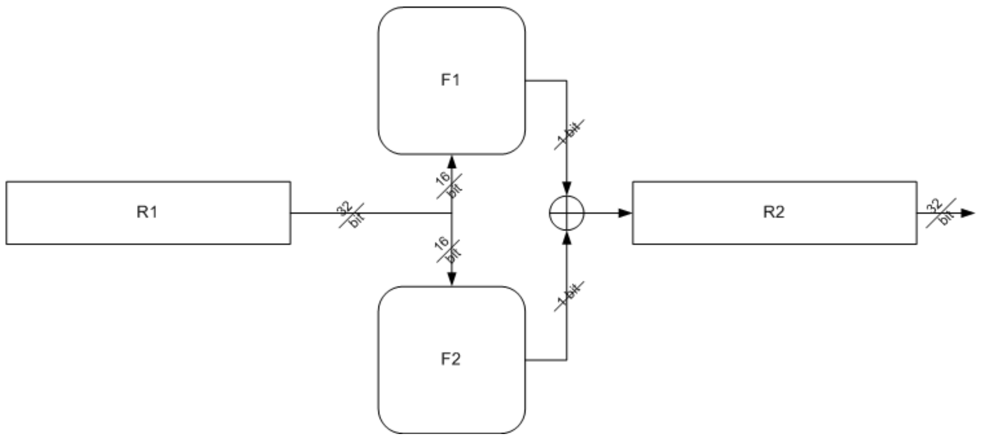

Figure 5: General Overview of one primitive.

The LFSR R2 in the data generation part of primitve 1 is of length 127, and contains 32-bit tuples in each state. The feedback polynomial for this LFSR is

$$x^{127} + x^{62} + x^{59} + x^{15} + 1.$$

This gives the LFSR the maximum period $2^{127} - 1$.

The LFSR R1 in the clock control part of primitve 2 is of length 103, and contains 32-bit tuples in each state. The feedback polynomial for this LFSR is

$$x^{103} + x^{66} + x^{43} + x^{34} + 1.$$

This gives the LFSR the maximum period $2^{103} - 1$.

The LFSR R2 in the data generation part of primitve 2 is of length 149, and contains 32-bit tuples in each state. The feedback polynomial for this LFSR is

$$x^{149} + x^{114} + x^{64} + x^{29} + 1.$$

This gives the LFSR the maximum period $2^{149} - 1$.

The functions $f1$ and $f2$ are in fact two balanced truth tables. These tables both contain 2^{16} elements. When accessing the truth table, the 16 most significant bits of the 32-bit output tupple is used as input for $f1$, and the 16 least significat bits as input to $f2$. The results from these two functions is then accessed with the XOR operator. If the result of the XOR operation is 1, the data generation LFSR is shifted one additional time before the output is used further in the algorithm.

Figure 5 shows the general primitive design.

3.1.3 Key expansion and Generator Initialization Algorithm

The cipher takes a 256 bit secret key as input and combines this with a 64 bit message key (which is publicly known) to initialize the state of the linear feedback shift registers. The message key is different for each encryption. The main purpose of the message key is to ensure that the same key never is used twice for a single encryption.

When the cipher is initialized, every bit of the secret key is bitwise XOR'ed with the message key, to generate 64 32 bit integer tupples. The initialisation of the LFSR states is done in the following way: First, the 256 bit secret key is converted into 8 32 bit integer tuples, and the secret key is converted into 2 32 bit tuples. For the first 32 bit secret key tuple, the tuple is bitwise XOR'ed with the first message key tuple. The second 32 bit secret key tuple is rotated once to the right, then bitwise XOR'ed with the first 32 bit message key. The third secret key tuple is rotated twice to the right, and so on. Then the same procedure is used together with the second message key tuple. Then the LFSRs are filled with these 64 32 bit tuples in a serialized way, meaning that when all the 32 bit tupples are used once, the first tupple is then again used to fill the next LFSR position which is to be initialized, and so on.

3.1.4 Re-keying Policies and Maintenance of the Cipher

The secret key should be changed once a month for safety reasons. Namely, there is a possibility for the output sequence to be repeated. In order to reduce the probability of this event, the secret key should be changed regulary (See for example [13]).

The message key is changed for each encryption.

3.2 Cipher Analysis and Results

3.2.1 Algebraic Analysis

An algebraic analysis over a cipher generator is usually done by creating a set of equations based on the secret input key and the output from the generator. The goal is to get a set of equations, which is as nonlinear as possible. A non-linear equation in the form $x_1, x_2, ..., x_l$ can be written mathematically as

$$\sum_{i=1}^{n} \left(\prod_{j=1}^{l} x_j^{c(i,j)} \right) = c$$

where $c \in \{0, 1\}$ is a constant and $c(i, j) \in \{0, 1\}$.

In other words the equations retrieved should consist of many high order products to document high non-linearity.

In principle, this process of generating nonlinear equations could be used to describe each output bit as a non-linear combination of the generator's seed. There are however two major problems with this approach. First, a nonlinear equation produced can at maximum be of the degree $d = l$, which in the worst case then can consist of 2^l monomials in one single equation. Working with nonlinear equations can only be done efficiently when handling not too many monomials in each equation, thus it is an almost impossible task to perform.

Secondly, solving systems of nonlinear equations is known to be NP-complete, which is a part of the NP-hard problem [88].

Instead of generating the equations, which would be impossible in our scenario, we try to document security by looking at the probability of high linear complexity in different parts of the cipher generator design. The output from a single linear feedback shift register is totally linear. When the period of the register is run through once (maximum $2^l - 1$ iterations, see [89]), the generator will cycle again and produce the same output. The Hamming distance between two identical outputs from the register will always be $2^l - 1$ (if the register is of maximum period). Because a single LFSR has total linear dependency on the output bits from its own state bits, high period LFSRs are not good as keystream generator by themselves, but can work as very good building blocks for a pseudo random keystream generator.

If we look at Figure 5, we can see that the output bits from the functions $F1, F2$ depend non-linearly on the state bits of the LFSR $R1$. This is because of the high non-linearity of the functions. Both functions are balanced and take two separate 16 bit inputs from the LFSR $R1$ to generate the output bits from both functions. Since the functions are balanced, the probability that the function will return the value 0 is 0.5.

The output sequence of the LFSR *R2* depends non-linearly on the LFSR *R1* state bits. This would even be the case if we did not have the functions $F1$ and $F2$. To introduce the functions adds even one more source of non-linearity. Another interesting property of the output from the LFSR $R2$ is that it does not depend linearly on its own state bits. This is because of the non-uniform decimation, or the irregular clocking of the register.

3.2.2 Period and Linear Complexity

A pseudorandom generator or a complete stream cipher scheme is a finite state machine, which can consist of at most 2^l inner states. A direct consequence of this is that the generator can give a maximum output of 2^l bits before it cycles (starts repeating the same sequence). This again might mean that the least significant output bits from the generator can be used to produce a recurrence relation to model the most significant bits of the output stream. If we can find a recurrence relation $x_i = R(x_{i-1}, ..., x_{i-k})$, where k is the length of R to describe the keystream, and we have at least k consecutive bits of output from the generator, we can predict and generate the entire keystream easily. Two particular recurrences are important to identify regarding the security of the cipher.

The *period* of the generator must be very long [61]. If we have an infinite bitstream from the generator $z = (z_0, z_1, ...)$, and we have two values $\rho, \theta \in N$ such that $z_i = z_{i+\rho}$ for all $i \geq \theta$ the sequence is said to be ρ-periodic with a pre-period of θ. Since the generator can have at most 2^l inner states, it holds that $\theta + \rho \leq 2^l$. Since for all $i \geq \theta + \rho$, an attacker can use the recurrence $z_i = z_{i+\rho}$ to predict additional bits of the keystream, it is paramount that no more than $\theta + \rho$ bits of keystream are generated with the same key.

In both primitives which build up our generator we have two LFSRs with high periods. Because of the irregular clocking, the period from each primitive will be much higher than the period of the LFSR $R2$.

$$Per_{P1} \gg \left(2^{l_{R2}} - 1\right)$$

The period of the entire generator will be higher than the highest period of the two primitives. This is because of the combination function in the end of the scheme which is a sum modulo 2^{32} [61]. In some cases, the periodic part of the bitstream sequence z can be described by a linear recurrence relation R such that $k < \rho$. The length k of the shortest linear recurrence is denoted as *linear complexity* or *linear equivalence* $LC(z)$. In other words the linear complexity is the length of the smallest LFSR which generates the bitstream sequence z. The linear complexity of a generator is then $LC(z) \leq \rho$, since the period ρ of the generator is also a linear recurrence.

Berlekamp and Massey have designed an effective algorithm [64], which constructs the shortest linear recurrence describing z. This algorithm only needs $2 * LC$ keystream bits and takes $O(LC^2)$ computational steps to generate the linear recurrence. Thus, we need high linear complexity when designing a keystream generator.

In our design, the linear complexity in both primitives will be very high. Since the primitives are irregulary clocked and the output from $R1$ is sent into two highly non-linear functions, we can state that the linear complexity is $LC \gg l_{R2}$. Again due to the properties of the sum modulo 2^{32}, where in fact only the bit with weight 0 is linearly dependent on the input bits of weight 0 because of the carry bit, we can claim that the linear complexity of the entire generator is greater than the primitive with the highest linear complexity.

$$LC > MAX(LC_{P1}, LC_{P2})$$

3.2.3 Correlation Analysis

When a cipher is correlation immune, it means that changing a few bits in an earlier state of the cipher should not make the cipher behave in any foreseen way and produce predictable output.

The most interesting part of the cipher to analyze for correlation dependence is the output function, sum modulo 2^{32} (the combination algorithm in Figure 4). The sum modulo 2^{32} function has much of the properties as a latin square [90], and is known to be a very correlation immune function. A latin square has the property that if all possible numbers generated from the function are plotted into a matrix, each row and each column in the matrix will only contain one instance of the same value. If the rows and columns in the matrix are mixed, the structure of the latin square will remain intact, i.e. each row and column will still be unique. Table 1 and Table 2 illustrate this property for $\oplus_n$, where $n = 4$.

$\oplus_n$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Table 1: Original Latin Square Matrix.

$\oplus_n$	0	1	2	3
0	2	0	3	1
1	1	3	2	0
2	0	2	1	3
3	3	1	0	2

Table 2: The Latin Square Matrix, Rows 1 and 3 are mixed, as well as Columns 2 and 3.

If we take the example further and convert the numbers to binary form, we can see that n bits of input to the latin square result in one bit of output for two functions $f1$ and $f2$. In other words, if we use the row and column numbers as a reference or index to an element in the latin square, we need both the column number and the row number to find the values for $f1$ and $f2$, i.e, there is no correlation between only the column number and a value in the latin square, nor a correlation between a row number and a value in the latin square. This property is illustrated in Table 3, where all possible combinations of rows and columns are shown for $n = 4$.

As an example, we can read that the row-column value 1101 results in $f1 = 0$ and $f2 = 1$ for this particular mix of columns and rows in the latin square. If only the row index value, 11, were known, we would not be able to tell anything about $f1$ or $f2$. Only knowing the column index value, 01, would neither tell anything about $f1$ or $f2$.

Row	Col	f1	f2
00	00	1	0
00	01	0	0
00	10	1	1
00	11	0	1
01	00	0	1
01	01	1	1
01	10	1	0
01	11	0	0
10	00	0	0
10	01	1	0
10	10	0	1
10	11	1	1
11	00	1	1
11	01	0	1
11	10	0	0
11	11	1	0

Table 3: Values in the latin square (f1, f2) when row index and column index are used as identifiers.

As we can conclude from the example, the sum modulo 2^{32} is a highly correlation immune function, which will lead to high correlation immunity for the custom cipher, and making it harder to perform correlation attacks on it.

3.2.4 Analysis of Keys

The message key should be unique for every encryption produced by this cipher design. The message key has a length of 64 bits. The complexity of the key is 2^{64}, meaning that we can have at most 2^{64} different keys before repeating the same key. Since the message key bits are used to initialize the generator, the key cannot be only zeroes. The message key is publicly known, so we have the possibility to have an algorithm implemented in the cipher which iterates through all the good values of the possible 2^{64} keys. Another possiblity might be to use a 64 bit cryptographic hash algorithm over the message which is to be encrypted as the message key. This however enables the birthday paradox. Thus, the same message key will be repeated after $2^{64/2}$ in average.

Making only small changes in the key or keys which the stream cipher takes as input to generate the keystream, should make the final output from the generator to be totally different than from the output generated from the almost identical key. There should not be any correlation between the keystream produced by the original key and the keystream produced by the altered but almost identical key. The output should also be balanced for both keys. This means that there should be almost identical numbers of zeroes and ones in both outputs. The ratio between zeroes and ones should also be balanced, meaning that the ratio should be about 0.5. To make a small indication of how well the custom cipher has these properties, a small experiment was conducted. In this experiment we produced 32000 bits of output from the generator using one specific message key. Then we altered only the least significant bit in the message key, and produced 32000 new bits of output. For the first message key we got 16065 zeroes and 15935 ones as output. The ratio between zeroes and ones is about 0.5 and is balanced, which indicates a good result. For the second message key we got 16145 zeroes and 15855 ones as output. The ratio between zeroes and ones is also for this key about 0.5

and is balanced, which indicates another good result. No correlation between the two results can easily be found, and the output sequence appears completely different from each other. Conclusion; the micro experiment conducted, indicates that the properties tested for the cipher seem correct. A complete statistical test was applied to the cipher, and the results are later presented in Chaper 4.

3.3 Implementation in C#

The implementation of the cipher simulation was made in C#. The language was chosen because it is a new and popular language, and the efficiency for handling different variable types and structures used to implement the LFSR and the logical functions seems to be about the same for both Java, C and C# [91].

The implementation is made with the consern to be efficient in software. Guidelines for fast implementation found in [92] were attempted to be followed.

The implementation in C# is just a simulation of the cipher. The cipher should be coded in asambler for optimal efficiency.

Appendix A shows some outlines of the source code, to give an idea of how the implementation was made.

4 Statistical Testing

4.1 Introduction

A truly random bit sequence should have the same properties as something which always has one out of two outcomes, and the probability is equally realistic for both outcomes, like for example a coin toss. Here the probability of getting heads or tail is exactly 0.5, and if we toss the coin a thousand times, it should result in about five hundred head tosses, and five hundred tail tosses. The output from a pseudo random generator should appear truly random. Thus the output should be about half zeros and half ones, hence a hypothetical output of an idealized generator of a truly random sequence can serve as a benchmark for the evaluation of a pseudorandom generator. This method of benchmarking pseudorandom generators is used in NIST's statistical test suite [93], and is a good tool to perform the statistical analysis of the new stream cipher, since the cipher generator should work as a pseudorandom generator.

As the output from a pseudorandom generator should appear totally random, the output from the generator should also be unpredictable if the seed[1] is unknown. The next output number from the output sequence should be unpredictable in spite of any knowledge of the previous random numbers in the sequence. This property is known as *forward unpredictability*. It should also be infeasible to determine the seed from knowledge of any previously generated numbers, i.e. *backward unpredictability* is also required for pseudorandom generators. There should be no correlation between the seed and any value generatad from that seed.

4.2 Testing Procedure

Randomness is a probabilistic property, meaning that the properties of a random sequence can be characterized and described in terms of probability.

There are an infinite number of possible statistical tests, each assessing the presence or absence of a pattern, which, if detected, would indicate that the sequence is nonrandom. Because there are so many tests for judging whether a sequence is random or not, there will never be a finite set or a complete set of tests to determine wheter a sequence appears truly random or not. Care must also be taken to give an absolute conclusion whether a sequence appears truly random or not based on statistical testing [93].

A statistical test is always formulated to test a specific *null hypothesis* (H_0). The null hypotesis is what we want to prove, and in this scenario, where we want to test a stream cipher generator, we want the sequence being tested to appear random. Associated with the null hypothesis is always an *alternative hypothesis* (H_1)[2]. When testing the custom stream cipher, the alternative hypothesis is that she tested sequence is not random. For each test run against the custom cipher, a decicion or conclusion is derived that accepts or rejects the null hypothesis, i.e., whether the stream cipher generator is producing random values or not.

[1]In the case of the new cipher, the seed is the key given for the specific encryption.

[2]H_1 is sometimes refered to as H_a in literature.

For each test conducted, a relevant randomness statistic must be chosen to determine whether or not to accept or reject the null hypothesis. If we assume randomness, such a statistic has a distribution of possible values. A mathematical model can be used to generate a theoretical reference of this statistical distribution under the null hypothesis. From this theoretical reference distribution, a *critical value* is determined. During a test, a test statistic value is computed from the sequence being tested. Then the test statistic value is compared to the critical value. If the test statistic value exceeds the critical value, the null hypothesis for randomness is rejected, and the H_1 hypothesis is accepted. Otherwise, the null hypothesis is accepted, and the H_1 hypothesis is rejected.

