



**LOCATING ENCRYPTED DATA HIDDEN AMONG NON-ENCRYPTED DATA
USING STATISTICAL TOOLS**

THESIS

Walter J. Hayden, Captain, USAF

AFIT/GCS/ENG/07-06

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/07-06

**LOCATING ENCRYPTED DATA HIDDEN AMONG NON-ENCRYPTED DATA
USING STATISTICAL TOOLS**

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Walter J. Hayden, BS

Captain, USAF

March 2007

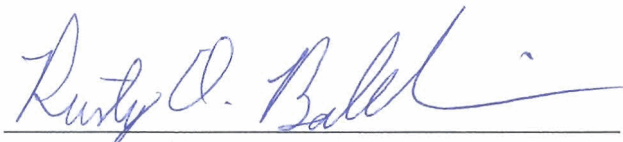
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**LOCATING ENCRYPTED DATA HIDDEN AMONG NON-ENCRYPTED DATA
USING STATISTICAL TOOLS**

Walter J. Hayden, BS

Captain, USAF

Approved:


Rusty O. Baldwin, PhD (Chairman)

5 Mar 07
Date


Timothy J. Halloran, Lt Col, USAF (Member)

5 Mar 07
Date


Gilbert L. Peterson, PhD (Member)

5 MAR 07
Date

Abstract

This research tests the security of software protection techniques that use encryption to protect code segments containing critical algorithm implementation to prevent reverse engineering. Using the National Institute of Standards and Technology (NIST) Tests for Randomness encrypted regions hidden among non-encrypted bits of a binary executable file are located. The location of ciphertext from four encryption algorithms (AES, DES, RSA, and TEA) and three block sizes (10, 100, and 500 32-bit words) were tested during the development of the techniques described in this research. The test files were generated from the Win32 binary executable file of Adobe's Acrobat Reader version 7.0.9.

The culmination of this effort developed a technique capable of locating 100% of the encryption regions with no false negative error and minimal false positive error with a 95% confidence. The encrypted region must be encrypted with a strong encryption algorithm whose ciphertext appears statistically random to the NIST Tests for Randomness, and the size of the encrypted region must be at least 100 32-bit words (3,200 bits).

*To my wife and children, without their love, support, and understanding this thesis would
not have been possible.*

Acknowledgments

I would like to express gratitude to my thesis advisor Dr. Baldwin for his guidance, support, and strict timeline throughout the course of this thesis effort. The insight and experience was certainly appreciated. I would also like to thank my committee members, Lt. Col. Halloran and Dr. Peterson for helping me to make this a better thesis.

Walter J. Hayden

Table of Contents

	Page
Abstract	iv
Acknowledgments.....	vi
List of Figures	ix
List of Tables	xi
I. Introduction	1
Motivation	1
Background.....	2
Research Focus	3
Objective.....	3
Approach	4
Overview	4
II. Literature Review	5
Chapter Overview.....	5
Encryption Algorithms	5
Encryption Modes	26
Padding Schemes.....	30
Statistical Methods	33
Previous Work	41
Summary.....	41
III. Methodology	43
Chapter Overview.....	43
Problem Definition	43
Performance Metrics	45
Parameters	47
Factors	50
Evaluation Technique.....	50
Experimental Design	51
Summary.....	58
IV. Analysis and Results.....	59
Chapter Overview.....	59

Size Test Results.....	59
Algorithm Significance Results.....	61
Padding Encryption File Results	67
Multiple Encrypted Regions Results.....	69
Summary.....	71
V. Conclusions and Recommendations	72
Chapter Overview.....	72
Conclusions of Research	72
Significance of Research	73
Recommendations for Action.....	73
Recommendations for Future Research.....	74
Summary.....	74
Bibliography	76
Vita.....	79

List of Figures

	Page
1. Indices for Bytes and Bits	10
2. Input array into state array [NIS01b]	11
3. Pseudo Code for Cipher's Transformation	12
4. Pseudo Code for Round Transformation	13
5. The byte substitution is applied to each individual byte [NIS01b]	13
6. Each row is cyclically shifted to the right by the shift amount	14
7. Matrix multiplication of <i>state array</i> columns	15
8. Matrix multiplication carried out	15
9. <i>MixColumn</i> transformation	15
10. <i>AddRoundKey</i> transformation	16
11. DES encryption transformation	18
12. DES Cipher Function	19
13. TEA Feistel Structure [Wik06]	21
14. Demonstration of the weakness of ECB Mode [Wik06a]	27
15. Electronic Code Book [NIS01b]	27
16. Cipher Block Chaining [NIS01b]	28
17. Cipher Feedback Mode [NIS01b]	30
18. Discrete Fourier Transform (Spectral) Test Results [NIS01]	37
19. Encryption Location System	44
20. Error Diagram	46

21. File Padding Technique Example	54
22. Size Test Results	60
23. Text Test Results.....	62
24. Text Encryption Example	63
25. Program Example (Plaintext).....	64
26. Program Example (Ciphertext)	65
27. Program Test Results	65
28. Padding technique Test Results	67
29. Selection Function Comparison.....	68
30. Multiple Encryption Regions Test Results	69
31. Encryption Location System Output.....	70

List of Tables

	Page
1. Simple S-Box Example.....	9
2. Size Test Average Numerical Results.....	60
3. Algorithm Significance Average Numerical Results (Text).....	63
4. Algorithm Significance Average Numerical Results (Program)	66
5. Average Bit Error of Encrypted Region	66
6. Multiple Encrypted Regions Average Numerical Results	69

LOCATING ENCRYPTED DATA HIDDEN AMONG NON-ENCRYPTED DATA USING STATISTICAL TOOLS

I. Introduction

Motivation

Future wars will be fought by warriors armed with computers. On those computers will be software to assist the soldier with navigation, communication, targeting, and a host of other tasks. This software will most assuredly contain classified data or algorithms requiring protection in the event the computer falls into enemy hands. Encryption is the only means of protection strong enough to protect this software.

Protecting software with encryption is necessary, however, encrypting the entire program is not. A program needs to be both secure and accessible to the soldier. The program may perform several frequent time-sensitive unclassified tasks and a few less frequent classified ones. Encrypting the entire program would hinder the soldier's ability to quickly access the time-sensitive tasks.

The solution is to protect only the areas of the program's code containing the classified data and algorithms. As the program is executed the soldier would have access to the common unclassified tasks, however, to perform a classified task the soldier would need to perform some authorization steps to unlock those tasks. This gives the soldier the accessibility they need, while protecting the software from the enemy.

Background

After a software program is written in a higher level language such as Java or C++, the code is compiled into a format that the processor can execute. The executable format is fairly difficult for a human to read and understand, however, techniques have been developed to transform the compiled code back into a readable format (even the original higher language). This process is called reverse engineering.

Reverse engineering has fostered an ethical debate about the protection of intellectual property in software programs. Thousands of dollars and man hours go into the development and testing of commercial software programs. Because of the fiscal investment and the need to protect classified information, reverse engineering has become a major concern for both commercial and government entities alike who want to protect their investments.

To thwart reverse engineering, a number of protection schemes are used. One of these is obfuscation. Obfuscation can be as simple as renaming variables and methods or as complex as flattening a programs abstract syntax tree. The primary objective behind obfuscation is to either make it very difficult to decompile by confusing decompilers, or by making the decompiled program too difficult for a human to comprehend.

In many situations code obfuscation is effective and can dissuade a determined reverse engineer. However, it is risky to use this to protect highly sensitive or extremely high cost software programs, such as programs that perform intelligence gathering tasks or target recognition. In these cases, encryption of critical program areas is warranted.

The next logical question is “How secure is it?” Encryption requires a way to decrypt the program’s code as the program is loaded into memory. For an operating system to load the program into memory, a special loader program must know the location of the encrypted segments, the encryption algorithm, the encryption key, and possibly an initialization vector. This information could be supplied by the user or embedded into the executable file. If this information is kept separate from the encrypted data, then it is secure. However, keeping the information separated may not be desirable, therefore leaving the programs security in question.

For the enemy to successfully “crack” this protection technique, they must locate the encryption regions, find the encryption parameters (encryption algorithm, key size, and block size), find the key, and possibly the initialization vector.

Research Focus

This research is focused on the first step, locating encrypted data among non-encrypted data using a statistical test package developed by the National Institute of Standards and Technology.

Objective

The goal is to find the NIST statistical tests that are most useful in locating encrypted data, the best parameters for those tests, and analytical techniques to interpret the results of the tests to accurately locate all of encrypted data with as little error as possible.

Approach

The general approach of this research is trying several solution and then hone the technique until the goals are met. The first experiment is to find the best size parameter for the NIST tests. Once the best size parameter is found it will be used to test the four encryption algorithms and three encrypted region block sizes. The lessons learned from this experiment will be used to refine the location technique until the goal is achieved.

Overview

The remainder of this document is organized as follows. Chapter 2 provides background information on each of the four encryption algorithms and statistical tests used in this research. Chapter 3 outlines the research methodology, goals, and hypothesis in greater detail. Chapter 4 presents the results of the experiments conducted during this research. Finally, Chapter 5 is the conclusion of this research and future work.

II. Literature Review

Chapter Overview

This chapter provides background information. The reader is assumed to have basic knowledge of math and computers. The topics covered in this chapter include an explanation of selected encryption algorithms, statistical techniques, and methods used to categorize these encryption algorithms.

Encryption Algorithms

An encryption algorithm is a mathematical function that transforms a message into a form that is unreadable without knowledge of the algorithm and/or the secret key used to encrypt the message. To facilitate further discussion of encryption algorithms, some basic terminology must be established. Most of the following terminology can be found in [MVV97, pg. 11].

Encryption Domain and Codomains:

- Alphabet of definition ($\mathcal{A}$): The alphabet of definition is a finite set of symbols. The most common alphabet is $\mathcal{A} = \{0,1\}$, also known as the binary alphabet because it is used in computing and can be used to encode any finite symbol.
- Message space ($\mathcal{M}$): The message space is a set of strings over an $\mathcal{A}$ that contains the message. Elements of $\mathcal{M}$ are called a *plaintext message* or *plaintext* for short. Examples of plaintext messages include binary strings, English text, and computer code.

- Ciphertext space ($\mathcal{C}$): The ciphertext space is a set of strings over $\mathcal{A}$ (which does not have to be the same $\mathcal{A}$ as $\mathcal{M}$) that contains the encrypted messages. An element of $\mathcal{C}$ is called a ciphertext.

