CHAPTER 1

INTRODUCTION

The privacy of stored information is a key issue regarding the information loss. Increasing proliferation due to the advancement of digital communication has led the invention of newer technology to make it secure. One way of having protection on such data is to encrypt it. We can't deny the fact of having information access by unwanted parties through various communication means despite having significant measures applied to protect them. Protecting data at rest in storage systems poses new challenges compared to protecting data in flight, which has been the focus of communication security for some time and is well understood today. One notable difference between these two problems is that communication channels typically use a streaming interface with first-in/first-out characteristic, whereas storage systems must provide random access to small portions of the stored data [1]. New techniques are needed to provide security in this context, particularly in order to protect the integrity of stored data efficiently. Ensuring data security is a big challenge for computer users. Businessman, professionals, and home users all have some important data that they want to make secure from others.

Cryptography is the study of method of hiding secret in the message whereas trying to figure out the secrets that someone else has hidden is known as cryptanalysis. History reveals many examples of cryptology that worked, and that didn't [2].

Storage encryption technologies use one or more cryptographic keys to encrypt and decrypt the data that they protect. The number of keys and the types of keys used are product and implementation-dependent. For example, public key cryptography uses a pair of keys, and symmetric cryptography uses a single key. Some products support the use of a recovery key that can be used to recover the encrypted data if the regular key is lost. Also, some technologies permit encrypted storage to be shared by multiple users, which could be enabled by having a different key for each user. Often, user's keys are not

directly used to decrypt their stored data; instead, those keys are used to decrypt another key, which in turn is used to decrypt the stored data [2]

1.1 Storage Security:

An *end user device* is a personal computer (desktop or laptop), consumer device (e.g., personal digital assistant [PDA], smart phone), or removable storage media (e.g., USB flash drive, memory card, external hard drive, writeable CD or DVD) that can store information. *Storage security* is the process of allowing only authorized parties to access and use stored information.

1.2 The Need of Storage Security

In today's computing environment, there are many threats to the confidentiality of information stored on end user devices. Some threats are unintentional, such as human error, while others are intentional. Intentional threats are posed by people with many different motivations, including causing mischief and disruption and committing identity theft and other fraud. One of the most common threats is *malware*, also known as *malicious code*, which refers to a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or OS. Types of malware threats include viruses, worms, malicious mobile code, Trojan horses, rootkits, and spyware. Malware can give attackers unauthorized access to a device, transfer information from the device to an attacker's system, and perform other actions that jeopardize the confidentiality of the information on a device. Another common threat against end user devices is device loss or theft. Someone with physical access to a device has many options for attempting to view the information stored on the device. This is also a concern for insider attacks, such as an employee attempting to access sensitive information stored on another employee's device. Another form of insider attack is a user attempting to access another user's files on a device that the two users share.[3]

Many threats against end user devices could cause information stored on the devices to be accessed by unauthorized parties. To prevent such disclosures of information, particularly of personally identifiable information and other sensitive data, the information needs to be secured. Securing other components of end user devices, such as OSs, is also necessary, but in many cases additional measures are needed to secure the stored information. Without implementing additional measures like encryption, authentication tools etc, an attacker that steals a device could use forensic tools and techniques to recover information directly from the storage media, circumventing the protections applied by the device's OS.

1.3 Security Controls for Storage

The primary security controls for restricting access to sensitive information stored on end user devices are encryption and authentication. Encryption can be applied granularly, such as to an individual file containing sensitive information, or broadly, such as encrypting all stored data. The appropriate encryption solution for a particular situation depends primarily upon the type of storage, the amount of information that needs to be protected, the environments where the storage will be located, and the threats that need to be mitigated. Storage encryption solutions require users to authenticate successfully before accessing the information that has been encrypted. Common authentication mechanisms are passwords, personal identification numbers (PIN), cryptographic tokens, biometrics, and smart cards. The combination of encryption and authentication helps control access to the stored information.

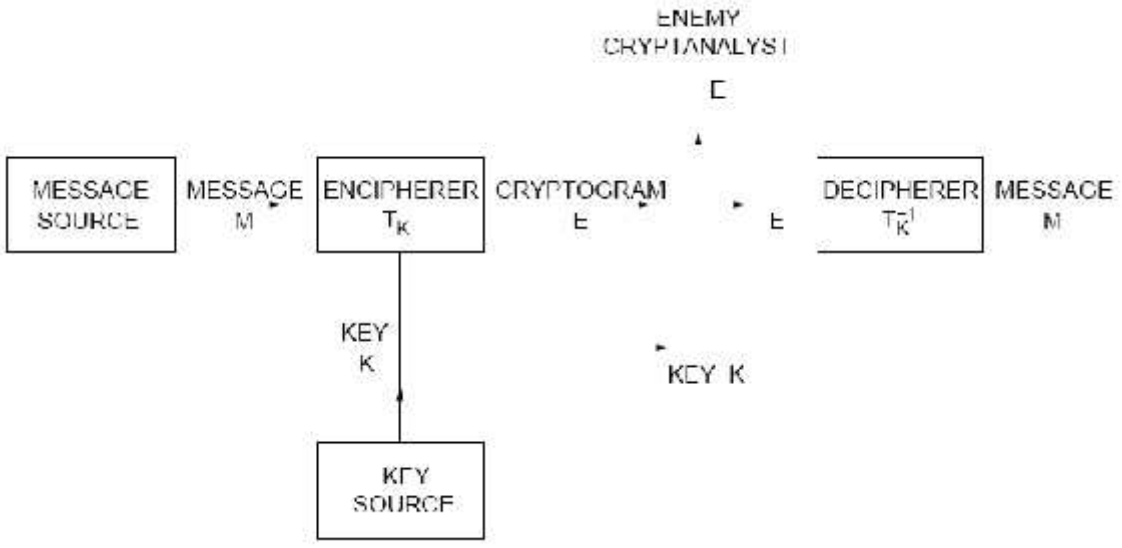
Organizations also need to consider the security of backups of stored information. Some organizations permit users to back up their local files to a centralized system, while other organizations recommend that their users perform local backups (e.g., burning CDs, external USB storage media). In the latter case, organizations should ensure that the backups will be secured at least as well as the original source. This could be done with

similar controls, such as encrypting the backups, or with different types of controls, such as storing backup tapes in a physically secured room within the organization's facilities.

Organizations should also implement other measures that support and complement storage encryption implementations such as full disk encryption, volume and virtual disk encryption and file/folder encryption. These measures help to ensure that storage encryption is implemented in an environment with the management, operational, and technical controls necessary to provide adequate security for the storage encryption implementation.

1.4 Secrecy System

A secrecy system is defined abstractly as a set of transformations of one space (the set of possible messages) into a second space (the set of possible cryptograms). Each particular transformation of the set corresponds to enciphering with a particular key. The transformations are supposed reversible (non-singular) so that unique deciphering is possible when the key is known [2]. A secrecy system can be represented in various ways. We can represent a secrecy system using a line diagram as



[Figure 1: Schematic diagram of simple secrecy system]

If M is the message, K the key, and E the enciphered message, we have,

$$E = f(M, K)$$

1.5 Valuation of Secrecy System

There are a number of different criteria that should be applied in estimating the value of a proposed secrecy system. The most important of these are:

Amount of Secrecy

There are some systems that are perfect—the enemy is no better off after intercepting any amount of material than before. Other systems, although giving him some information, do not yield a unique "solution" to intercepted cryptograms. Among the uniquely solvable systems, there are wide variations in the amount of labor required to affect this solution and in the amount of material that must be intercepted to make the solution unique.

Size of Key

The key must be transmitted by imperceptible means from transmitting to receiving points. Sometimes it must be memorized. It is therefore desirable to have the key as small as possible but should not be in such pattern that it is susceptible to cryptanalytic or brute force attack.

Complexity of Enciphering and Deciphering Operations

Enciphering and deciphering should be as simple as possible. If they are done manually, complexity leads to loss of time, errors, etc. If done mechanically, complexity leads to large expensive machines.

Propagation of Errors

In certain types of ciphers, an error of one letter in enciphering or transmission leads to a large number of errors in the deciphered text. The errors are spread out by the deciphering operation, causing the loss of much information and frequent need for repetition of the cryptogram. It is naturally desirable to minimize this error expansion.

Expansion of Message

In some types of secrecy systems the size of the message is increased by the enciphering process. This undesirable effect may be seen in systems where one attempts to swamp out message statistics by the addition of many nulls, or where multiple substitutes are used. It also occurs in many “concealment” types of systems.

1.6 Motivation

The need of having data being secure is the zest of cryptography. For this designing and devising new concept will never have a stoppage. There has been a great deal of discussion of the security of Data Encryption Standard (DES) in the open literature. Most of it has been favorable to DES [4], but there are a few indications that it would be wise to supplement the aging DES and replace it. DES has been in use since 1977, and has been used in a large number of applications where people have had many possible motivations for trying to break this cipher. During this time, it is possible that someone has discovered a computationally feasible method for doing so [5]. Under such circumstances, it is highly unlikely that such a discovery would be made known. One of the closest things of breaking DES is a story of the FBI successfully decrypting a file of drug transaction records that were encrypted on a PC using a DES board [6]. The DES board that the criminal used has an algorithm to generate a key from a word or phrase. By an exhaustive search of an English dictionary and key names from the criminal’s family

and friends using a supercomputer, the file was solved. This indicates some weakness in DES.

On the other hand, the weaknesses of DES have been overcome after evolution of Secure Hash Standard (SHS). This Standard specifies five secure hash algorithms [7] - SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 - for computing a condensed representation of electronic data (message). When a message of any length less than 2^{64} bits (for SHA-1, SHA-224 and SHA-256) or less than 2^{128} bits (for SHA-384 and SHA-512) is input to a hash algorithm, the result is an output called a **message digest**. The message digests range in length from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).

1.7 Approach

The basic approach is to implement SHA in a simplified way such that the new concept will fulfill requirements of SHS. The design of the algorithm and its basics has been defined in the proposed solution section. Different algorithms and variation on those algorithms have been studied in order to acquire the broad knowledge on the implementation of new concept such that the repetition of already accomplished idea is omitted.