Statistical hypothesis testing is a conclusion-generation procedure that always has one out of two possible outcomes, either accept H_0 or accept H_1. Based on a truth table a statistical hypothesis testing can have four possible outcomes, see Table 4.

Real Situation	Conclusion: Accept H_0	Conclusion: Accept H_1
Data is random (H_0 is true)	No error	Type 1 error
Data is not random (H_1 is true	Type II error	No error

Table 4: Hypothesis truth table

If the tested sequence is, in truth, random, then a conclusion to reject the null hypothesis will occur a small percentage of the time. To make this wrong conclusion is called a Type I error. On the other hand, if the data, in truth, is non-random, then a conclusion to accept the null hypothesis is called a Type II error. The other two possible outcomes from the test are correct conclusions.

The probability of a Type I error is often called the *level of significance* of the statistical test. This probability is most often set prior to a statistical test and is denoted as α. When testing for randomness, α is the probability that the test will indicate that the test sequence is not random when it really is random. A sequence apperars to have non-random properties even when a "good" random generator produced the sequence. Common values of α in cryptography are about 0.01 [93], and we shall use this value for the later statistical testing.

The probability of a Type II error is denoted as β. For a statistical test, β is the probability that the test will indicate that the sequence is random when it is not. A "bad" generator may produce a sequence that apperars to have random properties. Unlike α, β is not a fixed value. Because there are an infinite number of ways that a data stream can be non-random, β can take on equally many values.

One of the primary goals of the statistical testing is to minimize the probability of a Type II error, i.e., to minimize the probability of accepting a sequence being produced by a good generator when the generator was actually bad. Type II errors is most common to commit when it comes to almost all statistical testing, this because we often want the H_0 hypothesis to be correct. This is also the most dangerous mistake to make. When we perform statistical testing on random sequences, the probabilities α and β are related to each other and to the size n of the tested sequence in such a way that if two of them are specified, the third value is automatically determined. In practice, the usual procedure is to select a sample sequence of size n and a value for α. Then a critical point for a given statistic is selected that will produce the smallest β(the probability of a Type II error).

The cutoff point for acceptability is chosen such that the probability of falsely accepting a sequence as random has the smallest possible value.

Each test conducted on the generator is based on a calculated test statistic value, which is a function of the data. If the test statistic value is S and the critical value is t, then the Type I error probability is given by

$$P(S > t|H_0 istrue) = P(rejectH_0|H_0 istrue).$$

The Type II error probability is gived by

$$P(S \leq t|H_0 isfalse) = P(acceptH_0|H_0 isfalse).$$

The test statistic is used to calculate a *P-value* that summarizes the strength of the evidence against the null hypothesis. For these tests, each P-value is the probability that a perfect random number generator would have produced a sequense less random than the sequence that was tested, given the kind of non-randomness assessed by the test. If a P-value for a test is determined to be equal to 1, then the sequence appears to have perfect randomness. A P-value of zero indicates that the sequence appears to be completely non-random. The significance level α can be chosen for each test (Typically in the range $[0.001, 0.01]$). If $P - value \geq \alpha$, then the null hypothesis is accepted, and we conclude the sequence to appear random for the particular test. If $P - value < \alpha$ then the null hypothesis is rejected, and we conclude the sequence to appear non-random.

As an example, if α is set to 0.01, we would expect 1 sequence out of 100 sequences rejected by the test if the sequence was random. A $P - value \geq 0.01$ would mean that the sequence would be concidered to be random with a confidence of 99%. A $P - value < 0.01$ would lead to a conclusion that the sequence is non-random with a confidence of 99%.

There are three assumptions we have to make with respect to the random binary sequences being tested.

- *Uniformity:* At any point in the generation of random or pseudorandom bits, the occurrence of a zero or a 1 is equally likely, i.e., the probability of each is exactly 1/2. The expected number of zeroes (or ones) is $n/2$, where n is the sequence length.
- *Scalability:* Any test applied to a sequence can also be applied to subsequences extracted at random. If a sequence is random, then any such extracted subsequence should also be random. Hence, any extracted subsequence should pass any test for randomness.
- *Consistency:* The behavior of a generator must be consistent across starting values. It is inadequate to test a generator based on the output from a single seed.

4.3 Applied Tests

Beneath follow results obtained when applying different statistical tests from the NIST Special Publication 800-22 [93] on the custom cipher. A more detailed description of the tests can also be found in [93]. Ten bitstreams are generated from the custom stream cipher design, each with a different key as seed input for the generator. The bitstreams are 1,000,000 bits long. The significance level α is set to be 0.01 for all tests if otherwise not specified.

There are some standard functions used with the statistical results to generate the different P-values. The *Complementary Error Function* (denoted as the function erfc in ANSI C math.h library), given by

$$\mathrm{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-u^2} du.$$

The *Gamma Function* (igamc), given by

$$\Gamma(z) = \int_z^\infty t^{z-1} e^{-t} dt.$$

Based on the Gamma Function, is the *Incomplete Gamma Function*. Depending on the values of its parameters α and x, the incomplete gamma function may be approximated using either a continued fraction development or a series development.

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

, where $P(a,0) = 0$ and $P(a,\infty) = 1$.

$$Q(a,x) \equiv 1 - P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

, where $Q(a,0) = 1$ and $Q(a,\infty) = 0$.

Table 5 lists the symbols which are most used when describing the different tests.

Symbol	Description
n	The length of a given bitsequence
ε	The sequence of bits generated by the cipher
M	Block length

Table 5: Notation Symbols.

Below follows results obtained from different statistical tests applied on the custom cipher.

4.3.1 Frequency (Monobit) Test

The basic idea behind this test is to calculate the numbers of ones and zeroes in the bit sequence tested. The distribution should be about the same number of ones and zeroes. The test statistic for this test is given by

$$S_{obs} = \frac{|S_n|}{\sqrt{n}}$$

, where $|S_n|$ is the sum of the bit sequence if the bit 0 is replaced with -1, and n is the sequence length. The P-value is generated by applying the Complementary Error Function

$$P - value = \mathrm{erfc}\left(\frac{S_{obs}}{\sqrt{2}}\right).$$

Table 6 shows the different p-values calculated with the entire 1,000,000 bit bit-streams as input.

Bitstream #	Number of 0s	P-value
1	500457	0.360717
2	500433	0.386490
3	499946	0.913996
4	500358	0.473991
5	500199	0.690630
6	499866	0.788699
7	499959	0.934647
8	500264	0.597499
9	500303	0.544515
10	499678	0.519575

Table 6: Frequency Test, P-values with entire bitstream as input.

As we can see from Table 6, all bitstream P-values are ≥ 0.01, which is the signifcance level α. This indicates that all the ten tested sequences appear to be random, and pass this test.

If we generate 1000 substrings from the bitstreams, each of the length 1000 bits, we can look at the proportion of substring sequences, which pass this particular test. When α is set to 0.01, we assume that 1 out of 100 sequences fails to pass the test. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, results in that the proportion should lie above 0.9805607 for this particular test. Figure 6 shows the proportion value for the substrings. Looking at the figure we can conclude that the sequences tested for every bitstream appears to be random, and passes the frequency test.

The distribution of the p-values should also be uniform for the bitstreams generated. To examine the bitreams for uniformity, we can draw a simple histogram. In this histogram, the interval between 0 and 1 is divided into 10 sub-intervals, and the P-values that lie within each sub-interval are counted and displayed. Figure 7 and 8, shows the distribution of p-values for the bitstreams generated with respectively key 1 and key 7. The streams selected for evaluation were arbritary selected.

As we can conclude from the graphs, the distribution seems quite uniform. However the sub-interval between 0.8 and 0.9 has thee highest count of p-values in both keystreams. It might be interesting to see if this appears to be the case for more bitstreams generated with different keys.

4.3.2 Frequency Test within a Block

The focus of this test is the proportion of ones within M-bit blocks. The purpose is to determine wheter the frequency of ones in an M-bit block is approximately M/2, as would be expected under an assumption of randomness.

A measure of how well the observed proportion of ones within a given M-bit match

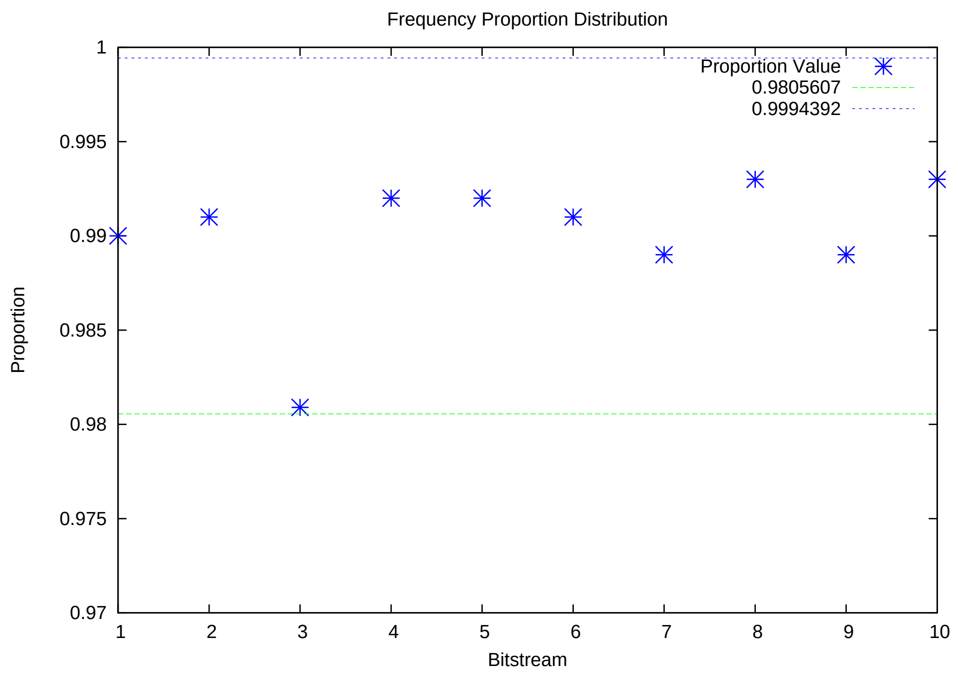

Figure 6: Frequency Test Proportion Distribution

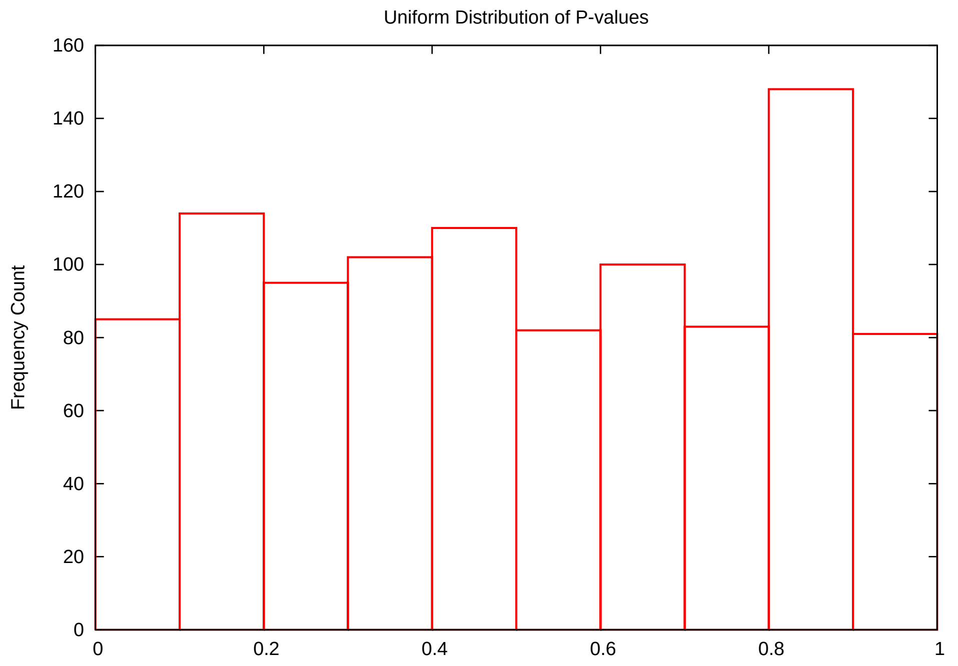

Figure 7: Frequency Test P-value Distribution for Bitsream 1

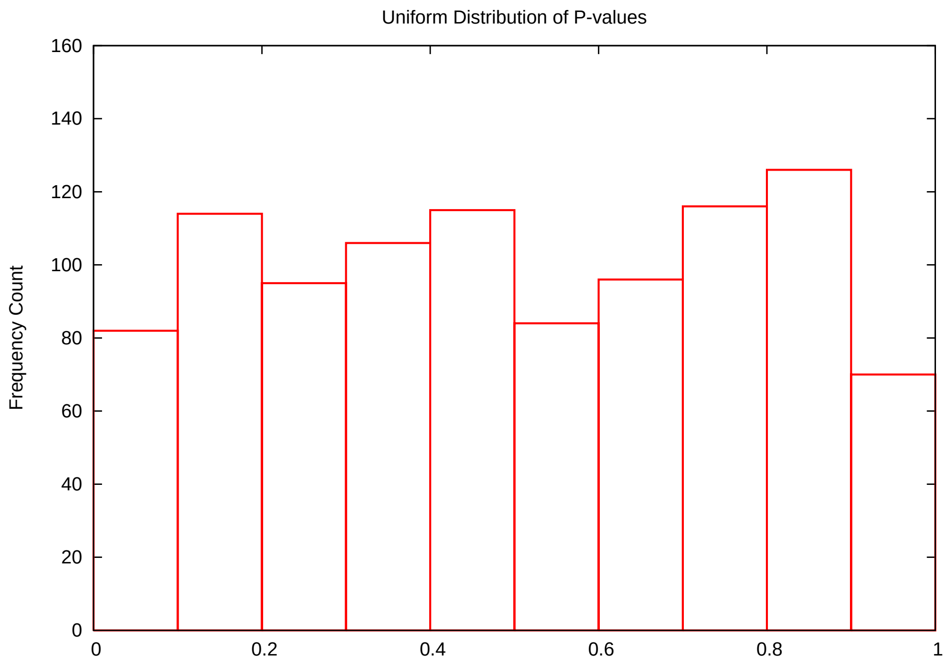

Figure 8: Frequency Test P-value Distribution for Bitstream 7

the expected proportion 1/2 is given by

$$X^2(obs) = 4M\sum_{i=1}^{N}(\pi_i - 1/2)^2$$

, where π_i is the proportion of ones in each M-bit block given by the equation

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)M+j}}{M},$$

for $(1 \leq i \leq N)$, M = blocklength, n = bitsequence length, $N = |\frac{n}{M}|$. non-overlapping blocks, ε is the sequence of bits. The P-value is generated by applying the Incomplete Gamma Function

$$P - value = Q(N/2, x^2(obs)/2.$$

We generate 1000 substrings from the bitstreams, each of the length 1000 bit. Then we divide each substring into $M = 50$ bit blocks. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again means that the proportion should lie above 0.9805607 for this particular test. Figure 9 shows the proportion value for the blocks within the substrings. Looking at the figure, we can conclude that the sequences tested for every bitstream appear to be random, and pass the frequency test, since all proportion values $\geq \alpha$

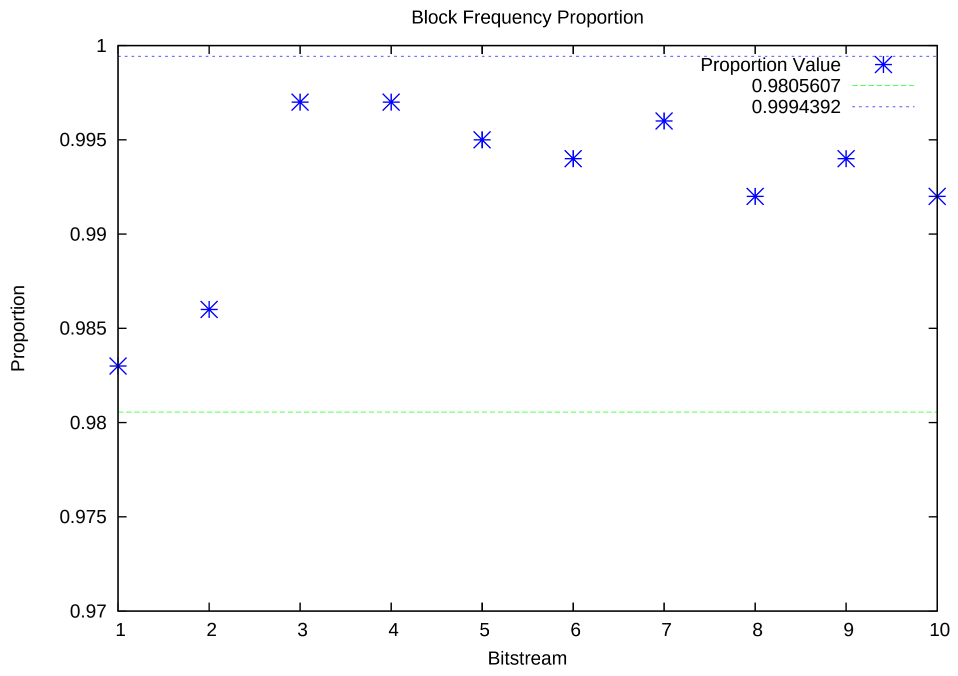

Figure 9: Frequency Block Test Proportion Distribution

Also here, we can draw a similar histogram of the distribution of the p-values to check uniformity. Figure 10 and 11, show the distribution of p-values for the bitstreams generated with respectively key 1 and key 7. The streams selected for evaluation were arbitrarily selected.