Encryption and decryption transformations:

- Key space ($\mathcal{K}$): The key space is the set of strings over an $\mathcal{A}$ for an E (*encryption algorithm*). An element of $\mathcal{K}$ is called the key.
- Encryption Key ($e \in \mathcal{K}$): The encryption key uniquely determines a bijection from $\mathcal{M}$ to $\mathcal{C}$ denoted by E_e . E_e is called an *encryption function* or *encryption transformation*.
- Decryption Key ($d \in \mathcal{K}$): The decryption key uniquely determines a bijection from $\mathcal{C}$ to $\mathcal{M}$ denoted by D_d . D_d is called a *decryption function* or *decryption transformation*.
- Encrypting or Encryption of m : The process of applying the transformation of E_e to a message $m \in \mathcal{M}$.
- Decrypting or Decryption of c : The process of applying the transformation of D_d to a message $c \in \mathcal{C}$.
- Encryption scheme, Encryption algorithm, or Cipher: The encryption scheme consists of a set of encryption transformations and a set of corresponding decryption transformations such that for every $e \in \mathcal{K}$ there is a unique $d \in \mathcal{K}$ where $D_d(E_e(m)) = m$ for all $m \in \mathcal{M}$.

- Key pair (e, d) : A key pair is the encryption key and corresponding decryption key of an encryption scheme. In certain types of encryption algorithms the two key are the same.

To develop or construct an encryption scheme an encryption transformation, decryption transformation, message space, cipher space, and a key space must be chosen ([MVV97, pg. 12]).

The remainder of this section explains the most common types of encryption algorithms or encryption schemes starting with the Symmetric Cipher.

Symmetric Key Algorithm

The most common encryption algorithm is the symmetric key algorithm. It derives its name from the symmetry of its key pair, that is, the encryption key and decryption key are the same (or one can very easily be derived from the other). This type of algorithm also goes by several other name such as *single-key*, *one-key*, *private-key*, and *conventional* encryption ([MVV97, pg. 15]).

Stream vs. Block Algorithms

Symmetric Key ciphers can be categorized into two specific types. The types differ based on how the encryption/decryption functions perform their tasks. The more common of the two types is called the *block* cipher.

The block cipher partitions plaintext message into groups of a fixed length (typically 128 or 256 bits for a binary alphabet), and transforms the whole block at one time. Some

block ciphers keep the blocks separate and independent while other use a previously encrypted block to encrypt subsequent blocks.

The second symmetric key type is called a *stream* cipher. The stream cipher is unique because the encryption function must encrypt the message one symbol or character at a time. It is similar to a block cipher with a block length of one, however, the key for a stream cipher differs slightly. The key for a stream cipher is made up of several sequential keys called a *keystream*. More specifically a keystream is a sequence of symbols $e_1e_2e_3\dots e_i \in \mathcal{K}$, where $\mathcal{K}$ is a key space for a set of encryption transformations ([MVV97, pg. 20]).

The stream cipher applies the encryption transformations in accordance with the keystream being utilized. The keystream can be produced randomly or from an algorithm (called a *keystream generator*) which generates the keystream from a small initial keystream (*seed*) ([MVV97, pg. 21]).

The advantage of a stream cipher is two fold. First, the encryption algorithm can be applied real-time. For example, in a communication circuit where the system can't wait for the arrival of a whole block before performing the encryption due to a lack of buffer space or the randomness of the incoming data stream. Second, the algorithm can be applied in devices that don't have the memory space to store a whole block of information.

Substitution and Transposition ciphers

There are two additional classes or categories of symmetric key ciphers: substitution and transposition. Many of the symmetric key ciphers fall within one or both of these categories. Considering them as a property, a cipher can exhibit is easier to understand than considering them as classes or categories.

A substitution cipher replaces an element of a message $\mathcal{M}$ with another symbol $\mathcal{A}_c$ (where $\mathcal{A}_c$ is the alphabet of the ciphertext space). Many advanced encryption algorithms use a substitution box (aka S-box) to protect against differential cryptanalysis (see below). The S-box is a table containing an array of substitutions based on the input. For example, Table 1 is a very simple S-box to illustrate the substitution principle.

Table 1. Simple S-Box Example

S_x		Last three bits							
		000	001	010	011	100	101	110	111
First	0	101	011	110	010	001	111	000	100
bit	1	011	010	111	000	100	101	110	001

If the input contained 1001 then the output of the S-box would be 010.

A transposition cipher rearranges the elements of the message. This is often weaker since the size of the output and the symbols are preserved. A good example of a transposition cipher is the scrambled words puzzles found in some newspapers.

Advance Encryption Standard (AES)

Most of the information in this section was obtained from [DaR99] and [NIS01b].

On the 12th of September, 1997, the National Institute of Standards and Technology (NIST) requested proposals for the Advanced Encryption Standard (AES) [NIS06]. On the 2nd of October, 2000, NIST announced Rijndael (pronounced Rhine-doll) the new encryption standard [NIS06]. Rijndael was developed by two Belgian cryptologists, Joan Daemen and Vincent Rijmen, who based their algorithm off an algorithm called SQUARE they created prior to developing Rijndael. Rijndael derives its name from the last name of its creators, **Rijmen and Daemen**. The AES and Rijndael are not exactly the same. While NIST adopted Rijndael as the AES standard, AES only supports a fixed block size of 128 bits and key lengths of 128, 192, or 256 bits [NIS01b] even though Rijndael is capable of additional block sizes and key lengths. AES from this point forward will be used interchangeably with Rijndael assuming a fixed block size of 128 bits.

AES uses the binary alphabet ($\{0,1\}$) for transformations. The input for AES is a one-dimensional array of 16 bytes with indices ranging from 0..15. Similarly the cipher key is a one-dimensional array of 16, 24, or 32 bytes depending on the key size. The bits within the bytes are in little-endian order. Figure 1 illustrates the indices for the bytes and bits.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
Byte number	0								1								2								...
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	...

Figure 1. Indices for Bytes and Bits

The interim block after each stage of the transformation is called a state. A state consists of a two-dimensional array of four rows and four columns where each cell contains 4 bytes. Nb is the number of 32-bit words in the block. This is always four. Nk is the number of 32-bit words in the key and can be 4, 6, or 8. Nr is the number of rounds of transformations performed. The number of rounds is a function of the block size and the key size, thus the number of rounds would be 10, 12, or 14 for key sizes of 128, 192, or 256 respectively since the block size is always four for AES.

The first step in the AES transformation moves bytes from the input array into the *state* array. Figure 2 illustrates this process. The i th and j th index of the *state* array are obtained from n by using the formulas in Figure 2, likewise n can be obtained from the i th and j th index of the *state* array.

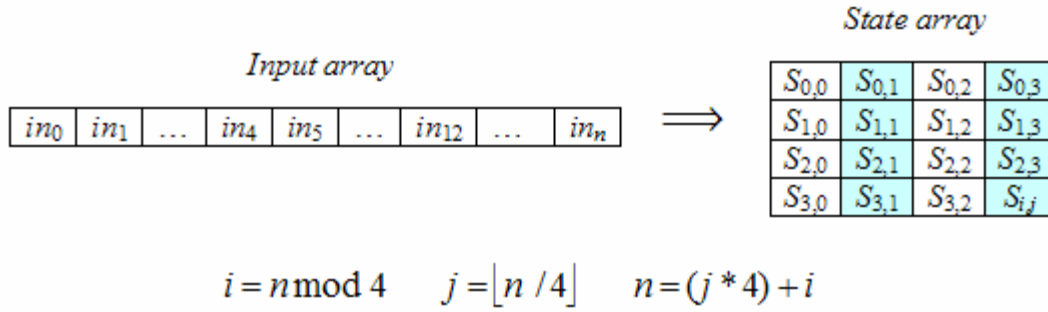


Figure 2. Input array into state array [NIS01b]

After the input is placed into the state array, a *KeyExpansion* function expands the cipher key into a series of *RoundKeys*, which are used in each of the Nr rounds. The *KeyExpansion* function is not discussed, but may be found in [DaR99] or [NIS01b]. The

KeyExpansion can be configured to store the *KeySchedule* into an array of 32-bit words commonly referred as $W[i]$, or the *KeySchedule* can be computed on the fly.

Below is the basic C-style pseudo code for the cipher's transformations. After preparing the *state array* the key is expanded to produce the *KeySchedule*. The algorithm performs a *AddRoundKey* transformation, followed by $Nr - 1$ rounds and a final round. Once the transformation is complete the *state array* is copied into a one-dimensional output array.

```
Rijndael (State, CipherKey)
{
    KeyExpansion (CipherKey, ExpandedKey);

    AddRoundKey (State, ExpandedKey);

    For (i = 1 ; i < Nr ; i++)
    {
        Round (State, ExpandedKey + Nb*i);
    }

    FinalRound (State, ExpandedKey + Nb*Nr);
}
```

Figure 3. Pseudo Code for Cipher's Transformation

Each of the individual rounds are made up of four distinct transformations with the exception of the final round which does not perform the *MixColumn* transformation. Below is C-style pseudo code for the round transformations. The four transformations *ByteSub*, *ShiftRow*, *MixColumn*, and *AddRoundKey* are each discussed individually in the paragraphs that follow.

```

Round (State, RoundKey)
{
    ByteSub (State) ;
    ShiftRow (State) ;
    MixColumn (State) ;
    AddRoundKey (State, RoundKey) ;
}

```

Figure 4. Pseudo Code for Round Transformation

ByteSub

ByteSub is a simple S-box byte substitution. The complicated part of this transformation is how the S-box is calculated. Since this is beyond the scope of this paper, it is not explained here; however the reader may refer to [DaR99] or [NIS01b]. Figure 5 illustrates this transformation.

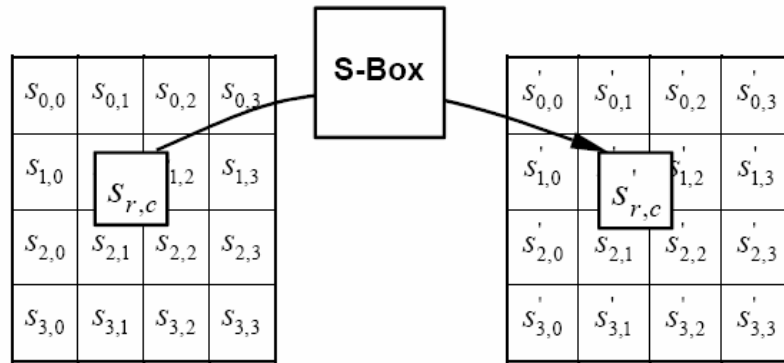


Figure 5. The byte substitution is applied to each individual byte [NIS01b]

Each of the individual bytes in the *state array* prior to the transformation are applied to the S-box to produce a new *state array* composed of the transformed bytes. The new *state array* replaces the previous *state array* as the cipher progresses to the next transformation.