CHAPTER 2

LITERATURE SURVEY

2.1 Cryptography

Cryptography, which is the study of methods for sending messages in secret (namely, in enciphered or disguised form) so that only the intended recipient can remove the disguise and read the message (or decipher it)[8]. Cryptography has, as its etymology, kryptos from the Greek, meaning hidden, and graphein, meaning to write. The original message is called the plaintext, and the disguised message is called the cipher text. The final message, encapsulated and sent, is called a cryptogram. The process of transforming plaintext into ciphertext is called encryption or enciphering. The reverse process of turning ciphertext into plaintext, which is accomplished by the recipient who has the knowledge to remove the disguise, is called decryption or deciphering. Anyone who engages in cryptography is called a cryptographer. On the other hand, the study of mathematical techniques for attempting to defeat cryptographic methods is called cryptanalysis. Those practicing cryptanalysis (usually termed the “enemy”) are called cryptanalysts. The term cryptology is used to embody the study of both cryptography and cryptanalysis, and the practitioners of cryptology are cryptologists. The etymology of cryptology is the greek kryptos meaning hidden and logos meaning word. Also, the term cipher is a method for enciphering and deciphering.

2.2 Cipher Models

Different cipher models are present in the world of cryptography. However they are more often categorized in two unique models. Symmetric Model is the single key, one key or conventional where as Asymmetric Model is public key cryptography.

2.2.1 Symmetric Model

In most symmetric algorithms, the encryption key and the decryption key are the same. These algorithms, also called secret-key algorithms, single-key algorithms or one-key algorithms, require that he sender and receiver agree on a key before they can communicate securely. The security of a symmetric algorithm rests in the key; divulging the key means that anyone could encrypt and decrypt messages. As long as the communication needs to remain secret the key must remain secret [9].

Encryption and decryption with a symmetric algorithm are denoted by:

$E_K(M)=C$

$D_K(C)=M$

Symmetric algorithms operate under two categories:

- Block Cipher: Symmetric algorithm operating on the plaintext in groups of bits called Block and the algorithms are called Block Ciphers.
- Stream Cipher: Symmetric algorithm operating on the plaintext a single bit(or sometime byte) at a time are called Stream Ciphers

2.2.1.1 Substitution Technique

A substitution technique [10] is one in which the letters of plain text are replaced by other letter or by numbers or symbols. If the plain text is viewed as a sequence of bits then substitution involves replacing plain text bit patterns with cipher text bit pattern. In this cipher each letter of the message is replaced by a fixed substitute, usually also a letter. Thus the message,

$M = m_1m_2m_3m_4\ldots$ (Where, m_1, m_2 … are the successive letters.)

Becomes:

$$E = e_1e_2e_3e_4\ldots$$

$$= f(m_1)f(m_2)f(m_3)f(m_4)\ldots$$

Where, the function f (m) is a function with an inverse. The key is a permutation of the alphabets.

The simple substitution cryptogram and a variation by Julius Caesar 'Caesar cipher' were popular classical cryptograms. The main weakness of substitution technique is that one can choose fixed permutation of the alphabet space of the plain text and if the frequency of occurrence of the characters is known, it is easy to recover the plain text with very less effort.

2.2.1.2 Transposition Technique

It refers to the changing of character position in the plain text to generate some cipher text[6] . A very different kind of mapping is achieved by performing some sort of permutation on the plain text letters. The message M is divided into groups of length d and a permutation applied to the first group, the same permutation to the second group, etc. The permutation is the key and can be represented by a permutation of the first d integers. Thus for d = 5, we might have 2 3 1 5 4 as the permutation. This means that:

$m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8\ m_9\ m_{10}$ …….

Becomes

$m_2\ m_3\ m_1\ m_5\ m_4\ m_7\ m_8\ m_6\ m_{10}\ m_9$ …….

Sequential application of two or more transpositions will be called compound transposition. A pure transposition cipher is easily recognized, because it has the same letter frequency as the original plain text.

A transposition cipher can be made more secure by performing more than one stage of transpositions.

2.2.2 Asymmetric Model:

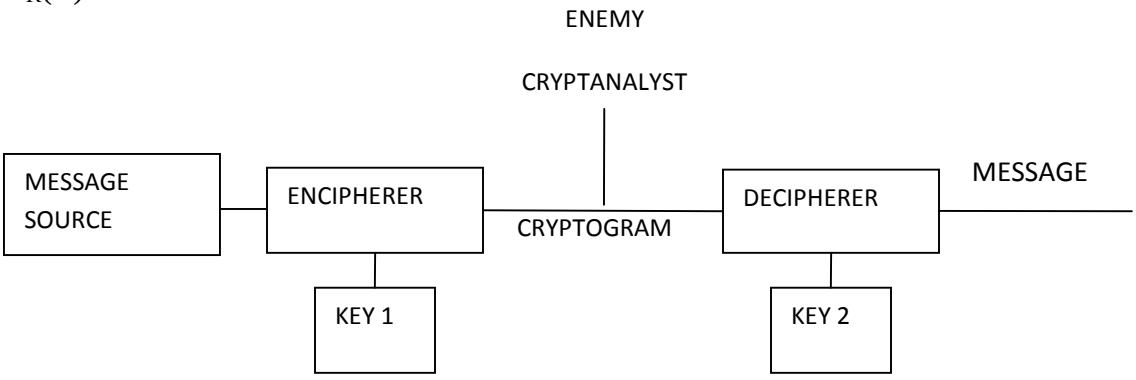
Asymmetric algorithms [4] or Public-Key algorithms are designed so that the key used for encryption is different from the key used for decryption. Furthermore, the decryption key cannot (at least in any reasonable amount of time) be calculated from the encryption key. The algorithms are called “public-Key” because the encryption key can be made public: A complete stranger can use the encryption key to encrypt a message, but only a specific person with the corresponding decryption key can decrypt the message. In these systems, the encryption key is often called the public key, and the decryption key is often called the private key. The private key is sometimes also called the secret key.

Encryption using public key K is denoted by:

$E_K(M)=C$

Even though the public key and private key are different, decryption with the corresponding private key is denoted by:

$D_K(C)=M$



[Figure 2. Asymmetric Cryptogram]

Since many encryption keys can be used to encrypt the plain text and a single decryption key is used to recover the plain text; this model some time known as public key

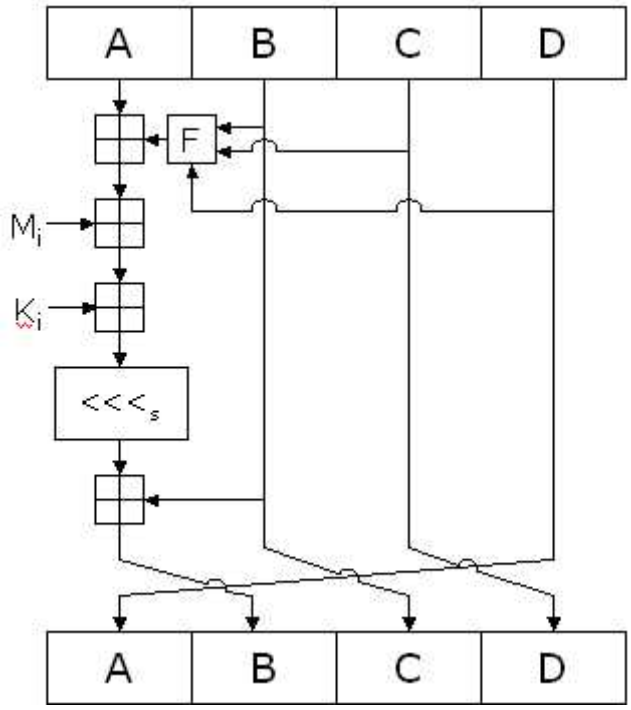
cryptography and the encryption and decryption keys are respectively known as public and private keys.

2.2.3 Hash Function

Hash Function are used to transform a message of arbitrary length into a “message digest” of a fixed, and relatively small, length. They are one-way functions and the output varies with even minor changes in a large document, so these are effective in Message Authentication Code (MAC) or Modification Detection Code (MDC). MD5 and SHA are commonly used hash function [11].

2.2.3.1 MD5

MD5[12] processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit little endian integers); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with a 64-bit integer representing the length of the original message, in bits.



[**Figure 3**. MD5 operation with one operation within a round]

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C* and *D*. These are initialized to certain fixed constants. The main algorithm then operates on each 512-bit message block in turn, each block modifying the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation. Figure illustrates one operation within a round. There are four possible functions *F*; a different one is used in each round:

$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$

$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$

$H(X,Y,Z) = X \oplus Y \oplus Z$

$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$

2.2.3.2 Secure Hash Algorithms 1 (SHA-1)

SHA-1 [12] produces a 160 bit hash and is built upon the same design principles as MD4 and MD5. These hash functions use an iterative procedure. The original message m is broken into a set of fixed size blocks, m = [m1,m2,…..,ml], where the last block is padded to fill out the block. The message blocks are then processed via a sequence of rounds that use a compression function h', which combines the current block and the result from the previous round. That is, we start with an initial value X_0 , and define $X_j = h'(X_{j-1}, m_j)$. The final X_l is the message digest.

The trick behind building a hash function is to device a good compression function. This compression function should be built in such a way as to make each input bit affect as many output bits as possible. Once main difference between SHA-1 and MD family is that for SHA-1 the input bits are used more often during the course of the hash function than they are for MD4 or MD5. This more conservative approach makes the design of SHA-1 more secure than either MD4 or MD5, but also makes it a little slower.

SHA-1 begins by taking the original message and padding it with a 1 bit followed by a sequence of 0 bits. Enough 0 bits are appended to make the new message 64 bits short of the next highest multiple of 512 bits in length. Following the appending of 1s and 0s, we append the 64-bit representation of the length T of the message. Thus, if the message is T bits, then the appending creates a message that consists of $L = \lfloor T/512 \rfloor + 1$ blocks of 512 bits. We break the appended message into L blocks $m_1, m_2, \ldots\ldots m_L$. The hash algorithm inputs these blocks one by one.

For example, if the original message has 2800 bits, we add a 1 and 207 0s to obtain a new message of length 3008 = 6 * 512 -64. Since 2800 = 101011110000_2 in binary, we append fifty-two 0s followed by 101011110000 to obtain a message of length 3072. This is broken into six blocks of length 512.

2.2.4 Quantum Encryption

G Brassard and C Bennett introduced BB84 protocol, based on the idea of using quantum mechanics to solve key distribution problem. It uses light particles to communicate instead of bits. A light particle 'photon' can have one of the four orientations, Horizontal, Vertical, 45^0 diagonal and -45^0 diagonal. Each of these represent a bit – and / represents a 0; and | & \ represents 1. Each bit in the plain text is converted randomly in one of the two orientations connected with that bit and is transferred via fiber optic cable.