As we can conclude from the graphs, the P-value distribution seems quite uniform.

4.3.3 Runs Test

This test focuses on the number of runs in a sequence, where a run is an uninterrupted sequence of identical bits. A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value. The purpose of this test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.

The test statistic for this test is given by

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$$

, where $r(K) = 0$ if $\varepsilon_k = \varepsilon_{k+1}$, and $r(k) = 1$ othervise. The P-value is given by

$$P - value = erfc\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n\pi(1-\pi)}}\right)$$

, where π is the pre-test proportion of ones in the input sequence: $\pi = \frac{\sum j\varepsilon j}{n}$.

We generate 1000 substrings from the bitstreams, each of the length 1000 bits. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

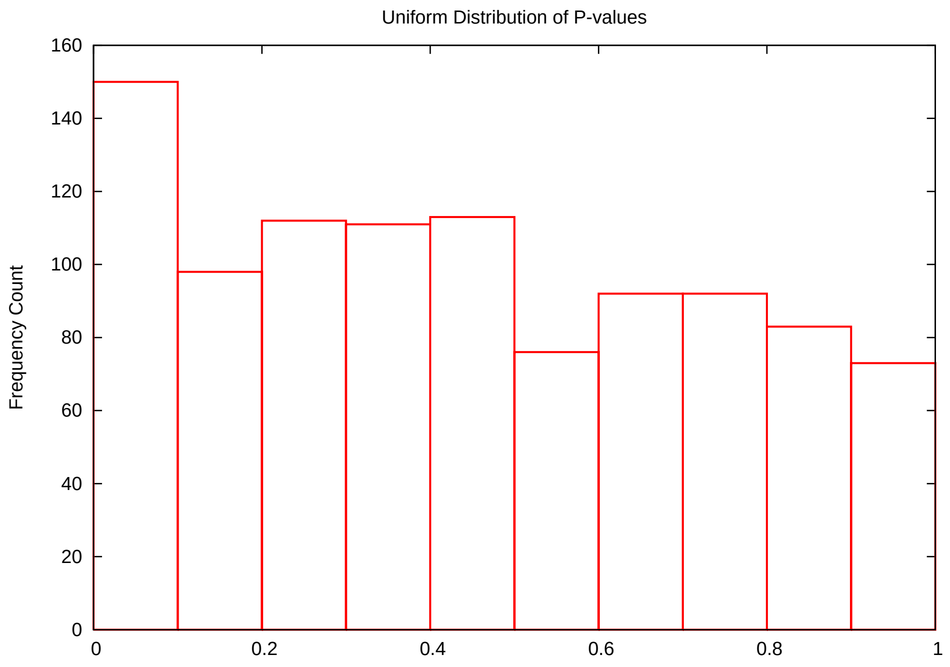

Figure 10: Block Frequency Test, P-value Distribution for Bitsream 1

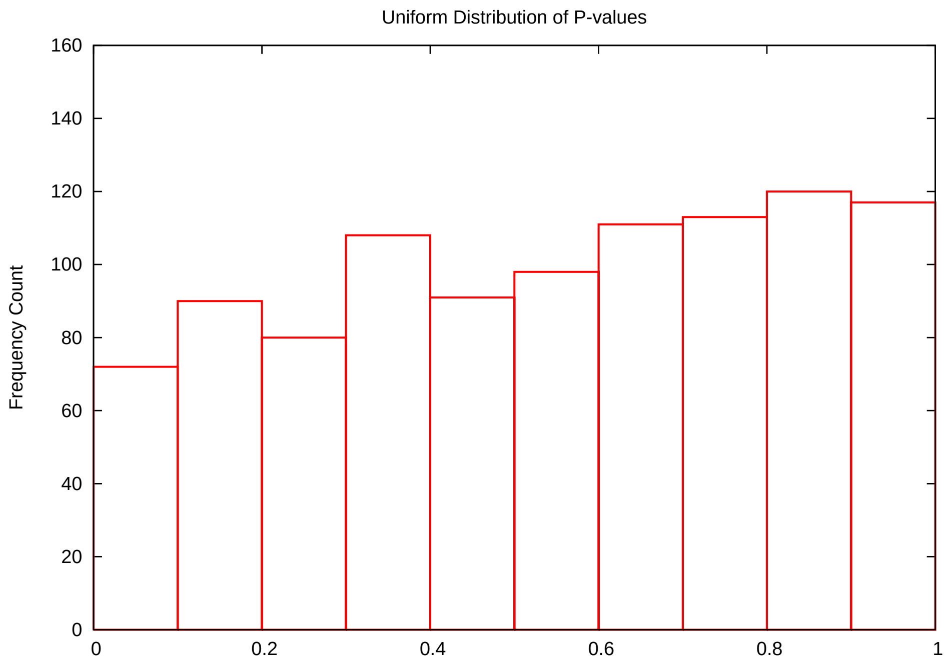

Figure 11: Block Frequency Test, P-value Distribution for Bitstream 7

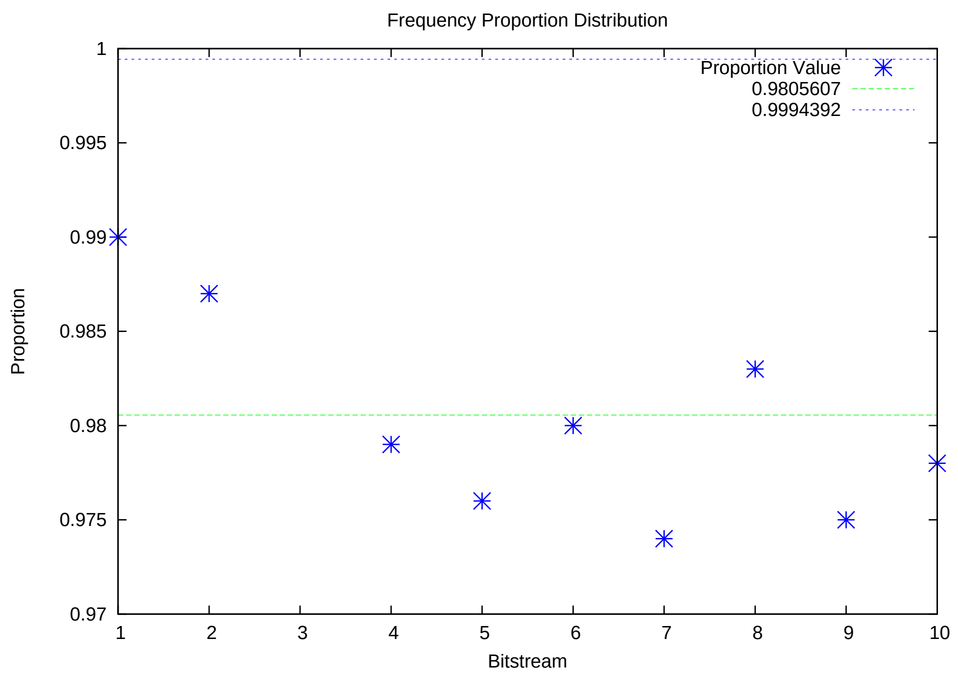

Figure 12: Runs Test Proportion Distribution

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test. Figure 12 shows the proportion value for the blocks within the substrings. Looking at the figure, we can conclude that the subsequences generated from three different bitstreams appear to be random. The subsequences generated from the other bitstreams have values which lie below the acceptance confidence interval used when 1000 subsequences are generated. When testing only 100 subsequences from each bitstream, the acceptance value becomes 0.960150. Each bitstream passed this test for 100 generated subsequences. The conclusion is that more bitstreams should be tested to give a clearer answer wheter the cipher passes the Runs test or not.

Also here, we can draw a similar histogram of the distribution of the p-values to check uniformity. Figures 13 and 14, show the distribution of p-values for the bitstreams generated with respectivly key 1 and key 7. The streams selected for evaluation were arbitrarily selected.

As we can conclude from the graphs, the P-value distribution seems quite uniform.

4.3.4 Test for the Longest Run of Ones in a Block

This test focuses on the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence.

M is pre-defined in the testing utility, and for a bitsequence of length 1000, M is set to 8.

The test statistic for this test $X^2(obs)$ measures how well the observed longest run

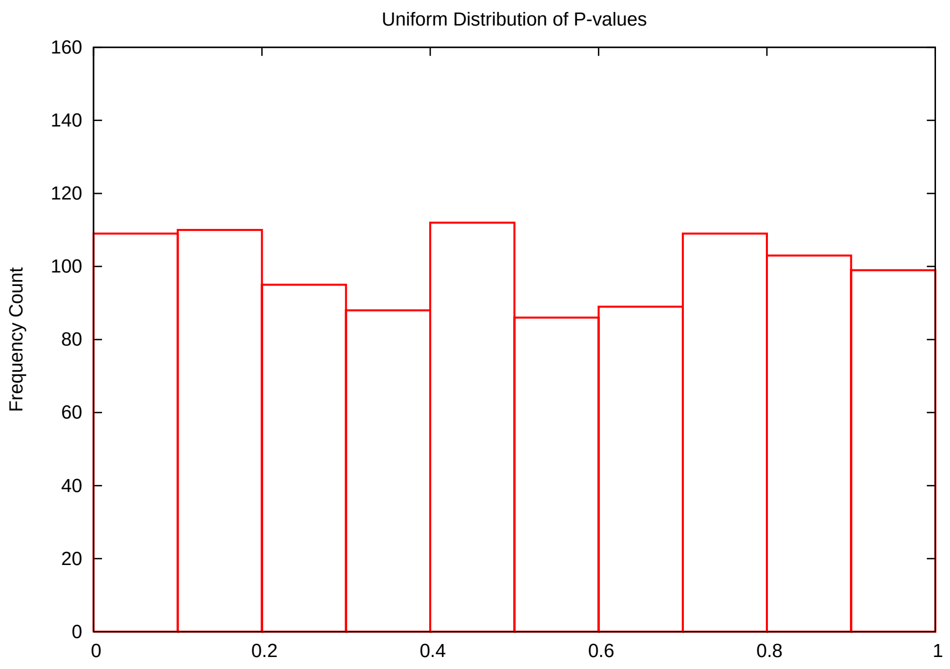

Figure 13: Runs Test, P-value Distribution for Bitsream 1

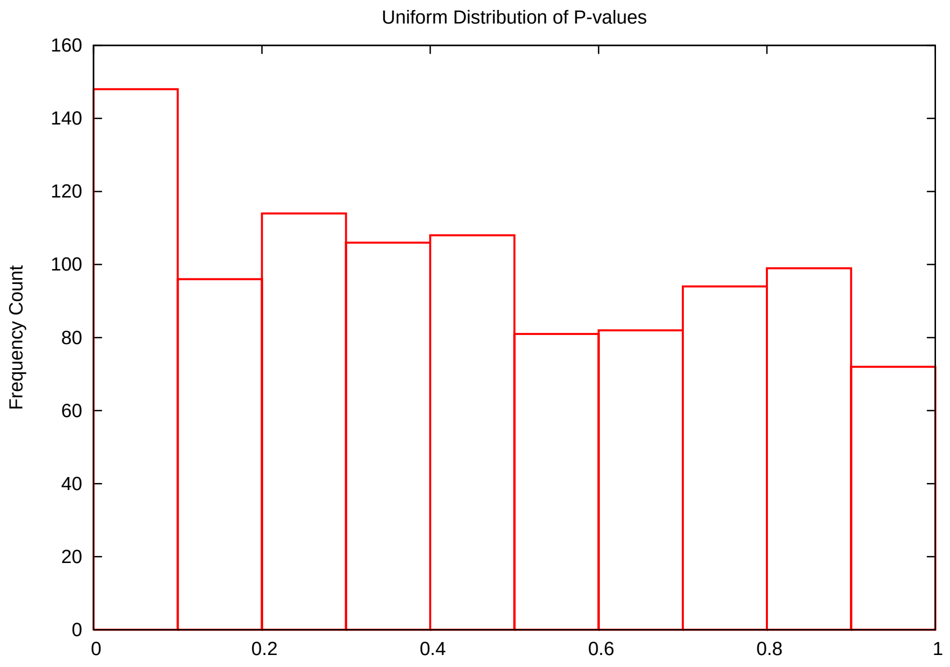

Figure 14: Runs Test, P-value Distribution for Bitstream 7

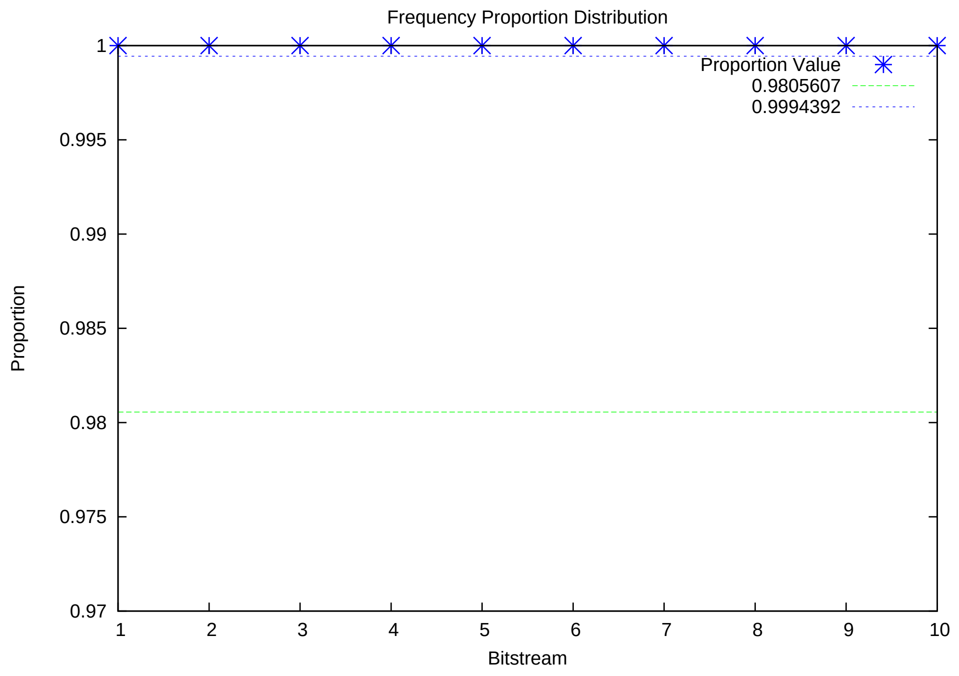

Figure 15: Longest Run Test Proportion Distribution

length within M-bit blocks matches the expected longest length within M-bit blocks. The function to calculate this value is found in [93]. The P-value is given by

$$P - value = \Gamma\left(\frac{3}{2}, \frac{X^2(obs)}{2}\right).$$

We generate 1000 substrings from the bitstreams, each of the length 1000 bit. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test. Figure 15 shows the proportion value for the blocks within the substrings. Looking at the figure, we can conclude that all the subsequences pass the test, and we can conclude that the sequences appear random for this test.

4.3.5 Binary Matrix Rank Test

The focus of this test is the rank of disjoint sub-matrices of the entire bit sequence. The purpose of this test is to check for linear dependence among fixed length substrings of the orginal sequence.

The test statistic for this test $X^2(obs)$ is a measure of how well the observed number of ranks of various orders match the expected number of ranks under the assumption of randomness.

