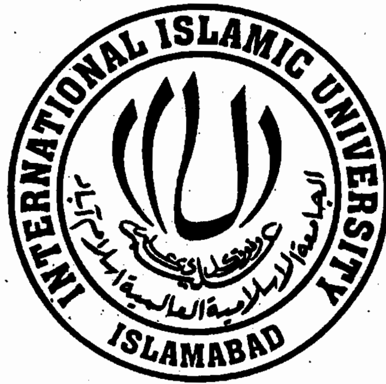
T04125

DIFFERENTIAL KEY ATTACKS ON RABBIT



Developed by

Asmara Safdar

Supervised by

Dr. Malik Sikandar Hayat Khiyal

Department of Computer Science
Faculty of Basic and Applied Sciences
International Islamic University, Islamabad.
2007

Department of Computer Sciences
Faculty of Basic and Applied Sciences
International Islamic University, Islamabad

Final Approval

Dated: Oct 5, 2007

It is certified that we have read the thesis report entitled "**Differential Key Attacks On Rabbit**" submitted by **Ms. Asmara Safdar** Registration Number **272-FBAS/MSCS/,** and it is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad. for the award of MS Degree in Computer Science.

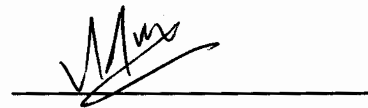
Committee

External Examiner

Dr.M.A Ansari
Associate Professor,
Department of Computer Scienes
Federal Urdu University,
Islamabad

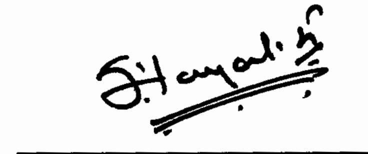
Internal Examiner

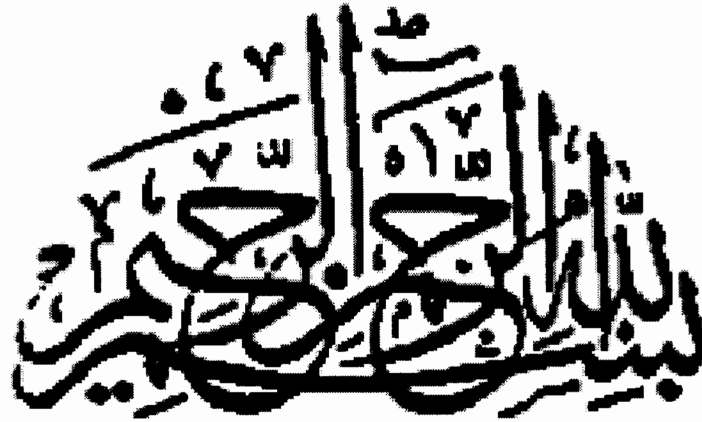
Ms.Muneera Bano
Lecturer,
Department of Computer Sienes
International Islamic Univrversity,
Islamabad.



Supervisor

Dr. Malik Sikander Hayat Khiyal
Professor,
Department of Computer Sienes
Fatima Jinnah University,
The Mall, Rawalpindi.





In The Name of

ALLAH ALMIGHTY

The Most Beneficent, The Most Merciful

A dissertation submitted to the
Department of Computer Science,
Faculty of Basic and Applied Sciences,
International Islamic University, Islamabad
as a partial fulfillment of the requirements for the award of the degree of

MS in Computer Science

To

Holy Prophet,

Prophet Muhammad

(صلى اللهعيه وسلم)

Who is role model for us in every aspect of life...

&

To

My Parents and Family

I am most indebted to my parents and family, whose affection has always been the source of encouragement for me, and whose prayers have always been a key to my success.

Declaration

I hereby declare and affirm that this software neither as a whole nor as a part thereof has been copied out from any source. It is further declared that I have developed this software and accompanied report entirely on the basis of my personal efforts, made under the sincere guidance of my teachers. If any part of this project is proven to be copied out or found to be a reproduction of some other, I shall stand by the consequences.

No portion of the work presented in this report has been submitted in support of an application for other degree or qualification of this or any other University or Institute of learning.

INTERNATIONAL ISLAMIC UNIVERSITY ISLAMABAD

Asmara Safdar

272-FBAS/MSCS/F-05

Acknowledgement

I bestow all praises to, acclamation and appreciation to Almighty Allah, The Most Merciful and Compassionate, The Most Gracious and Beneficent, Whose bounteous blessings enabled me to pursue and perceive higher ideals of life, Who bestowed me good health, courage, and knowledge to carry out and complete my work. Special thanks to our **Holy Prophet Muhammad (SAW)** who enabled us to recognize our Lord and Creator and brought us the real source of knowledge from Allah (SWT), the Qur'ān, and who is the role model for us in every aspect of life.

I reckon it a proud privileged to express my deepest gratitude and deep sense obligation to my reverend supervisor **Dr. Malik Sikandar Hayat Khiyal** who kept my morale high by his suggestions and appreciation. His motivation led me to this success. Without his sincere and obliging nature and precious guidance; I could never have been able to complete this task.

It will not be out of place to express my profound admiration and gratitude for my teacher **Mr. Syed Irfan-ul-Allah** for his dedication, untiring help and kind behavior throughout the project efforts and presentation of this manuscript. He did a lot of effort for my success.

Finally I must mention that it was mainly due to moral support and financial help of my father, **Safdar Hussain Randhawa**, during my entire academic career that made it possible for me to complete my work dedicatedly. I'm obliged to my wiser and great father who gave me permission to study herein the institute to vast my vision and perk up my Knowledge. I owe all achievements to my most loving mother, who means most to me, for her prayers are more precious before any treasure on the earth. I'm also thankful to **Mr.Jawad Hussain Randhawa**, my elder brother, for his moral support and help during the project, and to my loving brothers, caring sisters, sincere friends (AASSK &SJHS), and helpful class fellows whose prayers and apparently trivial yet great help have always been moral fiber for me.

Asmara Safdar

272-FBAS/MSCS/F05

PROJECT IN BRIEF

Project Title:	**Differential Key Attacks on Rabbit**
Organization	International Islamic University, Islamabad
Under Taken By:	**Asmara Safdar** **272-FBAS/MSCS/F05**
Supervised By:	**Dr.Malik Sikandar Hayat Khiyal.**
Starting Date:	**October, 2006.**
End Date:	**2007**
Tools used:	**Turbo C++ 3.0, Visual C++ 6.0**
Opereating System:	**Microsoft Windows XP professional**
System Used:	Pentium lV

ABSTRACT

With the explosive growth in information technology, the threats to security of the data being sent and received is increasing day by day. The awareness of users about the importance of confidentiality of the data has made them more demanding for their security. Now they want their data be encrypted before it's transferred for security purposes. So, different encryption algorithms have been developed for this purpose. These Algorithms take data as an input and give output with almost entirely different form of data to meet the purpose of confidentiality.

This Dissertation is concerned with Differential cryptanalysis of Rabbit. Stream cipher is one of the branches of Encryption Techniques. It has been claimed as one of the most secure stream ciphers. So far many different types of attacks have been designed to break it or compromise its security but there has been no remarkable success.

We propose a special type of differential key analysis based upon chosen plaintext attack that works in bottom up approach in such a way to break or recover a part of the key and then to proceed further. We believe that such type of attack is more beneficial to recover a key in full or partially.

Contents

Declaration iii

Acknowledgement iv

Abstract vi

1. INTRODUCTION 1

1.1 CRYPTOGRAPHY 2

1.1.1 Cryptographic goals 2

1.2 BASIC TERMINOLOGY AND CONCEPTS 4

1.2.1 Encryption domains and co domains 4

1.2.2 Encryption and decryption transformations 4

1.2.3 Achieving confidentiality 5

1.3 SYMMETRIC KEY ENCRYPTION 8

1.3.1 Substitution ciphers and transposition ciphers 8

1.3.2 Stream ciphers 8

1.4 DIGITAL SIGNATURES 13

1.5 PUBLIC-KEY ENCRYPTION 13

1.5.1 Symmetric-key vs. public-key cryptography 14

1.6 CLASSES OF ATTACKS AND SECURITY MODELS 15

1.6.1 Main Categories 15

1.6.2 Generic attacks on stream ciphers 16

1.6.1 Attacks on encryption schemes 17

1.6.2 Models for evaluating security 18

1.6.3 Perspective for computational security 20

2. LITERATURE SURVEY 21

2.1 Rabbit: A New High-Performance Stream Cipher 22

2.1.1 Security Analysis 23

2.1.1.1 Key Setup Properties 23

2.1.1.2 Counter Properties 23

2.1.1.3 Algebraic Analysis 24
2.1.1.4Linear Correlation Analysis 24
2.1.1.5 Statistical Tests 25
2.1.2 Resulting Attacks 25
2.1.2.1 Attacks on the Key Setup Function 25
2.1.3 Performance 26
2.1.4 Conclusion 26
2.1.5 Disadvantage 26
2.3 Cryptanalytic weakness in modern stream ciphers and recommendations for improving their security levels 26
2.3.1 Changes in Rabbit 27
2.3.2 Disadvantage 28
2.4. The Rectangle Attack-Rectangling the Serpent 28
2.4.1 Working 28
2.4.1.1 Boomerang Attack 28
2.4.1.2 Rectangling the Boomerang 28
2.4.2 Disadvantage 29
2.5 Problem Domain and Objective to Attain 29
2.5.1 Problem Statement 29
2.5.2 Objective to Attain 30
2.5.2.1 Differential key analysis 30
2.5.2.2 Ladder Approach 30
3. RESEARCH METHODOLOGIES 31
3.1 The Design of Rabbit 31
3.1.1The Cipher Algorithm 31
3.1.1.1 Key Setup Scheme 31
3.1.1.2 Next-state Function 32
3.1.1.3 Counter System 33
3.1.1.4 Extraction Scheme 34
3.1.1.5 Encryption/decryption Scheme 34
3.3 Our Approach to Attacks 34

3.3.1 Ladder Approach 34
3.3.1.1 Initial guess 35
3.3.1.2 Strong Correlation between the states 35
3.3.1.3 Weak encryption keys 35
3.3.1.4 Strong chosen plaintext 36
3.3.2 Limits of my attack 36
4. IMPLEMENTATION 37
4.1 Turbo C++ 3.0 37
4.2 Microsoft Visual C++ 6.0 37
4.3 Implementation of the Attack 38
4.3.1 Implementation in Turbo C++ 38
4.3.1The g-Function 38
4.3.2 The next_state Function 39
4.3.3 The Key Set Up Function 41
4.3.4The Cipher Function 43
4.3.6 Binary Function 46
4.3.7 The main function 47
4.3.8 Variables: 48
4.3.2 Implementation In Visual C++ 48
5 ANALYSIS AND RESULTS 38
5.1 In Turbo C++ 38
5.1.1 XOR'ing the int variables in Turbo C++ 38
5.1.2 The Affect, on the State Variables, of Calling Next State Function More Than Once 40
5.1.4 Repetition of Output key value from the different output 46
5.1.4.1 Single Input key 46
5.1.4.2 Character array of 8 elements 46
5.1.5 Affect of Length of Input Text on the cipher function 46
5.2 Implementation in Visual C++ 48
5.2.1 XOR'ing the int variables in VC++ 48
5.2.4 Repetition of Output key value from the different input. 51

6. CONCLUSIONS AND FUTURE ENHANCEMENT 52

6.1Conclusion: 52

6.1.1 Implementation in Turbo C++ 52

6.1.2 Implementation in VC++ 54

6.2Future enhancements: 54

Chapter 1

Introduction

1. Introduction

The concept of information will be taken to be an understood quantity. To introduce cryptography, an understanding of issues related to information security in general is necessary. Information security manifests itself in many ways according to the situation and requirement. Regardless of who is involved, to one degree or another, all parties to a transaction must have confidence that certain objectives associated with information security have been met. Some of these objectives are listed in Table 1.1 [23].

Privacy or confidentiality	Keeping information secret from all but those who are authorized to see it.
Data integrity	Ensuring information has not been altered by unauthorized or unknown means.
Entity authentication or identification	Corroboration of the identity of an entity (e.g., a person, a computer terminal, a credit card, etc.)
Message authentication	Corroborating the source of information; also known as data origin authentication.
Signature	A means to bind information to an entity.
Authorization	Conveyance, to another entity, of official sanction to do or be something.
Validation	A means to provide timeliness of authorization to use or manipulate information or resources.
Access control.	restricting access to resources to privileged entities
Certification	Endorsement of information by a trusted entity.
Time stamping	Recording the time of creation or existence of information.
Witnessing	Verifying the creation or existence of information by an entity other than the creator
Receipt	Acknowledgement that information has been received.
Confirmation	Acknowledgement that services have been provided.
Ownership	A means to provide an entity with the legal

	right to use or transfer a resource to others.
Anonymity	Concealing the identity of an entity involved in some process
Non-repudiation	Preventing the denial of previous commitments or actions.
Revocation	Retraction of certification or authorization

Table1.1: Some information security objectives. [23]

1.1 Cryptography

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques.

1.1.1 Cryptographic goals

Of all the information security objectives listed in Table 1.1, the following four form a framework upon which the others will be derived: [23]

- **Confidentiality** is a service used to keep the content of information from all but those authorized to have it. Secrecy is a term synonymous with confidentiality and privacy. There are numerous approaches to providing confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.
- **Data integrity** is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such things as insertion, deletion, and substitution.
- **Authentication** is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. For these reasons this aspect of cryptography is usually subdivided into two major classes: entity

authentication and data origin authentication. Data origin authentication implicitly provides data integrity (for if a message is modified, the source has changed).