2.2.5 Noise addition

Adding some extra or unnecessary data to the plain text or to the resulting cipher text makes the cipher text more difficult against cryptanalysis. The unnecessary data is known as noise. Such noise may be added on the plain text or it can also be added to the cipher text. One of the popular noise additions is sequencing in which the Shift Register Sequencing technique results some noise on cipher text to change the period of the space.

Such noise may be in the form of electrical signal, some sound waves or some color code.

2.2.6 Steganography

Steganography is the scheme in which the actual message is covered by some object and transmitted to the receiver **Error! Reference source not found.**. The basic model of steganography consists of Carrier, message, embedding algorithm and stego key.

The carrier is the object in which the message is embedded by using embedding algorithm along with stego key. The covered object with secretly embedded message is known as stego object [36].

2.2.7 The Needham- Schroeder Public Key protocol

This protocol aims to establish mutual authentication between initiator A and a responder B. Each agent A possesses a public key Ka, which any other agent can obtain from a key server. There will be a secret key, Ka-1 which is inverse of Ka.

The protocol uses nonces: random numbers generated with the purpose of being used in a single run of the protocol. The nonces can be denoted by Na and Nb generated by A and B respectively. The protocol can be described as

Message 1. A→ B : A.B.{Na. A}PK(B)

Message 2. B→A : B.A.{Na.Nb}PK(A)

Message 3. A→ B : A.B.{Nb}PK(B)

CHAPTER 3

PROBLEM DEFINITION

The fundamental aim of cryptography is to make the data or any message secure since there have been lots of effort by the eavesdroppers to infiltrate into the original data or message posing the risk of information loss or theft. Various kinds of attacks are being observed though out the internet world or even from the trusted person.

3.1 Problem Definition

Different cryptographic algorithms and developed tools have been studied. Despite of their application there have always been a key issue related to the security. Several attacks on the information have resulted in the design of new concept and methodology.

This work addresses the problem of individual privacy and confidentiality in cryptographic communication. Consider a situation in which the encrypted communication is intercepted by an authority, and subsequently the sender is coerced to reveal the keys that generated the cipher text; the result of which is that the content of the message sent is revealed.

In this work it has been tried to present a solution which tries to solve some of these issues. The solution would enhance a step further in a long run of designing a strong mechanism of security that results in very less damaging the privacy and confidentiality of the sent or stored information.

3.2 Proposed Solution

As the integrity and security is the prime concern of encryption mechanism, I have proposed a unique mechanism to enhance the security of data using multiple keys. There will be two keys Key1 and Key2. Key1 will be an integer value (1 to 4294967296) ie, from 2^0 to 2^{32} ,which will be seeded to the Pseudo Random Number Generator(PRNG)

and will be applied for authentication purpose. Key2 will be a pass phrase of string type which will be a shared public key. The overall procedure can be sub divided into four sub-procedures.

1. PseudoRandom Number Generation of SubKey
2. Generation of Hash values using SHA-256
3. Generation of Plaintext modulo 256
4. Encryption Process

In symmetric encryption, the decryption is the reverse process of encryption. The plaintext can be recovered by applying above algorithm in reverse order provided that Key 1 and Key 2 are available. Decryption starts with the first block which has a known H element and PRNG. p(1) will be recovered using k(1) and the first byte of h(1) since the value of a(0) will be 0. Since a(1) will be equal to p(1) and a(i) can be regenerated as each p(i) is recovered. After decryption of each 32 byte block, the hash for decrypting the next 32 byte block can be calculated.

CHAPTER 4

STUDY OF ALGORITHMS

4.1 Secure Hash Standard

The Secure Hash Standard(SHS) specifies five secure hash algorithms[7], SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. All five of the algorithms are iterative, one-way hash functions that can process a message to produce a condensed representation called a **message digest**. These algorithms enable the determination of a message's integrity: any change to the message will, with a very high probability, result in a different message digests. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers or bits[7].

Each algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves padding the padded message into m-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a message schedule from the padded message and uses that schedule, along with functions, constants, and word operation to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest.

The five algorithms differ most significantly in the security strengths that are provided for the data being hashed. The security strengths of these five hash functions and the system as a whole when each of them is used with other cryptographic algorithms, such as digital signature algorithms, and keyed- hash message authentication codes.

Additionally, the five algorithms differ in terms of the size of the blocks and words of data that are used during hashing. Figure represents the basic properties of these hash algorithms.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SH-512	$< 2^{128}$	1024	64	512

[**Figure 4**: Secure Hash Algorithm Properties]

4.2 SHA-256

SHA-256 is used to hash a message, *M*, having a length of *l* bits, where $0 \le l < 2^{64}$. The algorithm uses

1) a message schedule of sixty-four 32-bit words,

2) eight working variables of 32 bits each, and

3) a hash value of eight 32-bit words.

The final result of SHA-256 is a 256-bit message digest.

The words of the message schedule are labeled W_0, W_1,…, W_{63}. The eight working variables are labeled ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***. The words of the hash value are labeled $H_0^{(i)}$, $H_1^{(i)}$,….,$H_7^{(i)}$ which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-256 also uses two temporary words, *T*1 and *T*2.

4.2.1 SHA-256 Functions

SHA 256 use six logical functions, where each function operates on 32-bit words, which are represented as x,y, and z. The result of each function is a new 32-bit word.

$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$

$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$

$\sum_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$

$\sum_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$

$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$

$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

4.2.2 SHA-256 Constants

SHA-256 use the sequence of sixty-four constant 32-bit words, $K_0^{256}, K_1^{256}, \ldots\ldots K_{63}^{256}$. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are (from left to right) as follows:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

4.2.3 SHA-256 Preprocessing

Preprocessing take place before hash computation begins. This preprocessing consists of three steps: padding the message, *M,* parsing the padded message into message blocks, and setting the initial hash value, *H*(0).

4.2.3.1 Padding the message

The message, M, is padded before computing hash. The purpose of this padding is to ensure that the padded message is a multiple of 512 bits. Suppose that the length of the message, *M*, is *l* bits. Bit "1" is appended to the end of the message, followed by *k* zero bits, where *k* is the smallest, non-negative solution to the equation . Then the 64-bit block that is equal to the number expressed using a binary representation is appended.

4.2.3.2 Parsing the Padded Message

After the message has been padded, it is then parsed into N 512-bit blocks before the hash computation begins. Let each blocks be $M^{(1)}$, $M^{(2)}$,……,$M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block *i* are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$ and so on up to $M_{15}^{(i)}$

4.2.3.3 Setting the initial Hash value $H^{(0)}$

Before hash computation begins, the initial hash value $H^{(0)}$, must be set. The size and number of words in $H^{(0)}$ depends on the message digest size. For SHA-256, the initial hash value, H(0) consists of the following eight 32-bit words in hex:

$H_0^{(0)}$ = 6a09e667
$H_1^{(0)}$ = bb67ae85
$H_2^{(0)}$ = 3c6ef372
$H_3^{(0)}$ = a54ff53a
$H_4^{(0)}$ = 510e527f
$H_5^{(0)}$ = 9b05688c

$H_6^{(0)}$ = 1f83d9ab

$H_7^{(0)}$ = 5b30cd19

These words are obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

4.2.4 SHA-256 Initial Operation

Procedure

1. Pad the message, M, according to sec. 4.2.3.1
2. Parse the padded message into N 512-bit message blocks, $M^{(1)}$, $M^{(2)}$, ….., $M^{(N)}$ according to sec. 4.2.3.2
3. Set the initial hash value, H(0), as specified in Sec. 4.2.3.3

4.2.5 SHA-256 Hash Computation

The SHA-256 hash computation uses functions and constants. The addition (+) is performed modulo 2^{32} .

After preprocessing is completed, each message block $M^{(1)}$, $M^{(2)}$,……,$M^{(N)}$, is processed in order, using the following steps:

For I = 1 to N:
{

1. Prepare the message schedule, $\{W_t\}$:

   W_t = Mt(i) for $0 \leq t \leq 15$ and
   $W_t = \pm_1^{256}(W_{t-2}) + W_{t-7} + \pm_0^{256}(W_{t-15}) + W_{t-16}$ for $16 \leq t \leq 63$

2. Initialize the eight working variables, a, b, c, d, e, f, g, and h, with the $(i\text{-}1)^{st}$ hash value:

   a = $H_0^{(i-1)}$

   b = $H_1^{(i-1)}$

   c = $H_2^{(i-1)}$

   d = $H_3^{(i-1)}$

   e = $H_4^{(i-1)}$

$f = H_5^{(i-1)}$

$g = H_6^{(i-1)}$

$h = H_7^{(i-1)}$

3. For t = 0 to 63:

{

$T_1 = h + \sum_1^{256}(e) + Ch(e,f,g) + K_t^{256} + W_t$

$T2 = \sum_0^{256}(a) + Maj(a,b,c)$

$h = g$

$g = f$

$f = e$

$e = d + T_1$

$d = c$

$c = b$

$b = a$

$a = T_1 + T_2$

}

4. Compute the i^{th} intermediate hash value $H^{(i)}$

$H_0^{(i)} = a + H_0^{(i-1)}$

$H_1^{(i)} = b + H_1^{(i-1)}$

$H_2^{(i)} = c + H_2^{(i-1)}$

$H_3^{(i)} = d + H_3^{(i-1)}$

$H_4^{(i)} = e + H_4^{(i-1)}$

$H_5^{(i)} = f + H_5^{(i-1)}$

$H_6^{(i)} = g + H_6^{(i-1)}$

$H_7^{(i)} = h + H_7^{(i-1)}$

}

After repeating steps one through four a total of N times (i.e, after processing M(N)), the resulting 256-bit message digest of the message, M, is

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} \,||\, H_3^{(N)} \,||\, H_4^{(N)} \,||\, H_5^{(N)} \,||\, H_6^{(N)} \,||\, H_7^{(N)}$$

CHAPTER 5

DESIGN AND IMPLEMENTATION

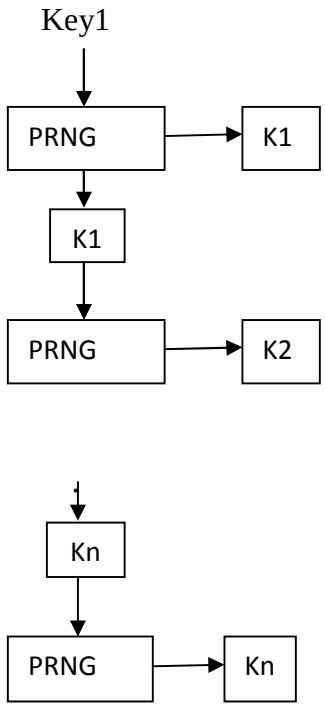
The detailed design of the proposed solution and its implementation methodology has been discussed in this chapter.