The P-value is given by

$$P - value = e^{-X^2(obs)/2}.$$

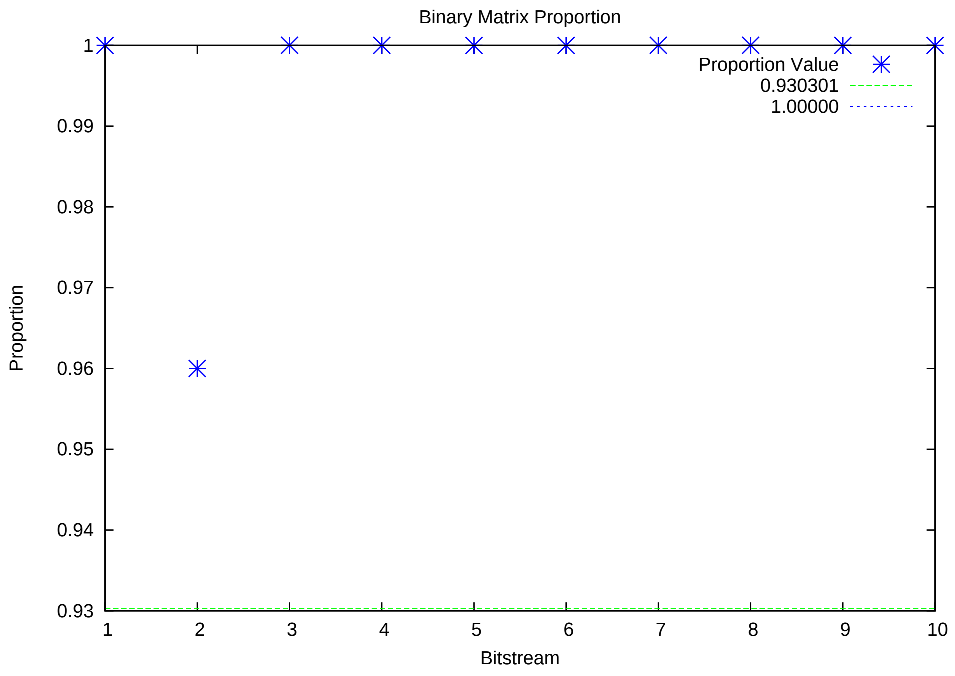

Figure 16: Binary Matrx Rank Test Proportion

In this test M is the number of rows in each matrix, this is default set to 32, and is what we shall operate with. Q is the number of colums in each matrix, and is also set to 32, which we have chosen to work with. $N = |\frac{n}{MQ}|$ disjoint blocks, and each row of this matrix blocks is filled with successive Q-bit blocks of the original sequence ε. A criterion for running this test, is that the minimum number of bits in each sequence to be tested are $n \geq 38MQ$, i.e., each subsequence we want to test must be at least 38,912 bits long.

We generate 25 substrings from the bitstreams, each of the length 38,912 bits. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{25}} = .99 \pm 0.05969$ for a sample size of 25. This again, gives that the proportion should lie above 0.93031 for this particular test.

Figure 16 shows the proportion value for the blocks within the substrings. Looking at the figure, we can conclude that the subsequences generated from all ten bitstreams appear to be random.

Also here, we can draw a similar histogram of the distribution of the p-values to check uniformity. Figures 17 and 18, show the distribution of p-values for the bitstreams generated with respectively key 1 and key 7. The streams selected for evaluation were arbitrarily selected.

As we can conclude from the graphs, the P-value distribution seems quite uniform. However, larger bit sequences should be tested to give a more certtain conclusion.

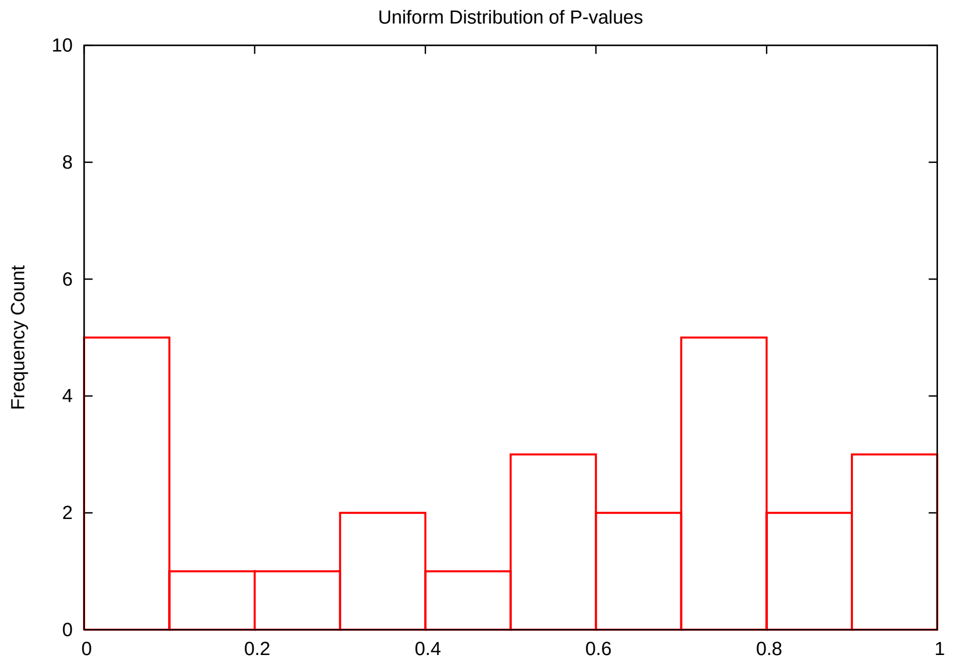

Figure 17: Binary Matrix Rank, P-value Distribution for Bitsream 1

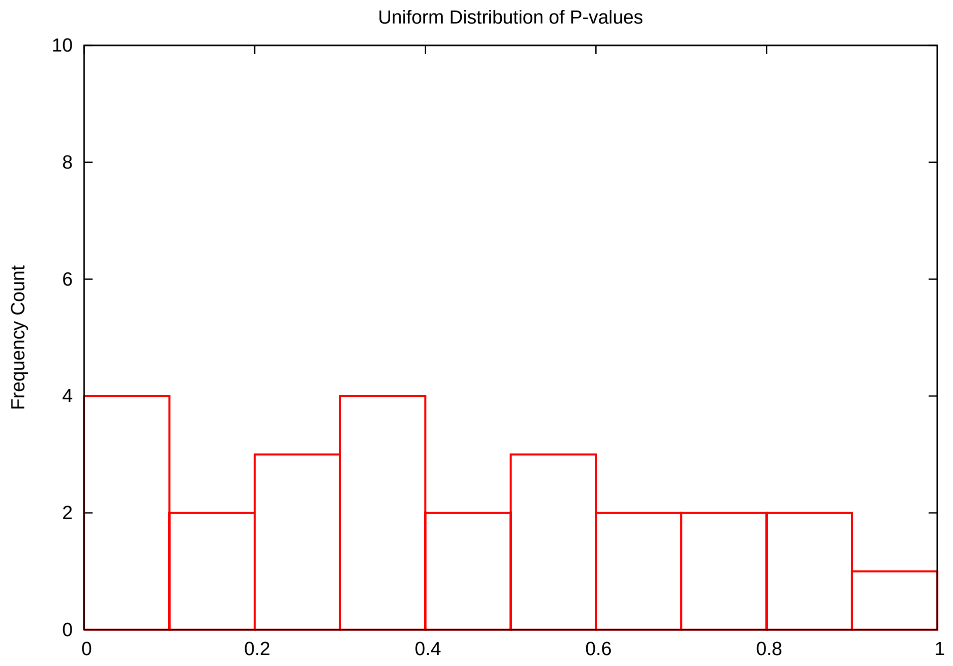

Figure 18: Binary Matrix Rank, P-value Distribution for Bitstream 7

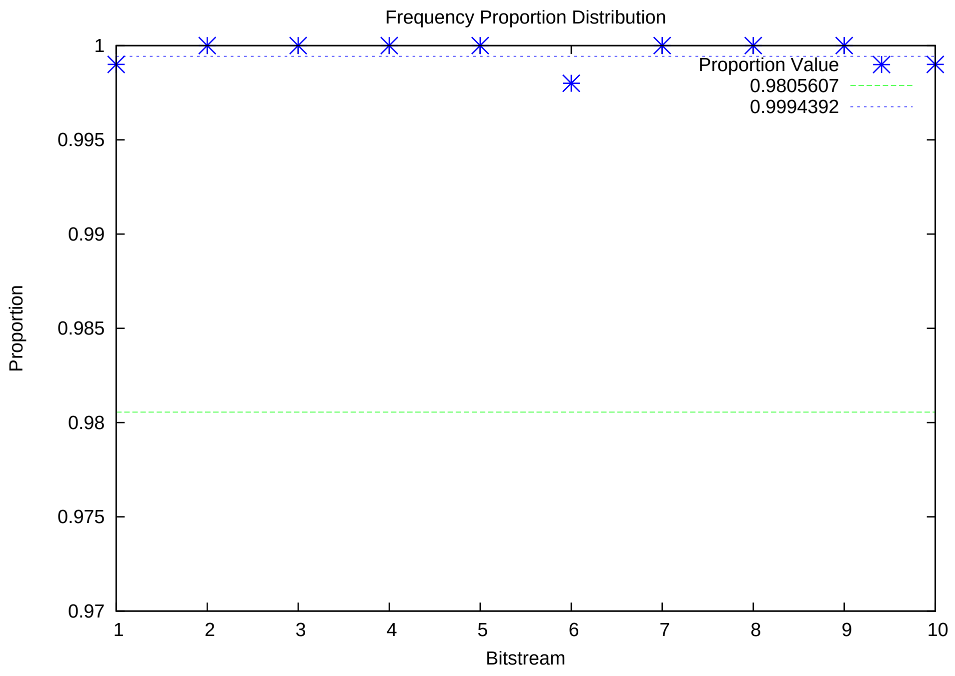

Figure 19: Discrete Fourier Transform Test Proportion

4.3.6 Discrete Fourier Transform (Spectral) Test

The focus of this test are the peak heights in Discrete Fourier Transform of the sequence. The purpose is to detect periodic repetitive patterns that are near each other in the tested sequence that would indicate a deviation from the assumption of randomness. The intention is to detect whether the number of peaks exceeding a 95% threshold is significantly different than 5%.

The test statistic for this test d, is the normalized difference between the observed and the expected number of frequency components that are beyond the 95 % threshold. The reference distribution for the test statistic is the normal distribution.

The P-value is given by

$$P - value = erfc\left(\frac{|d|}{\sqrt{2}}\right).$$

We generate 1000 substrings from the bitstreams, each of the length 1000 bit. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test.

Figure 19 shows the proportion value for the blocks within the substrings. Looking at the figure, we can conclude that the subsequences generated from all ten bitstreams appear to be random.

Also here, we can draw a similar histogram of the distribution of the p-values to check uniformity. Figures 20 and 21, show the distribution of p-values for the bitstreams

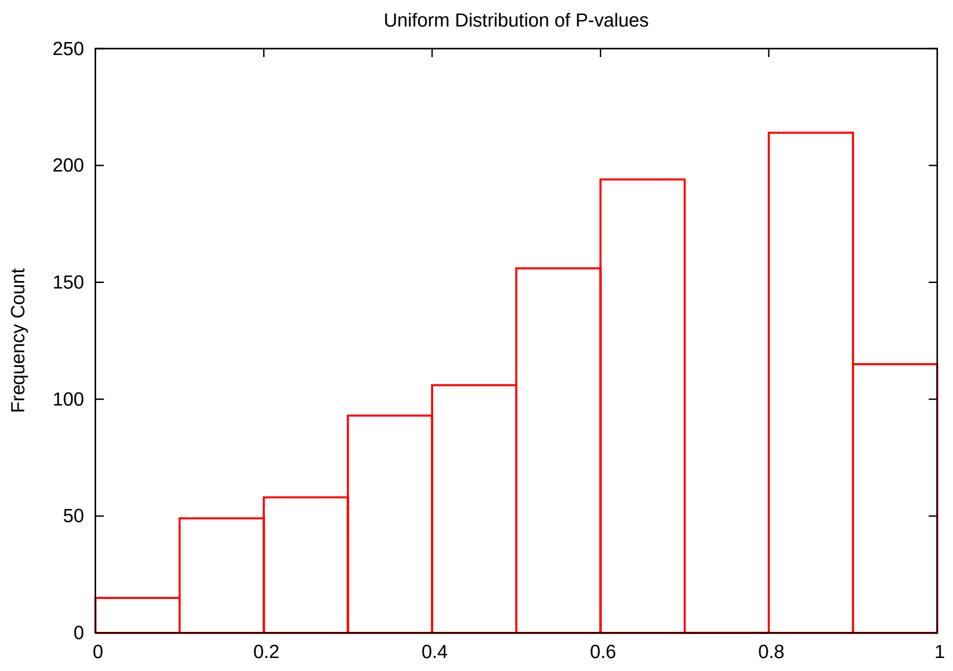

Figure 20: Discrete Fourier Transform, P-value Distribution for Bitsream 1

generated with respectively key 1 and key 7. The streams selected for evaluation were arbitrarily selected.

As we can conclude from the graphs, the P-value distribution seems quite similar for the two different bit streams. The interval between 0.7 and 0.8 however is 0 for all the bit streams. This is also the case for all the other algorithms which come with the testing suite which we have tested. The conclusion is that this behaviour is uniform for the test applied on different generators, and that the distribution of the P-values is good.

4.3.7 Non-overlapping Template Matching Test

The focus of this test is the number of occurrences of pre-specified target strings. The purpose of this test is to detect whether the generator produces too many occurrences of a given non-periodic (aperiodic) pattern. For this test, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the pattern, and the search resumes.

The test statistic for this test $X^2(obs)$, is a measure of how well the observed number of template "hits" matches the expected number of template "hits" (under the assumption of randomness). The reference distribution for the test statistic is the X^2 distribution.

The P-value is given by

$$P - value = \Gamma\left(\frac{N}{2}, \frac{X^2(obs)}{2}\right)$$

, where N is the number of independent blocks = 8.

This test is assuming a bit sequence of length 10^6, thus we apply the test on the entire bitstreams (no subsequences are generated). α is set to 0.01.

The range of acceptable proportions is determined using the confidence interval de-

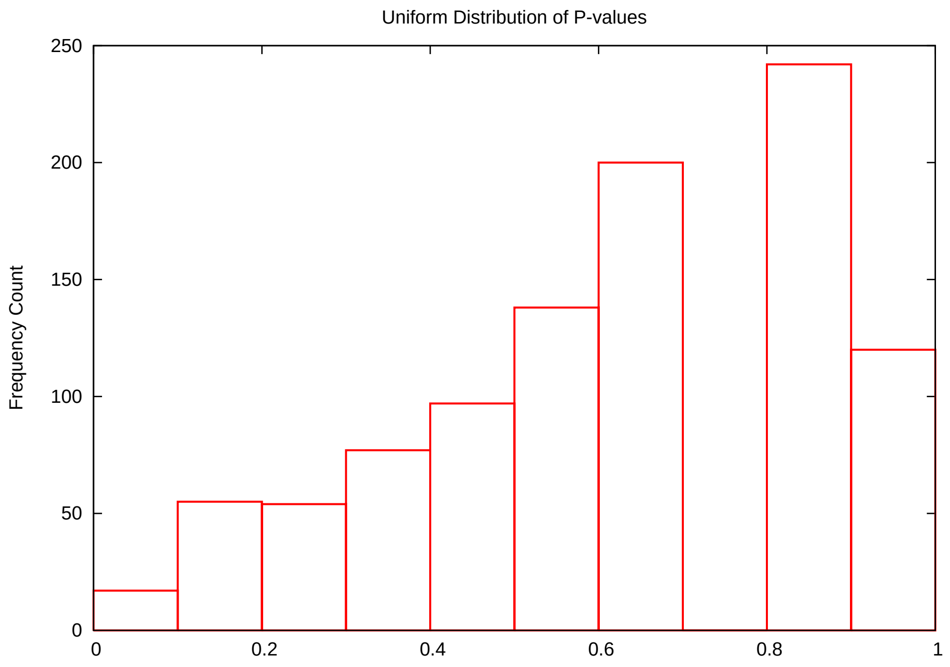

Figure 21: Discrete Fourier Transform, P-value Distribution for Bitstream 7

fined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

We have chosen the template length to be 9, which results in that a P-value will be generated for 148 different aperiodic templates. Table 7 shows the results for the ten bit sequences. The ten different bit sequences tested all fail to pass different aperiodic patterns than the other sequences. The second bitstream fails only for one of the 148 sequences. This should indicate that the stream would appear to be random, however larger sample sizes must be tested to give a more certain conclusion.