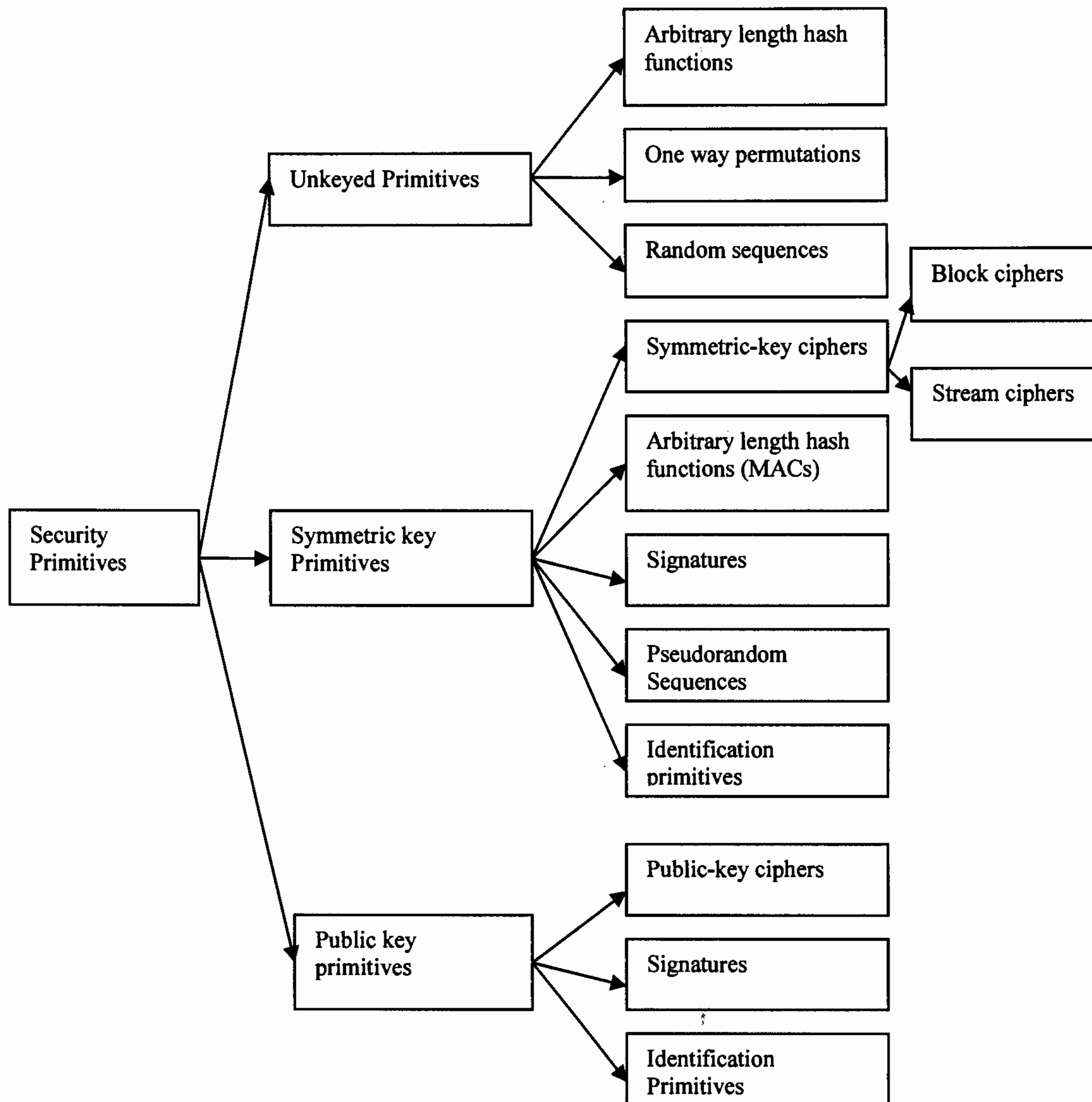


Figure 1.1: A taxonomy of cryptographic primitives. [23]

- **Non-repudiation** is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary.

A fundamental goal of cryptography is to adequately address these four areas in both theory and practice. Cryptography is about the prevention and detection of cheating and other malicious activities.

1.2 Basic terminology and concepts

The scientific study of any discipline must be built upon rigorous definitions arising from fundamental concepts.

1.2.1 Encryption domains and co domains

- $\boldsymbol{A}$ denotes a finite set called the *alphabet of definition.* For example, $\boldsymbol{A} = \{0, 1\}$, the binary alphabet, is a frequently used alphabet of definition. Note that any alphabet can be encoded in terms of the binary alphabet. For example, since there are 32 binary strings of length five, each letter of the English alphabet can be assigned a unique binary string of length five.
- $\boldsymbol{M}$ denotes a set called the message space. $\boldsymbol{M}$ consists of strings of symbols from an alphabet of definition. An element of $\boldsymbol{M}$ is called a *plaintext message* or simply a *plaintext*. For example, $\boldsymbol{M}$ may consist of binary strings, English text, computer code, etc.
- $\boldsymbol{C}$ denotes a set called the *ciphertext space.* $\boldsymbol{C}$ consists of strings of symbols from an alphabet of definition, which may differ from the alphabet of definition for $\boldsymbol{M}$. An element of $\boldsymbol{C}$ is called a *ciphertext.*[23]

1.2.2 Encryption and decryption transformations

- $\boldsymbol{K}$ denotes a set called the *key space*. An element of $\boldsymbol{K}$ is called a *key*.
- Each element $e \in \boldsymbol{K}$ uniquely determines a bijection from $\boldsymbol{M}$ to $\boldsymbol{C}$, denoted by $\boldsymbol{E_e}$.
  $\boldsymbol{E_e}$ is called an *encryption function* or an *encryption transformation.* Note that $\boldsymbol{E_e}$ must be a bijection if the process is to be reversed and a unique plaintext message recovered for each distinct ciphertext.
- For each $\boldsymbol{d} \in \boldsymbol{K}$, $\boldsymbol{D_d}$ denotes a bijection from $\boldsymbol{C}$ to $\boldsymbol{M}$ (i.e., $\boldsymbol{D_d} : \boldsymbol{C} \rightarrow \boldsymbol{M}$). $\boldsymbol{D_d}$ is called a decryption function or decryption transformation.

- The process of applying the transformation E_e to a message $m \in M$ is usually referred to as *encrypting* m or the *encryption* of m.
- The process of applying the transformation D_d to a ciphertext c is usually referred to as *decrypting* c or the *decryption* of c.
- An encryption scheme consists of a set $\{E_e : e \in K\}$ of encryption transformations and a corresponding set $\{D_d : d \in K\}$ of decryption transformations with the property that for each $e \in K$ there is a unique key $d \in K$ such that $D_d = E_e^{-1}$; that is, $D_d(E_e(m)) = m$ for all $m \in M$. An encryption scheme is sometimes referred to as a *cipher*.
- The keys e and d in the preceding definition are referred to as a *key pair* and sometimes denoted by (e, d). Note that e and d could be the same.
- To *construct* an encryption scheme requires one to select a message space M, a ciphertext space C, a key space K, a set of encryption transformations $\{E_e : e \in K\}$, and a corresponding set of decryption transformations $\{D_d : d \in K\}$

1.2.3 Achieving confidentiality

An encryption scheme may be used as follows for the purpose of achieving confidentiality. Two parties **A** and **B** first secretly choose or secretly exchange a key pair (e; d). At a subsequent point in time, if **A** wishes to send a message $m \in M$ to **B**, she computes $c = E_e(m)$ and transmits this to **B**. Upon receiving c, **B** computes $D_d(c) = m$ and hence recovers the original message m.

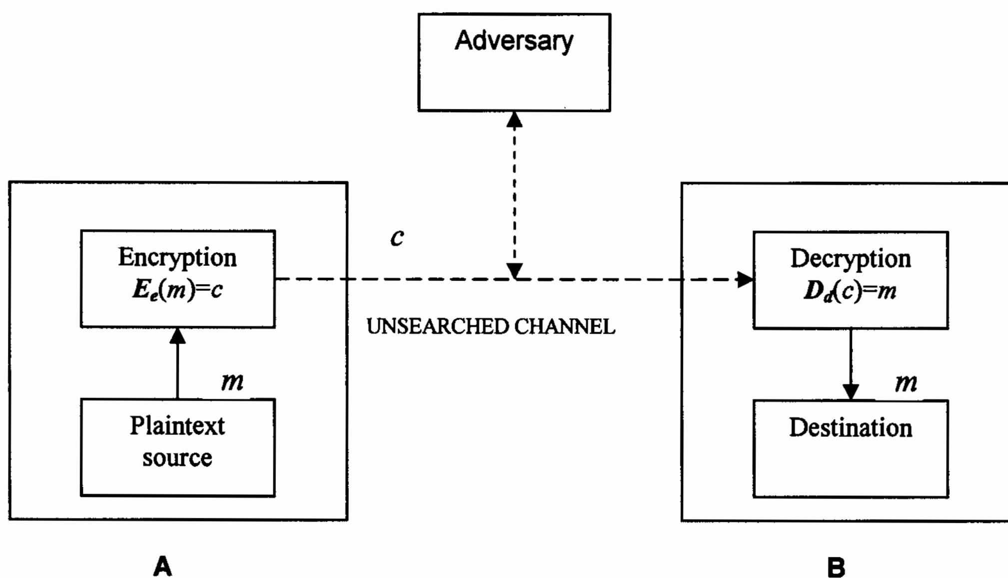


Figure 1.2: Schematic of a two-party communication using encryption. [23]

1.2.3.1 Communication participants

Referring to Figure 1.1, the following terminology is defined. [23]

- An *entity* or *party* is someone or something which sends, receives, or manipulates information. Alice and Bob are entities. An entity may be a person, a computer terminal, etc.
- A *sender* is an entity in a two-party communication which is the legitimate transmitter of information. In Figure 1.6, the sender is Alice.
- A *receiver* is an entity in a two-party communication which is the intended recipient of information. In Figure 1.6, the receiver is Bob.
- An *adversary* is an entity in a two-party communication which is neither the sender nor receiver, and which tries to defeat the information security service being provided between the sender and receiver. Various other names are synonymous with adversary such as enemy, attacker, opponent, tapper, eavesdropper, intruder, and interloper. An adversary will often attempt to play the role of either the legitimate sender or the legitimate receiver.

Channels

- A *channel* is a means of conveying information from one entity to another.
- A *physically secure channel* or secure channel is one which is not physically accessible to the adversary.
- An *unsecured channel* is one from which parties other than those for which the information is intended can reorder, delete, insert, or read.
- A *secured channel* is one from which an adversary does not have the ability to reorder, delete, insert, or read.

Security

A fundamental premise in cryptography is that the sets $\boldsymbol{M}$; $\boldsymbol{C}$;$\boldsymbol{K}$; $\{E_e : \boldsymbol{e} \in \boldsymbol{K}\}$, $\{\boldsymbol{D}_d : d \in \boldsymbol{K}\}$ are public knowledge. When two parties wish to communicate securely using an encryption scheme, the only thing that they keep secret is the particular key pair (e,d) which they are using, and which they must select. One can gain additional security by keeping the class of

encryption and decryption transformations secret but one should not base the security of the entire scheme on this approach. History has shown that maintaining the secrecy of the transformations is very difficult indeed.

Encryption scheme

An encryption scheme is said to be breakable if a third party, without prior knowledge of the key pair (e,d), can systematically recover plaintext from corresponding ciphertext within some appropriate time frame.

An appropriate time frame will be a function of the useful lifespan of the data being protected. For example, an instruction to buy a certain stockmay only needs to be kept secret for a few minutes whereas state secrets may need to remain confidential indefinitely.

An encryption scheme can be broken by trying all possible keys to see which one the communicating parties are using (assuming that the class of encryption functions is public knowledge). This is called an *exhaustive search* of the key space. It follows then that the number of keys (i.e., the size of the key space) should be large enough to make this approach computationally infeasible. It is the objective of a designer of an encryption scheme that this be the best approach to break the system.

Frequently cited in the literature are *Kerckhoffs' desiderata*, a set of requirements for cipher systems. They are given here essentially as Kerckhoffs originally stated them:

1. The system should be, if not theoretically unbreakable, unbreakable in practice;
2. Compromise of the system details should not inconvenience the correspondents;
3. The key should be rememberable without notes and easily changed;
4. The cryptogram should be transmissible by telegraph;
5. The encryption apparatus should be portable and operable by a single person; and
6. The system should be easy, requiring neither the knowledge of a long list of rules nor mental strain.

This list of requirements was articulated in 1883 and, for the most part, remains useful today. Point 2 allows that the class of encryption transformations being used be publicly known and that the security of the system should reside only in the key chosen.

Cryptology

- *Cryptanalysis* is the study of mathematical techniques for attempting to defeat cryptographic techniques, and, more generally, information security services.

- A *cryptanalyst* is someone who engages in cryptanalysis.
- *Cryptology* is the study of cryptography and cryptanalysis.
- A *cryptosystem* is a general term referring to a set of cryptographic primitives used to provide information security services. Most often the term is used in conjunction with primitives providing confidentiality, i.e., encryption.

Cryptographic techniques are typically divided into two generic types: symmetric-key and public-key.

1.3 Symmetric Key Encryption

1.3.1 Substitution ciphers and transposition ciphers

Substitution ciphers are block ciphers which replace symbols (or groups of symbols) by other symbols or groups of symbols.

A *block cipher* is an encryption scheme which breaks up the plaintext messages to be transmitted into strings (called *blocks*) of a fixed length t over an alphabet ***A***, and encrypts one block at a time. [23]

Another class of symmetric-key ciphers is the simple transposition cipher, which simply permutes the symbols in a block.

1.3.2 Stream ciphers

Stream ciphers form an important class of symmetric-key encryption schemes. They are, in one sense, very simple block ciphers having block length equal to one. What makes them useful is the fact that the encryption transformation can change for each symbol of plaintext being encrypted. In situations where transmission errors are highly probable, stream ciphers are advantageous because they have no error propagation. They can also be used when the data must be processed one symbol at a time (e.g., if the equipment has no memory or buffering of data is limited).

Definition Let ***K*** be the key space for a set of encryption transformations. A sequence of symbols $e_1 e_2 e_3$ _ _ _ $e_i \in K$, is called a keystream.

1.3.2.1 The key space

The size of the key space is the number of encryption/decryption key pairs that are available in the cipher system. A key is typically a compact way to specify the encryption

transformation (from the set of all encryption transformations) to be used. For example, a transposition cipher of block length t has $t!$ encryption functions from which to select. Each can be simply described by a permutation which is called the key.

It is a great temptation to relate the security of the encryption scheme to the size of the key space. The following statement is important to remember.

Fact A necessary, but usually not sufficient, condition for an encryption scheme to be secure is that the key space be large enough to preclude exhaustive search.

For instance, the simple substitution cipher in Example 1.25 has a key space of size $26! \approx 4 \times 10^{26}$. The polyalphabetic substitution cipher of Example 1.31 has a key space of size $(26!)^3 \approx 7 \times 10^{79}$. Exhaustive search of either key space is completely infeasible, yet both ciphers are relatively weak and provide little security.