5.1 Pseudo Random Number Generation of Key

The PRNG will be seeded with Key1 which is an integer value (System date or Key1 or any other entropy sources). Here I have defined the manual input of integer value ranging from 2^0 to 2^{32}. The PRNG will generate a 32 byte block.

PRNG = {k(1), k(2),…….,k(n)}

Where k(i) is the ith output of the PRNG



[**Figure 5**: Generation of k(i)'s]

5.2 Generation of Hash Elements using SHA-256

The input to the SHA-256 will be Key2 which is a pass phrase and will generate its respective hash value. The reason for adding initial random 32 bytes to the message is to thwart known plain text attack since this random string will diffuse into the rest of the message via the hashes. Further, encrypting the exact same message twice will result in completely different ciphertext being generated.

Procedure:

H={h(1), h(2),…..,h(n)} = SHA-256 hashes.

Where h(1) = SHA-256(Key2)

And each h(i)=SHA-256(Previous 32bytes of P with h(i-1) appended)

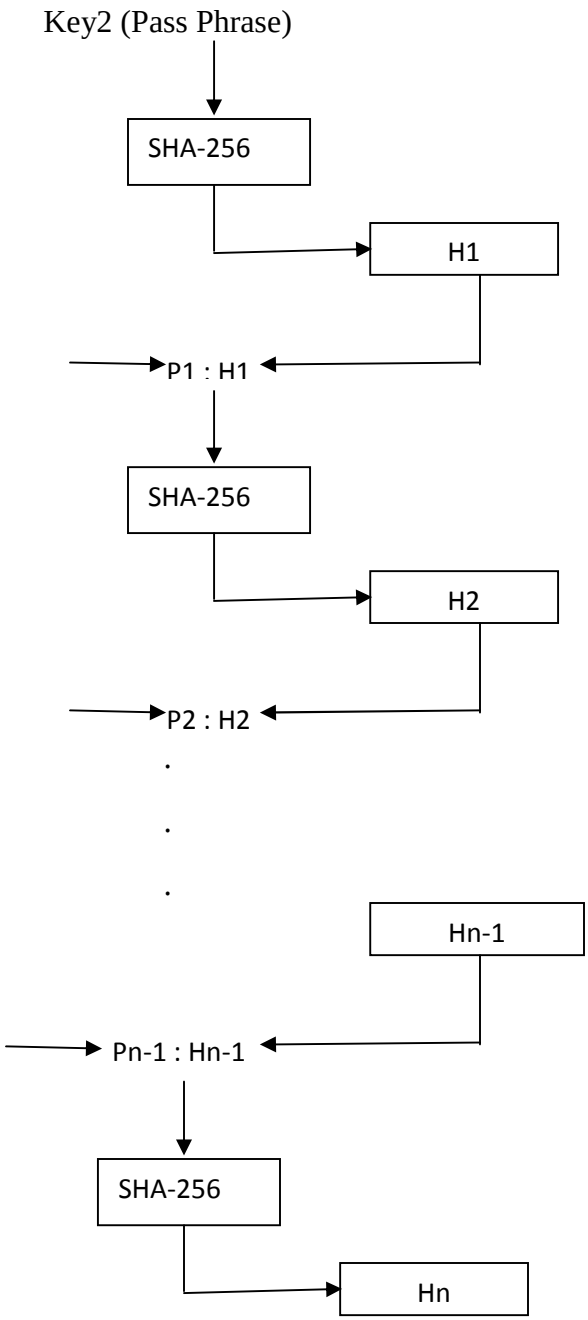
That is, h(2)=SHA-256(p(1) : h(1))

h(2)=SHA-256(p(2) : h(2))

….

h(n)=SHA256(p(n-1) : h(n-1)

The process can be defined by the following diagram



[**Figure 6**. Generation of h(i)'s]

5.3 Generation of Plaintext modulo 256

This section will define the generation of plaintext modulo 256. As the operation of algorithm will be carried out in SHA-256, the plain texts have been segmented using modulo function of 256 such that the range of binary value will be between 0 and 255.

Procedure :

P= {p(1),p(2),……p(n)} where P is the plain text and each p(i)'s are of 32 byte block.

A = {a(0), a(1),……..,a(n)}

where a(0)=0,

a(i)=(a(i-1)+p(i)) % 256 (5.3.1)

Now,

Define a(0)= 0

Then,

a(1)= (a(0)+p(1))%256

a(2)= (a(1)+p(2))%256

.

.

a(n)=(a(n-1)+p(n))%256

These a(i)'s are later XOR'ed with p(i)'s and k(i)'s to generate intermediate ciphertext.

5.4 Encryption Process

The encryption process is carried out in two sub processes. Initially the intermediate cipher text by XORing the chunks of plain text ie, P(i)'s, the PRNG values, ie, K(i)'s, and the plaintext modulo 256(A).

Procedure:

M= {m(1),m(2),…….,m(n) = Intermediate CipherText

Where

m(1) = p(1) XOR k(1) XOR a(0)

m(2) = p(2) XOR k(2) XOR a(1)

.

.

m(i) = p(i) XOR k(i) XOR a(i-1)

The intermediate cipher text so generated is now broken down into 32 bytes blocks. Now there is one-to-one correspondence between the M blocks and the 32 byte H elements.

Now the second part consists of generation of actual cipher Text C which is done by XORing each M block against its corresponding H elements.

ie, C = {c(1), c(2), ……., c(n)}

where, c(1) = m(1) XOR h(1)

c(2) = m(2) XOR h(2)

c(n) = m(n) XOR h(n)

5.5 Algorithm

Step 1 : PRNG will be keyed with some entropy sources (System Time or any integer value) (key1)

Step 2 : PRNG :- Generation of k = {k(1), k(2),……, k(n)

Step 3 : Input pass phrase (Key2)

Step 4 : Generate Hash value using SHA-256

Key2 →SHA-256→H elements

(H elements are of 32 byte)

Step 5 : Input Data Source (Plain text (P))

Step 6 : A(i)=a(i-1) + p(i)%256

Where a(0)= 0, a(1)=a(0)+(p1)%256 and so on….

Step 7 : Generate Intermediate cipher text (M)

Where M(i)= p(i) XOR k(i) XOR a(i-1)

(M is a stream cipher)

Step 8 : M is broken into 32 byte blocks (M=m1,m2……mn) ie, m1=1st block, m2=2nd block and so on

Step 9 : Cascading of the result of step 4 and step 8

m1 XOR h1 = c1

m2 XOR h2 = c2

Step 10 : Output result C, where C=C1,C2….Cn

5.6 Implementation of Algorithm

5.6.1 Tools

Borland C++ 5.02

C++ is an Object oriented programming language developed by Strups Bazarne. Borland C++ 5.02 compiler is used to compile the codes written in C++. It provided interactive and easy environment to implement the algorithm.

CHAPTER 6

TESTING AND ANALYSIS

The overall process and its implementation were tested for encryption. The example below is a test data supplied for the developed system. The size of the source file after encryption will be increased. The reason for increment in the size of cipher text is to thwart known plain text attacks since the string will diffuse into the rest of message via hashes.

6.1 Execution of Program

The program is executed as following:

C:\> scrambler [Data to be encrypted] [Key1] [Key2]

6.2 Testing

Testing has been carried out with various test data. Following are the output result supplied with different keys.

Sourcedata1:

AAAAAAAAAAAA……..

File Size: 4342 bytes

Key1 : 23423

Key2: hello

Ciphertext :

9NYIQ!<Dk]A\Ú--ÅONW.:E8iU6´0 :õ[LFm0N(ÉMaJÑ '-DUINm0Nkâxeüõ0'üwL m)R$÷nT}Å? $N_](4 * Y_Ë|Ñ:^TO(7Tk^DvÎ 9ÏRO……..

Ciphertext Size: 4374 bytes

Sourcedata2:

BBBBBBBBBBBBBBB……..

File Size: 5645 bytes

Key 1: 47678987

Key2: Jupiter

Ciphertext: OS>< "t*ƒ*ÐRÏÛ ^TSE#- 8}pá .?ü:Z. ,R9S^¹- öFX F? õ 9RPG+8S?‘ rg Ê5: ASI <)t>a(3…….

Ciphertext Size: 5677 bytes

Sourcedata3:

CCCCCCCCCCCCCCCCC…….

File Size: 3297 bytes

Key1: 548970

Key2: Kathmandu

CipherText :

ü&Û¹OgF‘ð»×¯ ÔxŽ↑ n©äüq½- nÃ¼Ôn <ë¿çƱ@Ür¯7¢©\-
!)öú.D¦ÏàÞ¢05uAíÕ→ éˆ2ž8^,° {pB3(¼‘y¯,K.‰¤v…G▪ └ äØŸ}åeû¥ ÎE……

CipherText Size: 3329 bytes

6.3 Analysis:

The algorithm has been fed with various test datasource along with keys. It has been observed that the algorithm produces the same output if the data source and the keys are same. It has also been observed that a minor change in key bit can generate huge variation in cipher text.

Following data depicts the variation in ciphertext when key bits are changed.

S.N	Number of bit changed in Key1	Number of bit changed in Key2	Number of bits changed in cipher text
1	1	0	57
2	2	0	63
3	3	0	126
4	0	1	26
5	0	2	41
6	0	3	94
7	1	1	137
8	1	2	189
9	2	2	231
10	2	3	47

[**Figure 7**. Bit Variation Analysis Chart]

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion :

The security of encrypted message is somehow vulnerable to the unprecedented attacks by unknown users. The main clue that the attackers find to decrypt the message is by analyzing the size of the encrypted file. This work has implemented a procedure to restrict such attack by increasing the original data by initial random 32 bytes such that the attackers will be unable to decrypt until and unless they knew the actual size of original data. The addition of 32 bytes has brought a significant secrecy in the encrypted file. Similarly the implementation of two keys with different types has moreover denied the chances of being decrypted or theft of the secured data.

7.2 Future Works

The cryptanalysis of above algorithm may be the interesting research work. Furthermore defining the algorithm with some variations in plaintext modulo 256 and use of other random number generators may be done in the future. As this work has been done on storage security, it can be further extended in data communication for confidentiality as well.