4.3.8 Overlapping Template Matching Test

The focus of this test is the number of occurrences of pre-specified target strings. The purpose of this test is to detect wheter the generator produces too many occurrences of a given non-periodic (aperiodic) pattern. For this test, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. The difference between this test and the test described in Section 4.3.7 is that when the pattern is found, the window slides only one bit before resuming the search.

The test statistic for this test $X^2(obs)$, is a measure of how well the observed number of template "hits" matches the expected number of template "hits" (under the assumption of randomness). The reference distribution for the test statistic is the X^2 distribution.

Bitstream	Templates not passed(F)	F/148
1	22	0.149
2	1	0.007
3	21	0.142
4	26	0.176
5	23	0.155
6	21	0.142
7	21	0.142
8	21	0.142
9	22	0.149
10	27	0.182

Table 7: Non-overlapping Templates Stats.

The P-value is given by

$$P - value = \Gamma\left(\frac{5}{2}, \frac{X^2(obs)}{2}\right).$$

This test is assuming a bit sequence of length 10^6, thus we apply the test on the entire bitstreams (no subsequences are generated). α is set to 0.01.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

We have chosen the template length to be 9. For this test, only one P-value is computed for each sequence. Table 8 shows the results for the ten bit sequences. The only bitstream which passes this test is the second bitstream, which mirrors the results obtained in Section 4.3.7. Larger sample sizes must be tested to give a more certain conclusion whether or not this indicates non-randomness for this particular test.

Bitstream	Templates passed/failed (P/F)
1	F
2	P
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F

Table 8: Overlapping Templates Stats.

4.3.9 Maurer's "Universal Statistical" Test

The focus of this test is the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random.

The test statistic for this test $X^2(obs)$, is a measure of how well the observed number of template "hits" matches the expected number of template "hits" (under the assumption of randomness). The reference distribution for the test statistic is the X^2 distribution.

The P-value is given by

$$P - value = \Gamma\left(\frac{5}{2}, \frac{X^2(obs)}{2}\right).$$

This test is assuming a bit sequence of length 10^6, thus we apply the test on the entire bitstreams (no subsequences are generated). α is set to 0.01.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, pre-defined values are set for different sequence lengths to be tested. We shall test sequences of length 10^6. For this length, the test blocksize is $L = 7$, and the number of initialization blocks is set to $Q = 10 \times 2^L = 1280$. Table 9 shows the results for the ten bitsequences. There were two bitstreams which did not pass this test. The P-values calculated for these two bitstreams 3 and 9, were respectively $P - value = 0.004126$ and 0.004128. Both values are $P - value \leq \alpha$, which might indicate non-randomness for this two particular sequences. However, 8 out of 10 sequences passed the test, thus larger sample sizes must be tested to give a more certain conclusion wheter or not this indicates non-randomness for the cipher.

Bitstream	Passed/Failed (P/F)
1	P
2	P
3	F
4	P
5	P
6	P
7	P
8	P
9	F
10	P

Table 9: Maurer's "Universal Statistical" Test Stats.

4.3.10 Lempel-Ziv Compression Test

The focus of this test is the number of cumulatively distinct patterns (words) in the sequence. The purpose of the test is to determine how much the tested sequence can be compressed. If a sequence can be significantly compressed, it is considered to be non-random. A random sequence will have a characteristic number of distinct patterns.

The test statistic for this test $W(obs)$, is the number of disjoint and cumulatively distinct words in the target sequence. The reference distribution for the test statistic is the normal distribution.

The P-value is given by

$$P - value = \frac{1}{2}\text{erfc}\left(\frac{\mu - W(obs)}{\sqrt{2\sigma^2}}\right)$$

, where $\mu = 69586.25$ and $\sigma = \sqrt{70.448718}$ for a sequence size $n = 10^6$.

This test is assuming a bit sequence of length 10^6, thus we apply the test on the entire bitstreams (no subsequences are generated). α is set to 0.01.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, it is recommended to use sample sizes $n \geq 10^6$, thus we apply the test on the entire ten bitstreams.

Table 10 shows the results for the ten bit sequences. There were three bitstreams which did not pass this test. The number of disjoint and cumulative words for the sequences ranges from 69558 to 69584. This indicates all the bitstreams are close to μ and close to pass this test for randomness. However, only 7 out of 10 sequences passed the test, thus larger sample sizes must be tested to give a more certain conclusion wheter or not this indicates randomness or not.

Bitstream	Passed/Failed (P/F)
1	F
2	F
3	P
4	F
5	P
6	P
7	P
8	P
9	P
10	P

Table 10: Lempel-Ziv Compression Test.

4.3.11 Serial Test

The focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of the test is to determine wheter the number of occurences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity, meaning that every m-bit pattern has the same chance of appearing as every other m-bit pattern. For $m = 1$, this test will be equivalent to the Frequency test described in Section 4.3.1.

The test statistic for this test $\nabla\Psi^2{}_m(obs)$ and $\nabla^2\psi^2{}_m$, is a measure of how well the observed frequencies of m-bit patterns match the expected frequencies of the m-bit patterns. The reference distribution for the test statistic is the X^2 distribution.

Two P-values for this test are computed, given by

$$P - value1 = \Gamma\left(2^{m-2}, \nabla\psi^2{}_m\right)$$

, and

$$P - value1 = \Gamma\left(2^{m-3}, \nabla\psi^2{}_m\right)$$

, where m is the length in bits of each block.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, it is recommended to choose m and n such that $m < |\log_2 n| - 2$. We first apply the test on the entire ten bitstreams. We choose $n = 10^6$ and $m = 2$. α is set to 0.01.

Table 11 shows the results for the ten bit sequences. The first two bitstreams passed this test (Both $P - values \geq \alpha$). However the last 8 bit sequences did not. The inner state of the first two bitsequences (which passed the test) were initialized with a human generated seed. The last eight sequences were initialized with automaticly generated seeds. This might or might not be a coincidence, however further investigation should be conducted on this property. The criterion $m < |\log_2 n| - 2$, also holds for bit sequences of length 1000. We generate 1000 subsequences of length 1000 for each bitstream to get further analysis.

The range of acceptable proportions for this test is determined by using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test.

Table 12 shows the resulting proportion of sub sequences which now passes this test. In this test, three out of the ten bit sequences fail, indicating a better result for the cipher. Further bit sequences should be tested to give a final conclusion for this test.

4.3.12 Approximate Entropy Test

The focus of this test is the frequency off all possible overlapping m-bit patterns across the entire sequence (as the test described in Section 4.3.11). The purpose of the test is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and $m + 1$) against the expected result for a random sequence.

Bitstream	P-value1	P-value2
1	0.291771	0.201956
2	0.357241	0.252624
3	0.000000	0.000000
4	0.000000	0.000000
5	0.000000	0.000000
6	0.000000	0.000000
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000
10	0.000000	0.000000

Table 11: Serial Test on samples $n = 10^6$.

Bitstream	Prop. passed for P-val.1	Prop. passed for P-val.2
1	0.9900	0.9910
2	0.9930	0.9890
3	0.9820	0.9730
4	0.9890	0.9820
5	0.9820	0.9810
6	0.9860	0.9870
7	0.9860	0.9770
8	0.9890	0.9840
9	0.9880	0.9780
10	0.9840	0.9810

Table 12: Serial Test on samples where $n = 10^3$.

The test statistic for this test $X^2(obs) = 2n[\log 2 - ApEn(m)]$, where $ApEn(m) = \varphi^{(m)} - \varphi^{(m+1)}$, is a measure of how well the observed value of $ApEn(m)$ matches the expected value.

The P-value for this test given by

$$P - value = igamc\left(2^{m-1}, \frac{X^2}{2}\right)$$

, where m is the length in bits of each block.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again means that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, it is recommended to choose m and n such that $m < \lfloor \log_2 n \rfloor - 2$. We first apply the test on the entire ten bitstreams. We choose $n = 10^6$ and $m = 2$. α is set to 0.01.

Table 13 shows the results for the ten bit sequences. The first two bitstreams passed this test($P{-}values \geq \alpha$). However the last 8 bit sequences did not. The inner state of the first two bitsequences (which passed the test) were initialized with a human generated seed. The last eight sequences were initialized with automaticly generated seeds. Again, this special property of the test requires more attention. The criterion $m < \lfloor \log_2 n \rfloor - 2$ also holds for bit sequences of length 1000. We generate 1000 subsequences of length 1000 for each bitstream to get further analysis.

The range of acceptable proportions for this test is determined by using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test.

Table 14 shows the resulting proportion of subsequences which now passes this test. In this test, one out of the ten bit sequences fails. This result looks promising for passing the test, however, even further bit sequences should be tested to give a final conclusion.

Bitstream	P-value
1	0.264642
2	0.563521
3	0.000000
4	0.000000
5	0.000000
6	0.000000
7	0.000000
8	0.000000
9	0.000000
10	0.000000

Table 13: Approximate Entropy Test on samples $n = 10^6$.

Bitstream	Proportion passed test
1	0.9910
2	0.9900
3	0.9840
4	0.9910
5	0.9790
6	0.9850
7	0.9890
8	0.9890
9	0.9880
10	0.9830

Table 14: Approximate Entropy Test on samples where $n = 10^3$.

4.3.13 Cumulative Sums (Cusum) Test

The purpose of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted (-1,+1) digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occuring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. This cumulative sum may be considered as a

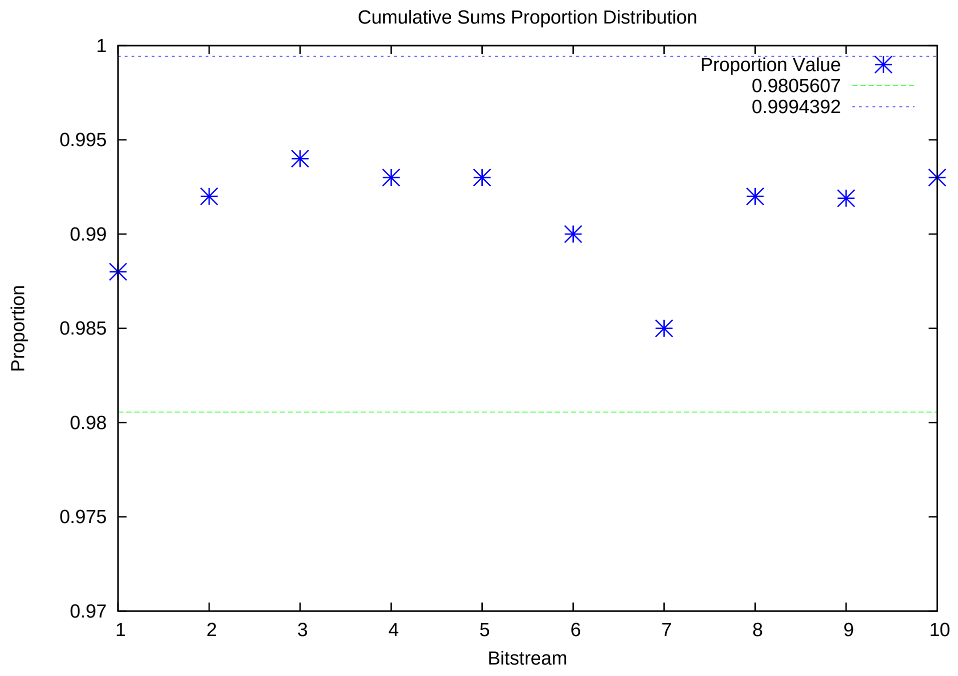

Figure 22: Forward Proportion Distribution

random walk. For a random sequence, the excursions of the random walk should be zero. For certain types of non-random sequences, the excursions of this random walk from zero will be large.

The test statistic for this test, z, is the largest excursion from the origin of the cumulative sums in the corresponding (-1,+1) sequence.

We generate 1000 substrings from the bitstreams, each of the length 1000 bits. α is set to 0.01. The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1000}} = .99 \pm 0.0094392$ for a sample size of 1000. This again, gives that the proportion should lie above 0.9805607 for this particular test.

This test generates two P-values. One when iterating forward through the sequence, and one going backwards in the sequence. Figure 22 shows the proportion of substrings pasing this test iterating forwards. Looking at the figure, we can conclude that the subsequences generated from the bitstreams appear to be random.

We can draw a histogram of the distribution of the p-values to check uniformity. Figures 23 and 24, show the distribution of p-values for the bitstreams generated with respectively key 1 and key 7. The streams selected for evaluation were arbritary selected.

As we can conclude from the graphs, the P-value distribution seems quite uniform.

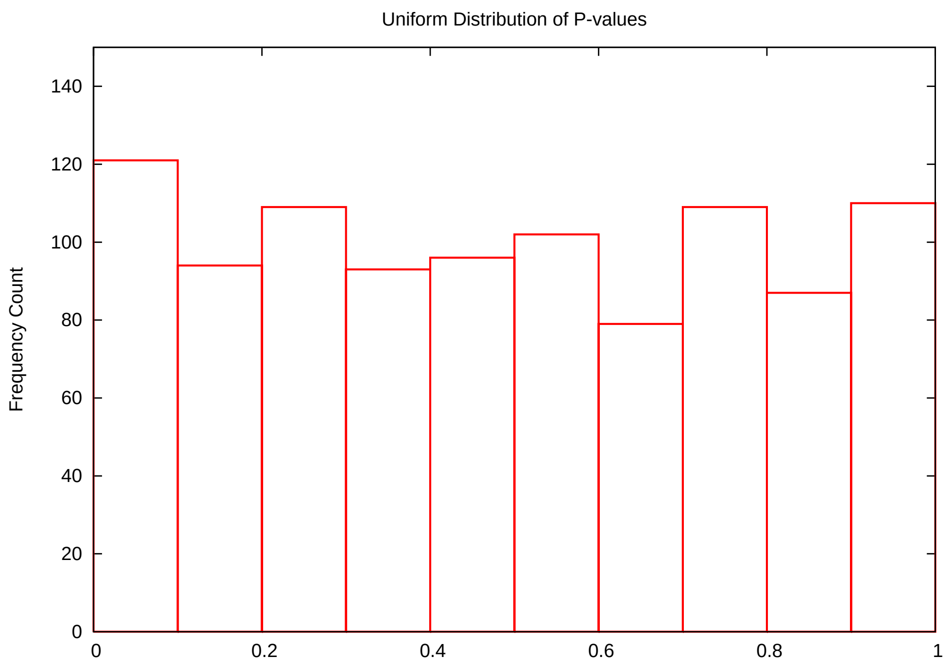

Figure 23: Cummulative Sums Test, P-value Distribution for Bitsream 1

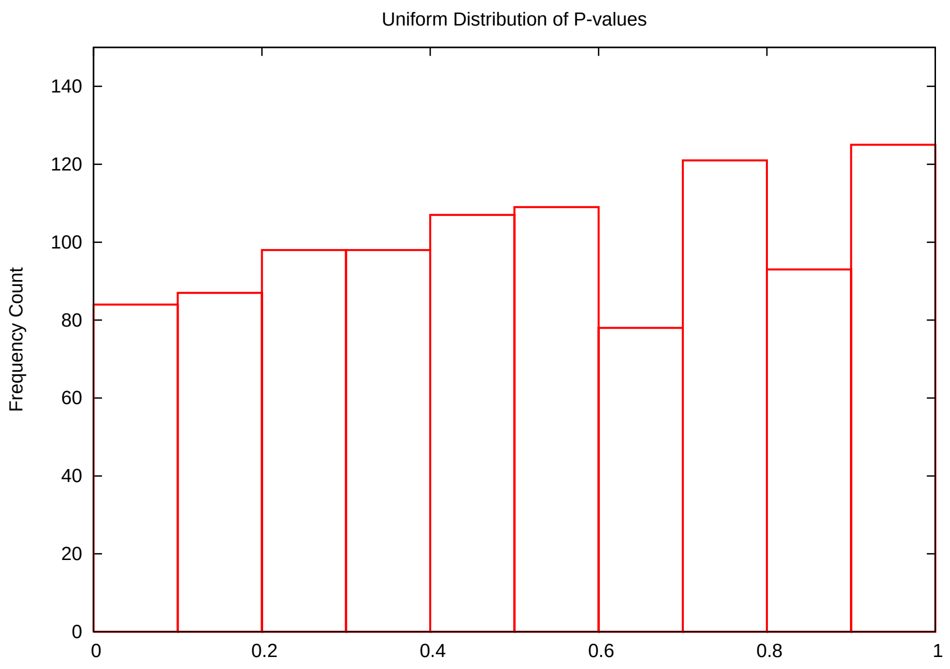

Figure 24: Cummulative Sums Test, P-value Distribution for Bitstream 7

4.3.14 Random Excursions Test

The focus of this test is the number of cycles having exactly K visits in a cumulative sum random walk. The cumulative random walk is derived from partial sums after the (0,1) sequence is transferred to the appropriate (-1,+1) sequence. A cycle of random walk consists of a sequence of steps of unit length taken at random that begin at, and return to the origin. The purpose of this test is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence. This test is in practice a series of eight tests and conclusions, one test and conclusion for each of the states: -4, -3, -2, -1, +1, +2, +3, +4.