ShiftRow

The *ShiftRow* transformation is perhaps the simplest of all. This transformation shifts the rows to the right 0, 1, 2, or 3 positions depending on which row the shift is being operated on. Since the transformation performs a cyclic shift, the right most bytes that fall off of the rightmost end are added to the front of the row. Figure 6 illustrates the *ShiftRow* transformation.

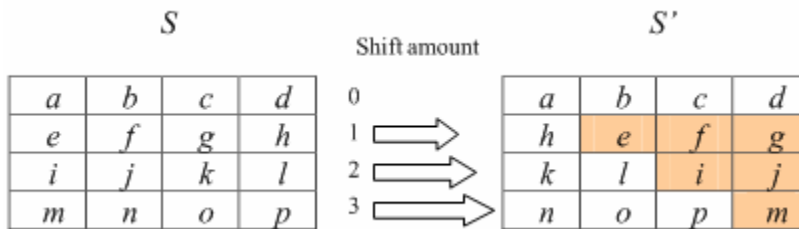


Figure 6. Each row is cyclically shifted to the right by the shift amount

MixColumn

The *MixColumn* operates on the *state array* one column at a time. It can be thought of as a matrix multiplication where the original *state array* column is multiplied by a special matrix by taking the product of two polynomials over $\text{GF}(2^8)$ (please refer to [NIS01b] for further explanation), the resultant matrix is the new column of the transformed *state*

array. Figure 7, Figure 8, and Figure 9 will help clarify the *MixColumn* transformation.

Figure 7 represents $s'(x) = a(x) \oplus s(x)$ in its matrix form where $s'(x)$ is the new *state array*, $a(x)$ is the special matrix, and $s(x)$ is the previous *state array*.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < Nb$$

Figure 7. Matrix multiplication of *state array* columns

Now Figure 8 shows the expanded matrix multiplication.

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}) \end{aligned}$$

Figure 8. Matrix multiplication carried out

Finally, Figure 9 shows *MixColumn* being applied across each column.

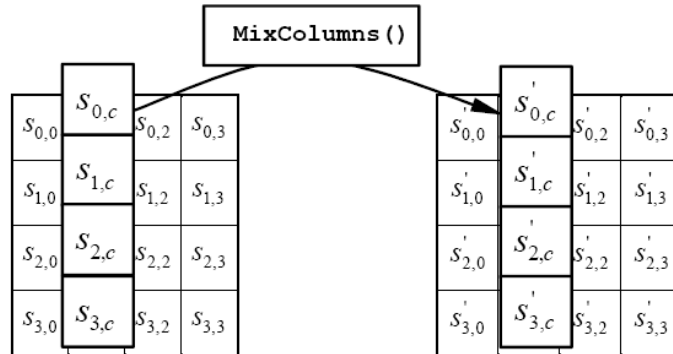


Figure 9. *MixColumn* transformation

AddRoundKey

The *AddRoundKey* is similar to the *ByteSub* transformation. Each byte of the *state* array is XORed with its corresponding expanded *RoundKey* to produce the resulting transformed *state* array as shown in Figure 10.

$$\begin{array}{|c|c|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|c|c|} \hline k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & k_{0,4} & k_{0,5} \\ \hline k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} & k_{1,5} \\ \hline k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} & k_{2,5} \\ \hline k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} & k_{3,5} \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} & b_{0,5} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ \hline \end{array}$$

Figure 10. *AddRoundKey* transformation

The decryption transformation for AES is similar and performs the inverse of each of the individual encryption transformations. For further explanation of the inverse transformations please refer to [NIS01b]. Appendix B of [NIS01b], has an excellent example of how to perform the above transformation and the resultant *state* array after each transformation of the AES encryption algorithm.

Data Encryption Standard (DES)

Most of the information in this section was obtained from [NIS99].

The Data Encryption Standard (DES) has a long history. It dates back to July 1977 when it first became a standard. It was reaffirmed as a standard in 1983, 1988, 1993, and in 1999. However, due to the discovery of a cryptanalysis attack which decrypted the cipher text within 24 hours, DES was improved by Triple DES (TDES), which runs a plaintext message through the DES transformation three distinct times.

The predecessor to DES was a group of ciphers called Lucifer developed by Horst Feistel and his colleagues at IBM [Fei06]. From Feistel's work, a class of ciphers known as *The Feistel Network* where the binary bits are transformed using substitution, transposition, and linear transformations such as exclusive-or (XOR), are all used in DES.

DES operates on a 64 bit block of data with a 64 bit key (56 bits used for the encryption and 8 used for error checking or ignored altogether). After the input is segregated into 64 bit blocks, a block is transposed using an initial permutation (IP) transformation. The block is then split into two separate sub-blocks of 32 bits appropriately called left (L_n) and right (R_n) where n is the round number. DES then performs 16 rounds of identical transformations before concatenating the two sub-blocks followed by a final permutation (IP^{-1}). Figure 11 illustrates the cipher transformation.

At the end of a round, where n is the round number, the two sub-blocks, L_n and R_n become

$$L_{n+1} = R_n \tag{1}$$

$$R_{n+1} = L_n \oplus f(R_n, K_n)$$

The new left sub-block is the same as the previous round's right sub-block and the new right sub-block is the previous left sub-block XORed with the result of the *Cipher Function*

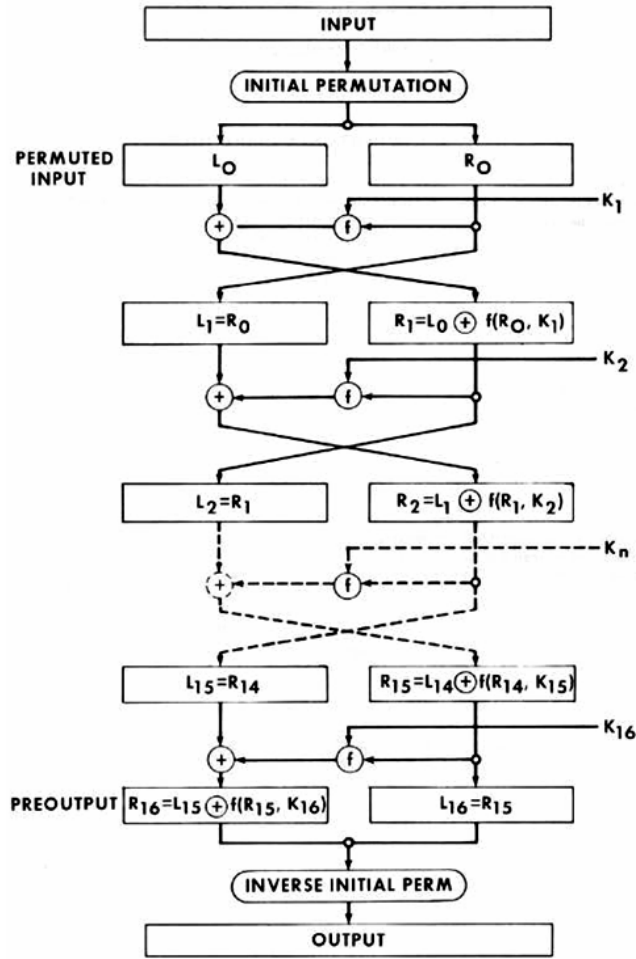


Figure 11. DES encryption transformation

(f) given the previous right sub-block and a *RoundKey* obtained from the key schedule.

The key schedule produces a 48-bit *RoundKey* from the round number and the cipher key.

$$K_{n+1} = KS(n, K_c). \quad (2)$$

The key schedule is not explained here but details are in [DES06]. The *Cipher Function* (f) sometimes referred to as the *Feistel function* as shown in Figure 12 transforms the

right sub-block by expanding the 32-bit sub-block into a 48-bit block by using an expansion table which repeats several of the bits in a systematic order.

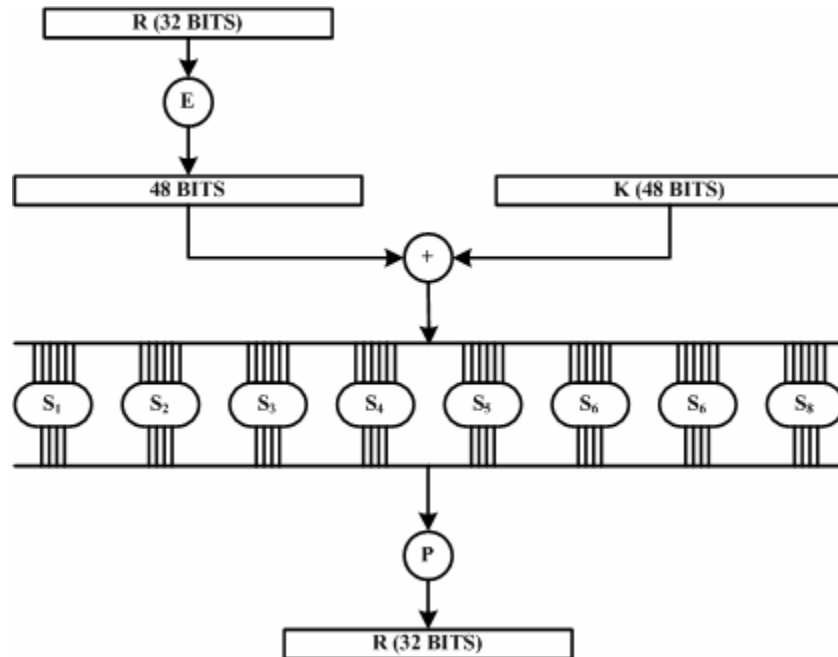


Figure 12. DES Cipher Function

The transformation XOR's the newly expanded block with the 48-bit *RoundKey* obtained from the key schedule. Next, the function separates the newly mixed 48-bit block into eight 6-bit segments to be processed by the eight S-boxes. The S-boxes ensure the cipher text is thoroughly diffused, linearity is significantly reduced, and produces a 4-bit output from the 6-bit input. The eight 4-bit segments are concatenated before being passed to the permutation function that transposes the bits in a predetermined fashion. The output of the *Cipher function* is passed on to the next round after XORing with the left sub-block as previously described.

The process of deciphering the ciphertext is the same as the encrypting the plaintext. This was a benefit of using DES, the symmetry of the algorithm reduced hardware cost because less circuitry was necessary. This simplicity may also have been its demise, because hardware conducted exhaustive key searches were possible within a 24-hour period.