APPENDIX

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <comutil.h>
#include <time.h>
#include <Winbase.h>
#include <Windows.h>


#ifndef _SHA2_H
#define _SHA2_H

#include <limits.h>



#if defined(__cplusplus)
extern "C"
{
#endif

#define BYTES_PER_BLOCK     32

#define SHA256_DIGEST_SIZE  32
#define SHA384_DIGEST_SIZE  48
#define SHA512_DIGEST_SIZE  64

#define SHA256_BLOCK_SIZE   64
#define SHA384_BLOCK_SIZE  128
#define SHA512_BLOCK_SIZE  128

#define SHA2_DIGEST_SIZE       SHA256_DIGEST_SIZE
#define SHA2_MAX_DIGEST_SIZE    SHA512_DIGEST_SIZE

#define SHA2_GOOD   0
#define SHA2_BAD    1

/* type to hold the SHA256 context                          */

typedef struct
{   sha2_32t count[2];
    sha2_32t hash[8];
    sha2_32t wbuf[16];
} sha256_ctx;

/* type to hold the SHA384/512 context                      */
```

```
typedef struct
{   sha2_64t count[2];
   sha2_64t hash[8];
   sha2_64t wbuf[16];
} sha512_ctx;

typedef sha512_ctx  sha384_ctx;

/* type to hold a SHA2 context (256/384/512)  */

typedef struct
{   union
   {   sha256_ctx  ctx256[1];
      sha512_ctx  ctx512[1];
   } uu[1];
   sha2_32t    sha2_len;
} sha2_ctx;

void sha256_compile(sha256_ctx ctx[1]);
void sha512_compile(sha512_ctx ctx[1]);

void sha256_begin(sha256_ctx ctx[1]);
void sha256_hash(const unsigned char data[], unsigned long len, sha256_ctx ctx[1]);
void sha256_end(unsigned char hval[], sha256_ctx ctx[1]);
void sha256(unsigned char hval[], const unsigned char data[], unsigned long len);

void sha384_begin(sha384_ctx ctx[1]);
#define sha384_hash sha512_hash
void sha384_end(unsigned char hval[], sha384_ctx ctx[1]);
void sha384(unsigned char hval[], const unsigned char data[], unsigned long len);

void sha512_begin(sha512_ctx ctx[1]);
void sha512_hash(const unsigned char data[], unsigned long len, sha512_ctx ctx[1]);
void sha512_end(unsigned char hval[], sha512_ctx ctx[1]);
void sha512(unsigned char hval[], const unsigned char data[], unsigned long len);

int sha2_begin(unsigned long size, sha2_ctx ctx[1]);
void sha2_hash(const unsigned char data[], unsigned long len, sha2_ctx ctx[1]);
void sha2_end(unsigned char hval[], sha2_ctx ctx[1]);
int sha2(unsigned char hval[], unsigned long size, const unsigned char data[], unsigned long len);

#if defined(__cplusplus)
}
#endif

#endif

#define MAX_WORDS 524288

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
```

```
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

unsigned long genrand_int32(void);

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
static unsigned long accum; /* the array for the state vector  */

/* define the hash functions that you need          */

#define SHA_2          /* for dynamic hash length  */
#define SHA_256
//#define SHA_384
//#define SHA_512

#include <string.h>     /* for memcpy() etc.        */
#include <stdlib.h>     /* for _lrotr with VC++     */

//#include "sha2.h"

/*  1. PLATFORM SPECIFIC INCLUDES */

#if defined(__GNU_LIBRARY__)
#  include <byteswap.h>
#  include <endian.h>
#elif defined(__CRYPTLIB__)
#  if defined( INC_ALL )
#    include "crypt.h"
#  elif defined( INC_CHILD )
#    include "../crypt.h"
#  else
#    include "crypt.h"
#  endif
#  if defined(DATA_LITTLEENDIAN)
#    define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#  else
#    define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#  endif
#elif defined(_MSC_VER)
#  include <stdlib.h>
#elif !defined(WIN32)
#  include <stdlib.h>
#  if !defined (_ENDIAN_H)
#    include <sys/param.h>
#  else
#    include _ENDIAN_H
#  endif
#endif

#define SHA_LITTLE_ENDIAN   1234 /* byte 0 is least significant (i386) */
#define SHA_BIG_ENDIAN      4321 /* byte 0 is most significant (mc68k) */
```

```
#if !defined(PLATFORM_BYTE_ORDER)
#if defined(LITTLE_ENDIAN) || defined(BIG_ENDIAN)
#  if defined(LITTLE_ENDIAN) && defined(BIG_ENDIAN)
#    if defined(BYTE_ORDER)
#      if   (BYTE_ORDER == LITTLE_ENDIAN)
#        define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#      elif (BYTE_ORDER == BIG_ENDIAN)
#        define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#      endif
#    endif
#  elif defined(LITTLE_ENDIAN) && !defined(BIG_ENDIAN)
#    define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#  elif !defined(LITTLE_ENDIAN) && defined(BIG_ENDIAN)
#    define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#  endif
#elif defined(_LITTLE_ENDIAN) || defined(_BIG_ENDIAN)
#  if defined(_LITTLE_ENDIAN) && defined(_BIG_ENDIAN)
#    if defined(_BYTE_ORDER)
#      if   (_BYTE_ORDER == _LITTLE_ENDIAN)
#        define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#      elif (_BYTE_ORDER == _BIG_ENDIAN)
#        define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#      endif
#    endif
#  elif defined(_LITTLE_ENDIAN) && !defined(_BIG_ENDIAN)
#    define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#  elif !defined(_LITTLE_ENDIAN) && defined(_BIG_ENDIAN)
#    define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#  endif
#elif 0     /* **** EDIT HERE IF NECESSARY **** */
#define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#elif 0     /* **** EDIT HERE IF NECESSARY **** */
#define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#elif (('1234' >> 24) == '1')
#  define PLATFORM_BYTE_ORDER SHA_LITTLE_ENDIAN
#elif (('4321' >> 24) == '1')
#  define PLATFORM_BYTE_ORDER SHA_BIG_ENDIAN
#endif
#endif

#if !defined(PLATFORM_BYTE_ORDER)
#  error Please set undetermined byte order (lines 159 or 161 of sha2.c).
#endif

#ifdef _MSC_VER
#pragma intrinsic(memcpy)
#endif

#define rotr32(x,n)   (((x) >> n) | ((x) << (32 - n)))

#if !defined(bswap_32)
#define bswap_32(x) (rotr32((x), 24) & 0x00ff00ff | rotr32((x), 8) & 0xff00ff00)
#endif
```

```
#if (PLATFORM_BYTE_ORDER == SHA_LITTLE_ENDIAN)
#define SWAP_BYTES
#else
#undef  SWAP_BYTES
#endif

#if defined(SHA_2) || defined(SHA_256)

#define SHA256_MASK (SHA256_BLOCK_SIZE - 1)

#if defined(SWAP_BYTES)
#define  bsw_32(p,n)        { int _i = (n);       while(_i--) p[_i] = bswap_32(p[_i]); }
#else
#define  bsw_32(p,n)
#endif

/* SHA256 mixing function definitions   */

#define ch(x,y,z)   (((x) & (y)) ^ (~(x) & (z)))
#define maj(x,y,z)  (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

#define s256_0(x) (rotr32((x),  2) ^ rotr32((x), 13) ^ rotr32((x), 22))
#define s256_1(x) (rotr32((x),  6) ^ rotr32((x), 11) ^ rotr32((x), 25))
#define g256_0(x) (rotr32((x),  7) ^ rotr32((x), 18) ^ ((x) >>  3))
#define g256_1(x) (rotr32((x), 17) ^ rotr32((x), 19) ^ ((x) >> 10))

/* rotated SHA256 round definition. Rather than swapping variables as in    */
/* FIPS-180, different variables are 'rotated' on each round, returning     */
/* to their starting positions every eight rounds                           */

#define h2(i) ctx->wbuf[i & 15] += \
   g256_1(ctx->wbuf[(i + 14) & 15]) + ctx->wbuf[(i + 9) & 15] + g256_0(ctx->wbuf[(i + 1) & 15])

#define h2_cycle(i,j)  \
   v[(7 - i) & 7] += (j ? h2(i) : ctx->wbuf[i & 15]) + k256[i + j] \
      + s256_1(v[(4 - i) & 7]) + ch(v[(4 - i) & 7], v[(5 - i) & 7], v[(6 - i) & 7]); \
   v[(3 - i) & 7] += v[(7 - i) & 7]; \
   v[(7 - i) & 7] += s256_0(v[(0 - i) & 7]) + maj(v[(0 - i) & 7], v[(1 - i) & 7], v[(2 - i) & 7])

/* SHA256 mixing data   */

const sha2_32t k256[64] =
{   n_u32(428a2f98), n_u32(71374491), n_u32(b5c0fbcf), n_u32(e9b5dba5),
   n_u32(3956c25b), n_u32(59f111f1), n_u32(923f82a4), n_u32(ab1c5ed5),
   n_u32(d807aa98), n_u32(12835b01), n_u32(243185be), n_u32(550c7dc3),
   n_u32(72be5d74), n_u32(80deb1fe), n_u32(9bdc06a7), n_u32(c19bf174),
   n_u32(e49b69c1), n_u32(efbe4786), n_u32(0fc19dc6), n_u32(240ca1cc),
   n_u32(2de92c6f), n_u32(4a7484aa), n_u32(5cb0a9dc), n_u32(76f988da),
   n_u32(983e5152), n_u32(a831c66d), n_u32(b00327c8), n_u32(bf597fc7),
   n_u32(c6e00bf3), n_u32(d5a79147), n_u32(06ca6351), n_u32(14292967),
   n_u32(27b70a85), n_u32(2e1b2138), n_u32(4d2c6dfc), n_u32(53380d13),
   n_u32(650a7354), n_u32(766a0abb), n_u32(81c2c92e), n_u32(92722c85),
```

```
    n_u32(a2bfe8a1), n_u32(a81a664b), n_u32(c24b8b70), n_u32(c76c51a3),
    n_u32(d192e819), n_u32(d6990624), n_u32(f40e3585), n_u32(106aa070),
    n_u32(19a4c116), n_u32(1e376c08), n_u32(2748774c), n_u32(34b0bcb5),
    n_u32(391c0cb3), n_u32(4ed8aa4a), n_u32(5b9cca4f), n_u32(682e6ff3),
    n_u32(748f82ee), n_u32(78a5636f), n_u32(84c87814), n_u32(8cc70208),
    n_u32(90befffa), n_u32(a4506ceb), n_u32(bef9a3f7), n_u32(c67178f2),
};