The test statistic for this test $X^2(obs)$, for a given state x, is a measure of how well the observed number of state visits within a cycle match the expected number of state visits within a cycle, under the assumption of randomness. The reference distribution for the test statistic is the X^2 distribution.

The P-value for this test given by

$$P - valuex = \Gamma\left(\frac{5}{2}, \frac{X^2(obs)}{2}\right).$$

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, it is recommended to choose n such that $n \geq 10^6$. α is set to 0.01.

Table 15 shows the results for the ten bit sequences. The first six bitstreams passed this test ($P - values \geq \alpha$) for all states. Bitstream number 7 had one state ($x = -1$) that did not pass, but the rest of the states passed. A complete evaluation for the last three bitstreams could not be computed because not enough cycles were produced for the sequences. The seven bitstreams that were completely evaluated indicate that the sequences produced by the cipher are random, however larger bit sequences should be tested, so that all sequences would be equaly evaluated to give a final conclusion.

Bitstream	Passed/Failed(-4,-3,-2,-1,+1,+2,+3,+4)
1	P,P,P,P,P,P,P,P
2	P,P,P,P,P,P,P,P
3	P,P,P,P,P,P,P,P
4	P,P,P,P,P,P,P,P
5	P,P,P,P,P,P,P,P
6	P,P,P,P,P,P,P,P
7	P,P,P,F,P,P,P,P
8	Not enough cycles produced
9	Not enough cycles produced
10	Not enough cycles produced

Table 15: Random excursions Test Statistics.

4.3.15 Random Excursions Variant Test

The focus of this test is the number of times that a particular state is visited in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of visits to various states in the random walk. This test is in fact a series of eighteen tests, one test and conclusion for each of the states: -9, -8, ..., -1, and +1, +2, ..., +9.

The test statistic for this test $X^2(obs)$, for a given state x, is a measure of how well the observed number of state visits within a cycle match the expected number of state visits within a cycle, under the assumption of randomness. The reference distribution for the test statistic is the X^2 distribution.

The P-value for this test given by

$$P - valuex = \Gamma\left(\frac{5}{2}, \frac{X^2(obs)}{2}\right).$$

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

, where $\hat{p} = 1 - \alpha$, and m is the sample size. This gives the confidence interval $.99 \pm 3\sqrt{\frac{.99(.01)}{1}} = .99 \pm 0.298496$ for a sample size of 1. This again, gives that the proportion should lie above 0.69150 for this particular test. However, since only one sequence is tested, the conclusion can only be that the sequence passes the test, or not.

For this test, it is recommended to choose n such that $n \geq 10^6$. α is set to 0.01.

Table 16 shows the results for the ten bit sequences. All states in bitstream 1, 3, 4, 5, 6, 7, passed this test ($P - values \geq \alpha$). Bitstream number 2 had one state ($x = +1$) that did not pass, but the rest of the states passed. A complete evaluation for the last three bitstreams could not be computed because not enough cycles were produced for the sequences. The seven bitstreams that were completely evaluated indicate that the sequences produced by the cipher are random, however larger bit sequences should be tested, so that all sequences would be equaly evaluated to give a final conclusion.

Bitstream	Not Passed States
1	All passed
2	X = +1
3	All passed
4	All passed
5	All passed
6	All passed
7	All passed
8	Not enough cycles produced
9	ANot enough cycles produced
10	Not enough cycles produced

Table 16: Random excursions Variant Test Statistics.

4.3.16 Conclusion

For most tests, the cipher seems to produce random sequences for all the different seeds tested. However, some characteristics are worth additional attention.

In the Serial Test, the bit sequences, which had seeds automatically generated using the system random function in C#, did not pass the test. More research should be conducted to see if this is a coincidence or not. The cipher dependency of the seeds should also be further investigated.

The results in general look very promising regarding the randomness of the cipher. Even though more bit streams should be tested, we conclude that the new bit stream generator is a good generator.

4.4 Efficiency Testing

4.4.1 Introduction

The presented cipher in this thesis is a custom designed intended for running very efficiently in software, and to have high level of security. To give a benchmark of how fast the cipher really is, the cipher is compared to some other popular ciphers. The custom cipher simulation is compared with two other implementations also in C#, the RC4 stream cipher [75] and the Rijndael (AES) block cipher [50]. The RC4 implementation was obtained at `http://www.codeproject.com`[3], and the Rijndael implementation was obtained at `http://msdn.microsoft.com`[4].

4.4.2 Efficency Results

The test is run on a laptop with the Operating System Windows XP SP2. The CPU is Intel Mobile 1,8 GHz, and the laptop has a physical memory of 512 MB.

Three different files with different lengths were used during the testing, file 1 with the size 1.13 MB, file 2 with the size 57.2 MB and file 3 with size 103 MB. The key length used for both the RC4 and Rijndael encryption were 256 bits. The custom design used a 256 bit key in addition to the 64 bit message key. Figure 25 shows the efficiency measured on the different files with the different ciphers. Table 17 shows the exact results from the test.

CIPHER	1.13 MB	57.2 MB	103 MB
Custom	0.320	10.876	19.448
AES	0.521	14.441	26.318
RC4	0.581	7.811	14.220

Table 17: Efficiency results obtained on a Intel Mobile 1.8 GHz Processor.

4.4.3 Conclusion

The results show that the newly designed cipher is fast. The simulation implementation is not optimized regarding the number of system function calls, and variable types have to be converted often, which also reduces the efficiency. The cipher is based on software simulated linear feedback shift registers, which can be implemented in many different ways. Further improvements of the LFSR simulation, such as the sliding window technique [94] can be applied to improve the efficiency. The cipher seems to be a little bit slower than the RC4 algorithm. This is probably because RC4 does not make use of LFSRs. The cipher is significantly faster than the AES implementation tested. The RC4 and AES implementation are not tested to be working correctly, however this is an

[3]http://www.codeproject.com/csharp/rc4csharp.asp
[4]http://msdn.microsoft.com/msdnmag/issues/03/11/AES

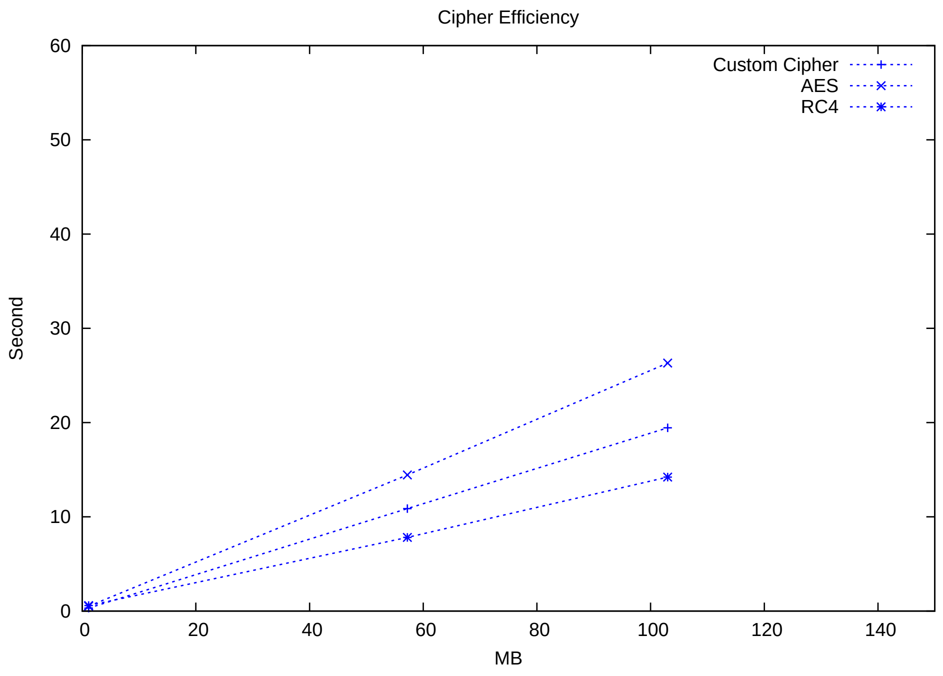

Figure 25: Efficiency results obtained on a Intel Mobile 1.8 GHz Processor.

assumption made, so the validity of the obtained results from these two implementations cannot be verified. The reliability however seems to be quite good, since the same result is produced in each run. The tests have been run on a laptop, where the operating system has full control over the memory and the virtual memory (SWAP) files. If too much data to fit in a certain percentage of the physical memory are accessed during the encryption, this may influence the results by being a third factor. We would not only be measuring the cipher efficiency, but also the operating system's procedures to generate and handle virtual files on the hard drive. To minimize this third factor, only files small enough to fit in the physical memory were encrypted and tested.

5 Discussion

5.1 Advantages/Disadvantages of Using a Custom Cipher Compared to Publicly Known Ciphers

A custom designed cipher, where the source code is closed, can be tested by statistical testing. The results from this testing can then be analyzed, and indications and conclusions can be made to indicate that the cipher is a good cipher or not. For any adversary wanting to break the cipher, to know the source code would be an advantage. To achieve this on a closed source custom cipher, reverse engineering has to be applied on it, i.e., reversing machine code back to readable source code. To conduct reverse engineering is possible to do, however it is often time demanding, especially if the source code was written with code obfuscation principles in mind. With enough effort, the source code would eventually be revealed. As an example was the RC4 algorithm reverse engineered as early as in 1994, see Section 2.2.5.3. However, on a closed source custom cipher, obtaining the source code will be another obstacle or a "speed bump" on the way to breaking the cipher. As claimed in [95], to make obstacles or "speed bumps" for adversaries, is the best way to improve security.

The security by obscurity principle (keeping the source code hidden or secret) [30] is a much debated principle in cryptographic research and information security literature. When discussing this matter, some claim that security by obscurity do not gain any improved security at all, others claim that it does. One factor that is important to consider when discussing this matter is the adversaries. For adversaries with unlimited resources, it probably would be easier to overcome the security by obscurity problem than for an adversary with limited resources. It is no doubt that security by obscurity makes the process of attacking a cipher or a source code kept secret to take longer time and demand greater resources than if all is publicly known. However, it is clear that it is important to be security conscious when designing ciphers and source code even though it is planned to be kept secret. If the security by obscurity principle should fail, the task of breaking the cipher should still be an impossible task to do.

Contrary to say that a cipher might be broken in secret, we can say that open source ciphers are available for everyone to crypt-analyze. A popular public cipher will be and have been thoroughly analyzed, and if no big security breaches are found on them, this can indicate that the cipher is a good design.

To generate a picture over a potential adversary is very difficult. Adversary modeling is a wide topic within information security, and to make any precise model over adversary capabilities is impossible. However, we can make some assumptions. Potential adversaries interested in getting hold of sensitive information, will always be on the same level of breaking a public known cipher as the open literature written on it. In addition, any adversaries may have made discoveries and conducted successful attacks in secret. Different ciphers may be completely broken in secret, so by using a open source cipher, we might rely on a cipher which are already broken. By using a custom made cipher, we can be sure that it is not already is broken. By maintaining good policies when using a cipher such as the one described in Chapter 3, we can be pretty sure that data encrypted with it will remain confidential, and breaking the cipher will be hard.

5.2 Efficiency/Security Trade off in Cryptographic Algorithms for PC protection

Traditionally, it seems to be a relationship between efficiency and security. The higher security we want, the less efficient is the method providing the security. Each operation a CPU has to perform takes time. Some operations are more complex and more time demanding than others. The ideal case is to use simple operations, but to combine these operations in a way to achieve a high level of security.

In software, large storage of data is faster to process, than data which have to be calculated first, and then processed. On modern computers, storage space is almost never a problem.

To know the processor which will execute the cipher running on it is also an advantage when designing it. To know how many bits the CPU processes at the time, can be used to gain higher efficiency. The custom cipher produces 32 bits at the time for optimization on 32 bit CPU computers.

6 Conclusions

Most software applications, which provide encryption of sensitive data uses passwords or pass phrases to protect the keys used for encryption and decryption. The key is often derived directly from the password or pass phrase in some key generation method. To be able to retrieve the password results in finding the encryption key. Since the entropy in such a password or pass phrase most likely is much less than in the key generated, the security of such systems fall dramatically. Estimates show that the entropy of a 8 character long password are about the same as for a 32-bit key, hence if a 128 bit key is wanted, we need a 98 character long password or pass phrase [96] to obtain the same level of entropy. Thus, such software should be used with care and investigated before given any trust.

In this thesis a new stream cipher design is presented and an implementation of the cipher was written in C#. The cipher uses well known and analyzed techniques to build up the cipher, such as irregular clocking and linear feedback shift registers.

Cryptanalysis performed on the cipher indicates that all the known weak properties a stream cipher can have, which reduces the ciphers security, is handled in a good way.

Statistical testing applied to the cipher proves that the bit sequences produced by the cipher generator appear to be random. Based on the statistical results obtained during testing, we can conclude that the cipher generator is a good generator. Thus, the cipher seems to be secure.

Practical experiments conducted show that the cipher runs fast in software on a 32 bit CPU. Even though the implementation of the cipher is not extensively optimized, the testing shows that it has good efficiency statistics when compared to other ciphers.

7 Future Work

The implementation of the cipher is in C#, and though the implementation is made with efficiency in mind, there is still room for further optimization. Other techniques, especially the LFSR implementation in the C# simulation, can be implemented and tested. To maximize the efficiency of the cipher, a good x86 Assembler implementation would be necessary. New efficiency tests on the Assembler implementation of the cipher would give much better results for the optimal efficiency for the cipher implemented in software.

The security strength of the custom cipher can be further assessed and analyzed. A more thorough cryptanalysis can be conducted, and the number of stitistical tests can be expand. More sequences, both larger and generated with a larger width of seeds (keys) can be applied to the already used statistical tests, to give even better and a more certain conclusion if the generator is good or not.

It would be interesting to encrypt some files or data with the custom algorithm, and then apply a greater penetration testing experiment on them. This would have given a more practical assessment of the cipher's security strength.

To make a user interface for easy usage of the custom cipher is necessary if it is to be used for people with limited knowledge of programming and computers in general. Since the cipher is implemented with object oriented design in mind, this should not be a too difficult task to do for other not knowing the inner workings of the cipher design.