Tiny Encryption Algorithm (TEA)

The Tiny Encryption Algorithm (TEA) was written and developed by David Wheeler and Roger Needham from the Computer Laboratory of Cambridge University. TEA was designed to be a simple to implement (only a few lines of code for most languages) and fast. It incorporates Shannon's principles of *confusion* and *diffusion* [Sha49] through the use of integer addition and bit-wise XORs instead of using substitution or transposition. This ensures a fast set-up time with little use of memory making it ideal for use in smaller electronic components.

The algorithm uses a Feistel structure similar to DES. Figure 13 illustrates a round of TEA divided into two cycles. TEA operates on a 64-bit block segregated into two 32-bit sub-blocks. The key for TEA is 128 bits and is also separated into four 32-bit sub-keys. In many of the implementations of TEA, each 32-bit sub-block or sub-key is stored as a 32-bit integer making the algorithm easier to implement. In Figure 13, the square boxes with the cross-bars represents integer addition modulo 2^{32} , the circle with the cross-bar is an XOR, and the double greater-than or less-than signs is a bit shift in the appropriate

direction. The Δ_i is a constant used to diffuse the bits independent of the key. The constant is generated from the following formula: $(\sqrt{5} - 1)2^{31}$ [WhN94].

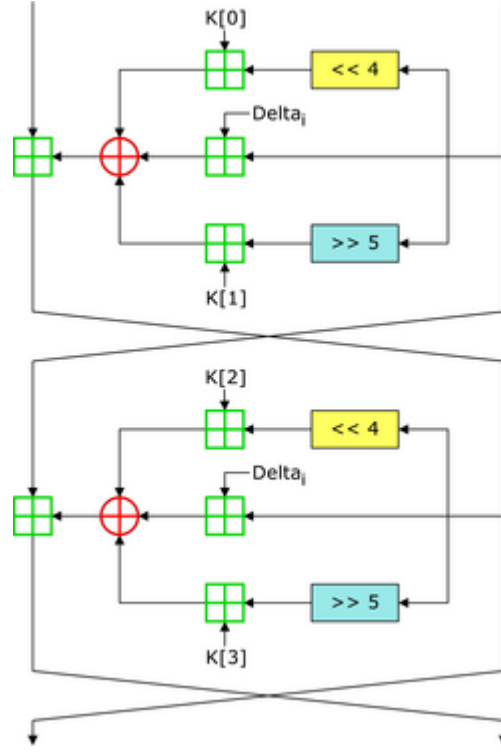


Figure 13. TEA Feistel Structure [Wik06]

TEA starts by storing the two sub-blocks into two integers. The easiest way to understand the two cycles illustrated in Figure 13 is with the mathematical formulas. For continuity the following equations use the same notation as the description of DES where i is the round number and the two cycles are accomplished by

$$sum_i += \Delta_i, \quad (3)$$

$$L_{i+1} = L_i + ((R_i \ll 4) + k_0) \text{ XOR } (R_i + sum_i) \text{ XOR } ((R_i \gg 5) + k_1), \text{ and} \quad (4)$$

$$R_{i+1} = R_i + ((L_{i+1} \ll 4) + k_2) \text{ XOR } (L_{i+1} + sum_i) \text{ XOR } ((L_{i+1} \gg 5) + k_3) \quad (5)$$

TEA typically performs 32 rounds each containing two cycles as described in (3) – (5). The *sum* is essentially *delta* multiplied by the round number. The first cycle adds the previous value of the left sub-block to the right sub-block after three different transformations XORed together as described by (4). The second cycle is essentially the same except the right and left sub-blocks are switched.

The developers of TEA claim the algorithm is very secure assuming the recommended 32 rounds are used. However, TEA suffers from some weaknesses. Andem [And03] discovered TEA has the strength of a 126-bit key and not the full strength of 128 bits due to the weakness of integer addition modulo 2^{32} . Kelsey et al. [KSW97] discovered several possible attacks on TEA some of which require 2^{30} known plaintexts. While this may seem impractical, for persistent data it could be significant.

Asymmetric Key Algorithm

Another category of encryption algorithms is the *asymmetric key* algorithms. Its name comes from the keys used to encrypt and decrypt a message. Asymmetric key algorithms are sometimes mistakenly called public-key algorithms. Although public-key algorithms are asymmetric key algorithms, the term is not all inclusive. In other words, there are some asymmetric key algorithms that do not follow the public-key paradigm. However, when referring to asymmetric key algorithms it is understood public-key algorithms are being referred to.

A public-key algorithm scheme has two distinct keys. The first is the *public key* (hence the name) and the other is the *private key* because it is kept secret while the public

key can be disclosed. A two key system eliminates the problem of transporting keys which is an issue with block ciphers. Using the public key, a message can be encrypted by anyone, however, the message can only be decrypted by those who possess the private key. In some public-key schemes a key is a tuple containing two numbers: a multiplier and a modulus.

Since the public key is available to anyone, the public-key algorithm scheme has some unique properties not found in the block cipher algorithms. Anyone can securely send the owner of the public key a private message which is decrypted using the private key. This is convenient for one-to-one message, but is not practical for widely distributed messages. In addition, if the message (or a portion of the message) was encrypted by the private key, then by using the public key, anyone could verify the message came from the owner of the private key, thus acting like a digital signature.

One drawback to the public-key encryption algorithm is its relatively slow speed because of the complex mathematical calculations of the encryption/decryption functions. To overcome this, some hybrid schemes combine the ease of key transportation with the speed and security of block ciphers. Pretty Good Privacy, for example, uses a private key to encrypt the symmetric key the block cipher algorithm uses [PGP05].

The security of the public-key algorithm depends on the difficulty of factoring large prime numbers, to be more exact, factoring a large number into its two prime factors. For example, assume 77 was a large number whose prime factors are 7 and 11. Given just 77, finding the prime factors of 7 and 11 is thought to take several hundred or thousands

of years using today's technology. However, with advances in number theory, this may change.

Rivest, Shamir, Adleman (RSA)

Most of the information in this section was obtained from [RSA78].

RSA is one of the most well known public-key encryption algorithms. It was developed by R.L. Rivest, A. Shamir, and L. Adleman and is described in [RSA78]. The algorithm derives its name from the first letter of each of the developers' last names. The encryption algorithm performs mathematical transformations on a numerical representation of the plaintext and produces a numerical representation of the ciphertext. The cipher text can then be stored in the preferred encoding. To decrypt the ciphertext it is first decoded into numeric form and the inverse mathematical transformation is used to recover the plaintext.

The keys for RSA are derived from large prime numbers of relatively the same size and are called p and q . The size of p and q determine the strength of the algorithm and RSA is thought to be secure when p and q have a size of 512 bits or more. Large prime numbers are difficult to calculate, but it is fairly easy to test a number for primality. Thus to generate the key pair a random number is generated and then tested primality using a deterministic or a probabilistic test such as the Rabin-Miller's test for primality.

Once the two prime numbers are generated, the modulo n can be calculated by multiplying the two prime numbers together. The modulus is used in both the *public* and

private key. The totient, $\phi(n)$, is calculated by multiplying $p-1$ and $q-1$ and a large random integer that is relatively prime to the totient and whose value is greater than one and less than n is found. For an integer to be relatively prime with respect to the totient, it cannot have any common factors with the totient other than 1. The *private* key consists of d and n . To finish calculating the *public* key the multiplier e is calculated where e is the “multiplicative inverse” of d modulo $(p-1)(q-1)$. Finding e is easily done by finding an integer x that results in an integer or

$$e = (x(p-1)(q-1) + 1) / d. \quad (6)$$

To encrypt text with RSA, the plaintext is encoded into an integer value and then the following transformation is applied

$$C = \mathcal{M}^e \bmod n \quad (7)$$

where C is the ciphertext encoded as an integer, and $\mathcal{M}$ is encrypted e . Thus, to decrypt the ciphertext the *private* key, d , and n are needed in the inverse transformation

$$\mathcal{M} = C^d \bmod n. \quad (8)$$

The following is a small example to demonstrate the RSA algorithm:

Choose $p = 11$ and $q = 13$

$n = 143$ and $\phi(n) = (p-1)(q-1) = 120$

Choose $d = 43$ arbitrarily and test for relative primality to $\phi(n)$

Next find e by choosing x such that e is an integer in

$$e = (x(p-1)(q-1) + 1) / d$$

Let $x = 24$, thus $e = 67$

Let $M = 45$, thus $C = M e \bmod n = 4567 \bmod 143 = 111$

Now use (8) to decrypt: $M = C d \bmod n = 11143 \bmod 143 = 45$

Even though RSA is currently considered secure, there are caveats. Due to the nature of the mathematical function used in the transformation, a plaintext that encodes to 0 or 1 will always encrypt to 0 or 1 respectively. To prevent this, a padding scheme is used to ensure the plaintext never encodes to 0 or 1.

RSA is not very efficient as a block cipher, however it can be used as one. The block size for RSA is the modulo size divided by 8. For example, if the modulo size is 1024 bits then the block size is 128 bits. The input size must be slightly less to accommodate the padding and the exact size of the input block depends on the padding scheme.

Encryption Modes

A block cipher can suffer from a slight weakness if the individual blocks are kept separate. This weakness occurs because like plaintext blocks will encrypt to like ciphertext blocks giving an attacker more information than is desired. This effect is illustrated in Figure 14 and is known as Electronic Code Book Mode (ECB) [NIS01b]. As can be seen in this ECB mode illustration, the pixels maintain their non-random appearance. Even though neither the key nor the data is compromised, the algorithm is vulnerable to non-conventional attacks. To combat this phenomenon, NIST has

published different encryption modes which link one block to another, thus eliminating the “sameness” of ciphertext with same plaintext.

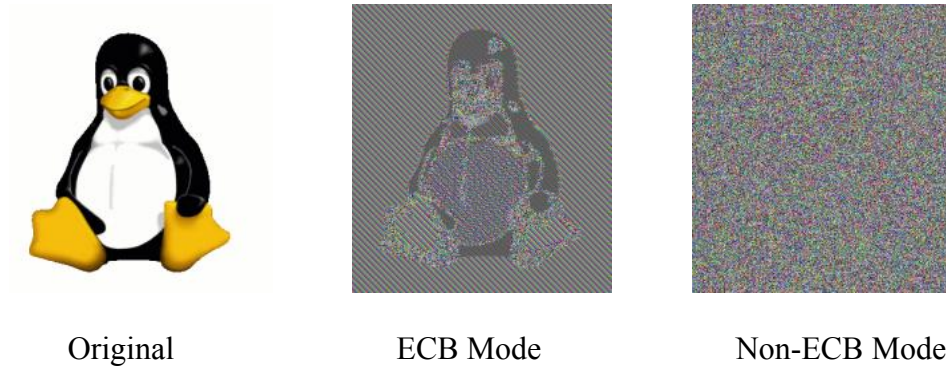


Figure 14. Demonstration of the weakness of ECB Mode [Wik06a]

Electronic Code Book Mode

In the ECB mode as shown in Figure 15, the encryption transformation is applied independently to each block with no interaction between them. The blocks are encrypted in parallel for faster encryption.