/* SHA256 initialisation data */

const sha2_32t i256[8] =
{
    n_u32(6a09e667), n_u32(bb67ae85), n_u32(3c6ef372), n_u32(a54ff53a),
    n_u32(510e527f), n_u32(9b05688c), n_u32(1f83d9ab), n_u32(5be0cd19)
};


void sha256_begin(sha256_ctx ctx[1])
{
    ctx->count[0] = ctx->count[1] = 0;
    memcpy(ctx->hash, i256, 8 * sizeof(sha2_32t));
}

/* Compile 64 bytes of hash data into SHA256 digest value   */
/* NOTE: this routine assumes that the byte order in the    */
/* ctx->wbuf[] at this point is in such an order that low   */
/* address bytes in the ORIGINAL byte stream placed in this */
/* buffer will now go to the high end of words on BOTH big  */
/* and little endian systems                                */

void sha256_compile(sha256_ctx ctx[1])
{   sha2_32t         v[8], j;

    memcpy(v, ctx->hash, 8 * sizeof(sha2_32t));

    for(j = 0; j < 64; j += 16)
    {
        h2_cycle( 0, j); h2_cycle( 1, j); h2_cycle( 2, j); h2_cycle( 3, j);
        h2_cycle( 4, j); h2_cycle( 5, j); h2_cycle( 6, j); h2_cycle( 7, j);
        h2_cycle( 8, j); h2_cycle( 9, j); h2_cycle(10, j); h2_cycle(11, j);
        h2_cycle(12, j); h2_cycle(13, j); h2_cycle(14, j); h2_cycle(15, j);
    }

    ctx->hash[0] += v[0]; ctx->hash[1] += v[1]; ctx->hash[2] += v[2]; ctx->hash[3] += v[3];
    ctx->hash[4] += v[4]; ctx->hash[5] += v[5]; ctx->hash[6] += v[6]; ctx->hash[7] += v[7];
}

/* SHA256 hash data in an array of bytes into hash buffer   */
/* and call the hash_compile function as required.          */

void sha256_hash(const unsigned char data[], unsigned long len, sha256_ctx ctx[1])
{   sha2_32t pos = (sha2_32t)(ctx->count[0] & SHA256_MASK),
             space = SHA256_BLOCK_SIZE - pos;
```

```
    const unsigned char *sp = data;

    if((ctx->count[0] += len) < len)
        ++(ctx->count[1]);

    while(len >= space)     /* tranfer whole blocks while possible  */
    {
        memcpy(((unsigned char*)ctx->wbuf) + pos, sp, space);
        sp += space; len -= space; space = SHA256_BLOCK_SIZE; pos = 0;
                        bsw_32(ctx->wbuf, SHA256_BLOCK_SIZE >> 2)
        sha256_compile(ctx);
    }

    memcpy(((unsigned char*)ctx->wbuf) + pos, sp, len);
}

/* SHA256 Final padding and digest calculation  */

static sha2_32t  m1[4] =
{
    n_u32(00000000), n_u32(ff000000), n_u32(ffff0000), n_u32(ffffff00)
};

static sha2_32t  b1[4] =
{
    n_u32(80000000), n_u32(00800000), n_u32(00008000), n_u32(00000080)
};

void sha256_end(unsigned char hval[], sha256_ctx ctx[1])
{   sha2_32t    i = (sha2_32t)(ctx->count[0] & SHA256_MASK);

            bsw_32(ctx->wbuf, (i + 3) >> 2)

    if(i > SHA256_BLOCK_SIZE - 9)
    {
        if(i < 60) ctx->wbuf[15] = 0;
        sha256_compile(ctx);
        i = 0;
    }
    else    /* compute a word index for the empty buffer positions  */
        i = (i >> 2) + 1;

    while(i < 14) /* and zero pad all but last two positions      */
        ctx->wbuf[i++] = 0;


    ctx->wbuf[14] = (ctx->count[1] << 3) | (ctx->count[0] >> 29);
    ctx->wbuf[15] = ctx->count[0] << 3;

    sha256_compile(ctx);


    for(i = 0; i < SHA256_DIGEST_SIZE; ++i)
```

```
        hval[i] = (unsigned char)(ctx->hash[i >> 2] >> 8 * (~i & 3));
}

void sha256(unsigned char hval[], const unsigned char data[], unsigned long len)
{   sha256_ctx  cx[1];

    sha256_begin(cx); sha256_hash(data, len, cx); sha256_end(hval, cx);
}

#endif

#if defined(SHA_2)

#define CTX_256(x)  ((x)->uu->ctx256)

/* SHA2 initialisation */

int sha2_begin(unsigned long len, sha2_ctx ctx[1])
{   unsigned long   l = len;
    switch(len)
    {
        case 256:   l = len >> 3;
        case  32:   CTX_256(ctx)->count[0] = CTX_256(ctx)->count[1] = 0;
                    memcpy(CTX_256(ctx)->hash, i256, 32); break;

        default:    return SHA2_BAD;
    }

    ctx->sha2_len = l; return SHA2_GOOD;
}

void sha2_hash(const unsigned char data[], unsigned long len, sha2_ctx ctx[1])
{
    switch(ctx->sha2_len)
    {
        case 32: sha256_hash(data, len, CTX_256(ctx)); return;

    }
}

void sha2_end(unsigned char hval[], sha2_ctx ctx[1])
{
    switch(ctx->sha2_len)
    {
        case 32: sha256_end(hval, CTX_256(ctx)); return;

    }
}

int sha2(unsigned char hval[], unsigned long size,
                        const unsigned char data[], unsigned long len)
{   sha2_ctx    cx[1];
```

```
    if(sha2_begin(256, cx) == SHA2_GOOD)
    {
        sha2_hash(data, len, cx); sha2_end(hval, cx); return SHA2_GOOD;
    }
    else
        return SHA2_BAD;
}

#endif
```

// The random number generator is based on the Mersenne Twistor

```
/* initializes mt[N] with a seed */
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    accum = 0;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
                (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;

    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
          - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
```

```
    if (i>=N) { mt[0] = mt[N-1]; i=1; }
  }

  mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */

  for (i=0; i<64; i++) accum += genrand_int32();

}

/* generates a random number on [0,0xffffffff]-interval */
unsigned long genrand_int32(void)
{
  unsigned long y;
  static unsigned long mag01[2]={0x0UL, MATRIX_A};
  /* mag01[x] = x * MATRIX_A  for x=0,1 */

  if (mti >= N) { /* generate N words at one time */
    int kk;

    if (mti == N+1)   /* if init_genrand() has not been called, */
      init_genrand(5489UL); /* a default initial seed is used */

    for (kk=0;kk<N-M;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
    }
    for (;kk<N-1;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
    }
    y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

    mti = 0;
  }

  y = mt[mti++];

  /* Tempering */
  y ^= (y >> 11);
  y ^= (y << 7) & 0x9d2c5680UL;
  y ^= (y << 15) & 0xefc60000UL;
  y ^= (y >> 18);

  accum += y;

  return accum;
}
/* Main coding starts here*/
#define ENCRYPT 0
#define DECRYPT 1
```

```
int main(int argc, char * argv[])
{
        unsigned long requested_seed;
   FILE * fd_in, * fd_out;
   unsigned char buffer_area[MAX_WORDS], keystream;
        size_t bytes_read = 0;
        int i, j, eof, filename_size, keysize = 0, direction, muddle = 0;
   int last_char, start_fill, blocks_processed = 0;
        char output_filename[1000];
        const char * my_out = output_filename;
        unsigned long multipart_key[N];
        long file_size;
   sha256_ctx m_sha256, m2_sha256;
   unsigned char pBuf[4096], pTemp[256];
   unsigned long uRead = 0;

   FILETIME idleTimePtr, kernelTimePtr, userTimePtr;

        memset(buffer_area, 0, sizeof(buffer_area));
        memset(output_filename, 0, sizeof(output_filename));
   memset(mt, 0, N);
   memset(multipart_key, 0, N);
/*
** for encryption
** get high entropy seed for PRNG
** use PRNG to generate 32 byte block
** reseed the PRNG with KEY
*/
   //GetSystemTimes(&idleTimePtr, &kernelTimePtr, &userTimePtr);
   multipart_key[0] = idleTimePtr.dwLowDateTime;
   multipart_key[1] = idleTimePtr.dwHighDateTime;
   multipart_key[2] = kernelTimePtr.dwLowDateTime;
   multipart_key[3] = kernelTimePtr.dwHighDateTime;
   multipart_key[4] = userTimePtr.dwLowDateTime;
   multipart_key[5] = userTimePtr.dwHighDateTime;
   multipart_key[6] = time(NULL);
   multipart_key[7] = clock();
   init_by_array(&multipart_key[0], 8);
   memset(&multipart_key[0], 0, sizeof(multipart_key));
   for (i = 0; i < BYTES_PER_BLOCK; i++) {
      buffer_area[i] = genrand_int32() >> 24;
   }

        if (argc == 1) {
                printf("Invoke encryption using\n\n");
                printf("   \"%s file key1 key2\"\n", argv[0]);
                printf("Where key1 is a number between 1 and 4294967296 and key2 \nis a string of
characters (no whitespace).\n");
                printf("Encrypted/decrypted output will be the filename with an\nunderscore
appended/removed\n");
      printf("The encrypted file will be 32 bytes larger than the plain text file\n");
                exit(0);
        } else {
```

```
            if (argc < 3) {
                printf("too few arguments\n");
                    exit(0);
                }
            }


            if (argc >= 4) {
        char * string = argv[3];
        strcpy((char *)&pBuf[0], argv[3]);
        sha256_begin(&m_sha256);
        sha256_hash(pBuf, (unsigned long)strlen(argv[3]), &m_sha256);
        sha256_end(pTemp, &m_sha256);
        multipart_key[0] = atoi(argv[2]);
                    for (i = 0; i < strlen(argv[3]); i++) {
           multipart_key[1+(i>>2)] |= string[i] << (8*(i&3));
                    }
        keysize = (i >> 2) + 1;
        if (i&3) keysize++;
        init_by_array(&multipart_key[0], keysize);