Bibliography

[1] Preneel, B., Biryukov, A., Oswald, E., Rompay, B. V., Granboulan, L., Dottax, E., Murphy, S., Dent, A., White, J., Dichtl, M., Pyka, S., Schafheutle, M., Serf, P., Biham, E., Barkan, E., Dunkelman, O., Ciet, M., Sica, F., Knudsen, L., Raddum, H., & Parker, M. NESSIE security report. Technical report, NESSIE, `http://www.cryptonessie.org`, 2003. 1.3, 2.2.3.1, 2.2.4, 2.2.5.4

[2] Kaliski, B. 2000. Pkcs #5: Password-based cryptography specification version 2.0. 2.1.1, 2.1.2

[3] 1993. PKCS#8: Private-Key Information Syntax Standard Version 1.2. `http://www.rsasecurity.com/rsalabs/node.asp?id=2130`. 2.1.1

[4] 1999. PKCS#12 v1.0: Personal Information Exchange Syntax. `http://www.rsasecurity.com/rsalabs/node.asp?id=2138`. 2.1.1

[5] Partition Manager 6.0. `http://www.partition-manager.com/`. 2.1.1

[6] Cole, E. 2003. *Hiding in Plain Sight : Steganography and the Art of Covert Communication*. Wiley. 2.1.1

[7] Microsoft Windows XP. `http://www.microsoft.com/windowsxp/`. 2.1.1

[8] Linux. http://www.kernel.org. 2.1.1

[9] Eastlake, D. 2001. RFC 3174: US Secure Hash Algorithm 1 (SHA1). `http://www.ietf.org/rfc/rfc3174.txt`. 2.1.1, 2.2.3.5

[10] Rivest, R. 1992. RFC 1321: The MD5 Message Digest Algorithm. `http://www.ietf.org/rfc/rfc1321.txt`. 2.1.1

[11] Viega, J. & McGraw, G. 2002. *Building Secure Software - How to Avoid Security Problems the Right Way*. Addison-Wesley. 2.1.1

[12] Storing Your Valuables: Warehousing Your Data. `http://www.enterprisestorageforum.com/`. 2.1.1

[13] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. 1996. *Handbook of applied cryptography*. CRC Press. 2.1.1, 2.2.3.1, 3.1.4

[14] Stinson, D. 2002. *Cryptography: Theory and Practice*. CRC Press, second edition edition. 2.1.1, 7

[15] Algorithms integrated in Mozilla. http://www.mozilla.org/projects/ security/pki/nss/algorithms.html. 2.1.2

[16] Garfinkel, S. 1994. *PGP : Pretty Good Privacy*. O'Reilly. 2.1.2

[17] Callas, J., Donnerhacke, L., & H. Finney, R. T. 1998. RFC 2440: OpenPGP. `http://www.ietf.org/rfc/rfc2440.txt`. 2.1.2

[18] Password Manager XP. `http://www.cp-lab.com/`. 2.1.2

[19] Secret Keeper. `http://www.hilarytech.com/`. 2.1.2

[20] Cypherus. `http://www.cypherus.com/`. 2.1.2

[21] Anderson, R. & Needham, R. 1995. Advances in Cryptology. *Lecture Notes in Computer Science*, 963, 236–248. Publisher: Springer-Verlag Heidelberg. 2.1.3

[22] Cramer, R. & Shoup, V. 2003. Design and Analysis of Practical Public-Key Encryption Schemes Secure Against Adaptive Chosen Cipertext Attack. *SIAM J. COMPUT.*, 33(1), 167Ű–226. 2.1.3

[23] Fluhrer, S., Mantin, I., & Shamir, A. 2001. Weaknesses in the Key Scheduling Algorithm of RC4. *Lecture Notes in Computer Science*, 2259. Publisher: Springer-Verlag Heidelberg. 2.1.3

[24] Dunn, A. 2003. Environment-independent performance analyses of cryptographic algorithms. In *Proceedings of the twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, 265–274. Australian Computer Society, Inc. 2.1.3

[25] Guttman, E., Leong, L., & Malkin, G. 1999. Users' Security Handbook. 2.1.3

[26] Wroblewski, G. 2002. General Method of Program Code Obfuscation. 2.1.4

[27] Dahll, G., Barnes, M., & Bishop, P. December 1990. Software diversity: way to enhance safety? *Information and Software Technology*, 32(10), 677–685. Publisher: Butterworth-Heinemann. 2.1.4

[28] Link, H. & Neumann, W. Clarifying Obfuscation: Improving the Security of White-Box Encoding. 2.1.4

[29] Wang, C., Hill, J., Knight, J., & Davidson, J. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. 2.1.4

[30] Mercuri, R. T. & Neumann, P. G. November 1990. Inside Risks Security by Obscurity. *Communications of the ACM*, 46(11), 160. 2.1.4, 5.1

[31] Chikofsky, E. J. & II, J. H. C. January 1990. Reverse Engineering and Design Recovery: A Taxonomy. 7(1), 13–17. From the IEEE Archive. 2.1.4

[32] van Oorschot, P. C. December 2003. Revisiting Software Protection. *Information Security: 6th International Conference, ISC 2003, Bristol, UK, October 1-3, 2003*, 2851, 1–13. 2.1.4, 2.1.5

[33] Ghosh, A. K. & McGraw, G. 1998. An Approach for Certifying Security in Software Components. *21st National Information Systems Security Conferance, October 5-9, 1998*. 2.1.4

[34] Amy Carroll, Mario Juarez, J. P. & Leininger, T. Microsoft 'Palladium' A Business Overview. Technical report, Microsoft, http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp, August 2002. 2.1.4

[35] Kerckhoffs, A. January, February 1883. La cryptographie militaire. *Journal des sciences militaires*, IX, 5–38, 161–191. 2.2.1

[36] Shannon, C. 1949. Communication theory of secrecy systems. *Bell Systems Techn. Journal*, 28, 656–715. 2.2.1

[37] Shannon, C. 1948. A mathematical theory of communication. *Bell Systems Techn. Journal*, 27, 623–656. 2.2.1

[38] Rivest, R. L., Shamir, A., & Adleman, L. M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, 120–126. 2.2.1

[39] Feistel, H. May 1973. Cryptography and computer privacy. In *Scientific American*, volume 228, 15–23. Springer-Verlag. 2.2.2.1

[40] FIPS PUB 46-2, DATA ENCRYPTION STANDARD (DES). Technical report, National Bureau of Standards, 1993. http://www.itl.nist.gov/fipspubs/fip46-2.htm. 2.2.3.1

[41] Diffie, W. & Hellman, M. June 1977. Exhaustive cryptanalysis of the NBS Data Encryption Standard. *IEEE Computer*, 10(6), 74–84. 2.2.3.1

[42] van Oorschot, P. & Wiener, M. J. 1996. Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude. *Lecture Notes in Computer Science*, 1109, 229–236. 2.2.3.1

[43] Rivest, R. 95, 96. Personal Communication. 2.2.3.1, 2.2.6.1

[44] Merkle, R. C. & Hellman, M. E. 1981. On the security of multiple encryption. *Commun. ACM*, 24(7), 465–467. 2.2.3.1

[45] Kelsey, J., Schneier, B., & Wagner, D. 1996. Key-schedule cryptanalysis of 3-WAY IDEA, G-DES, RC4, SAFER and Triple-DES. In *Proceedings of Crypto 96*, volume 1109 of *Lecture Notes in Computer Science*, 243–253. Springer-Verlag. 2.2.3.1

[46] Lucks, S. 1998. Attacking triple encryption. In *Proceedings of Fast Software Encryption 98*, volume 1372 of *Lecture Notes in Computer Science*, 239–253. Springer-Verlag. 2.2.3.1

[47] Lai, X. & Massey, J. L. April 1990. A proposal for a new block encryption standard. In *Proceedings of Eurocrypt 90*, volume 473 of *Lecture Notes in Computer Science*, 389–404. Springer-Verlag. 2.2.3.2

[48] Biham, E., Biryukov, A., , & Shamir, A. 1999. Miss in the Middle Attacks on IDEA and Khufu. In *Proceedings of Fast Software Encryption 99*, 124–138. Springer-Verlag. 2.2.3.2

[49] Biryukov, A., Jr, J. N., Preneel, B., & Vandewalle, J. 2002. New weak-key classes of IDEA. In *Proceedings of ICICS 02*, volume 2513 of *Lecture Notes in Computer Science*, 315–326. Springer-Verlag. 2.2.3.2

[50] Daemen, J. & Rijmen, V. AES proposal: Rijndael. Selected as the Advanced Encryption Standard. Available from `http://www.nist.gov/aes`. 2.2.3.3, 4.4.1

[51] Daemen, J., Knudsen, L. R., & Rijmen, V. 1997. The block cipher Square. In *Proceedings of Fast Software Encryption 97*, Springer-Verlag, ed, volume 1297 of *Lecture Notes in Computer Science*, 149–165. 2.2.3.3

[52] Knudsen, L. R. & Wagner, D. 1997. Integral cryptanalysis. In *Proceedings of Fast Software Encryption 02*, Springer-Verlag, ed, volume 2365 of *Lecture Notes in Computer Science*, 112–127. 2.2.3.3

[53] Gilbert, H. & Wagner, D. April 2000. A collision attack on seven rounds of Rijndael. In *Proceedings of the Third Advanced Encryption Standard Conference*, NIST, ed, 230–241. 2.2.3.3

[54] Massey, J. L., Khachatrian, G., & Kuregian, M. K. Nomination of SAFER++ as candidate algorithm for the new European Schemes for Signatures, Integrity, and Encryption (NESSIE). Technical report, Cylink Corp., September 2000. 2.2.3.4

[55] Shaked, Y. & Wool, A. June 2005. Cracking the Bluetooth PIN. In *Proceedings of the Third Annual International Conference on Mobile Systems, Applications and Services: MobiSys*. `http://www.eng.tau.ac.il/~yash/shaked-wool-mobisys05/index.html`. 2.2.3.4

[56] Jr, J. N., Preneel, B., , & Wandewalle, J. 2001. Linear cryptanalysis of reduced-round SAFER++. In *Proceedings of the Secound NESSIE Workshop*. 2.2.3.4

[57] Handschuh, H. & Naccache, D. Shacal. Technical report, Gemplus, 2000. 2.2.3.5

[58] Wang, X., Yin, Y. L., & Yu, H. Collision Search Attacks on SHA1. `http://cryptome.org/sha1-attacks.htm`, February 2005. 2.2.3.5

[59] Biham, E., Keller, N., & Dunkelman, O. 2003. Rectangle attacks on SHACAL-1. In *Proceedings of Fast Software Encryption 03*, volume 2887 of *Lecture Notes in Computer Science*. Springer-Verlag. 2.2.3.5

[60] Englund, H. & Johansson, T. February 2005. A New Distinguisher for Clock Controlled Stream Ciphers. Proceedings in Fast Software Encryption 2005. 2.2.4, 2.2.5.4

[61] Rueppel, R. A. 1986. Analysis and Design of stream ciphers. Springer-Verlag. 2.2.4, 3.2.2

[62] Blum, L., Blum, J., & Shub, M. May 1986. A simple unpredictable pseudorandom number generator. *SIAM Journal on Computing*, 15(2), 364–383. 2.2.4.1

[63] Biryukov, A. & Shamir, A. 2000. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Proceedings of Asiacrypt 00*, volume 1976 of *Lecture notes in computer science*, 1–13. Springer-Verlag. 2.2.4.2

[64] Massey, J. January 1969. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, IT-15, 122–127. 2.2.4.2, 3.2.2

[65] Siegenthaler, T. 1985. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, 34(1), 81–85. 2.2.4.2

[66] Meier, W. & Staffelbach, O. 1989. Fast correlation attacks on certain stream ciphers. *J. Cryptol.*, 1(3), 159–176. 2.2.4.2

[67] Courtois, N. T. 2002. Higher order correlation attacks, XL algorithm, and cryptoanalysis of Toyocrypt. In *Proceedings of ICISC 02*, volume 2587 of *Lecture Notes in Information Security*. Springer-Verlag. 2.2.4.2

[68] Ekdahl, P. & Johansson, T. 2000. SNOW: a new stream cipher. In *Proceedings of First open NESSIE Workshop*. 2.2.5.1

[69] Coppersmith, D., Halevi, S., & Jutla, C. S. 2002. Cryptanalysis of stream ciphers with linear masking. In *Proceedings of Crypto 02*, volume 2442 of *Lecture Notes in Computer Science*, 512–532. Springer-Verlag. 2.2.5.1

[70] Hawkes, P. & Rose, G. G. 2002. Guess-and-determine attacks on SNOW. In *Proceedings of Selected Areas in Cryptography 02*, volume 2595 of *Lecture Notes in Computer Science*. Springer-Verlag. 2.2.5.1

[71] Ekdahl, P. & Johansson, T. 2002. A new version of the stream cipher SNOW. In *Proceedings of Selected Areas in Cryptography 02*, Lecture Notes in Computer Science, 47–61. Springer-Verlag. 2.2.5.1

[72] Hawkes, P. & Rose, G. G. SOBER. Technical report, September 2000. Primitive submitted to NESSIE by Qualcomm International. 2.2.5.2

[73] Ekdahl, P. & Johansson, T. 2002. Distinguishing attacks on SOBER-t16 and t32. In *Proceedings of Fast Software Encryption 02*, volume 2365 of *Lecture Notes in Computer Science*, 210–224. Springer-Verlag. 2.2.5.2

[74] Cannière, C. D., Lano, J., Preneel, B., & Vandewalle, J. 2002. Distinguishing attacks on SOBER-t32. In *Proceedings of he Third NESSIE Workshop*. 2.2.5.2

[75] Rivest, R. L. The RC4 encryption algorithm. Technical report, RSA Security Inc, March 1992. 2.2.5.3, 4.4.1

[76] Mantin, I. & Shamir, A. 2002. A Practical Attack on Broadcast RC4. In *Proceedings of Fast Software Encryption 01*, volume 2355 of *Lecture Notes in Computer Science*, 152. Springer-Verlag. 2.2.5.3

[77] Fluhrer, S., Mantin, I., & Shamir, A. 2002. Weaknesses in the key scheduling algorithm of RC4. In *Proceedings of Selected Areas in Cryptography 01*, volume 2259 of *Lecture Notes in Computer Science*, 1–24. Springer-Verlag. 2.2.5.3

[78] Fluhrer, S., Mantin, I., & Shamir, A. 2003. Atacks on RC4 and WEP. 2.2.5.3

[79] Dawson, E., Clark, A., Golic, J., Millan, W., Penna, L., & Simpson, L. 2000. The LILI-128 keystream generator. 2.2.5.4

[80] Simpson, L., Dawson, E., Golic, J. D., & Millan, W. 2000. LILI keystream generator. In *Proceedings of the Seventh Annual Workshop on Selected Areas in Cryptology 00*, volume 2384 of *Lecture Notes in Computer Science*, 25–39. Springer-Verlag. 2.2.5.4

[81] Babbage, S. 2001. Cryptanalysis of the LILI-128 stream cipher. In *Proceedings of the Second NESSIE Workshop 01*. 2.2.5.4

[82] Saarinen, M.-J. O. 2002. A time-memory trade-off attack against LILI-128. In *Proceedings of Fast Software Encryption 02*, volume 2365 of *Lecture Notes in Computer Science*, 231–236. Springer-Verlag. 2.2.5.4

[83] Jönsson, F. & Johansson, T. A fast correlation attack on LILI-128. Technical report, Lund University, 2001. 2.2.5.4

[84] Clark, A., Dawson, E., Fuller, J., Golic, J., Lee, H.-J., Millan, W., Moon, S.-J., & Simpson, L. 2002. The LILI-II keystream generator. In *Proceedings of ACISP 02*, volume 2384 of *Lecture Notes in Computer Science*, 25–39. Springer-Verlag. 2.2.5.4

[85] Charles P. Wright, J. D. & Zadok, E. 2003. Cryptographic File Systems Performance: What You DonŠt Know Can Hurt You. In *Proceedings of the 2003 IEEE Security In StorageWorkshop (SISW 2003)*. 2.2.6

[86] Wright, H. 2001. The Encrypting File System. How Secure is it? 2.2.6.1

[87] Håkan Englund, M. H. & Johansson, T. February. Correlation Attacks Using a New Class of Weak Feedback Polynomials. In *Proceedings of the 2004 Fast Software Encryption*. 3.1.2

[88] Gary, M. & Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. 3.2.1

[89] Golomb, S. W. & Golomb, S. 1981. *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, USA. 3.2.1

[90] Camion, P., Carlet, C., Charpin, P., & Sendrier, N. 1992. On Correlation-Immune Functions. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, 86–100, London, UK. Springer-Verlag. 3.2.3

[91] Gundersen, G. & Steihaug, T. 2004. On the Efficiency of Arrays in C, C# and Java. 3.3

[92] Schneier, B. & Whiting, D. 1997. Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor. In *Proceedings of Fast Software Encryption 97*, volume 1267 of *Lecture Notes in Computer Science*, 242. Springer-Verlag. 3.3

[93] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., & Vo, S. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical Report NIST SP 800-22, National Institute of Standards and Technology, 2001. 4.1, 4.2, 4.2, 4.3, 4.3.4

[94] Rose, G. 1998. A stream cipher based on linear feedback over GF(2^8). In *Proceedings of ACISP 98*, Lecture Notes in Computer Science, 150. Springer-Verlag. 4.4.3

[95] Odlyzko, A. September 2003. Economics, psychology, and sociology of security. In *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, 182–189. Springer-Verlag. 5.1