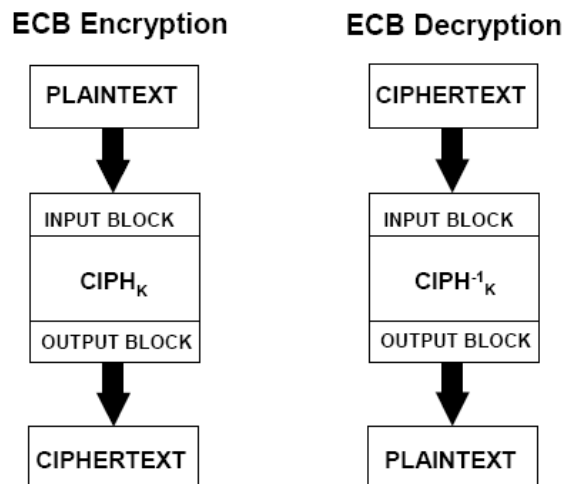


Figure 15. Electronic Code Book [NIS01b]

Cipher Block Chaining Mode

The Cipher Block Chaining (CBC) Mode links the plaintext of a current block with the ciphertext of the previous block with the XOR operator. Thus, if several blocks of plaintext are the same, after XORing the plaintext with the previous blocks ciphertext and applying the encryption transformation, the two ciphertext blocks will be different. Since the first block of plaintext does not have any ciphertext to XOR with, an initialization vector (IV) is generated. The IV can be any binary sequence of the same size as the algorithm's block size. The IV does not have to be secret, but should not be reused with the same encryption key and it must be unpredictable. With CBC the blocks can no longer be encrypted in parallel because one block depends on a previous block. Figure 16 illustrates the CBC process.

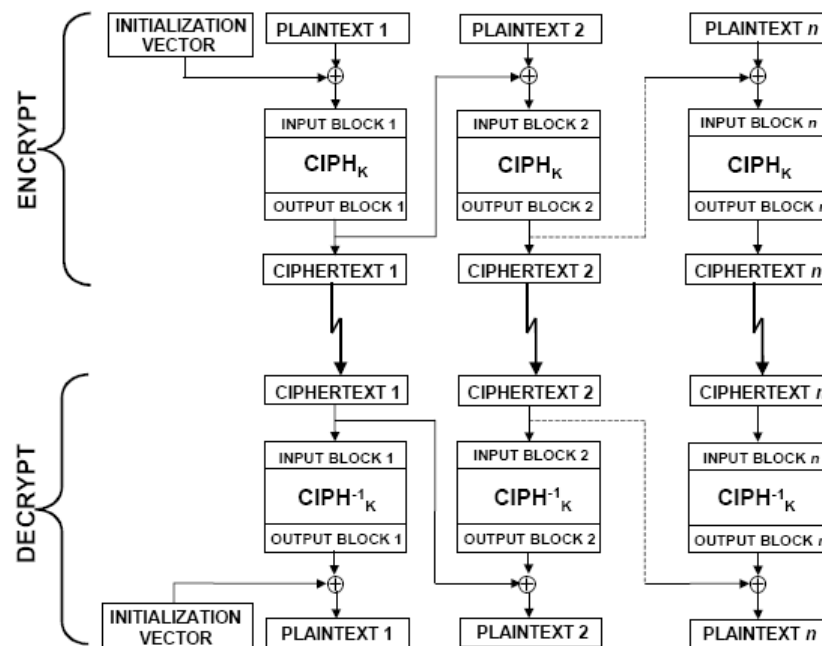


Figure 16. Cipher Block Chaining [NIS01b]

Cipher Feedback Mode

The Cipher Feedback Mode (CFB) feeds segments of the previous rounds input block back into the input block of the next round along with ciphertext of the previous text. The block size for the encryption algorithm remains the same, however only s new bits from the plaintext are fed into the cipher during each iterative step.

CFB also starts out with an IV that must be unpredictably generated and need not be kept secret. The IV is encrypted with the selected algorithm and key. The output of the cipher is segregated into two segments, the first being s most significant bits where s is an input to the CFB. The second segment consists of $b - s$ significant bits (where b is the length of the block). The second segment is discarded and the first segment is XORed to the next s bits of the plaintext to produce the ciphertext of that iteration. The input to the next block is the $b - s$ least significant bits of the previous iteration's input, in the case of the second iteration this would be the IV. The least significant bits from the previous input would be suffixed by the ciphertext from the previous round (which is s bits long) producing the next block for the encryption transformation. Figure 17 illustrates the CFB graphically. There are several additional encryption modes such as the Output Feedback and Counter Mode that are not discussed but can be access at [NIS01b].

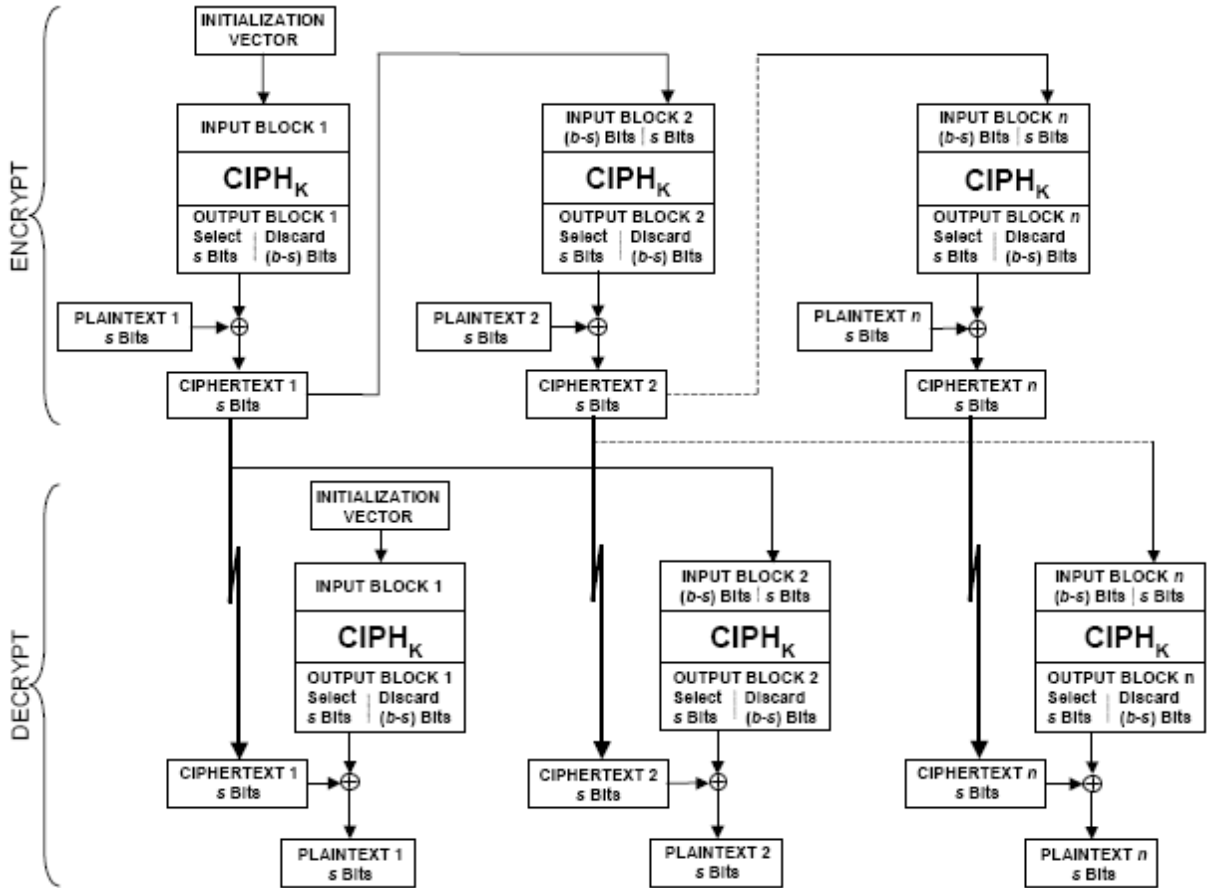


Figure 17. Cipher Feedback Mode [NIS01b]

Padding Schemes

In the realm of block ciphers the plaintext doesn't always evenly divide into the selected algorithm's block size. To overcome this, padding schemes fill the unused portion of the block. For a padding scheme to work effectively, the decryption transformation must realize a padding scheme is being employed and must be able to distinguish the padding bits from the message bits. Like the encryption modes, there are several padding schemes to choose from and only a select few will be discussed in this paper.

PKCS#5

This padding scheme is described in [RSA99] and is very common. It works effectively with block sizes up to 32 bytes wide [RSA93]; however [RSA99] describes a padding scheme of 8 bytes wide. For greater generality the padding scheme described in [RSA93] is presented below.

The encryption block (EB) of k bytes, $1 < k \leq 32$, is

$$EB = M \parallel PS \quad (9)$$

where M is the message, and PS is a padding string of $k - (\|M\| \bmod k)$ bytes where $\|M\|$ is the length of the plaintext message. Each byte has a binary representation of $k - (\|M\| \bmod k)$. For example,

$k = 16$, $M = \mathbf{34\ 87\ A4\ 1F\ DC}$, thus $\|M\| = 5$

$k - (\|M\| \bmod k) = 16 - (5 \bmod 16) = 11$ and in hex $11 = \mathbf{0B}$

Thus the padded message is:

34 87 A4 1F DC 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B

EB , then, would take on one of the following

$EB = M \parallel 01$ — if $k - (\|M\| \bmod k) = 1$;

$EB = M \parallel 02\ 02$ — if $k - (\|M\| \bmod k) = 2$;

.

.

.

$EB = M \parallel k_h \dots k_h$ — if $k - (\|M\| \bmod k) = k$

where k_h is the result of $k - (\|M\| \bmod k)$ encoded in hex. So the padding scheme remains unambiguous, if a message does not require padding, an additional padded block filled with k_h bytes is added anyways. To unpad the message, the last byte is read and that many are removed bytes from the end of the message ensuring each byte removed is the same as the last.