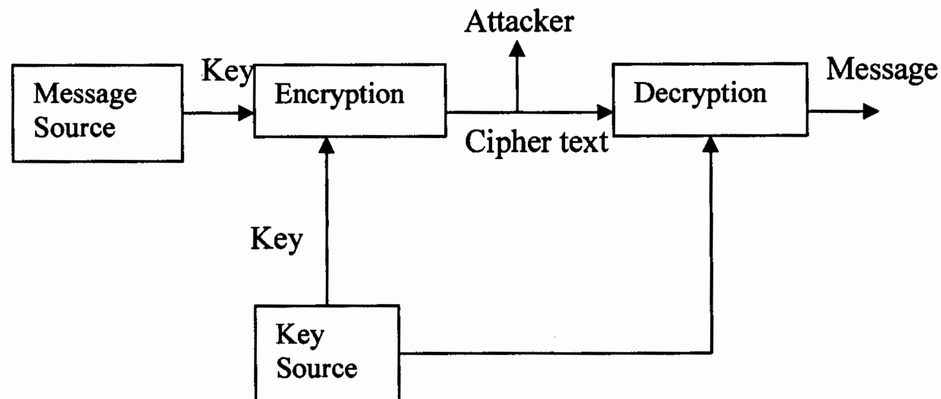


Fig1.1: Symmetric-key secrecy system

Due to Shannon [22], a secrecy system1 between a transmitter and a receiver can be best illustrated by Fig. 1.1. At the transmitting end, the ciphertext is produced by encrypting the message using the key. Upon reception of the ciphertext, it is decrypted using the same key to obtain the original message. As the channel between the transmitter and the receiver is insecure, the ciphertext is subject to falling on the hand of the attacker. The attacker's task is to reconstruct the message from the ciphertext. In order to describe the ideal cryptosystem where knowledge of the ciphertext leaks no information about the message itself to the

attacker, Shannon first introduced the notion of perfect secrecy. Mathematically speaking, let **X**, **Y** be the plaintext and ciphertext respectively. Perfect secrecy means for all **X**, **Y** we always have

$$Pr(Y \mid X) = Pr(Y):$$

Main characteristics of stream ciphers can be summarized as the following:

- **Speed:** faster in hardware,
- **Hardware implementation cost:** low,
- **Error propagation:** limited or no error propagation,
- **Synchronization requirement:** to allow for proper decryption, the sender and receiver must be synchronized (i.e. using the same key and operating at the same position within the key). As detailed in Section 2.1.1, stream ciphers are commonly classified as synchronous stream ciphers and self synchronizing stream ciphers according to their capability of re-establishing proper decryption automatically after loss of synchronization.

1.3.2.2Classification

Stream ciphers can be either symmetric-key or public-key.

A stream cipher generates successive elements of the keystream based on an internal state.

There are different types of stream ciphers [24]:

- **Synchronous stream ciphers**

  A synchronous stream cipher is one in which the keystream is generated independently of the plaintext message and of the ciphertext. That's:

  $$\delta_{i+1} = F1(\delta_i, K)$$

  For example, the OFB mode of a block cipher is a synchronous stream cipher

  It has following distinctive features:

  - Synchronization requirements (In case of loss of synchronization).
  - No error propagation.
  - Active attacks.
- **Self-synchronizing stream ciphers**

A self-synchronizing (a.k.a. asynchronous) stream cipher is one in which the keystream is produced as a function of the key and a fixed number of previous ciphertext digits. The typical mode is the cipher feedback mode

$$\delta_i = F_2(K, y_{i-N}, y_{i-N+1}, :::, y_{i-1})$$

Where N is a constant. For example, a block cipher in one-bit cipher feedback mode is an asynchronous stream cipher. Accordingly, two basic properties of asynchronous stream ciphers include: as the name implies, self-synchronization is enabled in case of loss of synchronization; it suffers limited error propagation only.

1.3.2.3 LFSR-based stream ciphers

Linear feedback shift registers (LFSRs) [24] are popular components in stream ciphers as they can be implemented cheaply in hardware, and their properties are well-understood. The use of LFSRs on their own, however, is insufficient to provide good security[24]. Various schemes have been proposed to increase the security of LFSRs.

- **Non-linear combining functions**

  Use several LFSRs in parallel. The keystream is generated as a nonlinear function f of the outputs of the component LFSRs

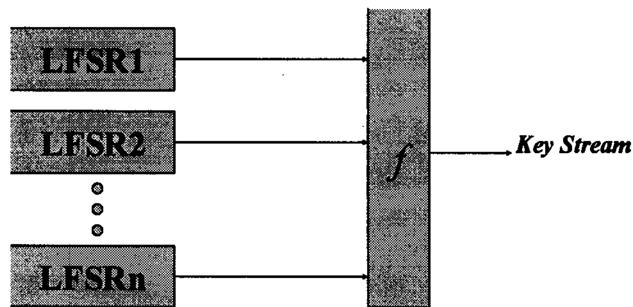


Fig.1.2 LFSR using nonlinear combining function[24]

- **Clock-controlled generators**

  Generate the keystream as some nonlinear function of the stages of a single LSFR

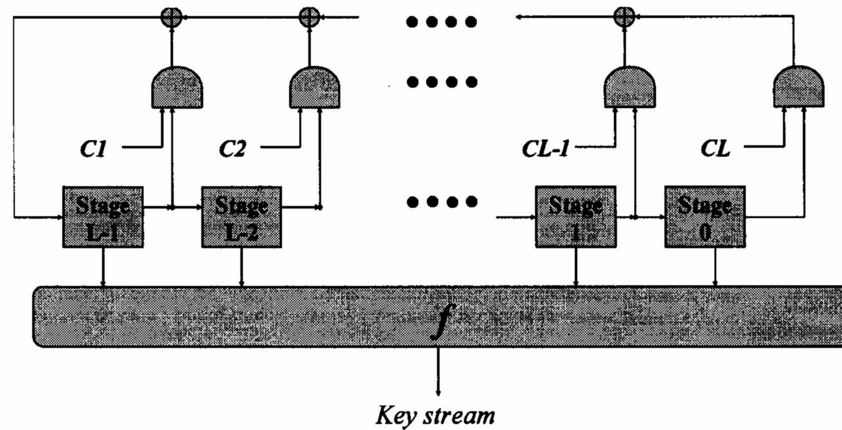


Fig1.3 Clock-controlled generators[24]

- **Filter generator**

  Using the output of one (or more) LFSRs to control the clock of one (or more) other LFSRs

1.3.2.4Necessary Design Principles

Below is provided a collection of necessary, but by no means sufficient design principles known so far for the LFSR-based keystream generator.

- **Pseudo randomness**: The output of the keystream generator should not be distinguishable from a truly random sequence; otherwise, the attacker can not only mount a ciphertext-only attack (to decrypt), but also play a more dangerous game to impersonate the encryptor with high probability of success.

  Pseudo randomness is the common criterion that all keystream generators should comply with.
- **Linear Complexity**: The length of the shortest possible equivalent LFSR to generate a given binary sequence of finite length. The notable Berlekamp- Massey algorithm [22] is a very efficient algorithm to determine the linear complexity of a finite binary sequence of bit length $\boldsymbol{n}$ within $\boldsymbol{O}(\boldsymbol{n}^2)$ bit operations.
- **Nonlinearity**: It was first studied in [18] as a security measure of cryptographic Boolean functions. A function with low nonlinearity is prone to the linear attack or the best affine approximation attack. Note that nonlinearity is also an important parameter for combination generators.

- **Correlation Immunity**: with the advent of Siegenthaler's proposed correlation attacks [11], Siegenthaler proposed the concept of correlation immunity for a cryptographic Boolean function in [20] to describe the existence of the correlation between the minimum number of input variables and output. High correlation immunity implies that many input variables must be considered jointly in the divide-and-conquer scenario, and thus is expected to increase the complexity of correlation attacks.
- **Others**: To resist newly-emerging algebraic attacks, the algebraic degree of a Boolean function should be high.

The problem of how to construct a good Boolean function to achieve the best possible tradeoff among above criteria is difficult, which stimulates lots of research work and has still a long way to go.

1.4 Digital signatures

A cryptographic primitive which is fundamental in authentication, authorization, and nonrepudiation is the *digital signature*. The purpose of a digital signature is to provide a means for an entity to bind its identity to a piece of information. The process of *signing* entails transforming the message and some secret information held by the entity into a tag called a *signature*.

1.5 Public-key encryption

Let $\{E_e : e \in K\}$ be a set of encryption transformations, and let $\{D_d : d \in K\}$ be the set of corresponding decryption transformations, where K is the key space. Consider any pair of associated encryption/decryption transformations $(E_e;D_d)$ and suppose that each pair has the property that knowing E_e it is computationally infeasible, given a random ciphertext $c \in C$, to find the message $m \in M$ such that $E_e(m) = c$. This property implies that given e it is infeasible to determine the corresponding decryption key d. (Of course e and d are simply means to describe the encryption and decryption functions, respectively.) E_e is being viewed here as a trapdoor one-way function (Definition 1.16) with d being the trapdoor information necessary to compute the inverse function and hence allow decryption. This is unlike symmetric-key ciphers where e and d are essentially the same.

1.5.1 Symmetric-key vs. public-key cryptography

Symmetric-key and public-key encryption schemes have various advantages and disadvantages, some of which are common to both. This section highlights a number of these and summarizes features pointed out in previous sections.

1.5.1.1 Advantages of symmetric-key cryptography

1. Symmetric-key ciphers can be designed to have high rates of data throughput. Some hardware implementations achieve encrypts rates of hundreds of megabytes per second, while software implementations may attain throughput rates in the megabytes per second range.
2. Keys for symmetric-key ciphers are relatively short.
3. Symmetric-key ciphers can be employed as primitives to construct various cryptographic mechanisms including pseudorandom number generators, hash functions, and computationally efficient digital signature schemes, to name just a few.
4. Symmetric-key ciphers can be composed to produce stronger ciphers. Simple transformations which are easy to analyze, but on their own weak, can be used to construct strong product ciphers.
5. Symmetric-key encryption is perceived to have an extensive history, although it must be acknowledged that, notwithstanding the invention of rotor machines earlier, much of the knowledge in this area has been acquired subsequent to the invention of the digital computer, and, in particular, the design of the Data Encryption Standard in the early 1970s.

1.5.1.2 Disadvantages of symmetric-key cryptography

1. In a two-party communication, the key must remain secret at both ends.
2. In a large network, there are many key pairs to be managed. Consequently, effective key management requires the use of an unconditionally trusted TTP.
3. In a two-party communication between entities ***A*** and ***B***, sound cryptographic practice dictates that the key be changed frequently and perhaps for each communication session.
4. Digital signature mechanisms arising from symmetric-key encryption typically require either large keys for the public verification function or the use of a TTP.

1.6 Classes of attacks and security models

Over the years, many different types of attacks on cryptographic primitives and protocols have been identified. The discussion here limits consideration to attacks on encryption and, to a small extent, on protocol.

1.6.1 Main Categories

The attacks these adversaries can mount may be classified as follows:

1.6.1.1 Passive attack It's one where the adversary only monitors the communication channel. A passive attacker only threatens confidentiality of data.

1.6.1.2 Active attack It's one where the adversary attempts to delete, add, or in some other way alter the transmission on the channel. An active attacker threatens data integrity and authentication as well as confidentiality.

According to the purpose of the attack, it can be divided into three categories.

1.6.1.3 Distinguishing attack

It's to distinguish the output of the keystream generator from a truly random sequence.

1.6.1.4 Predicting attack

It's to predict the output of the keystream generator with or without a keystream of limited length.

1.6.1.5 Key-recovery attack

It's to obtain the key.

The predicting attack implies a distinguishing attack. The key-recovery attack implies the predicting attack (and hence the distinguishing attack). Obviously, the most powerful attack of all is the key-recovery attack.

Meanwhile, according to the assumptions of the cryptanalyst, the attack can be either a known-plaintext attack or a ciphertext-only attack. The former implies the keystream is known. The latter essentially involves investigation of the redundancy in the plaintext and thus is application-dependent, which is less common in cryptanalysis. Now, we discuss generic attacks on stream ciphers.

1.6.2 Generic attacks on stream ciphers

1.6.2.1 Time-memory Tradeoff

The attack is not only effective to stream ciphers, but also applicable to block ciphers. In the landmark paper [5] in 1980, it was shown for the first time that a tradeoff can be achieved between time complexity and memory complexity of attacking a general cryptosystem. The idea was lately adjusted in the case of stream ciphers for a tradeoff [6,7] between time, memory and data complexities

1.6.2.2 Guess and Determine

The basic idea is to guess a few part of the key, and use the knowledge of the keystream generator to solve the rest of the key that generates the target keystream. It is especially suitable to attack LFSR-based stream ciphers, where only the states of a few shortest LFSRs are guessed.

1.6.2.3 Algebraic Attack

The algebraic attack [8, 9] is comparatively new in the research literature but has received lots of attention; it is also applicable to block ciphers (e.g. see [10]). In short, when there is a multivariate relation involving only the key and the keystream output, the key can be found by using either the linearization method or XL method to solve the (over defined) system of multivariate equations. The LFSR-based stream ciphers are potentially vulnerable against this attack and it has been successfully demonstrated that the algebraic attack against a series of stream ciphers is very practical and efficient [8, 9]. One major drawback of this method, however, is the difficulty in complexity estimate for both time and data complexity, which arises from the tough underlying problem of solving the equations.

1.6.2.4 Correlation Attack

Vast body of intensive research literature covers this kind of attacks for two decades. Initially targeting at the nonlinear combiners, Siegenthaler first introduced the correlation attacks [11] in the middle of the 1980's. The basic idea is to \divide and conquer" when the keystream output is correlated to the individual LFSR output sequence due to the poor choice of the combining function. That is, instead of the naive exhaustive search on all possible combination of the initial states of the component LFSRs, we only perform an exhaustive search on each individual LFSR independently and test the correlation between each LFSR output sequence and the keystream. The optimum (deterministic) maximum likelihood

decoding strategy yields the answer for the initial state of the LFSR.. Note that by viewing the nonlinear filter generator as the nonlinear combiner with memory, the idea [10] of Siegenthaler's correlation attacks on nonlinear combiners can be applied to attack nonlinear filter generators (e.g. [12, 13]).

Apparently, the time complexity of the basic correlation attack [11] grows exponential in the length of the LFSR, which is impractical for a long LFSR. As a matter of fact, in coding theory, the maximum likelihood decoding problem for linear codes, according to [14], was shown to be NP-complete. The focus of cryptographers has been on the general problem where the individual LFSR may be arbitrarily long. In order to speed up the attack for the general setting, Meier and Staffelbach [15, 16] used the probabilistic iterative decoding strategy to refine the basic correlation attack into a so-called \fast correlation attack" to reconstruct each individual LFSR. A critical factor for the efficiency of the fast correlation attack is the novel use of the multiple polynomial of the LFSR's feedback polynomial with low weight (and low degree).