[96] Schneier, B. 2000. *Secrets & Lies*. Wiley Publishing, Inc. 6

A Source Code

```
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;

namespace TheCipherProject
{
    /// <summary>
    /// Summary description for class LFSR. LFSR is a class which simulates different shift ↙
     registers.
    /// </summary>
    public class LFSR
    {
        /// <summary>
        /// The LFSR struct. Contains the relevant data for a LFSR.
        /// </summary>
        public struct register
        {
            /// <summary>
            /// Register buffer
            /// </summary>
            public uint[] buf;
            /// <summary>
            /// register pol. const.
            /// </summary>
            public int[] con;
            /// <summary>
            /// register taps
            /// </summary>
            public int[] tap;

        //  public register(int BUFFER_LENGTH, int MAX_CONS, int MAX_TAPS)
        //  {
        //      this.buf = new int[BUFFER_LENGTH];
        //      this.con = new int[MAX_CONS];
        //      this.tap = new int[MAX_TAPS];
        //  }
        }

        /// <summary>
        /// An instance of the register struct.
        /// </summary>
        public register myregister;

        /// <summary>
        /// Constructor to initialize The LFSRs
        /// </summary>
        /// <param name="bl">The length of the buffer.</param>
        /// <param name="mc">The number of polynominal constants.</param>
        /// <param name="mt">The number of Taps.</param>
        /// <param name="pol">The specific feedback polynominal constants.</param>
        public LFSR(int bl, int mc, int mt, int [] pol)
        {
            //Initialize the register
            myregister.buf = new uint[bl];
            myregister.con = new int[mc];
            myregister.tap = new int[mt];

            //fixed (register *lfsreg = myregister)
            //{
            //  *lfsreg->buf = new int[bl];
            //}
            //lfsreg =& myregister;


            //Initialize feedback polynomial
            for ( int i = 1; i <= pol.GetLength(0); i++)
            {
                myregister.con[i] = pol[i-1];
            }
        }
        /// <summary>
        /// This function generates a file rndLFSR1.dat which is used to initialze the      ↙
    shift register.
```

```
        /// Testing purposes only.
        /// </summary>
        /// <param name="reg_len"> The Length of the register (X^length).</param>
        public void Init_file(int reg_len)
        {
            MT mt = new MT();
            StreamWriter sw = new StreamWriter ( "rndLFSR1.dat", false ) ;
            for (int i = 1; i <= reg_len; i++)
                {
                    sw.WriteLine( mt.Random() );
                }
            sw.Close();
        }
        /// <summary>
        /// This function generates a file rndLFSR2.dat which is used to initialze the ↙
    shift register.
        /// Testing purposes only.
        /// </summary>
        /// <param name="reg_len"> The Length of the register (X^length).</param>
        public void Init_file2(int reg_len)
        {
            MT mt = new MT();
            StreamWriter sw = new StreamWriter ( "rndLFSR2.dat", false ) ;
            for (int i = 1; i <= reg_len; i++)
            {
                sw.WriteLine( mt.Random() );
            }
            sw.Close();
        }
        /// <summary>
        /// This function generates a file rndLFSR3.dat which is used to initialze the ↙
    shift register.
        /// Testing purposes only.
        /// </summary>
        /// <param name="reg_len"> The Length of the register (X^length).</param>
        public void Init_file3(int reg_len)
        {
            MT mt = new MT();
            StreamWriter sw = new StreamWriter ( "rndLFSR3.dat", false ) ;
            for (int i = 1; i <= reg_len; i++)
            {
                sw.WriteLine( mt.Random() );
            }
            sw.Close();
        }
        /// <summary>
        /// This function generates a file rndLFSR4.dat which is used to initialze the ↙
    shift register.
        /// Testing purposes only.
        /// </summary>
        /// <param name="reg_len"> The Length of the register (X^length).</param>
        public void Init_file4(int reg_len)
        {
            MT mt = new MT();
            StreamWriter sw = new StreamWriter ( "rndLFSR4.dat", false ) ;
            for (int i = 1; i <= reg_len; i++)
            {
                sw.WriteLine( mt.Random() );
            }
            sw.Close();
        }
        /// <summary>
        /// This function initialze the first shift register from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_lfsr1_from_file()
        {
            StreamReader sr = new StreamReader(@"rndLFSR1.dat" ) ;
            int i = 1;
            string str;
            do
            {
                str = sr.ReadLine() ;
                myregister.buf[i++] = System.Convert.ToUInt32(str,10);
            } while ( str != null ) ;

            sr.Close() ;


        }
        /// <summary>
        /// This function initialze the secound shift register from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_lfsr2_from_file()
        {
            StreamReader sr = new StreamReader(@"rndLFSR2.dat" ) ;
            int i = 1;
            string str;
            do
            {
                str = sr.ReadLine() ;
                myregister.buf[i++] = System.Convert.ToUInt32(str,10);

            } while ( str != null ) ;

            sr.Close() ;

        }
        /// <summary>
        /// This function initialze the third shift register from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_lfsr3_from_file()
        {
            StreamReader sr = new StreamReader(@"rndLFSR3.dat" ) ;
            int i = 1;
            string str;
            do
            {
                str = sr.ReadLine() ;
                myregister.buf[i++] = System.Convert.ToUInt32(str,10);

            } while ( str != null ) ;

            sr.Close() ;


        }
        /// <summary>
        /// This function initialze the fourth shift register from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_lfsr4_from_file()
        {
            StreamReader sr = new StreamReader(@"rndLFSR4.dat" ) ;
            int i = 1;
            string str;
            do
            {
                str = sr.ReadLine() ;
                myregister.buf[i++] = System.Convert.ToUInt32(str,10);

            } while ( str != null ) ;

            sr.Close() ;

        }

        /// <summary>
        /// Description for LFSR_shift method. This method shifts the linear shift register ↙
     once.
        /// Only the pointers to the feedback and taps positions are shifted to the left   ↙
    modulo the length of the
        /// buffer.</summary>
        /// <param name="input_byte"> Number of input byte</param>
        /// <param name="lfsr"> The initialized register struct</param>
        /// <param name="nc"> Number of feedback taps</param>
        /// <param name="nt"> Number of output taps</param>
        public uint LFSR_shift( uint input_byte,ref register lfsr, int nc, int nt )
        {
            uint feedback_value;
            int i, buf_len;
            buf_len = lfsr.buf.GetLength(0) -1;
            feedback_value = input_byte;
            //Update the feedback pointers
            for (i = 1; i <= nc;i++)
                {
                    feedback_value ^= lfsr.buf[lfsr.con[i]];
                    lfsr.con[i] = -- lfsr.con[i] & buf_len;
                }
            lfsr.buf[lfsr.con[0]] = feedback_value;
            lfsr.con[0]  = -- lfsr.con[0] & buf_len;
            //Update tap pointers
            for (i = 1;i <= nt; i++)
            {
                lfsr.tap[i] = -- lfsr.tap[i] & buf_len;
            }
            return feedback_value;
        }

    }

    /// <summary>
    /// The General cipher class
    /// </summary>
    public class SAFEII
    {
        const int BUFFER_LENGTH = 256;
        const int MAX_CONS = 10;
        const int MAX_TAPS = 10;
        const int LFSR1_LENGTH = 107;
        const int LFSR2_LENGTH = 127;
        const int LFSR3_LENGTH = 103;
        const int LFSR4_LENGTH = 149;
        const int F_LENGTH = 65536;
        int[] pol1 = {27,62,92,107};
        int[] pol2 = {15,59,62,127};
        int[] pol3 = {34,43,66,103};
        int[] pol4 = {29,64,114,149};
        uint[] m_key = new uint[2];
        uint[] s_key = new uint[8];
        /// <summary>
        /// The lookup boolean/bit table f1.
        /// </summary>
        public bool[] f1;
        /// <summary>
        /// The lookup boolean/bit table f2.
        /// </summary>
        public bool[] f2;
        /// <summary>
        /// The lookup boolean/bit table f11.
        /// </summary>
        public bool[] f11;
        /// <summary>
        /// The lookup boolean/bit table f21.
        /// </summary>
        public bool[] f21;
        LFSR mylfsr1;
        LFSR mylfsr2;
        LFSR mylfsr3;
        LFSR mylfsr4;
        /// <summary>
        /// Constructor to start an encryption/decryption process
        /// </summary>
        public SAFEII()
        {
            //Initialize the first LFSR.
            mylfsr1 = new LFSR(BUFFER_LENGTH,MAX_CONS,MAX_TAPS, pol1);
            //Initialise the Secound secound LFSR
            mylfsr2 = new LFSR(BUFFER_LENGTH,MAX_CONS,MAX_TAPS, pol2);
            //Initialize the third LFSR.
            mylfsr3 = new LFSR(BUFFER_LENGTH,MAX_CONS,MAX_TAPS, pol3);
            //Initialise the fourth LFSR
            mylfsr4 = new LFSR(BUFFER_LENGTH,MAX_CONS,MAX_TAPS, pol4);

            //Initilize the look up tables
            f1 = new bool[F_LENGTH];
            f2 = new bool[F_LENGTH];
            f11 = new bool[F_LENGTH];
            f21 = new bool[F_LENGTH];
            /*mylfsr1.Init_file(ref mylfsr1.myregister, LFSR1_LENGTH);
              mylfsr2.Init_file2(ref mylfsr1.myregister, LFSR2_LENGTH);*/

            //Initialise the content of the first LFSR from file
            //mylfsr1.Init_lfsr1_from_file();
            //Initialise the content of the Secound IClocking  LFSR from file
            //mylfsr2.Init_lfsr2_from_file();

            //Initialize the bool array to simulate the f1 table from file
            Init_f1();
            //Initialize the bool array to simulate the f2 table from file
            Init_f2();
            //Initialize the bool array to simulate the f11 table from file
            Init_f11();
            //Initialize the bool array to simulate the f21 table from file
            Init_f21();


        }
        /// <summary>
        /// This function rotates a 32 bit source steps positions to the right
        /// </summary>
        /// <param name="source">The source to rotate</param>
        /// <param name="steps">Number of steps to rotate</param>
        /// <returns></returns>
        uint rotate_right(uint source, int steps)
        {
            uint tmp = source >> steps;
            uint tmp2 = Convert.ToUInt32(tmp | (source << (32 - steps))) ;
            return tmp2;
        }
        /// <summary>
        /// This function generates the 32 bit output from the primitive s1
        /// </summary>
        /// <param name="feedback_value">The feedback value from the first LFSR</param>
        /// <returns></returns>
        uint generate_output_from_s1(uint feedback_value)
        {
            uint feedback = mylfsr2.LFSR_shift(0,ref mylfsr2.myregister, 4,1);
            uint addr1 = (feedback_value & 0xffff0000)>>16 ;
            uint addr2 = feedback_value & 0x0000ffff ;
            return (Gen_output_f1(addr1) ^ Gen_output_f2(addr2)) == 0? feedback : mylfsr2. ↙
    LFSR_shift(0,ref mylfsr2.myregister, 4,1);
        }
        /// <summary>
        /// This function generates the 32 bit output from the primitive s2
        /// </summary>
        /// <param name="feedback_value">The feedback value from the third LFSR</param>
        /// <returns></returns>
        uint generate_output_from_s2(uint feedback_value)
        {
            uint feedback = mylfsr4.LFSR_shift(0,ref mylfsr4.myregister, 4,1);
            uint addr1 = (feedback_value & 0xffff0000) >>16 ;
            uint addr2 = feedback_value & 0x0000ffff ;
            return (Gen_output_f11(addr1) ^ Gen_output_f21(addr2)) == 0 ? feedback :       ↙
    mylfsr4.LFSR_shift(0,ref mylfsr4.myregister, 4,1);
        }
        /// <summary>
        /// Generate 32 bit output from the complete generator.
        /// </summary>
        /// <returns>Returns one 32 bit output from the generator.</returns>
        ulong generate_final_keystream_output()
        {
            return Convert.ToUInt32(Convert.ToUInt64(generate_output_from_s1(mylfsr1.       ↙
    LFSR_shift(0,ref mylfsr1.myregister,4,1)) + generate_output_from_s2(mylfsr3.LFSR_shift ↙
    (0,ref mylfsr3.myregister,4,1))) % 4294967296) ;
        }
        /// <summary>
        /// Read the keys into the keystream class
        /// </summary>
        /// <param name="s">referance to the keystream class</param>
        void read_keys(ref SAFEII s)
        {
            StreamReader sr = new StreamReader(@"m_key.dat" ) ;
            string str;
            str = sr.ReadLine();
            if (str != null)
            {
                s.m_key[0] = Convert.ToUInt32(str.Substring(0,32),2);
                s.m_key[1] = Convert.ToUInt32(str.Substring(32,32),2);
            }
            sr.Close() ;
            sr = new StreamReader(@"s_key.dat");
            str = sr.ReadLine();
            if (str != null)
            {
                int pos = 0;
                for(int i = 0; i < 8; i++)
                {
                    s.s_key[i] = Convert.ToUInt32(str.Substring(pos,32),2);
                    pos += 32;
                }
            }
            sr.Close();
        }
        /// <summary>
        /// Initialize the primitives based on the secret key and the message key
        /// </summary>
        /// <param name="s">a referance to the keystream class</param>
        void init_primitives(ref SAFEII s)
        {
            int i,j,s12 = LFSR1_LENGTH + LFSR2_LENGTH, s123 = s12 + LFSR3_LENGTH;
            uint[] c = new uint[512] ;
            for(i = 0; i < 2; i++)
                for (j = 0; j < 8; j++)
                    c[i*8+j] = s.m_key[i] ^ s.rotate_right(s_key[j], j);
            for(i=16;i<486;i++)
                c[i] = c[i-16] ;
            for(i=1;i<=LFSR1_LENGTH;i++)
                mylfsr1.myregister.buf[i] = c[i-1];
            for(i=1;i<=LFSR2_LENGTH;i++)
                mylfsr2.myregister.buf[i] = c[LFSR1_LENGTH+i-1];
            for(i=1;i<=LFSR3_LENGTH;i++)
                mylfsr3.myregister.buf[i] = c[s12+i-1];
            for(i=1;i<=LFSR4_LENGTH;i++)
                mylfsr4.myregister.buf[i] = c[s123+i-1];
            uint test ;
            for(int k = 0; k < 256; k++)
            {
                s.generate_final_keystream_output();
            }
        }
        /// <summary>
        /// This function initialze the boolean function f1 from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_f1()
        {
            StreamReader sr = new StreamReader(@"f1.dat" ) ;
            int i = 0;
            string str;
            do
            {
                str = sr.ReadLine();
                if (str != null)
                {
                    str = str.Replace(" ","");
                    for (int j = 0; j < str.Length; j++)
                    {
                        f1[i++] = System.Convert.ToBoolean(System.Convert.ToInt16(str.    ↙
    Substring(j,1)));
                    }
                }
            } while ( str != null ) ;

            sr.Close() ;

        }
        /// <summary>
        /// This function initialze the boolean function f2 from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_f2()
        {
            StreamReader sr = new StreamReader(@"f2.dat" ) ;
            int i = 0;
            string str;
            do
            {

                str = sr.ReadLine() ;
                if ( str != null)
                {
                    str = str.Replace(" ","");
                    for (int j = 0; j < str.Length; j++)
                        f2[i++] = System.Convert.ToBoolean(System.Convert.ToInt16(str.    ↙
    Substring(j,1)));
                }
            } while ( str != null ) ;

            sr.Close() ;

        }
        /// <summary>
        /// This function initialze the boolean function f1 from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_f11()
        {
            StreamReader sr = new StreamReader(@"f11.dat" ) ;
            int i = 0;
            string str;
            do
            {
                str = sr.ReadLine();
                if (str != null)
                {
                    str = str.Replace(" ","");
                    for (int j = 0; j < str.Length; j++)
                    {
                        f11[i++] = System.Convert.ToBoolean(System.Convert.ToInt16(str.    ↙
    Substring(j,1)));
                    }
                }
            } while ( str != null ) ;

            sr.Close() ;

        }
        /// <summary>
        /// This function initialze the boolean function f21 from file.
        /// Testing purposes only.
        /// </summary>
        public void Init_f21()
        {
            StreamReader sr = new StreamReader(@"f21.dat" ) ;
            int i = 0;
            string str;
            do
            {

                str = sr.ReadLine() ;
                if ( str != null)
                {
                    str = str.Replace(" ","");
                    for (int j = 0; j < str.Length; j++)
                        f21[i++] = System.Convert.ToBoolean(System.Convert.ToInt16(str.    ↙
    Substring(j,1)));
                }
            } while ( str != null ) ;

            sr.Close() ;

        }
        /// <summary>
        /// This function returns 1 bit output from the look up table f1
        /// </summary>
        /// <param name="feedback_value">16 bit input from the LFSR</param>
        /// <returns></returns>
        public int Gen_output_f1(uint feedback_value)
        {
            if ( f1[feedback_value] )
                return 1;
            else
                return 0;
        }
        /// <summary>
        /// This function returns 1 bit output from the look up table f2
        /// </summary>
        /// <param name="feedback_value">16 bit input from the LFSR</param>
        /// <returns></returns>
        public int Gen_output_f2(uint feedback_value)
        {
            if ( f2[feedback_value] )
                return 1;
            else
                return 0;
        }
        /// <summary>
        /// This function returns 1 bit output from the look up table f11
        /// </summary>
        /// <param name="feedback_value">16 bit input from the LFSR</param>
        /// <returns></returns>
        public int Gen_output_f11(uint feedback_value)
        {
            if ( f11[feedback_value] )
                return 1;
            else
                return 0;
        }
        /// <summary>
        /// This function returns 1 bit output from the look up table f21
        /// </summary>
        /// <param name="feedback_value">16 bit input from the LFSR</param>
        /// <returns></returns>
        public int Gen_output_f21(uint feedback_value)
        {
            if ( f21[feedback_value] )
                return 1;
            else
                return 0;
        }
```