ISO-10126

ISO-10126 is a padding standard described in [XML02] and is the W3C recommended standard. This padding scheme is similar to the PKCS#5. The number of padding bytes is calculated using the same formula ($k - (\|M\| \bmod k)$). The last byte of the block is the number of padding bytes (n), and the remaining $n-1$ bytes are filled with random bytes. Using the padding example from the previous section, the final padded message would be:

34 87 A4 1F DC 52 AE B4 9C 2B FE 7A 61 8D 62 0B

Theoretically this padding scheme could be extended to any block size, but having a block size greater than 32 bytes complicates the padding scheme a bit. To unpad the message, the last byte is stripped (assuming $k \leq 32$), and encoded as an integer. That number of bytes is removed (which also includes the byte for the padding count). Unfortunately, the error checking benefits from PKCS#5 is lost in this padding scheme.

Other Padding Schemes

There are a number of other padding schemes. Some are as simple as padding the block with null bytes or blank spaces (encoded as 20 in ASCII), and some are quite

complicated utilizing hash or other such functions. But the overall objective remains: (1) fill the last block so the block cipher can perform its encryption transformation and (2) make asymmetric ciphers more secure.

Statistical Methods

The nature of cryptography makes analyzing ciphertext difficult. In most cases, the encryption algorithm used to produce the ciphertext is known. That is not the case for this research which analyzes the ciphertext of several encryption algorithms. Therefore, statistical analysis is used heavily. The first analysis determines the randomness of the ciphertext. A strong algorithm will produce ciphertext that is nearly random. However, it is hoped the research will reveal a signature or footprint of the algorithm. The signature may not come from the random number analysis alone. New statistical testing techniques by Eric Filiol may reveal possible biases that can be used to develop a signature for an encryption algorithm [Fil02].

Random Numbers

A random number is a number that is generated from a sequence of numbers such that the next number of the sequence is unpredictable and exhibits no pattern relating it to the other numbers of the sequence. Statistically the numbers must be independent and are from a uniform distribution.

Often it is more convenient to determine if a sequence of numbers is not random. To do this, patterns that signify the bits are non-random are searched for. This might include

too many ones or zeros, or the oscillation between ones and zeros occur too fast or too slow to name a few.

What is Statistical Analysis?

Statistical analysis uses of statistical algorithms and techniques to describe or interpret a set of data. This research uses statistical techniques to interpret patterns in the ciphertext produced by different encryption algorithms to identify the algorithm from the ciphertext alone. Statistical techniques are divided into two main categories; descriptive and inferential.

Descriptive statistics are statistical algorithms that describe a population or sample of data. The most common descriptive statistics are mean, median, mode, standard deviation, distribution, all of which describes some aspect of the data set.

Inferential statistics report information about the patterns found within the data. Such statistics include hypothesis testing, analysis of variance, correlation, and regression modeling. From inferential statistics decision about future performance or results can be made. For example, consider a set of data collected from a manufacturing machine. Inferential statistics can determine whether the machine is performing within the required specifications and determine whether it should be about recalibrated.

Statistical Analysis Tests and Tools

To determine if randomness is present in a set of binary bits, statistical tests and tools determine if the bit patterns are independent of one another and if they are uniformly

distributed. There are several packages available, however the NIST Statistical Tests for Random Number Generators is used in this research.

NIST Test for Random Numbers

The NIST test suite was developed by a team of engineers and statisticians [NIS01]. The package provides 16 different tests to determine randomness. A brief explanation of each of the tests is given below:

Frequency (Monobit) Test

This test determines if the 1's and 0's are uniformly distributed as expected in an random set of bits. The two should be approximately equal in count.

Frequency Test within a Block

This test determines if the 1's and 0's are uniformly distributed within a M -bit block. The two should be approximately equal in count within each block.

Runs Test

This test determines if series of identical bits in the bit sequence is too long or too short. It does this by counting the number of oscillations (change from 1 to 0 or 0 to 1) and determining if the number of runs is as expected.

Longest Runs of Ones in a Block

This test determines if the longest run of 1's within a block of M -bits is as expected in a random set of bits. Since an irregularity in the longest run of ones would indicate an irregularity in the longest run of zeros, another test determines the longest run of zeros.

Binary Matrix Rank Test

This test looks for linear dependencies among fixed length blocks of the bit sequence. The sequence is divided into a series of $M \cdot Q$ blocks that are rearranged into M by Q matrices. The rank of each matrix is calculated to determine if the rank is as expected of a random sequence (Appendix A of [NIS01] describes how to calculate the rank of a matrix). This test use a predetermined matrix size of 32 by 32, but requires an input bit sequence that is larger than any input bit sequence used in this research.

Discrete Fourier Transform (Spectral) Test

This test uses a Discrete Fourier Transform to produce a landscape of the bit sequence looking for patterns that would not be consistent with a random sequence. Figure 18 graphically illustrate the result of this test. The first graph is appears random where there is a clear pattern in the second.

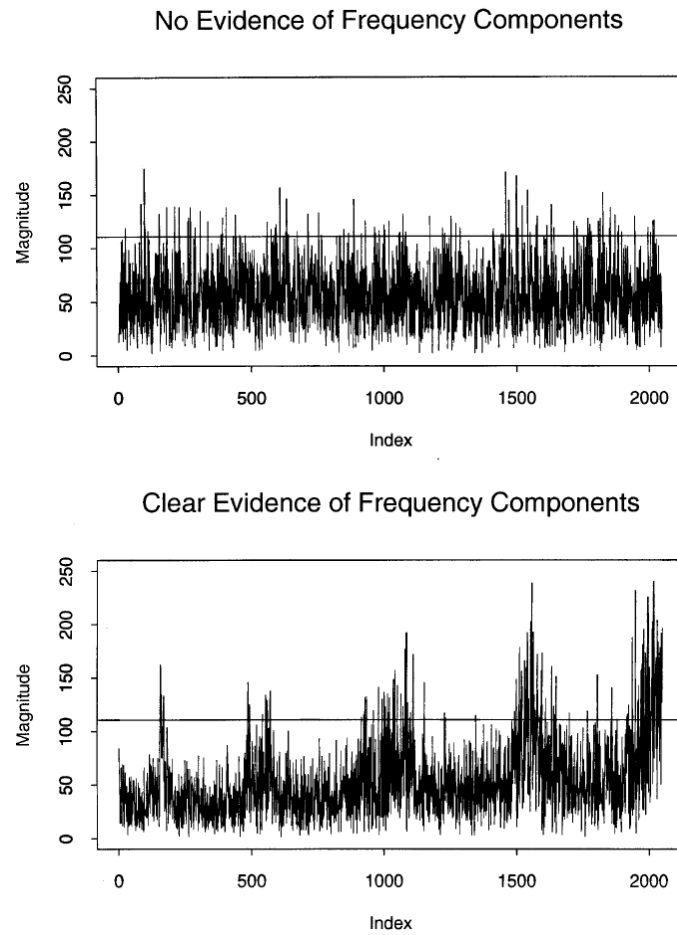


Figure 18. Discrete Fourier Transform (Spectral) Test Results [NIS01]

Non-overlapping Template Matching Test

This test searches for preset M -bit string patterns. The search starts at the beginning of the bit sequence, if a pattern is not found then the search continues at the next bit, if a pattern is found then the search continues at the first bit after the pattern.

Overlapping Template Matching Test

This test searches for preset M -bit string patterns. The search starts at the beginning of the bit sequence, search continue at the next bit whether a pattern is found or not found.

Maurer's "Universal Statistical" Test

This test determines if the bit sequence could be compressed. It does this by determining the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). If a bit sequence can be significantly compressed, then it is not random.

Lempel-Ziv Compression Test

This test also determines the compressibility of the bit sequence. If the bit sequence can be significantly compressed, then it is not random.

Linear Complexity Test

This test determines the length of a linear feedback shift register. If the length is too short, the bit sequence is not random. The Berlekamp-Massey algorithm [MVV97] calculates the length of the linear feedback shift register.

Serial Test

This test will determine whether the number of occurrences of a $2m$ m -bit overlapping patterns is approximately the same as would be expected for a random sequence. For

example, if $m = 3$, then occurrence of 3 bits (000 ... 111), 2 bits (00...11), and 1 bit (0, 1) patterns are counted and used in the test.

Approximate Entropy Test

This test is very similar to the Serial Test, except the bit patterns are derived from the input sequence. For example, if the bit sequence is 0111010 and $m = 3$, the possible test bit patterns of 3 bits are: 011, 111, 110, 101, and 010 derived from the bit sequence. Each occurrence of the full set of 3 bit patterns are counted and used in the test. In this example the full bit patterns are: #000 = 0, #001 = 0, #010 = 1, #011 = 1, #100 = 0, #101 = 1, #110 = 1, and #111 = 1. The above example simply illustrates the bit patterns and does not accurately reflect the testing algorithm.

Cumulative Sums (Cusum) Test

This test calculates the cumulative sums of the bit sequence to determine if it is consistent with a random bit sequence. It does this by converting the 0's in the bit sequence to -1 and then adding the first two bits, followed by the first three, and so on. It then compares the cumulative sums of the bit sequence in reverse. Thus the last two are added together followed by the last three, and so on. The forward test indicates a bit sequence with too many or too few ones in the first half where the backward cusum tests for too many or too few ones in the second half of the bit sequence.

Random Excursions Test

This test calculates the cumulative sum similar to the previous test, but only in the forward direction. The results of the cumulative sums are stored in a new set maintaining their order with an additional zero appended to the beginning and to end. The test then walks forward through the set calculating the number of cycles and generating the *random walk states*. The total number of cycles in which state x occurs exactly k times is counted. These counts are used to determine if they are consistent with a random bit sequence.

Random Excursions Variant Test

This test is almost the same as the Random Excursions test except only the number of times a particular state is entered is counted.

Many of the above tests require a large number of (10^6+) bits, and are not practical to use when trying to locate ciphertext within plaintext.

Statistical Cryptanalysis

Statistical cryptanalysis uses statistical methods and techniques to “break” a cipher, where “breaking” a cipher is extracting the plaintext and/or key just the ciphertext. There are many known cryptanalysis techniques, and even though this research may be considered an attack on the encryption algorithm, it should not be considered a cryptanalysis technique considering the goal is not to extract the plaintext or the key.

New Statistical Analysis

A new statistical test for testing randomness has been developed by Eric Filiol [Fil02]. It claims to show a strong bias when used to analyze output from both DES and AES encryption algorithms.