This fast correlation attack of [15, 16] was improved by a series of variant fast correlation attacks. Recently, various (still probabilistic) decoding techniques have proved very successful to further improve the performance of the fast correlation attack (e.g. [17]).

1.6.1 Attacks on encryption schemes

The objective of the following attacks is to systematically recover plaintext from ciphertext, or even more drastically, to deduce the decryption key.

1.6.1.1 A ciphertext-only attack is one where the adversary (or cryptanalyst) tries to deduce the decryption key or plaintext by only observing ciphertext. Any encryption scheme vulnerable to this type of attack is considered to be completely insecure.

1.6.1.2 A known-plaintext attack is one where the adversary has a quantity of plaintext and corresponding ciphertext. This type of attack is typically only marginally more difficult to mount.

1.6.1.3 A chosen-plaintext attack is one where the adversary chooses plaintext and is then given corresponding ciphertext. Subsequently, the adversary uses any information deduced in order to recover plaintext corresponding to previously unseen ciphertext.

1.6.1.4 An adaptive chosen-plaintext attack is a chosen-plaintext attack wherein the choice of plaintext may depend on the ciphertext received from previous requests.

1.6.1.5 A chosen-ciphertext attack is one where the adversary selects the ciphertext and is then given the corresponding plaintext. One way to mount such an attack is for the adversary to gain access to the equipment used for decryption (but not the decryption key, which may be securely embedded in the equipment). The objective is then to be able, without access to such equipment, to deduce the plaintext from (different) ciphertext.

1.6.1.6 An adaptive chosen-ciphertext attack is a chosen-ciphertext attack where the choice of ciphertext may depend on the plaintext received from previous requests.

1.6.2 Models for evaluating security

The security of cryptographic primitives and protocols can be evaluated under several different models. The most practical security metrics are computational, provable, and ad hoc methodology, although the latter is often dangerous. The confidence level in the amount of security provided by a primitive or protocol based on computational or ad hoc security increases with time and investigation of the scheme. However, time is not enough if few people have given the method careful analysis.

1.6.2.1Unconditional security

The most stringent measure is an information-theoretic measure – whether or not a system has *unconditional security*. An adversary is assumed to have unlimited computational resources, and the question is whether or not there is enough information available to defeat the system. Unconditional security for encryption systems is called *perfect secrecy*.

For perfect secrecy, the uncertainty in the plaintext, after observing the ciphertext, must be equal to the a priori uncertainty about the plaintext – observation of the ciphertext provides no information whatsoever to an adversary.

A necessary condition for a symmetric-key encryption scheme to be unconditionally secure is that the key be at least as long as the message. The one-time pad is an example of an unconditionally secure encryption algorithm. In general, encryption scheme do not offer perfect secrecy, and each ciphertext character observed decreases the theoretical uncertainty in the plaintext and the encryption key. Public-key encryption schemes cannot be

unconditionally secure since, given a ciphertext *c*, the plaintext can in principle be recovered by encrypting all possible plaintexts until *c* is obtained.

1.6.2.2 Complexity-theoretic security

An appropriate model of computation is defined and adversaries are modeled as having polynomial computational power. (They mount attacks involving time and space polynomial in the size of appropriate security parameters.) A proof of security relative to the model is then constructed. An objective is to design a cryptographic method based on the weakest assumptions possible anticipating a powerful adversary. Asymptotic analysis and usually also worst-case analysis is used and so care must be exercised to determine when proofs have practical significance. In contrast, polynomial attacks which are feasible under the model might, in practice, still be computationally infeasible.

Security analysis of this type, although not of practical value in all cases, may nonetheless pave the way to a better overall understanding of security. Complexity-theoretic analysis is invaluable for formulating fundamental principles and confirming intuition. This is like many other sciences, whose practical techniques are discovered early in the development, well before a theoretical basis and understanding is attained.

1.6.2.3 Provable security

A cryptographic method is said to be *provably secure* if the difficulty of defeating it can be shown to be essentially as difficult as solving a well-known and *supposedly difficult* (typically number-theoretic) problem, such as integer factorization or the computation of discrete logarithms. Thus, "provable" here means provable subject to assumptions. [23]

This approach is considered by some to be as good a practical analysis technique as exists. Provable security may be considered part of a special sub-class of the larger class of computational security considered next.

1.6.2.4 Computational security

This measure the amount of computational effort required, by the best currently-known methods, to defeat a system; it must be assumed here that the system has been well-studied to determine which attacks are relevant. A proposed technique is said to be *computationally secure* if the perceived level of computation required to defeat it (using the best attack known) exceeds, by a comfortable margin, the computational resources of the hypothesized adversary.

Often methods in this class are related to hard problems but, unlike for provable security, no proof of equivalence is known. Most of the best known public-key and symmetric key schemes *secure* in current use are in this class. This class is sometimes also called *practical security.*

1.6.2.5 Ad hoc security

This approach consists of any variety of convincing arguments that every successful attack requires a resource level (e.g., time and space) greater than the fixed resources of a perceived adversary. Cryptographic primitives and protocols which survive such analysis are said to have heuristic security, with security here typically in the computational sense. Primitives and protocols are usually designed to counter standard. While perhaps the most commonly used approach (especially for protocols), it is, in some ways, the least satisfying. Claims of security generally remain questionable and unforeseen attacks remain a threat.

1.6.3 Perspective for computational security

To evaluate the security of cryptographic schemes, certain quantities are often considered.
Definition The *work factor* W_d is the minimum amount of work (measured in appropriate units such as elementary operations or clock cycles) required to compute the private key d given the public key e, or, in the case of symmetric-key schemes, to determine the secret key ***k***. More specifically, one may consider the work required under a ciphertext-only attack given n ciphertexts, denoted $W_d(n)$.

If W_d is t years, then for sufficiently large t the cryptographic scheme is, for all practical purposes, a secure system. To date no public-key system has been found where one can prove a sufficiently large lower bound on the work factor W_d

Chapter 2

Literature Survey

2. Literature Survey

Stream ciphers are an important class of encryption algorithms. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation which varies with time [24]. There is a vast body of theoretical knowledge on stream ciphers, and various design principles for stream ciphers have been proposed and extensively analyzed. However, there are relatively few fully-specified stream cipher algorithms in the open literature. This unfortunate state of affairs can partially be explained by the fact that most stream ciphers used in practice tend to be proprietary and confidential. [2]

The goal of cryptanalysis is to find some weakness or insecurity in a cryptographic scheme, thus permitting its subversion or evasion. Cryptanalysis might be undertaken by a malicious attacker, attempting to subvert a system, or by the system's designer (or others) attempting to evaluate whether a system has vulnerabilities, and so it is not inherently a hostile act. **Differential cryptanalysis** is a general form of cryptanalysis applicable primarily to block ciphers, but also to stream ciphers and cryptographic hash functions. *In the broadest sense, it is the study of how differences in an input can affect the resultant difference at the output.*[26] Differential cryptanalysis is usually a chosen plaintext attack, meaning that the attacker must be able to obtain encrypted ciphertexts for some set of plaintexts of his choosing. There are, however, extensions that would allow a known plaintext or even a ciphertext-only attack. The basic method uses pairs of plaintext related by a constant *difference*; difference can be defined in several ways, but the eXclusive OR (XOR) operation is usual. The attacker then computes the differences of the corresponding ciphertexts, hoping to detect statistical patterns in their distribution. The resulting pair of differences is called a differential. Their statistical properties depend upon the nature of the S-boxes used for encryption, so the attacker analyses differentials (Δ_X, Δ_Y), where $\Delta_Y = S(X) \oplus S(X \oplus \Delta_X)$ (and $\oplus$ denotes exclusive or) for each such S-box S. In the basic attack, one particular ciphertext difference is expected to be especially frequent; in this way, the cipher can be distinguished from random. More sophisticated variations allow the key to be recovered faster than exhaustive search.

2.1 Rabbit: A New High-Performance Stream Cipher

Boesgaard et al.[1] represent a design of a secure stream cipher which is efficient in software; The Rabbit Stream Cipher. Rabbit [1] takes a 128-bit secret key as input and generates, for each iteration, an output block of 128 pseudo-random bits from a combination of the internal state bits. The encryption/decryption is carried out by XOR'ing the pseudo-random data with the plaintext/ciphertext. The size of the internal state is 513 bits divided between eight 32-bit state variables, eight 32-bit counters and one counter carry bit. The eight state variables are updated by eight coupled non-linear integer valued functions. The counters secure a lower bound on the period length for the state variables.

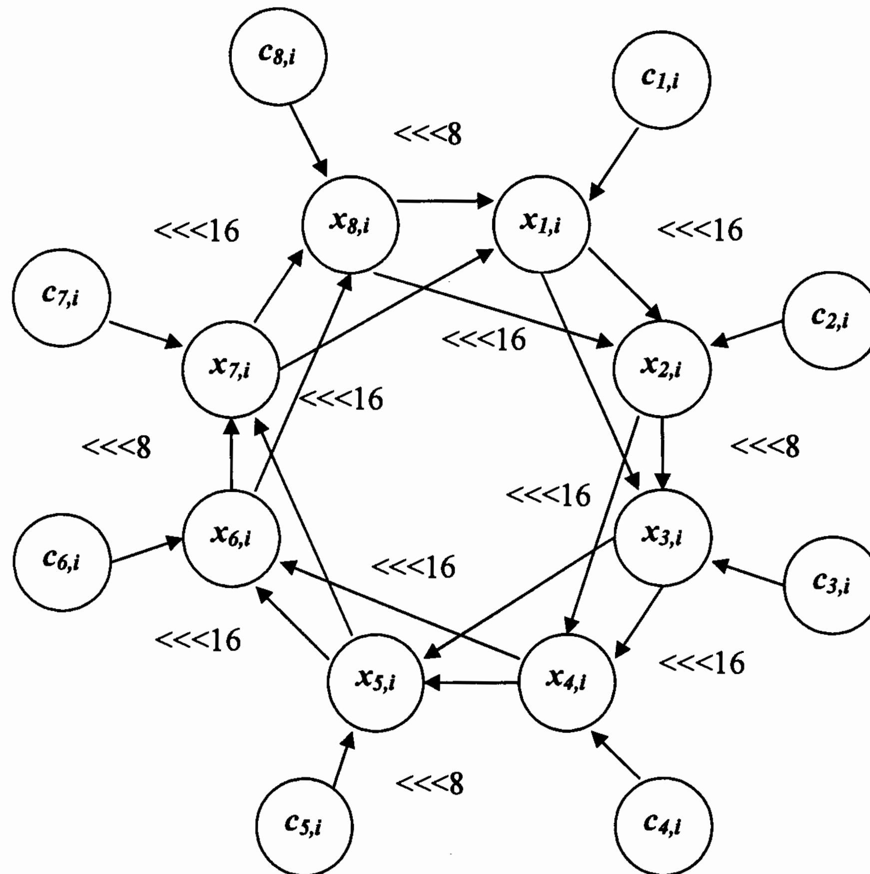


Fig 2.1: Graphical illustration of the system [1]

2.1.1 Security Analysis

2.1.1.1 Key Setup Properties

The setup can be divided into three stages: Key expansion, system iteration and counter modification.[1]

Key Expansion

In the key expansion stage we ensure two properties.

- A one-to-one correspondence between the key, the state and the counter, which prevents key redundancy.
- The other property is that after one iteration of the next-state function, each key bit has affected all eight state variables.

System Iteration

The key expansion scheme ensures that after two iterations of the next-state function; all state bits are affected by all key bits with a measured probability of 0.5. A safety margin is provided by iterating the system four times.

Counter Modification

The counter modification makes it hard to recover the key by inverting the counter system as this would require additional knowledge of the state variables.

2.1.1.2 Counter Properties

- *Period Length*

  The adopted counter system in Rabbit has a period length of 2^{256-1}. Since it can be shown that the input to the g-functions has at least the same period, a highly conservative lower bound on the period of the state variables, $N_x > 2^{158}$, can be secured

- *Probabilities for Bit-flips in the Counters*

  For a 256-bit counter incremented by one, the period length for bit position i is 2^{i+1}. This implies that the least significant bit has a bit-flip probability of 1 and the most significant bit has a bit-flip probability of 2^{-255}.

.The most important findings are as follows.

For the chosen $\boldsymbol{a_j}$ constants bit-flip probabilities for the individual bit positions are all in the interval [0.17; 0.91]. Furthermore, the probabilities are unique for each bit position. Since all

the counter bits have full period and unique bit-flip probabilities, it seems difficult to predict bit patterns of the counter variables.

2.1.1.3 Algebraic Analysis

[1] Analyze a given output byte's dependence on its input bytes. All output bytes of the next-state function depend on the maximal 12 input bytes. Consequently, removing any of those input bytes will result in nearly maximal entropy of error of the output bytes.

2.1.1.4Linear Correlation Analysis

The aim of the correlation analysis is to find the best linear approximations between bits in the input to the next-state function and the extracted output. Each of the eight next-state functions takes three 32-bit state variables and three 32-bit counter values as input and returns the corresponding updated 32-bit state variable. Each bit position in $x_{j,\,i+1}$ defines a binary function from $\{0,\ 1\}^{192}$ to $\{0,\ 1\}$. Thus, assuming that all 192 input bits are independently and uniformly distributed random variables, all correlations from output bits to linear combinations of input bits can be found via the Walsh-Hadamard Transform (WHT) [19, 20].

- *The g-function*

  An exhaustive investigation of all 2^{32} possible convolutions of WHT spectra from the individual output bits in the g-function is not feasible [1]. However, investigations of all convolutions of 16-, 18- and 20-bit g-functions show that the largest resulting correlation coefficients are of similar magnitude as the non-combined output bits and we expect the 32-bit g-function to behave similarly. [1]

- *Linearly Combined Output-Bits*

  It was found that all combinations of output bytes depend on at least four different g-functions which can only be obtained by combining at least five extracted output bits [1]. On the other hand, it was found that by combining two extracted output bits, the least number of g-function dependencies is five. Replacing addition with XOR [1] obtain: The largest corresponding correlation coefficient is $2^{-59.8}$. All other combinations of two output bits depending on five g-functions have smaller correlation coefficients. One of the examples of a linear approximation that only depends on four g-functions has a

largest correlation coefficient of $2^{-59.2}$. All other byte-wise combinations of five output bits depending on four g-functions have smaller correlation coefficients.

2.1.1.5 Statistical Tests

Tests were performed on the internal state as well as on the extracted output. Furthermore, [1] also conducts various statistical tests on the key setup function. Finally, we performed the same tests on a version of Rabbit where each state variable and counter variable was only 8 bit wide. No weaknesses were found in any case. [1]

2.1.2 Resulting Attacks

This subsection discusses relevant attacks based on the above analysis.