            } else {
        printf("insufficient arguments\n");
        exit(0);
            }
            if (_access((const char *)argv[1], 0) != 0) {
                    printf("file does not exist\n");
                    exit(0);
            } else {
                    filename_size = strlen(argv[1]);
                    strcpy(output_filename, argv[1]);
                    if (output_filename[filename_size-1] == 0x5f) {
           direction = DECRYPT;
                            output_filename[filename_size-1] = 0;
                    } else {
           direction = ENCRYPT;
                       output_filename[filename_size] = 0x5f;
                    }
            }

       fd_in = fopen((const char *)argv[1], (const char *)("rb"));
       fd_out = fopen(my_out, (const char *)("wb"));
    /*
    ** for each 32 byte block
    **    compute next blocks hash
    **    for each byte in block
    **       a(i) = a(i-1) + p(i) & 0xff
    **       apply p(i) ^= a(i-1)
    **       apply p(i) ^= k(i)
    **    endfor
    **    block ^= current blocks hash
    ** endfor
    */
```

```
   if (ENCRYPT == direction) start_fill = BYTES_PER_BLOCK;
   else start_fill = 0;
          do {
     bytes_read = fread( &buffer_area[start_fill], 1, (MAX_WORDS-start_fill), fd_in);
     bytes_read += start_fill;
     blocks_processed++;
     i = 0;
     while (i < bytes_read) {
        if (ENCRYPT == direction) {
           if ((i % 32) == 0) {
              for (j = i; j <= i+31; j++) {
                 pBuf[j%32] = buffer_area[i+(j%32)];
              }
           }
           if (((i+1) % 32) == 0) { // calculate hash for next block
              for (j=0; j<8; j++) {
                 pBuf[32+4*j] = (m_sha256.hash[j] >> 24) & 0xff;
                 pBuf[33+4*j] = (m_sha256.hash[j] >> 16) & 0xff;
                 pBuf[34+4*j] = (m_sha256.hash[j] >> 8) & 0xff;
                 pBuf[35+4*j] =  m_sha256.hash[j] & 0xff;
              }
              sha256_begin(&m2_sha256);
              sha256_hash(pBuf, 64, &m2_sha256);
              sha256_end(pTemp, &m2_sha256);
           }
           last_char = buffer_area[i];
           keystream = genrand_int32() >> 24;
                       buffer_area[i] ^= keystream^muddle;
           buffer_area[i] ^= m_sha256.hash[((i%32)/4)+(i%4)];
           muddle = (muddle + last_char) & 0xff;
           if ((++i % 32) == 0) { // transfer new hash into current hash
              memcpy(&m_sha256, &m2_sha256, sizeof(m_sha256));
           }
        } else { // DECRYPT
/*
** to decrypt
** for each block
** decrypt muddle and stream and hash
** calculate next hash
*/
           keystream = genrand_int32() >> 24;
           buffer_area[i] ^= keystream^muddle;
           buffer_area[i] ^= m_sha256.hash[((i%32)/4)+(i%4)];
           muddle = (muddle + buffer_area[i]) & 0xff;
           if (((i+1) % 32) == 0) {
              for (j = i-31; j <= i; j++) {
                 pBuf[0 + (j-i+31)] = buffer_area[j];
              }
              for (j=0; j<8; j++) {
                 pBuf[32+4*j] = (m_sha256.hash[j] >> 24) & 0xff;
                 pBuf[33+4*j] = (m_sha256.hash[j] >> 16) & 0xff;
                 pBuf[34+4*j] = (m_sha256.hash[j] >> 8) & 0xff;
                 pBuf[35+4*j] =  m_sha256.hash[j] & 0xff;
```

```
                }
                sha256_begin(&m_sha256);
                sha256_hash(pBuf, 64, &m_sha256);
                sha256_end(pTemp, &m_sha256);
            }
            i++;
        }
    }
    if ((DECRYPT == direction) && (blocks_processed == 1)) {
            fwrite(&buffer_area[32],1,bytes_read-32,fd_out);
    } else {
            fwrite(&buffer_area[0],1,bytes_read,fd_out);
    }
    start_fill = 0;
        eof = feof(fd_in);
    } while (eof == 0);