The test is based on a χ^2 distribution and is called Statistical Möbius Analysis. The test analyses the monomials with exactly d degree in the *Algebraic Normal Form* (ANF) of all of the Boolean functions modeling each of the output bits. The ANF is practically computed using the Möbius transformation and the results are compared to what would be expected of a random sequence.

Unfortunately implementations of this test have not been made public and the description of the test in [Fil02] was not sufficient enough to develop an implementation.

Previous Work

It is understood by experts in the field that encryption can be found by looking for randomness, however, there is no published research in this field or any known tool that can. Therefore, this research is not the first to suggest searching for randomness as a way to locate ciphertext in a binary file, but it is the first to publish a technique to.

Summary

This chapter provided the necessary background information to facilitate the understanding of this research. The key definitions used through out this paper are

plaintext and ciphertext. Plaintext is the message to be encrypted and the results of the decryption transformation. Ciphertext is the message after being encrypted; it is also the results of the encryption transformation.

The four encryption algorithms were another key part of this research. Background information was presented for each of the algorithms; the key concepts in this section were the algorithm types for each of the algorithms. AES, DES, and TEA are symmetric block algorithms, and RSA is an asymmetric algorithm that was used as a block cipher which is typically out of character for RSA.

The last key concept of this chapter is the NIST Statistical Tests for Randomness. The NIST test package consists of 16 tests. Each test analyzes a bit pattern using statistical methods to determine if the bit pattern is consistent with a random pattern. If several of the test results are positive for randomness, then with a fair level of confidence the bit pattern can be declared random.

III. Methodology

Chapter Overview

The chapter defines and explains the experimental methodology used to test the hypothesis and to meet the goals of this research.

Problem Definition

Goals and Hypothesis

The goal of this research is to find ciphertext hidden among plaintext within a binary executable file. It is presumed that only a few lines of machine code within the text segment is encrypted, while the rest of the text segment is presumed to be unencrypted, thus aiding in the protection of the encrypted segments by making this protection technique appear to be “stealthy.”

The hypothesis of this thesis is an encrypted region within a file can be identified with low probability of false negative or false positive error.

Approach

The approach used to isolate encrypted code segments from plaintext segments is to look for randomized bits. The NIST software segregates a file into groups of uniform blocks whose sizes are user defined and tests those blocks for randomness. If the block is determined to be random, it is assumed to be encrypted. Once an encrypted block is found, its offset from the beginning of the file is recorded and the search continues. A

number of location selection functions are evaluated and refined until the ciphertext is found.

System Boundaries

The Encryption Location System is made up of the following components: a collection of statistical tests and a location selection function. The system accepts the program to be tested as input. The system produces the location of the encrypted data. The parameters of the system are the acceptance level for the location selection function and parameters for the statistical test. Figure 19 is a visual representation of the Encryption Location System.

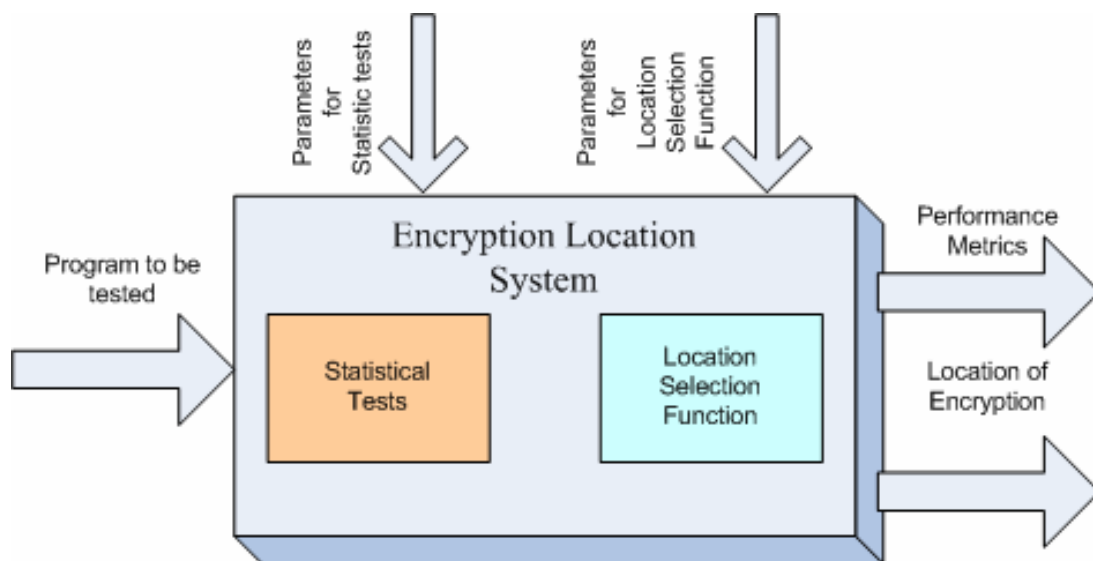


Figure 19. Encryption Location System

The statistical test component consists of the statistical tests used to evaluate the program to be tested. The selection function uses the results from the statistical test to determine the location of encrypted data (if any).

Performance Metrics

This research uses three metrics to identify how accurate the Encryption Location System locates ciphertext. These metrics are: (1) percent of encryption correctly located, (2) the percent of false negatives or the amount of encrypted bits that are missed, and (3) the percent of false positive or the amount of bits identified as encrypted but are not.

The percent of correctly identified encrypted bits is the location function presented in percentage form. For example, in Figure 20, in lines 6 – 12 is where the ciphertext is located and lines 9 – 14 is where the location function determined the ciphertext to be located, then the percent correctly identified would be 57.14%. There are 68 bits (between line 9 and 12) that were correctly identified out of a total of 119 encrypted bits. Thus the percent correctly identified is the number of ciphertext bits found divided by the total number of actual ciphertext bits.

False negatives occur between lines 6 and 8 inclusive. This is where the location function missed some of the ciphertext. This metric is stated as a percent of the total encrypted bits. Thus, the percent of false negative error would be 42.85%. The sum of the percent correct and the percent of false negative error is 100% (barring any rounding errors).

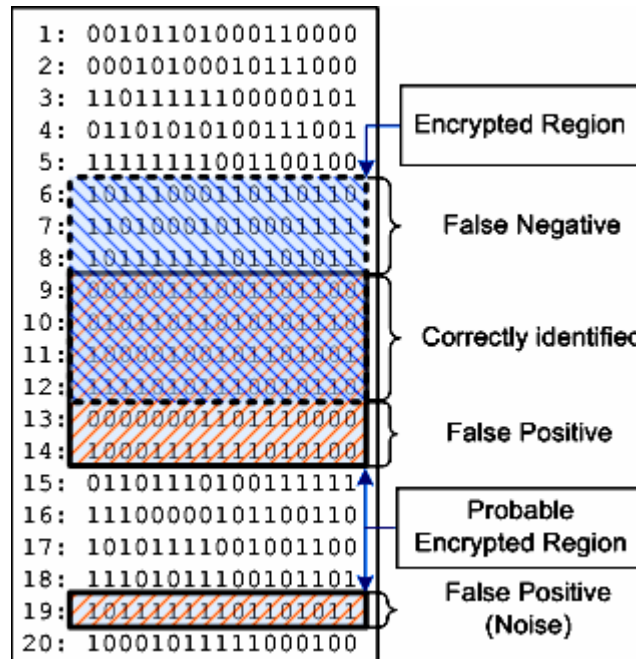


Figure 20. Error Diagram

The last metric is the percentage of false positives. The false positive error differs from the other two metrics by the divisor used to calculate the percentage. Since the location function could (in theory) falsely identify the entire file as encrypted (assuming the file does not contain any ciphertext), its divisor must be the total number of bits in the file.

There are two types of false positives this research is concerned with. The first will be referred to simply as false positive. This occurs within the probable encrypted region (the region the location function identified as ciphertext) that occurs within the encrypted region. Finding the exact starting and stopping point of the ciphertext is a more difficult problem to solve. Thus, there is more tolerance for this type of false positive error.

The other type of false positive error is when the location function identifies a probable encrypted region not associated with the encrypted region. This type of false positive error occurs when the bits in the file are determined to be random but are not. This type of false positive error is called noise, because it causes the location function to consider false encrypted regions rather than focusing on actual encrypted regions.

However, calculating the percentage of false positives, there is no distinction made between the two types. The sum of all false positive bits is divided by the total number of bits in the file. Thus, the percent of false positive error in Figure 20 is 15%, where the sum of the false positive bits is 51 and the total number of bits in the file is 340.

Parameters

System

The Encryption Location System has several system parameters including the computer system used to run the statistical test, the version of NIST statistical tests, the selection of statistical tests along with their associated parameters, and the parameter for the selection function. In later experiments the selection function was improved, requiring an additional parameter for selecting the sensitivity of the encryption region selection function.

In the Encryption Location System the computer system contributes little to the systems ability to locate the encrypted segments. Thus, any modern computer system will suffice; however, to simplify the running of hundreds of tests, a Linux operating

system was chosen for this research. As a result of choosing a Linux system the version of the NIST statistical tests used was version sts-1.5.

The NIST test suite contains 16 different tests; however most of the tests require an input data size of 10^6 or more bits. The presumed ciphertext input size for this research is at most half that size. Thus, only tests that operate effectively on small input size are used to locate hidden ciphertext. The following test can be conducted on small input size and were used: Frequency, Runs, Cumulative Sums, Discrete Fourier Transform, and the Binary Matrix Rank. The Binary Matrix Rank implementation in version sts-1.8 does not accept small input sizes. Thus, it should be excluded from the test suite if the Encryption Location System is implemented on a computer system running a Windows operating system.

The parameters for each of the above tests are the same. The parameters are the test-block size, the number of blocks contained within the file, and the format of the input data, which is always binary. To determine the optimal test-block size, a series of experiments are conducted. These experiments are described in the Experimental Design section with the results described in Chapter 4. The number of blocks contained within a file is the file size divided by the test-block size.

The selection function interprets the results of the statistical tests. The results of the statistical tests are stored in a binary grid representing the success or failure of the individual tests used, the more tests that pass, the greater the chance the test-block is truly random and therefore ciphertext. Thus, the selection function parameter is the number of

tests that must pass before accepting it as random. If the number is too small, then the chance of false positive rise, and if the number is too large, then there is a greater chance for false negatives.

The second generation of the selection function uses the padding technique as described on page 53 and requires an additional parameter. This fine tunes the way the selection function determines which of the probable encrypted regions are determined to be encrypted or false positive. This padding technique adds padding bits to the beginning of several copies of the target file. Each of these files are tested by the NIST tests and analyzed. The parameter is the number of the padded files an encrypted region must be present in to be consider a valid encryption region. This number cannot be larger than the number of padded files.