2.1.2.1 Attacks on the Key Setup Function

- [1] Conclude that due to the four iterations after key expansion and the final counter modification, both the counter bits and the state bits depend strongly and highly non-linearly on the key bits. This makes attacks based on guessing parts of the key difficult. Furthermore, even if the counter bits were known after the counter modification, it is still hard to recover the key.
- *Divide-and-Conquer Attack:*

  [1] Conclude: It is not possible to verify a guess on fewer bits than the key size.
- *Guess-and-Determine Attack:*

  [1] described the strategy for this attack is to guess a few of the unknown variables of the cipher and from those deduce the remaining unknowns. The attacker must guess more than 128 bits before the determining process can begin, thus, making the attack infeasible.
- *Distinguishing and Correlation Attacks:*

  [1] Narrate that in case of a distinguishing attack the attacker tries to distinguish a sequence generated by the cipher from a sequence of truly random numbers. This attack is also not believed to be feasible.

2.1.3 Performance

The measured encryption/decryption performance was 3.7 clock cycles per byte on a Pentium III processor and 10.5 clock cycles per byte on an ARM7 processor.

2.1.4 Conclusion

In terms of security, Guess and Determine attacks, Divide-and-Conquer attacks as well as Distinguishing and Correlation attacks were considered, but no attack better than exhaustive key search was found. The measured encryption/decryption performance was 3.7 clock cycles per byte on a Pentium III processor and 10.5 clock cycles per byte on an ARM7 processor. [1]

2.1.5 Disadvantage

During the security analysis of Rabbit, [1] conclude that knowing the values of the counters may significantly improve both the Guess-and-Determine attack, the Divide-and-Conquer attacks as well as a Distinguishing attack even though obtaining the key from the counter values is prevented by the counter modification in the setup function.
In terms of security attacks, Guess and-Determine attacks, Divide-and-Conquer attacks as well as Distinguishing and Correlation attacks were considered, but no attack better than exhaustive key search was found [1]. There is mentioned no analysis of Rabbit in terms of differential attacks.

2.3 *Cryptanalytic weakness in modern stream ciphers and recommendations for improving their security levels*

Irfan Ullah et al.[2] discuss security problems in some modern stream ciphers. They [2] discuss SNOW, Scream and Rabbit. Some efforts have been made to overcome the problems those were pointed out in these cryptosystems by different cryptanalysts. The stream ciphers are faster and efficient than block ciphers but comparatively less secure [2]. [2] Make some compromise on efficiency but to get more security.

In cryptanalysis of Rabbit, there are some deficiencies in the model specifically due to some weaknesses in the key scheduling algorithm of the model [2]. There is a possibility of related key attacks that exploit the symmetries of the next state and key setup function.

2.3.1 Changes in Rabbit

[2] Make some changes in its design that generates the key stream as such to minimize the attacks on it.

In the next state functions every X affects the next second state which is easily traceable so [2] make the following changes so that even more organized attacks may fail against this model:

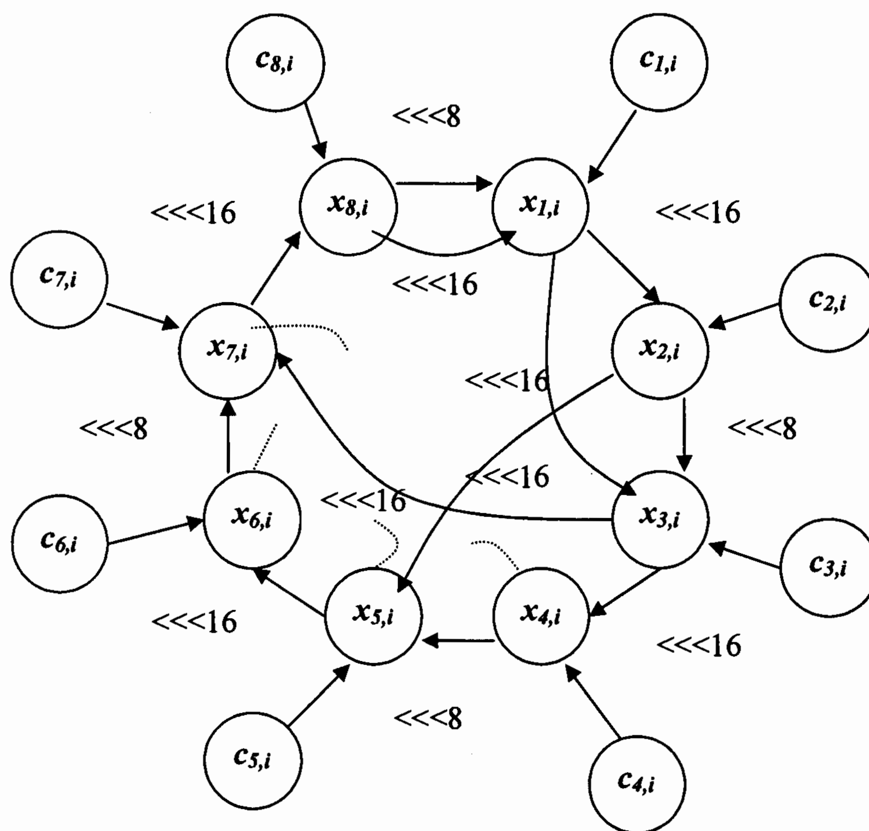


Figure 2.2: The New model of Rabbit [2]

a. Make the map as such that the current state X affects the next X state and that X affects the next second state and that affects the next third state. Thus there is no linear increase in the

state scheduling. At the fifth state the process is repeated once again as for the first four states.

b. The process continues till all states change their values at least once.

2.3.2 Disadvantage

[2] Claims that the differential analysis is impossible because we are not always sure after how much iteration every state is affected, if we increase the number of states for achieving more security the mathematical representation is only possible by making some logics.
Again there is no detailed proof against this claim. Every model leaves some very minor weaknesses those may be considered as ignorance of designers or there may be some other things that every designer had left some peeping point to make it always possible for them to crack their own models.

2.4. The Rectangle Attack-Rectangling the Serpent

Biham et al. [3] introduce different attacks on the Serpent including 10-round attack on 256-bit keys variants that's the best published attack on the cipher. The attack enhances Amplified Boomerang attack and uses better differentials.

2.4.1 Working

Here Serpent is a block cipher with 128-bit block size and key length is 0-256 bit.

2.4.1.1 Boomerang Attack

[3] Gives the main idea behind Boomerang attack; *Use two short differential characteristic instead of one long characteristic.* It's very useful when we have good short differential characteristics and very bad long ones.

2.4.1.2 Rectangling the Boomerang

[3] Gives three steps of improvement:

- Instead of requiring a specific γ ,we can count on all possible γ' values for which $\gamma' \rightarrow \delta$ by E_1
- Instead of discarding pairs with wrong β value, we sort the pairs into piles according to the output difference β of E_0.For each possible pile we perform the original attack.

- Third is based on first 2.We can take into consideration more quartets. Assume that for the first pair The difference α causes some difference a and for the second pair $a \rightarrow b$ we can count also characteristic for which $\gamma \rightarrow \delta$ and $\gamma \oplus a \oplus b \rightarrow \delta$

2.4.2 Disadvantage

Though the improvement in Boomerang attack given by [3] counts all the quartets with plaintext differences α and ciphertext difference β .However it's very hard to do exact calculation.

2.5 Problem Domain and Objective to Attain

2.5.1 Problem Statement

While talking in terms of cryptographic differential attacks in both of the stream ciphers specially Rabbit [1], there is no proof for security. Although Rabbit is stream ciphers and differential attacks are mostly known attacks for block cipher however, they can also be applied to stream ciphers [19]

We consider **Boesgaard et al.** [1] Rabbit model that is one of the state of the art stream ciphers. Many different attacks have been designed to break its security to some extent but no remarkable success has been achieved by any of the designers so far. In other words its security against these many different attacks has almost been confirmed. There have also been designed Differential attacks to menace the security but again there can not be any remarkable success. The reason is the brilliance of the model designers. But this is not the end. History of cryptography shows that many strong ciphers have been attacked differently by different crypt analyzers despite the claims of security by their designers. So we'll consider further such minor weaknesses in the models. Same types of attacks can be applied on the same model differently to analyze different aspects of the cipher. This will further direct towards the more security of the model in terms of the confiscation of flaws being exploited in the analysis.

T-4125

2.5.2 Objective to Attain

As we have studied techniques of Biham et al. [3] to implement differential attacks. Though this technique is designed for differential cryptanalysis of block ciphers, we'll use this to see its impact on the described stream cipher. For this we have to come to know about the small differential properties of Rabbit [1] model. We'll consider this and some other ways to try to take advantage of hidden flaws in Rabbit and hence its security.

To get the differential properties of the model [1] we will perform:

2.5.2.1 Differential key analysis

Although the key generated is truly random however, we will try to see the differences in the out put key by the input key values and also try to correlate the certain internal state values.

2.5.2.2 Ladder Approach

We'll try to analyze the model with Ladder Approach. In this approach, we'll divide the key into 8bits and then analyze them by examining different values against the input key values and the correlation between the different octets of the key generated and the input key, and also between the various state variables during key generation process.

The assumptions about the ladder approach are:

Initial guess: We modify cipher to plaintext and on the basis of similarity we modify our guess.

Strong Correlation Equations: Our equations give us recursive result (***P***) in bottom up approach. The difference is, we store the previous result also. After production of every state we are going to find out the correlation b/w actual chosen plaintext and recovered chosen plaintext ***P'***, ***P"***

Weak encryption keys: We apply weak encryption keys and check the attack and see after how much attempts we are supposed to recover the key.

Strong chosen plaintext: We'll have a probabilistic approach towards the selection of the plaintext that is to be chosen for the attack.

Chapter 3

Research Methodologies

3. Research Methodologies

Stream ciphers are an important class of symmetric encryption algorithms. Their basic design philosophy is inspired by the One-time-Pad cipher, which encrypts by XOR'ing the plaintext with a random key. However, the need for a key of the same size as the plaintext makes the One-Time-Pad impractical for most applications. Instead, stream ciphers expand a given short random key into a pseudo-random keystream, which is then XOR'ed with the plaintext to generate the ciphertext. Consequently, the design goal for a stream cipher is to efficiently generate pseudo-random bits which are indistinguishable from truly random bits. Rabbit was introduced by Boesgaard et al [1] and is one of the state of the art stream ciphers known. It's much secure cryptographically as claimed by its designers and the ones who have analysed it in some respects.

However, a great work in respect of analysis has been done and also yet to be done to prove the security of the cipher in different aspects

3.1 The Design of Rabbit

3.1.1The Cipher Algorithm

The internal state of the stream cipher consists of 513 bits. 512 bits are divided between eight 32-bit state variables $x_{j,i}$ and eight 32-bit counter variables $c_{j,i}$, where $x_{j,i}$ is the state variable of subsystem j at iteration i, and $c_{j,i}$ denote the corresponding counter variables. There is one counter carry bit, $\phi_{7,i}$, which needs to be stored between iterations. This counter carry bit is initialized to zero. The eight state variables and the eight counters are derived from the key at initialization.[1]

3.1.1.1 Key Setup Scheme[1]

The algorithm is initialized by expanding the 128-bit key into both the eight state variables and the eight counters such that there is a one-to-one correspondence between the key and the initial state variables, $x_{j,0}$, and the initial counters, $c_{j,0}$.

The key, $K^{[127..0]}$, is divided into eight subkeys: $k_0 = K^{[15..0]}$, $k_1 = K^{[31..16]}$, ..., $k_7 = K^{[127..112]}$. The state and counter variables are initialized from the subkeys as follows:

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \mid k_j & \text{for j even} \\ k_{(j+5 \bmod 8)} \mid k_{(j+4 \bmod 8)} & \text{for j odd} \end{cases} \qquad (1)$$

and

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \mid k_{(j+5 \bmod 8)} & \text{for j even} \\ k_j \mid k_{(j+1 \bmod 8)} & \text{for j odd.} \end{cases} \qquad (2)$$

The system is iterated four times, according to the next-state function defined below, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter values are re-initialized according to:

$$c_{j,4} = c_j \oplus x_{(j+4 \bmod 8),4} \qquad (3)$$

to prevent recovery of the key by inversion of the counter system.

3.1.1.2 Next-state Function[1]

The core of the Rabbit algorithm is the iteration of the system defined by the following equations:

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} <<< 16) + (g_{6,i} <<< 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} <<< 8) + g_{7,i} \\
x_{2,i+1} &= g2,i + (g_{1,i} <<< 16) + (g_{0,i} <<< 16) \\
x_{3,i+1} &= g3,i + (g_{2,i} <<< 8) + g_{1,i} \\
x_{4,i+1} &= g4,i + (g_{3,i} <<< 16) + (g_{2,i} <<< 16) \qquad (4) \\
x_{5,i+1} &= g5,i + (g_{4,i} << 8) + g_{3,i} \\
x_{6,i+1} &= g6,i + (g_{5,i} << 16) + (g_{4,i} << 16) \\
x_{7,i+1} &= g7,i + (g_{6,i} <<< 8) + g_{5,i}
\end{aligned}$$

$g_{j,i} = ((x_{j,i} + c_{j,i})^2 \; ((x_{j,i} + c_{j,i})^2 >>32)) \bmod 2^{32}$ (5)

Where all additions are modulo 2^{32}. Before iteration the counters are incremented as described below:

3.1.1.3 Counter System [1]

The dynamics of the counters is defined as follows:

$$C_{0,i+1} = C_{0,i} + a_0 + \phi_{7,i} \bmod 2^{32}$$

$$C_{1,i+1} = C_{1,i} + a_1 + \phi_{0,i+1} \bmod 2^{32}$$

$$C_{2,i+1} = C_{2,i} + a + \phi_{1,i+1} \bmod 2^{32}$$

$$C_{3,i+1} = C_{3,i} + a_3 + \phi_{2,i+1} \bmod 2^{32}$$

$$C_{4,i+1} = C_{4,i} + a_4 + \phi_{3,i+1} \bmod 2_{32} \quad (6)$$

$$C_{5,i+1} = C_{5,i} + a_5 + \phi_{4,i+1} \bmod 2^{32}$$

$$C_{6,i+1} = C_{6,i} + a_6 + \phi_{5,i+1} \bmod 2^{32}$$

$$C_{7,i+1} = C_{7,i} + a_7 + \phi_{6,i+1} \bmod 2^{32}$$

where the counter carry bit, $\phi_{j,i+1}$, is given by

$$\phi_{j,i+1} = \begin{cases} 1 \text{ if } c_{0,i} + a_0 + \phi_{7,i} \, , >= 2^{32} \wedge j = 0 \\ 1 \text{ if } c_{j,i} + a_j + \phi_{j-1,i+1} \, , >= 2^{32} \wedge j > 0 \\ 0 \text{ otherwise.} \end{cases} \quad (7)$$

Furthermore, the a_j constants are defined as:

a_0 = 0x4D34D34D a_1 = 0xD34D34D3
a_2 = 0x34D34D34 a_3 = 0x4D34D34D (8)
a_4 = 0xD34D34D3 a_5 = 0x34D34D34

$a_6 = 0x4D34D34D \quad a_7 = 0xD34D34D3.$

3.1.1.4 Extraction Scheme [1]

After each iteration 128 bits of output are generated as follows:

$$
\begin{aligned}
s_i^{[15..0]} &= x_{0,I}^{[15..0]} \oplus x_{5,i}^{[31..16]} & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & s_i^{[63..48]}{}_i &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned} \quad (9)
$$

where s_i is the 128-bit keystream block at iteration i.