  fclose(fd_in);
    fclose(fd_out);
    exit(0);
}
```

REFERENCES

[1] Carl E. Landwehr, *"The Best Available Technologies for Computer Security,"* Computer, Vol. 16, No. 7, July 1983,pp. 86—100, Los Alamitos, CA: IEEE Computer Society.

[2] C. E. Shannon, "A *Communication Theory of Secrecy Systems*," Bell System Technical Journal, Volume 28, 1949.

[3] K.Scarfone, M.Souppaya, M.Sexton; "Guide to Storage Encryption Technologies for end users", Special Publication 800-111, pp2-1. http://csrc.nist.gov/publications/nistpubs/800-111/SP800-111.pdf

[4] Carl H. Meyer & Stephen M. Matyas, *"Cryptography: a New Dimension in Computer Data Security"* — A Guide for the Design and Implementation of Secure Systems, New York, Chichester, Brisbane, Toronto, Singapore: John Wiley & Sons, 1982. Z103.M55.1982

[5] D Elminaam, M A Kader, M Hadhoud "*Performance Evaluation of Symmetric Encryption Algorithms*", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.12, December 2008

[6] Vin McLellan, *"Drugs and DES: A New Connection,"* Digital Review, August 24, 1987.

[7] http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[8] Alan G. Konheim, *"Cryptography: A Primer*", New York: John Wiley & Sons. Z103.K66 [KON]

[9] Fred Piper, *"Stream Ciphers"* ;Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[10] Henk C. A. van Tilborg, *"An Introduction to Cryptology*", Boston, Dordrecht, Lancaster: Kluwer Academic Publishers, 1988. Z103.T54.1987

[11] Shailendra Khadka Yadav, "*Content Hiding: An Image Approach*." Project Work in Partial Fulfillment of The Requirements For Master Degree in Computer Science and Information Technology, TU, 2008

[12] http://en.wikipedia.org/wiki/md5

[13] Mehmet Kivanc Mihcak (2002), "*Information Hiding Codes and Their Applications to Images and Audio*", PhD Thesis, Graduate College of the University of Illinois at Urbana-Champaign, US, 2002

BIBILIOGRAPHY

[1] Henk C.A.van Tilborg, "*Fundamentals of Cryptology*", Eidhoven University of Technology, netherdlands, 19

[2] "*Performance Comparison of Data Encryption Algorithms*," IEEE [Information and Communication Technologies, 2005. ICICT 2005. First International Conference, 2006-02-27, PP. 84- 89.

[3] A. M. Jackson, N. A. McEvoy, and B. B. Newman, "*Project Universe Encryption Experiment*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[JAC]

[4] C. B. Brookson and S. C. Serpell, "*Security on the British Telecom Satstream Service*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[BRO]

[5] C. E. Shannon, "*A Mathematical Theory of Communication*," Bell System Technical Journal, page 623, 1948

[6] C. J. Mitchell, "*A Comparison of the Cryptographic Requirements for Digital Secure Speech Systems Operating at Different Bit Rates,"* in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[MIT]

[7] C. P. Schnorr, *"Is the RSA Scheme Safe?"* in Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983. [SCH]

[8] Cipher A. Deavers and Louis Kruh, "*Machine Cryptography and Modern Cryptanalysis*", Norwood, MA: Artech House, Inc., 1985. Z103.D43.1985[DEA]

[9] D Elminaam, M A Kader, M Hadhoud "*Performance Evaluation of Symmetric Encryption Algorithms*", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.12, December 2008

[10] D. G. Haenshke, D. A. Kettler, and E. Oberer, "*Network Management and Congestion in the U. S. Telecommunications Network,"* IEEE Transactions on Communications, volume COM-29,pp. 376–385, April 1981. [HAE]

[11] D. J. Bond, "*Practical Primality Testing,*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[BON]

[12] Dave Bursky, "*Protect Your EEPROM Data and Gain More Flexibility,*" Electronic Design, 26 May 1988, pp.43—48. Waseca, MN: Brown Printing Co.[BUR]

[13] David B. Newman, Jr. and Raymond L. Pickholtz, "*Cryptography in the Private Sector,*" IEEE Communications Magazine, August 1986, Volume 24, Number 8, pp. 7—10. New York, NY: The Institute of Electrical and Electronic Engineers, Inc. [NEW]

[14] David Kahn, "*The Codebreakers — The Story of Secret Writing*", New York: The MacMillan Company, 1987. Z103.K28[KAH]

[15] Dorothy Elisabeth Robling Denning, "*Cryptography and Data Security*", Menlo Park, CA; London; Amsterdam; Don Mills, Ontario; Sydney: Addison-Wesley Publishing Company, 1982. QA76.9.A25.D46.1982[DEN]

[16] Douglas J. Albert and Stephen P. Morse, "*Combating Software Piracy by Encryption and Key Management,*" Computer, Vol. 21, No. 5, April 1984, pp. 68—73 Los Alamitos, CA: IEEE Computer Society. [ALB]

[17] Friedrich L. Bauer, "*Cryptology — Methods and Maxims,*" in Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983. [BAU]

[18] G. Goos and J. Hartmans, "*Lecture Notes in Computer Science*." Z102.5.W67.1982[GOO]

[19] Gilbert Held and Thomas R. Marshall, *"Data Compression"*, Second Edition; Chichester, New York, Brisbane,Toronto, Singapore: John Wiley & Sons, 1987. QA76.9.D33.H44.1987[HEL]

[20] H. J. Beker, "*A Survey of Encryption Algorithms,*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Instituteof Electrical Engineers, 1984. TK5102.5.I519.1984[BEK]

[21] H. J. Beker, "*A Survey of Encryption Algorithms,*" International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984.

[22] H. J. Beker, J. M. K. Friend, and P. W. Halliden, *"Simplifying Key Management in Electronic Fund Transfer Point of Sale Systems,"* in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[FRI]

[23] H. R. Chivers, "*A Practical Fast Exponentiation Algorithm for Public Key,*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[CHI]

[24] H. Retkin, *"Multi-Level Knapsack Encryption,"* in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[RET]

[25] H. Retkin, *"Multi-Level Knapsack Encryption,"* in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[RET]

[26] Hardjono, "*Security In Wireless LANS And MANS,*" Artech House Publishers 2005.

[27] Henk C.A. van Tilborg, "*Fundamentals of Cryptology*", Eindhoven University of Technology, Netherlands, 19

[28] J. A. Gordon and H. Retkin, *"Are Big S-Boxes Best?"* in Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983. [GOR]

[29] J. Sattler and C. P. Schnorr, *"Ein Effizienzvergleich Der Faktorisierungsverfahren von Morrison-Brillhart und Schroeppel,"* in Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983. [SAT]

[30] J. Vandewalle, R. Govaerts, W. De Becker, M. Decroos, & G. Speybrouk, *"RSA–Based Implementation of Public Key Cryptographic Protection in Office Systems"* in Proceedings of 1986 International Carnahan Conference on Security Technology: Electronic Crime Countermeasures, August 12-14, 1986. QA76.9.A25.C41.1986[VAN]

[31] James A. Storer, "*Data Compression Methods and Theory*". Rockville, MD: Computer Science Press, 1988. QA76.9.D33S76 [STO]

[32] Jui-Cheng Yen and Jiun-In Guo, *"A new image encryption algorithm and its VLSI architecture,"* in Proc. IEEE Work-shop Signal Processing Systems, 1999, pp. 430–437.

[33] Leslie S. Chalmers, "*An Analysis of the Differences Between the Computer Security Practices in the Military and Private Sectors,*" in Proceedings of the IEEE 1986 Symposium on Securi ty and Privacy, Oakland, CA, Apri l 7-9, 1986. QA76.9.A25.595.1986[CHA]

[34] Lester J. Fraim, *"Scomp: A Solution to the Multilevel Security Problem,"* Computer, Vol. 16, No. 7, July 1983,pp. 26—34, Los Alamitos, CA: IEEE Computer Society. [FRM]

[35] Lester S. Hill, "*Cryptography in an Algebraic Alphabet,"* American Mathematical Monthly, June 1929, pages 306-312, the American Mathematical Society. Republished electronically by Tony Patti by permission. [HIL]

[36] Mehmet Kivanc Mihcak (2002), "*Information Hiding Codes and Their Applications to Images and Audio*", PhD Thesis, Graduate College of the University of Illinois at Urbana-Champaign, US, 2002

[37] Michael Paul Johnson, "*The Diamond2 Block Cipher*", University of Colorado, 1995

[38] N. Lodge, B. Flannaghan, and R. Morcom, "*Vision Scrambling of C-MAC DBS Signals,*" in International Conference on Secure Communication Systems, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984[LOD]

[39] National Bureau of Standards, Federal Information Processing Standards Publication Number 46 Dated 15 January 1977. [DES]

[40] Ole Immanuel Franksen, Mr. Babbage's Secret — "*The Tale of a Cipher and APL,"* Englewood Cliffs, NJ: Prentice-Hall, 1985. Z103.B2.F72.1985[FRA]

[41] P.K. Jha, "*A Bit Level Symmetric Encryption Technique Through Recursive Addition of Block (RAB)"*, International Conference on Electronic Commerce in the 21st Century; Kathmandu, 2008.

[42] Phil Katz, PKARC FAST! Archive Create/Update Utility Version 3.5 04-27-87, published electronically in PKX35A35.EXE. [KAT]

[43] R. H. Cooper, "*Linear Transformations in Galois Fields and their Application to Cryptography*," Cryptologia Magazine, Volume 4, Number 3, July 1980, pages 184-188. Republished electronically by Tony Patti. [COO]

[44] Ralph C. Merkle, "*Secrecy, Authentication, and Public Key Systems,"* Ann Arbor, MI: UMI Research Press,1982. QA76.9.A25M47.1982[MER]

[45] Richard Doherty, "*FBI Nabs Pirated VCII,"* Electronic Engineering Times, Manhasset, N. Y.: CMP Publications, Inc., Issue 500, August 22, 1988. [DOH]

[46] Stanley R. Ames, Jr., Morrie Gasser, and Roger R. Schell, "*Security Kernel Design and Implementation: An Introduction*," Computer, Vol. 16, No. 7, July 1983, pp. 14—22 Los Alamitos, CA: IEEE Computer Society. [AME]

[47] Tekla S. Perry and Paul Wallich, *"Can Computer Crime be Stopped?,"* IEEE Spectrum, May, 1984, pp.34—49. New York, NY: The Institute of Electrical and Electronic Engineers, Inc. [PER]

[48] Terry Costlow, "*Global Computer Network Cracks Cryptographic Mathematics Barrier*," Electronic Engineering Times, Issue 508, 17 October 1988, p. 14, Manhasset, NY: CMP Publications, Inc. [COS]

[49] Thomas Beth, "*Introduction to Proceedings of the Workshop on Cryptography at Burg Feuerstein*", Germany, March 29 — April 2, 1982, Berlin, Heidelberg, New York: Springer-Verlag, 1983. [BET]

[50] Tony Patti, *"A Galois Field Cryptosystem, 1986."* Published electronically by Tony Patti, editor of Cryptosystems Journal, 9755 Oatley Lane, Burke, VA 22015. [PAT]

[51] Ultron Labs Corporation, *"Crypto Module Processes TOP SECRET Data,"* EE Product News, October 1988, p. 16, Overland Park, KS: Intertec Publishing. [ULT]

[52] W. P. Lu and M. K. Sandareshan, *"A Hierarchical Key Management Scheme for End-to-End Encryption in Internet Environments"* in Proceedings of the IEEE 1986 Symposium on Securi ty and Privacy, Oakland, CA, Apri l 7-9, 1986. QA76.9.A25.595.1986[LU]

[53] W. W. Rouse Ball & H. S. M. Coxeter, "*Mathematical Recreations & Essays*". New York: The MacMillan Company, 1962. [BAL]

[54] William Stallings, "*Cryptography and Network Security"*, Fourth Edition, Prentice Hall of India; ISBN-978-81-203-3018-4.

[55] Wladyslaw Kozaczuk, "*Enigma — How the German Machine Cipher was Broken and How it was Read by the Allies in World War Two*", University Publications of America, Inc., 1984.D810.C88.K6813.1984[KOZ]

[56] Ross Anderson, "*Searching for the Optimum Correlation Attack"*, FSE'94, LNCS 1008, Springer, pp137-143.

[57] Frederik Armknecht, *"A Linearization Attack on the Bluetooth Key Stream Generator"*, Available on http://eprint.iacr.org/2002/191/.

[58] Frederik Armknecht, Matthias Krause, *"Algebraic Atacks on Combiners with Memory"*, Crypto 2003, LNCS 2729, pp. 162-176, Springer.

[59] Steve Babbage, *"Cryptanalysis of LILI-128"*, Nessie project internal report, available at https://www.cosic.esat.kuleuven.ac.be/nessie/reports/.

[60] Eli Biham, *"A Fast New DES Implementation in Software"*, FSE'97, Springer, LNCS 1267, pp. 260-272.

[61] Paul Camion, Claude Carlet, Pascale Charpin and Nicolas Sendrier, *"On Correlation-immune Functions"*, Crypto'91, LNCS 576, Springer, pp. 86-100.

[62] Computer Security, Matt Bishop, Chapter9.3 p.605

[63] Nicolas Courtois and Jacques Patarin, *"About the XL Algorithm over GF(2)"*, Cryptographers' Track RSA 2003, LNCS 2612, pages 141-157 ,Springer 2003.

[64] Nicolas Courtois, "*Higher Order Correlation Attacks, XL algorithm and Cryptanalysis of Toyocrypt*", ICISC 2002, November 2002, Seoul, Korea, LNCS 2587, pp. 182-199, Springer.

[65] Nicolas Courtois: *"Fast Algebraic Attacks on Stream Ciphers with Linear Feedback"*, Crypto 2003, LNCS 2729, pp: 177-194, Springer.

[66] Eric Filiol: *"Decimation Attack of Stream Ciphers"*, Indocrypt 2000, LNCS 1977, pp. 31-42, 2000. Available on eprint.iacr.org/2000/040.

[67] Jovan Dj. Golic: "*On the Security of Nonlinear Filter Generators*", FSE'96, LNCS 1039, Springer, pp.173-188.

[68] Jovan Dj. Golic: "*Fast low order approximation of cryptographic functions*", Eurocrypt'96, LNCS 1070, Springer, pp. 268-282.

[69] Feistel, H. "*Cryptography and Computer Privacy*." Scientific American, May 1973.

[70] Willi Meier and Othmar Staffelbach: "*Nonlinearity Criteria for Cryptographic Functions*", Eurocrypt'89, LNCS 434, Springer, pp.549-562, 1990.

[71] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone: "*Handbook of Applied Cryptography*",CRC Press.

[72] M. Mihaljevic, H. Imai: "*Cryptanalysis of Toyocrypt-HS1 stream cipher*", IEICE Transactions on Fundamentals, vol. E85-A, pp. 66-73, Jan. 2002. Available at http://www.csl.sony.co.jp/ATL/papers/IEICEjan02.pdf.

[73] Jacques Patarin: "*Cryptanalysis of the Matsumoto and Imai Public Key Scheme"* Eurocrypt' 88,Crypto'95, Springer, LNCS 963, pp. 248-261, 1995.

[74] Rainer A. Rueppel: "*Analysis and Design of Stream Ciphers"*, Springer, New York, 1986.

[75] Palash Sarkar, Subhamoy Maitra: "*Nonlinearity Bounds and Constructions of Resilient Boolean Functions*", Crypto 2000, LNCS 1880, Springer, pp. 515-532.

[76] Palash Sarkar: "*The Filter-Combiner Model for Memoryless Synchronous Stream Ciphers*", Crypto 2002, LNCS 2442, Springer, pp. 533-548.

[77] Alex Biryukov, Adi Shamir: "*Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers*", Asiacrypt 2000, LNCS 2248, Springer, pp. 1-13.

[78] Adi Shamir, Jacques Patarin, Nicolas Courtois and Alexander Klimov: "*Efficient Algorithms for solving Overdefined Systems of Multivariate Polynomial Equations*", Eurocrypt'2000, LNCS 1807, Springer, pp. 392-407.

[79] Simpson, E. Dawson, J. Golic and W. Millan: "*LILI Keystream Generator*" Springer, pp. 248-261, Cf. www.isrc.qut.edu.au/lili/.

[80] Fredrik Jonsson, Thomas Johansson: "*A Fast Correlation Attack on LILI-128"*, Can be found at http://www.it.lth.se/thomas/papers/paper140.ps

[81] Markku-Juhani Olavi Saarinen: "*A Time-Memory Tradeoff Attack Against LILI-128",* FSE 2002, LNCS 2365, Springer, pp. 231-236, available at http://eprint.iacr.org/2001/077/.

[82] http://books.google.com.np/books?id=p1swqJNYA_QC&pg=PA224&lpg=PA224&dq=Secure+Hash+algorithm+theory&source=bl&ots=-M2zu7_c8C&sig=DhCydQZmdY-Iox1__OWoZaUR58&hl=ne&ei=W_n5TIazIcLJrAe89YiaCA&sa=X&oi=book_result&ct=result&resnum=6&ved=0CDAQ6AEwBQ#v=onepage&q=Secure%20Hash%20algorithm%20theory&f=false