3.1.1.5 Encryption/decryption Scheme [1]

The extracted bits are XOR'ed with the plaintext/ciphertext to encrypt/decrypt.

$$c_i = p_i \oplus s_i, \quad (10)$$

$$p_i = c_i \oplus s_i, \quad (11)$$

where c_i and p_i denote the *i*th ciphertext and plaintext blocks, respectively.

3.3 Our Approach to Attacks

Although the key generated is assumed to be truly random however, we have tried to see the differences in the out put key by the input key values and also to correlate the certain internal state values.

3.3.1 Ladder Approach

We've analyzed the model with Ladder Approach. In this approach, we divide the key into 8bits and then analyze them by examining different values against the input key values and the correlation between the different octets of the key generated and the input key, and also between the various state variables during key generation process.

During this, we did the following

- One by one changing the octet of the input key

- Noting down the result for each different output
- Saw the affect of changing key on the state variable
- Saw the affect of changing key value the ciphertext
- Found some correlation between state variables and the output text
- Then checked the correlation, thus found, against the different values to generalize the result.
- Tried to recover some part of key on that basis.

The assumptions about the ladder approach are:

3.3.1.1 Initial guess

We modify cipher to plaintext and on the basis of similarity we modify our guess.

3.3.1.2 Strong Correlation between the states

We try to find out the strong correlation between the states as for example in the case of implementation in Turbo C++ the equation (4) of the next state function of Rabbit is reduced to:

$$
\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i}) + (g_{6,i}) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \\
x_{2,i+1} &= g2,i + (g_{1,i}) + (g_{0,i}) \\
x_{3,i+1} &= g3,i + (g_{2,i} \lll 8) + g_{1,i} \\
x_{4,i+1} &= g4,i + (g_{3,i}) + (g_{2,i}) \qquad \text{(A)} \\
x_{5,i+1} &= g5,i + (g_{4,i} \ll 8) + g_{3,i}
\end{aligned}
$$

3.3.1.3 Weak encryption keys

We apply weak encryption keys and check the attack and see after how much attempts we are supposed to recover the key. That is, in the beginning we have tried to take the out put key with simple key values.

e.g. In the beginning we used the input key as {00,00,00,00,00,00,00,00}, then saw the values of the last updated state variables and the output text and then changed the input key

values gradually, i.e. {01, 00, 00, 00, 00, 00, 00, 00},{01, 01, 00, 00, 00, 00, 00, 00}, {01, 01, 01, 00, 00, 00, 00, 00}, {01, 01, 01, 01, 00, 00, 00, 00}, {01, 01, 01, 01, 01, 00, 00, 00}, {01, 01, 01, 01, 01, 01, 00, 00}, {01, 01, 01, 01, 01, 01, 01, 00}, {01, 01, 01, 01, 01, 01, 01, 01} to see if they make any difference and the extent to which they are affecting the output.

3.3.1.4 Strong chosen plaintext

We have a probabilistic approach towards the selection of the plaintext that is to be chosen for the attack.

In the beginning we used the input text with small input values so they may not make any difference to the output text and hence the ease for the analysis.

e.g. To find out the correlation between various state variables and the output text, I took the small values of the input text as {'a',00,00,00}.Here the last three input text elements are not going to affect the second third and fourth part of the out put text. And so they let us find some relation, if possible, between the output text and the state variables.

3.3.2 Limits of my attack

During the analysis procedure I adopted, following are the limits to which I bound my implementation:

- Most of the correlations I found were XOR based.
- I focused mainly on the values of the last updated state variables and not the values in earlier iterations as the state variables being XOR'ed with input text are the last updated.
- I found the results mainly on the basis of the code given to check their functionality on the two plate forms i.e. Turbo C++ and VC++

Chapter 4

Implementation

4. Implementation

Implementation includes all the details that were required to make the system operational. The development tools and technologies to implement the system and also reasons for selecting particular tool are discussed. Then the algorithm being translated into the implementation tool will be descried.

Software selection is very important step in developing a computer based system. The software that is used is capable of meeting the requirement of the proposed system. After considering the number of tools available these days such as Turbo C++, Visual Basic, Visual C++.Net, Visual C ,MS Excel, Visual Basic.Net and MATLAB we chose Turbo C ++ v3.0 and Visual C++ 6.0

4.1 Turbo C++ 3.0

Turbo C++ is a Borland C++ compiler and Integrated Development Environment (IDE), famous for its high compilation & linkage speed - hence the term "Turbo". It was a part of Borland's highly popular family of compilers including Turbo Pascal, Turbo Basic, Turbo Prolog and Turbo C. Turbo C++ was a successor of Turbo C.
Turbo C++ 3.0 was released in 1991 (shipping on November 20), and came in amidst expectations of the coming release of Turbo C++ for Microsoft Windows.

4.2 Microsoft Visual C++ 6.0

Visual C++ has features such as syntax highlighting, IntelliSense (a coding autocompletion feature) and advanced debugging functionality. For example, it allows for remote debugging using a separate computer and allows debugging by stepping through code a line at a time. The "edit and continue" functionality allows changing the source code and rebuilding the program during program debugging, without restarting the debugged program. The compile and build system feature, precompiled header files, "minimal rebuild" functionality and incremental link: these features significantly shorten turn-around time to edit, compile and link the program, especially for large software projects.

Visual C++ 6.0, which included MFC 6.0, was released in 1998. The release was somewhat controversial since it did not include an expected update to MFC. Visual C++ 6.0 is still quite popular and often used to maintain legacy projects.

4.3 Implementation of the Attack

We now give implementation of the attack.

Here I am going to describe the code of rabbit I modified a little to analyze by getting outputs of all the state variables and the sub_keys and hence to implement the attack:

4.3.1 Implementation in Turbo C++

Note that the basic program code has been taken from Rabbit [1[illegible]] with some amendments (but not in the way that the whole code sense was changed) to analyze it.

4.3.1The g-Function

Square a 32-bit number to obtain the 64-bit result and return the upper 32 bit XOR the lower 32 bit

```
*/=================================*/
*/===========g_function============*/
/*=================================*/

uint32 g_func(uint32 val)
{
// Construct high and low argument for squaring
uint32 a = val&0xFFFF;
uint32 b = val>>16;// Calculate high and low result of squaring
uint32 high = ((((a*a)>>17) + (a*b))>>15) + b*b;
uint32 low = val*val;
        cout<<"\nhigh="<<h<<"\nlow="<<l<<"\n";
 // Return high XOR low;
return high^low;
```

}

4.3.2 The next_state Function

Calculate the next internal state

```
/*=============================*/
/*======== next_state_function=====*/
/*=============================*/
void next_state(t_instance *p_instance)
{
//short s[8]={0,0,0,0,0,0,0,0};
// Temporary data
uint32 g[8], c_old[8], i;
// Save old counter values
for (i=0; i<8; i++)
c_old[i] = p_instance->c[i];
// Calculate new counter values
p_instance->c[0] += 0x4D34D34D + p_instance->carry;
p_instance->c[1] += 0xD34D34D3 + (p_instance->c[0] <
c_old[0]);
p_instance->c[2] += 0x34D34D34 + (p_instance->c[1] <
c_old[1]);
p_instance->c[3] += 0x4D34D34D + (p_instance->c[2] <
c_old[2]);
p_instance->c[4] += 0xD34D34D3 + (p_instance->c[3] <
c_old[3]);
p_instance->c[5] += 0x34D34D34 + (p_instance->c[4] <
c_old[4]);
p_instance->c[6] += 0x4D34D34D + (p_instance->c[5] <
c_old[5]);
```

```
p_instance->c[7] += 0xD34D34D3 + (p_instance->c[6] <
c_old[6]);
p_instance->carry = (p_instance->c[7] < c_old[7]);
// Calculate the g-functions
for (i=0;i<8;i++)
{g[i] = g_func(p_instance->x[i] + p_instance->c[i]);
//cout<<"g["<<i<<"]="<<g[i]<<"\n";//by asmara
}
// Calculate new state values
p_instance->x[0] = g[0] + _rotl(g[7],16) + _rotl(g[6],16);
p_instance->x[1] = g[1] + _rotl(g[0], 8) + g[7];
p_instance->x[2] = g[2] + _rotl(g[1],16) + _rotl(g[0],16);
p_instance->x[3] = g[3] + _rotl(g[2], 8) + g[1];
p_instance->x[4] = g[4] + _rotl(g[3],16) + _rotl(g[2],16);
p_instance->x[5] = g[5] + _rotl(g[4], 8) + g[3];
p_instance->x[6] = g[6] + _rotl(g[5],16) + _rotl(g[4],16);
p_instance->x[7] = g[7] + _rotl(g[6], 8) + g[5];
//for(int cnt=0;cnt<8;cnt++)
//s[cnt]=p_instance

//for ( cnt=0;cnt<8;cnt++) //asmara
{//cout<<"\nx["<<cnt<<"]"<<p_instance->x[i]; //asmara
//ex1=(p_instance->x[2*cnt]<<16)^(p_instance-
>x[(2*cnt+5)%8]>>16);
//cout<<"ex1="<<ex1;

cout<<"\nX0="<<p_instance->x[0];binary(p_instance->x[1]);
cout<<"\nX1="<<p_instance->x[1];binary(p_instance->x[2]);
cout<<"\nX2="<<p_instance->x[2];binary(p_instance->x[3]);
cout<<"\nX3="<<p_instance->x[3];binary(p_instance->x[4]);
cout<<"\nx4="<<p_instance->x[4];binary(p_instance->x[5]);
```

```
cout<<"\nx5="<<p_instance->x[5];binary(p_instance->x[6]);
cout<<"\nX6="<<p_instance->x[6];binary(p_instance->x[7]);
cout<<"\nX7="<<p_instance->x[7];binary(p_instance->x[8]);

/*s=(p_instance->x[i]^
(p_instance->x[(i+5)%8]))>>16)^
(p_instance->x[i]^(p_instance->x[(i+3)%8]<<16);*/

//ex2=(p_instance->x[2*cnt]>>16)^(p_instance-
>x[(2*cnt+3)%8]<<16);
//cout<<"ex2="<<ex2;
}
 }
```

4.3.3 The Key Set Up Function

This function takes the pointers to the initial key value as an input and does the following tasks:

- Divides the initial key into four subset keys and gives their output in parallel.
- Generates the initial state variables and prints their values.
- Generates the counter variables and prints their value.

This function is being called from the **main** function and it calls the **next_state_function** five times to iterate the system and hence to make the state variable more random and nonlinear.

```
/*=====================================*/
/*==========key_setup_function==========*/
/*=====================================*/
// key_setup
 void key_setup(t_instance *p_instance,const byte *p_key)
{
// Temporary data
uint32 k0, k1, k2, k3, i;
```

```
// Generate four subkeys
k0 = *(uint32*)(p_key+ 0);
cout<<"k0="<<k0<<endl;
k1 = *(uint32*)(p_key+ 4);
cout<<"k1="<<k1<<endl;
//k2=45;
k2 = *(uint32*)(p_key+ 8);
cout<<"k2="<<k2<<endl;
k3 = *(uint32*)(p_key+12);
//k3=786;
cout<<"k3="<<k3<<endl;

// Generates initial state variables
p_instance->x[0] = k0;
p_instance->x[2] = k1;
p_instance->x[4] = k2;
p_instance->x[6] = k3;
p_instance->x[1] = (k3<<16) | (k2>>16);
p_instance->x[3] = (k0<<16) | (k3>>16);
p_instance->x[5] = (k1<<16) | (k0>>16);
p_instance->x[7] = (k2<<16) | (k1>>16);

//outputs state variables
cout<<"\tKx[0]="<<p_instance->x[0];
cout<<"\tKx[2]="<<p_instance->x[2];
cout<<"\tKx[4]="<<p_instance->x[4];
cout<<"\tKx[6]="<<p_instance->x[6];
cout<<"\tKx[1]="<<p_instance->x[1];
cout<<"\tKx[3]="<<p_instance->x[3];
cout<<"\tKx[5]="<<p_instance->x[5];
cout<<"\tKx[7]="<<p_instance->x[7];
```

```
/*for(int k=0;k<8;k++)
cout<<"\tKx{"<<k<<"]="<<p_instance->x[k];*/

// Generate initial counter values
p_instance->c[0] = _rotl(k2,16);
p_instance->c[2] = _rotl(k3,16);
p_instance->c[4] = _rotl(k0,16);
p_instance->c[6] = _rotl(k1,16);
p_instance->c[1] = (k0&0xFFFF0000) | (k1&0xFFFF);
p_instance->c[3] = (k1&0xFFFF0000) | (k2&0xFFFF);
p_instance->c[5] = (k2&0xFFFF0000) | (k3&0xFFFF);
p_instance->c[7] = (k3&0xFFFF0000) | (k0&0xFFFF);

// Reset carry flag
p_instance->carry = 0;

// Iterate the system four times
for (i=0;i<4;i++)
next_state(p_instance);
// Modify the counters
for (i=0;i<8;i++)
p_instance->c[(i+4)&0x7] ^= p_instance->x[i];
}
```

4.3.4The Cipher Function

This function takes, as arguments, the value where the input data to be stored, the input data and the size of data.

This accomplishes the following tasks:

- Stores the value of the key extracted into a variable.

- XOR's the plaintext/Ciphertext with the key extracted to recover the plaintext/ciphertext.

```
/*==================================*/
/*===========cipher function========= */
/*================================ */

void cipher(t_instance *p_instance, const char *p_src,
byte *p_dest, size_t data_size)
{
uint32
i,fr,*frPtr,*secPtr,*thdPtr,*fthPtr,sec,thd,fth,dec1=0,dec2=0,
dec3=0,dec4=0,var1,var2,var3,var4;

for (i=0; i<data_size; i+=16)
     {
     //next_state(p_instance);

     // Encrypt 16 bytes of data
     var1=p_instance->x[0] ^
     (p_instance->x[5]>>16) ^
     (p_instance->x[3]<<16);
     fr=*(uint32*)(p_dest+ 0) = *(uint32*)(p_src+ 0) ^var1;

     cout<<"\n\nFirst="<<fr<<" Binary:";
     binary(fr);

     var2= p_instance->x[2] ^
     (p_instance->x[7]>>16) ^
     (p_instance->x[5]<<16);
     sec=*(uint32*)(p_dest+ 4) = *(uint32*)(p_src+ 4) ^var2;
```

```
    cout <<"\tSec="<< sec <<" binary: ";
    binary(sec);

    var3=p_instance->x[4] ^
    (p_instance->x[1]>>16) ^
    (p_instance->x[7]<<16);
    thd=*(uint32*)(p_dest+ 8) = *(uint32*)(p_src+ 8) ^var3;
    cout<<"\nThd="<<thd<<" binary:";
    binary(thd);

    var4=p_instance->x[6] ^
    (p_instance->x[3]>>16) ^
    (p_instance->x[1]<<16);
    fth=*(uint32*)(p_dest+12) = *(uint32*)(p_src+12) ^var4;
    cout<<"\tFourth="<<fth<<" binary:";
    binary(fth);

    //Increment pointers to source and destination data
    p_src += 16;
    p_dest += 16;
}

//decrypt the data

frPtr=&fr;
dec1=*(uint32*)(p_dest+0)= *(uint32*)frPtr^var1;
cout<<"\tDEC1:\t"<<dec<<" binary: ";
binary(dec1);

secPtr=&sec;
dec2=*(uint32*)(p_dest+ 4) = *(uint32*)secPtr ^var2;
```

```
cout <<"\tDEC2="<<dec2<<" binary: ";
binary(dec2);

thdPtr=&thd;
dec3=*(uint32*)(p_dest+ 8) = *(uint32*)thdPtr ^
var3;
cout<<"\nDEC3="<<dec3<<" binary:";
binary(dec3);
fthPtr=&fth;
dec4=*(uint32*)(p_dest+12) = *(uint32*)fthPtr ^
var4;
cout<<"\tDEC="<<dec4<<" binary:";
binary(dec4);

}

     }
```

4.3.6 Binary Function

It's an additional function to get the values of different variables in binary form for the ease of correlations and analysis.

```
//conversion from uint32 to binary
void binary(uint32 nmbr) {
 uint32 remainder;
 cout<<".";
 if(nmbr <= 1) {
 cout << nmbr;
 return;
}
```

```
remainder = nmbr%2;
binary(nmbr >> 1);
cout << remainder;
}
```

4.3.7 The main function

I wrote the main function to work as described in the Rabbit algorithm by declaring variables, calling functions by passing the parameters accordingly.

```
/*================================*/
/*===========main function=========*/
/*================================*/

void main(void)
{
clrscr();
uint32 sec,x2,rec;

t_instance instnc[8]={0,0,0,0,0,0,0,0};
t_instance *abc;
abc=instnc;

byte key[8]={01,01,01,01,01,01,01,01};
const byte* ky=key;

char dta[4]={'A',00,00,00};
const byte *def,*source=dta;

byte *destntn;
byte des=0;
destntn=&des;
```

```
size_t dt_sz=16;

key_setup(abc,ky);
      cipher(abc, source,destntn, dt_sz);

     getch();
}
```

4.3.8 Variables:

Beside the variables declared in the functions, the following variables have been defined in the Rabbit header file.

- ***Uint32*** is defined in the Rabbit header file as of type ***unsigned int***
- ***Byte*** is defined as ***unsigned char***
- ***t_instance*** is the structure to store the instance data (internal state) ***uint32 x[8], uint32 c[8], uint32 carry.***

4.3.2 Implementation In Visual C++

About all the code was same as that in Turbo C++

Chapter 5

Analysis and Results

5 Analysis and Results

5.1 In Turbo C++

5.1.1 XOR'ing the int variables in Turbo C++

In the given Rabbit code the following variables are defined as of type unsigned int with the type defined as ***uint32***

- State variables
- Counter variables

The following were defined as of type unsigned char with the type name as ***byte***

- The initial key value

In Turbo C++ v3 the ***int*** is of 2 byte and ***char*** is of 1 byte on this processor and hence if a variable is shifted left or to right by 16 bits it gives a simple zero result for example:

0101111101011011
If this is shifted to left the result is shown bit by bit as:
(0101111101011011) <<16
1011111010110110
0111110101101100
1111101011011000
1111010110110000
1110101101100000
1101011011000000
1010110110000000
0101101100000000
1011011000000000
0110110000000000
1101100000000000
1011000000000000

0110000000000000

1100000000000000

1000000000000000

0000000000000000

if rotated towards right then:

(0101111101011011)>>16

0010111110101101

0001011111010110

0000101111101011

0000010111110101

0000001011111010

0000000101111101

0000000010111110

0000000001011111

0000000000101111

0000000000010111

0000000000001011

0000000000000101

0000000000000010

0000000000000001

0000000000000000

0000000000000000

That's what being done in the ***cipher_function*** during key extraction from the state variables. By this,before the XORing with plaintext, the extraction scheme of the code is trimmed as:

```
C1= P1 ^ p_instance->x[0]
C2= P2 ^ p_instance->x[2]          (B)
C3= P3 ^ p_instance->x[4]
C4= P4 ^ p_instance->x[6]
```

So the encryption text is XOR'ed with some of the values of state variables directly, ***Key extraction system*** is no more handy here.

On this basis, most of the following results find clues towards weaknesses.

5.1.2 The Affect, on the State Variables, of Calling Next State Function More Than Once

Simply, it means simple iteration of the system more than once.

a)

When there was no function call to ***key-setup*** function and just a ***next_state*** function from the ***cipher*** function, the results were:

Input key value:{0,0,0,0,0,0,0,0}

Input text:{'A',00,00,00,00,00,00,00}

The values of State Variables are:

X0=28731................1111010011010111

X1=62679................1011110110100010

X2=48546................1111101101100100

X3=64356................1011110110100010

X4=48546...............100000100011110

X5=16670................1011110110100010

X6=48546................1110000101111110

X7=57726................1101001101001101

Ciphertext output is:

C1=28794 binary:...............111000001111010

C2=48546 binary: 1011110110100010

C3=48546 binary:................1011110110100010

C4=20898 binary:...............101000110100010

Result:

- Here, for the values of the state variables, it's obvious that

X2= X4=X6

It's because of the fact that the input key is not taken to initialize the sub-key and hence to initial state variables

- Beside this If we note the second , third and fourth part of the cipher text,

C2=X2

C3=X4

The reason is that if we go the previous trimmed function as described in equ(A),The first part of the cipher is dependent on X[0], the second on X[2], the third on X[4], and the fourth on X[6].

The general result was same for any value of input text and for any value of input key.

b)

By deleting the "***next_ state***" function call from the function "cipher" and simply calling ***key_setup*** function(function that actually initializes the state variables and also calls the next state function four times) from the ***main***, the results are shown in table 5.1

Input		**Output**			
key	text	sub-key	Initial state variables	Last updated state variables	Cipher texts
{00,00,00,00,00,00,00,00}	{'A',00,00,00}	k0=0 k1=0 k2=0 k3=0	Kx[0]=0 Kx[2]=0 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=42338 X1=29562 X2=24211 X3=39214 X4=57907 X5=10505 X6=42011 X7=15259	C1=42275 C2=24211 C3=57907 C4=18971
{11,00,00,00,00,00,00,00}	{'A',00,00,00}	k0=11 k1=0 k2=0 k3=0	Kx[0]=11 Kx[2]=0 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=50178 X1=28072 X2=2101 X3=14324 X4=23826 X5=10583 X6=47966 X7=59605	C1=50243 C2=2101 C3=23826 C4=21854

{11,11,00,00,00,00,00,00}	{'A',00,00,00}	k0=2827 k1=0 k2=0 k3=0	Kx[0]=2827 Kx[2]=0 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63365 X1=19778 X2=18613 X3=51744 X4=27851 X5=17033 X6=19462 X7=22355	C1=63428 C2=18613 C3=27851 C4=41478
{11,11,11,00,00,00,00,00}	{'A',00,00,00}	k0=2827 k1=11 k2=0 k3=0	Kx[0]=2827 Kx[2]=11 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63645 X1=18447 X2=49334 X3=16346 X4=36078 X5=14413 X6=14078 X7=1583	C1=63708 C2=49334 C4=55550 C3=36078
{11,11,11,11,00,00,00,00}	{'A',00,00,00}	k0=2827 k1=2827 k2=0 k3=0	Kx[0]=2827 Kx[2]=2827 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=40781 X1=56176 X2=39774 X3=56624 X4=13838 X5=44959 X6=34158 X7=46413	C1=40716 C2=39774 C3=13838 C4=27502
		k0=2827 k1=2827 k2=11 k3=0	Kx[0]=2827 Kx[2]=2827 Kx[4]=11 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=55618 X1=12686 X2=28202 X3=11487 X4=19203 X5=52744 X6=21909 X7=47675	C1=55555 C2=28202 C3=19203 C4=48021
{11,11,11,11,11,11,00,00}	{'A',00,00,00}	k0=2827 k1=2827 k2=2827 k3=0	Kx[0]=2827 Kx[2]=2827 Kx[4]=2827 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=38106 X1=62208 X2=21754 X3=1049 X4=13418 X5=50216 X6=61861 X7=2669	C1=38043 C2=21754 C3=13418 C4=8101
{11,11,11,11,11,11,11,00}	{'A',00,00,00}	k0=2827 k1=2827 k2=2827 k3=11	Kx[0]=2827 Kx[2]=2827 Kx[4]=2827 Kx[6]=11 Kx[1]=0 Kx[3]=0	X0=51417 X1=26344 X2=40610 X3=52220 X4=59696 X5=35258	C1=51352 C2=40610 C3=59696 C4=54125

			Kx[5]=0 Kx[7]=0	X6=15725 X7=6075	
{11,11,11,11,11,11,11,11}	{'A',00,00,00}	k0=2827 k1=2827 k2=2827 k3=2827	Kx[0]=2827 Kx[2]=2827 Kx[4]=2827 Kx[6]=2827 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=41337 X1=55000 X2=52433 X3=56549 X4=16816 X5=43181 X6=32509 X7=13608	C1=41272 C2=52433 C3=16816 C4=37117

Table 5.1

Conclusion:

- The values of state variables reflect the input key properties if the system is not iterated four times
- After getting different values of state variables against different input values show that the values of state variables and hence the output key repeat for each element value by *val*%256 where *val* is any input key value of the element.

Noe: The results shown above are found after some changes of the destination and the source bits of the Rabbit code according to the size of variables rather the one given in the Rabbit code to see the impact of key_set_up function exclusively.

5.1.3 Partial recovery of the key from state variables.

Wetook the Input key value as an array of unsigned char as defined in the Rabbit header file. It consisted of 8 elements of 8*8=64 bits (imaginary 8*16=128bits).

To get another interesting fact yet a flaw of the Rabbit code,wechecked the affect of changing values of different elements of the key and saw the result on the final state variables and hence on the output. By analyzing the results, it was possible to recover the certain elements of the input key partially. The results are shown in table 5.2

#	**Input**	**Output**

	key	Text	sub-key	Initial state variables	Last updated state variables	Cipher texts
a	00,00,00,00,00,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C4=21706 C3=20580 C2=13357 C1=2184
b	01,00,00,00,00,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C1=2184 C2=13357 C3=20580 C4=21706
c	01,01,00,00,00,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C1=2184 C2=13357 C3=20580 C4=21706
d	01,01,01,00,00,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C4=21706 C3=20580 C2=13356 C1=2184
e	01,01,01,01,00,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C3=21706 C3=20580 C2=13612 C1=2184

f	01,01,01,01,01,00,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C1=2184 C2=13612 C3=20580 C4=21706
g	01,01,01,01,01,01,00,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C4=21706 C3=20580 C2=13612 C1=2184
h	01,01,01,01,01,01,01,00	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C4=21706 C3=20581 C2=13612 C1=2184
i	01,01,01,01,01,01,01,01	'A',00,00,00	k0=0 k1=344 k2=0 k3=0	Kx[0]=0 Kx[2]=344 Kx[4]=0 Kx[6]=0 Kx[1]=0 Kx[3]=0 Kx[5]=0 Kx[7]=0	X0=63358 X1=2779 X2=13357 X3=25539 X4=20580 X5=40711 X6=21906 X7=8620	C1=2184 C2=13612 C3=20837 C4=21706

Table 5.2

Conclusion:

1. In case (c), we can clearly see the following relation:

$$C2 \oplus X2 = k3$$

K3 is the value of input key of the third element of the input key array

The above result is same for the following form of input key:

Input key ={ k1,k2,k3,0,0,0,0,0}

Where k1, k2, k3 are any value for the first second and third element of the input key array respectively

It's also important to mention that values are taken of the form (*key %255*) here key is the given input key value

2.In case (h), there is an obvious relation:

$$C3 \oplus X4 = k7$$

K7 is the value of input key of the seventh element of the input key array.

The above result is same for the following form of input key:

Input key={k1,k2,k3,k4,k5,k6,k7,0}

Where k1-k7 are any value for the first to seventh element of the input key array respectively.

5.1.4 Repetition of Output key value from the different output

5.1.4.1 Single Input key

In this case the key is repeated simply after 256 as it is the maximum number range for ASCII character in this window system.

5.1.4.2 Character array of 8 elements

If the code is run as it's done in 5.1.3 , the results are obvious in that case.

5.1.5 Affect of Length of Input Text on the cipher function

I got the following results by changing the value of input text. The results are as in table 5.3

Input	Output	
Text	Last updated state variables	Cipher texts

"a"	X0=58302 X1=4190 X2=22472 X3=37594 X4=1237 X5=43520 X6=33038 X7=44703	C1=58335 C2=22472 C3=1237 C4=39182
"ab"	X7=44703 X6=33038 X5=43520 X4=1237 X3=37594 X2=22472 X1=4190 X0=58302	C1=33247 C2=22442 C3=1237 C4=33038
"abc"	X0=58302 X1=4190 X2=22472 X3=37594 X4=1237 X5=43520 X6=33038 X7=44703	C1=33247 C2=13482 C3=1206 C4=33038
"abcd"	X0=58302 X1=4190 X2=22472 X3=37594 X4=1237 X5=43520 X6=33038 X7=44703	C1=33247 C2=13482 C3=24758 C4=33130

Table 5.3

Conclusion:

- In the first case the C2 and C4 are not affected by the input or in other words(according to equ B) C2=X2, C3=X4
- In the second case C2 $\oplus$X2 gives the input text, and also C3=X4, C4= X6
- In the Third case C3$\oplus$X4=input text on the corresponding place, and C4=X6.

- In the Fourth Case $C4 \oplus X6$=intput text on the corresponding place.

The above results can be written as:

For each nI there is a relation such that

$$C_{n+1}=X' \text{ and } C_{n+2}=X''$$

$$\text{And also } C_n \oplus X'=I_n$$

Here

nI is the number of input characters

C_n is the nth output cipher

X' and X'' indicate the corresponding state variable affecting the output according to equation B.

I_n is the nth input character

It shows the clear evidence of the fact that the state variables are not being mixed randomly before they are XOR'ed with the input text.

Note:In the above analysiswelimit the input text length up to four characters only.

5.2 Implementation in Visual C++

5.2.1 XOR'ing the int variables in VC++

In Visual C++ the int variable is of 4 bytes and a char variable is of 2 bytes so the problem for an int variable to reduce to zero after 16 left or right shifts is eliminated here and the key extraction function does a good work to extract the key from different values of the state variables.

So now the game is as:

(10100010111101110011100101110111)>>16=00000000000000001010001011110111

(10100010111101110011100101110111)<<16=00111001011101110000000000000000

in the first step 16 most significant bits are shifted to the least significant location while in the latter 16 least significant bits are shifted to the most significant location.

5.2.2 The Affect, on the State Variables, of Calling Next State Function More Than Once

When there was no function call to ***key-setup*** function and just a ***next_state*** function from the ***cipher*** function, the results were as in table 5.4

#	Input		Output	
	key	Text	Last updated state variables	Cipher texts
a	00,00,00,00,00,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=308037007 C2=377978021 C3=216370198 C4=129571995
b	99,00,00,00,00,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=3779780112(affecte d) C3=2163701980 C4=1295719957
c	99,99,00,00,00,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=3779771664(affecte d) C3=2163701980 C4=1295719957
d	99,99,99,00,00,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=377608976(affected) C3=2163701980 C4=1295719957

e	99,99,99 ,99,00,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=2183773456(affecte d) C3=2163701980 C4=1295719957
f	99,99,99,99,99,00,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=218377345(as previous) C3=2163701951(affecte d) C4=1295719957
g	99,99,99,99,99,99,00,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=218377345(as previous) C3=2163678143(affecte d) C4=1295719957
h	99,99,99,99,99,99,99,00	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=218377345(as previous) C3=2157190097(affecte d) C4=1295719957
i	99,99,99,99,99,99,99,99	'A',00,00,00, 00,00,00,00	X0=3520701919 X1=340762410 X2=2194122249 X3=3045877329 X4=3158142099 X5=4011418509 X6=3523434189 X7=2021276874	C1=3080370070 C2=218377345(as previous) C3=3818134463(affecte d) C4=1295719957

Table 5.4

Conclusion:

Since there is no key_setup_function call, any value of input key does not affect on state variables.

However, the input key value does make a change in the out put text in a way that a change in the first 8 bytes of the key changes the value of C2, while any change in the last 8 bytes of the input key makes a change in the value of C3 leaving the values of C1 and C4 unchanged.

Note:

- The result was tested by different values of input key with a constant vale of input text.
- The results are gotten by the same code as that used in the 5.1.2 a) case.

5.2.4 Repetition of Output key value from the different input.

- We took the code we changed in the case of 5.1.2 b) and got the results against different input values to get the output
  - When there is no key setup function call then the result of the above case is obvious from 5.2.2 where the output values are repeated as they don't remain dependent on the input key
- If the Rabbit code is implemented smoothly it gives that:
  - If the input key in form of single byte variable, it's repeated after every 256 value.
  - Whereas if it's in the form of an array element, the values of the output key generated is repeated for the value of each of the elements after every 256 provided that rest of the elements are constant. In short, for each element the affective value of the input key in decimal is ***val%256***

Chapter 6

Conclusion and Future Work

6. Conclusions and Future Enhancement

We analyzed the security of the stream cipher Rabbit against cryptanalytic attacks, i.e. attacks based on the given code of the rabbit and the mathematical description of the algorithm at two platforms.

6.1Conclusion:

6.1.1 Implementation in Turbo C++

As a result of implementation of the Code in Turbo C++ we are able to redefine the two equations formerly defined in the Rabbit [18] as:

A.

$$x_{0,i+1} = g_{0,i} + (g_{7,i}) + (g_{6,i})$$

$$x_{1,i+1} = g_{1,i} + (g_{0,i}<<<8) + g_{7,i}$$

$$x_{2,i+1} = g2,i + (g_{1,i}) + (g_{0,i})$$

$$x_{3,i+1} = g3,i + (g_{2,i}<<<8) + g_{1,i}$$

$$x_{4,i+1} = g4,i + (g_{3,i}) + (g_{2,i}) \qquad \text{(A)}$$

$$x_{5,i+1} = g5,i + (g_{4,i}<<8) + g_{3,i}$$

Here $x_{i,j}$ is the state variable i at iteration j and $g_{i,j}$ is the g_function defined in the Rabbit[18]

The equation (A) gives the core equation for next state function that's believed to introduce the nonlinearity in the output key, but in this case it's simple mixing of g function. By this the formerly described model of rabbit will be reduced to as shown in fig 6.1.

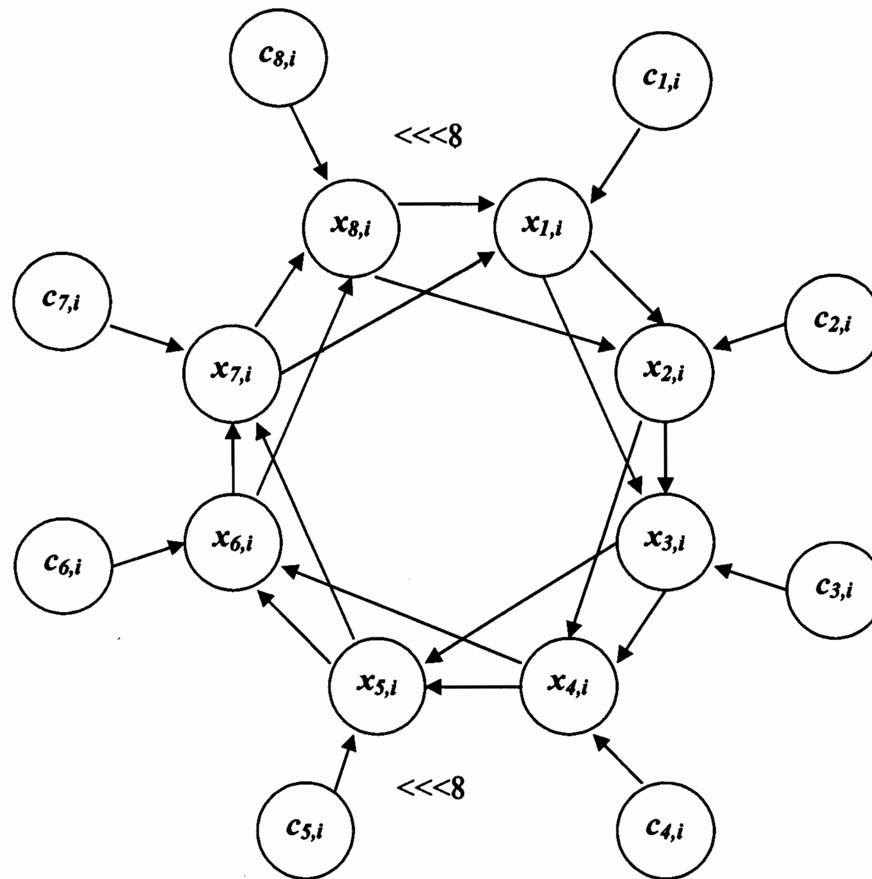


Fig 6.1: The model of Rabbit code when implemented in Turbo C++

B. The second equation thus formed from the described code is:

C1= P1 ^ p_instance->x[0]

C2= P2 ^ p_instance->x[2] (B)

C3= P3 ^ p_instance->x[4]

C4= P4 ^ p_instance->x[6]

Here C_i is the cipher text, and P_i is the plaintext and p_instance->x[i] is the values of state variable.

This equation leaves no transparency for the values of state variables those are needed for the encryption/decryption of plaintext/ciphertext

6.1.2 Implementation in VC++

- Although the weaknesses don't appear easily in this case yet one of them is the affective input key value is limited to *val* %256 where *val* is any value of the input key element.
- Beside that, without the explicit call of the key_setup function, as is in Rabbit code[18], the input key values don't affect the state variables and hence on the output key generation. To make it function properly we have to incorporate the keyset_up function explicitly rather a function call of next state function given in cipher_function of Rabbit [18]

6.2Future enhancements:

We openly invite the world to further analyze the cipher to get the key recovered fully or partially even in more efficient way.

References

References

[1] Martin Boesgaard, Mette Vesterager,Thomas Pedersen, Jesper Christiansen, and Ove Scavenius **Rabbit: A New High-Performance Stream Cipher** CRYPTICO A/S Fruebjergvej 32100 Copenhagen, Denmark.

[2] Syed Irfan Ullah, Ahmad Ali Tabassam, Sikandar Hayat Khiyal **Cryptanalytic weakness in modern stream ciphers and recommendations for improving their security levels** - IEEE --- 2005 International Conference on Emerging Technologies September 17-18, Islamabad.

[3] Eli Biham Orr Dunkleman Nathan Keller **The Rectangle Attack-Rectangling the Serpent-**Computer Science department, Technion-Israel Institute of Technology, Haifa 32000.

[4] Willi Meier and Othmar Sta_elbach. **Fast correlation attacks on certain stream ciphers.** Journal of Cryptology, 1(3):159-176, 1989.

[5] Martin E. Hellman. **A cryptanalytic time-memory tradeoff**. IEEE Transactions on Information Theory, IT-26(4):401-406, July 1980.

[6] Steve Babbage. **Improved exhaustive search attacks on stream ciphers**. European Convention on Security and Detection, IEE conference publication, No. 408, pages 161-166, May 1995.

[7] Alex Biryukov and Adi Shamir. **Cryptanalytic time/memory/data tradeoffs for stream ciphers.** In T. Okamoto, editor, Advances in Cryptology - ASIACRYPT2000, volume 1976 of Lecture Notes in Computer Science, pages 1-13. Springer-Verlag, 2000.

[8] Frederik Armknecht, Matthias Krause, and Dirk Stegemann. **Design principles for combiners with memory.** In S. Maitra, C. E. V. Madhavan, and R. Venkatesan, editors, to appear in the proceedings of Progress in Cryp tology - INDOCRYPT2005, Lecture Notes in Computer Science. Springer- Verlag.

[9] Frederik Armknecht, Joseph Lano, and Bart Preneel. **Extending the resynchronization attack.** In H. Handschuh and A. Hasan, editors, Selected Areas in Cryptography2004, volume 3357 of Lecture Notes in Computer Science, pages 19-38. Springer-Verlag, 2005.

[10] Nicolas T. Courtois and Josef Pieprzyk. **Cryptanalysis of block ciphers with overdefined systems of equations.** In Y. Zheng, editor, Advances in Cryptology SIACRYPT2002, volume 2501 of Lecture Notes in Computer Science, pages 267-287. Springer-Verlag, 2002.

[11] Thomas Siegenthaler. **Decrypting a class of stream ciphers using ciphertext only.** IEEE Transactions on Computers, C-34(1):81{85, Jan. 1985.

[12] Rejane Forre. **A fast correlation attack on nonlinearly feedforward filtered shift-register sequences.** In J. J. Quisquater and J. Vandewalle, editors, Advances in Cryptology - EUROCRYPT'89, volume 434 *of Lecture Notes in Computer Science, pages 586{595. Springer-Verlag, 1990.*

[13] Jovan Dj. Golic. **On the security of nonlinear filter generators.** In D. Gollmann, editor, Fast Software Encryption'96, volume 1039 of Lecture Notes in Computer Science, ages 173-188. Springer-Verlag, 1996.

[14] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. Van Tilborg. **On the inherent intractability of certain coding problems.** IEEE Transactions on Information Theory, IT-24(3):384-386, May 1978.

[15] Willi Meier and Othmar Stafelbach. **Fast correlation attacks on stream ciphers (extended abstract)**. In C. Gunther, editor, Advances in Cryptology EUROCRYPT'88, volume 330 of Lecture Notes in Computer Science, pages 301{314. Springer-Verlag, 1988.

[16] **Differential properties of the g-function**, white paper, version 1.0, www.cryptico.com.

[17] Anne Canteaut and Eric Filiol. **Ciphertext only reconstruction of stream ciphers based on combination generators.** In B. Schneier, editor, Fast Software Encryption2000, volume 1978 of Lecture Notes in Computer Science, pages 165-180. Springer-Verlag, 2001.

[18] Willi Meier and Othmar Staffelbach. **Nonlinearity criteria for cryptographic functions.** In J. J. Quisquater and J. Vandewalle, editors, Advances in Cryptology UROCRYPT'89, volume 434 of Lecture Notes in Computer Science, pages 549{562. Springer-Verlag, 1990.

[19] R. A. Rueppel: **Analysis and Design of Stream Ciphers,** *Springer*, Berlin (1986).

[20] Thomas Siegenthaler. **Correlation-immunity of nonlinear combining functions for cryptographic applications.** IEEE Transactions on Information Theory, IT-30(5):776-780, Sep. 1984.

[21] Claude E. Shannon. **Communication theory of secrecy systems.** Bell System Technical Journal, 28:656-715, 1949.

[22] James L. Massey. **Shift-register synthesis and BCH decoding.** IEEE Transactions on Information Theory, IT-15(1),122-127, Jan. 1969.

[23] **A Handbook of Applied Cryptography**. by A. Menezes, P. van Oorschot and S. Vanstone. Chapter 1

[24] **A Handbook of Applied Cryptography**. by A. Menezes, P. van Oorschot and S. Vanstone. Chapter 6

[25] **Stream cipher** - *Wikipedia, the free encyclopedia.htm*

[26] **Differential cryptanalysis** - *Wikipedia, the free encyclopedia.htm*

[27] Chapter 5: **Propagation and Correlation**, J. Daemen, Annex to AES Proposal (1998).