Workload

The parameters associated with the workload are used during the validation experiments. Thus, the system does not use the following parameters when locating hidden ciphertext within a target file.

In some of the selected encryption algorithms, the block size and/or the key size can be varied. The first parameter is the block and key size of a particular algorithm. A different block size or key size should not make any difference in locating ciphertext; therefore a standard block size and key size was used to minimize the number of experiments.

The last workload parameter is the characteristic of the plain text data. This technique for locating ciphertext should work for either object code or standard written text such as the novel Moby Dick. To test this assumption this technique was applied to both object code and to written text.

Factors

There are two factors in the Encryption Location System. The first is the selected encryption algorithms and the second is the length of the ciphertext.

This system is validated using four different algorithms. A high quality encryption algorithm's ciphertext (in a binary format) will appear to most statistical tests as random bits. This is evident in Soto's NIST report [Sot99a] for the round one testing of the AES candidates. Thus, AES and DES should both be easy to detect with the statistical tests, however it is not known if RSA and TEA will be as easy to detect.

Understanding the limits of this system is a secondary goal of this research. The larger the block of ciphertext (random data), the easier it will be to find. To test that theory three block sizes were selected (10, 100, and 500 32-bit word blocks). It is presumed the smallest size will be very difficult to find, the largest block size will be fairly easy, and the middle size is a hopeful challenge.

Evaluation Technique

After conducting the statistical tests, the results are analyzed and evaluated by the location selection function to determine if a block of data passed enough of the statistical

test as determined by the location selection parameter. If a block is found to be random, the whole block is assumed to be ciphertext. Further analysis is necessary to segregate the plaintext from the ciphertext within a block assuming it is possible.

Experimental Design

This section presents the details of the four experiments conducted in this research such as how the encryption was performed, a description of the software used to analyze the test result, and details of the location selection functions. The purpose and design of each of the four experiments is also given.

Encryption Software

The software used to perform all of the encryption was written using the Java Cryptography Architecture to perform the encryption of AES, DES, and RSA. The code for performing the encryption of the fourth algorithm (TEA) was originally done by Saurav Chatterjee; however, modifications were made to correct the implementation of the algorithm to conform to Wheeler and Needham's original algorithm [WhN94].

The encryption blocks for AES and DES used the Cipher Block Chaining Mode with a randomly selected initialization vector. The encryption blocks for the RSA algorithm were encrypted using RSA as a block algorithm. This caused the size of the RSA encryption block to be much larger than the AES and DES blocks. Like wise, the TEA encryption blocks were also slightly larger because of the input size for the TEA algorithm.

The three algorithms encrypted using the Java Cryptography Architecture used PKCS#5 as the padding scheme. A test file was encrypted and then decrypted by each of the algorithms to ensure the encrypted blocks were valid and reversible. All of the files used were produced from the same block of plaintext data with respect to the file type. The randomness between each trial was achieved by the random selection of a new key. The initialization vector was the same for each file, but it too was randomly selected. The location of the encrypted block was arbitrarily chosen, but was verified to be within the text segment using a disassembler.

Analysis Software

After gathering the encrypted data files for each experiment, the files were copied to the Linux machine for statistical analysis. A script was used to automate the process of running the NIST test on each file. The resultant grid files were stored into a file for further analysis.

The analysis program was written in Java. It reads the metadata (file size, encryption location, etc.) for each file type (i.e., AES_10, DES_100, etc.) and stored it in an orderly fashion for later use. The program then reads each grid file produced by the NIST software. The grid file was a series of 1 and 0 representing an individual's pass or failure of a particular NIST test. Each of the grid rows represented a test-block which was usually 2048 bits wide.

Each row of the grid was mapped to its location in the encrypted test file which is how a particular test-block was associated with a particular location in the test file. If a

block's sum was greater than or equal to the selection function's parameter, metadata from the corresponding block in the encrypted test file was stored in the data structure produced from loading of the file type's metadata.

After the analysis program processed each of the grid files, the program's data structure stored the actual location of the encrypted block and the perceived location obtained from the NIST test and corresponding analysis. The analysis program printed the statistics of each test to include the percent correct, percent of false negative and false positive.

Selection Function

The selection function is the core of the analysis program. It analyzes the list of encryption regions gathered from the run through the grid file, and determines which region or regions are likely to be encrypted. There were two distinct selection functions developed. The first is a rather basic version and the second is more sophisticated.

The first selection function assumes there was only one encrypted region. The encrypted region was assumed to be larger than any false positive noise. Thus, the selection function searched the list for the largest encrypted region previously discovered. This worked well for the larger 500 word blocks, but for the 100 and 10 word blocks it didn't work as well.

The second selection function uses a new padding technique. The padding technique adds non-random bits to the front of a test file to produce another file which is run

through the NIST test. The additional padding shifts the bits in the original file causing a different set of bits to be tested within a test-block. Figure 21 demonstrates the padding technique used in conjunction with the second generation selection function.

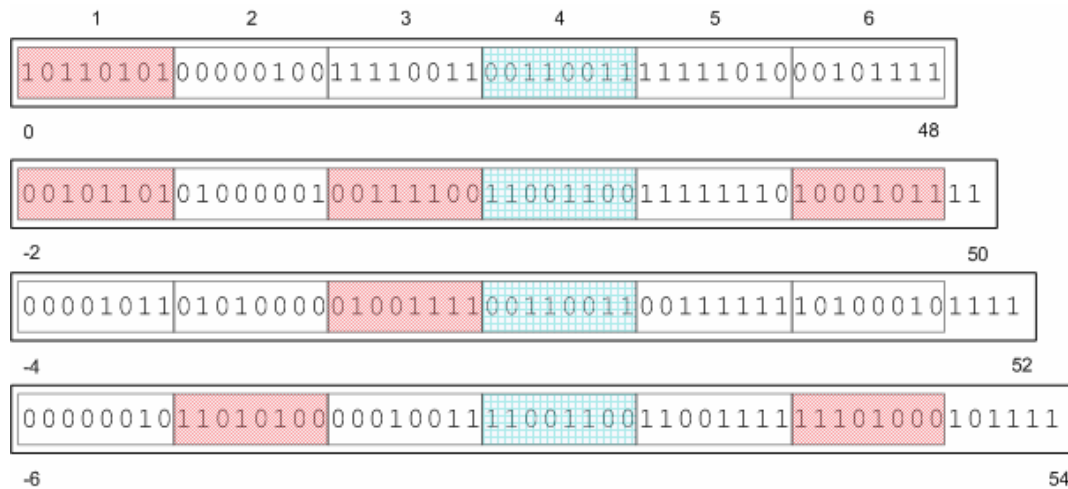


Figure 21. File Padding Technique Example

In the original file (the top one), block one is a false positive noise and block 4 is the location of the encrypted region. The second file is padded with two zero bits at the beginning. This padding in turn shifts the bits to the right causing a different set of bits to be tested within each test-block. The shift wasn't enough to eliminate the false positive noise and two more false positive noises showed up (in block 3 and 6). File 3 and 4 are likewise padded with two more bits than the previous file. As the bits are shifted the false positive noise moves, but the real encrypted regions remain.

The false positive noise occurs because the plaintext bits appear random to the statistical test. When the bits are shifted the false random bit patterns are broken up eliminating the false randomness. However, since the encrypted region was encrypted

with a strong encryption algorithm, its bits maintain their randomness as the bits are shifted. In this small example, the encrypted region stayed in block 4. In practice, the region sometimes shifts forward or backwards; however, its relative size and location remains the same (meaning the block should only shift forward or backwards one test-block).

As the analysis software comes across a probable encrypted region, it stores the region in the data structure, but will include a counter. If a probable region is encountered in the next file, its location is compared to the encrypted regions already contained within the data structure. If the probable encrypted region's starting location is within $\pm (1 \times \text{testBlockSize})$, then the two encrypted regions are merged. That is, the stored region's start is updated to the smaller starting point while the stopping bit is updated to the largest stopping bit and the counter is incremented by one.

After all of the padded files are analyzed the selection function searches the data structure for any encrypted region whose count is less than a predetermined sensitivity parameter. If the region's count is less than the sensitivity parameter, then the region is eliminated from the data structure. For this research, four files are used (one non-padded and three padded where each padded file is padded with one fourth the testBlockSize more than the previous file), and the sensitivity parameter is set to 3. Any remaining encrypted regions in the data structure are presumed encrypted.

In the process of merging the probable encrypted regions together, the region gets larger. This is to ensure the probable encrypted region captures the actual encrypted

region. This increases false positives associated with the encrypted region while eliminating the false positive noise. Even so, false positive added is significantly less than the noise eliminated.

Experiments Details

This section provides more detail on each of the experiments performed. Tests are conducted on both a text file and a Win32 executable file. The text file was an ASCII formatted file of the Constitution of the United States. In all of the experiments, the program test files were generated from the Adobe Reader 7.0 executable. These files were arbitrary chosen, mostly based on their availability to the general public.

Test-Block Size Test

The purpose of this experiment was to determine the best test-block size for NIST statistical tests. The NIST testing program reads a predetermined number of bits (called a test-block) from the selected file. The program initiates each of the selected statistical tests on that block recording the passing or failing of each test to the grid file before proceeding to the next block.

To determine the best block size, several tests were conducted using block sizes of 32, 160, 320, 640, 1280, and 2048. The block sizes were selected based on their divisibility of 32, the size of one instruction from a 32-bit instruction set. The best block size maximize the detection of ciphertext while minimizing false positive/negative errors.

After running each of the NIST tests, the results were analyzed using the first generation selection function to determine the number of errors and whether the known location of ciphertext was captured by a block passing the NIST tests for randomness. The block size that produced the best results was used in the Encryption Location System.

Algorithms Significance Test

After determining the test-block size, the next task was to determine if there was an encryption algorithm that was harder to find than others. The Algorithm Significance experiment was a full factorial design testing two file types, one text based and one program based. Each of these files were encrypted with one of the four algorithms (AES, DES, RSA, and TEA) and one of three block sizes (10, 100, and 500) a total of twenty times. Each of the 480 trials were analyzed using the first generation selection function.

Padded Encryption Files Test

The first selection function was fairly good, but produced too many errors. This experiment tested the padding technique and second selection function. From the results of the Algorithm Significance experiments, it was evident each of the four encryption algorithms produced fairly random data, thus only AES was selected for the following two experiments.

This experiment used files generated from the executable file encrypted with AES using each of the three block sizes with five trials. The 15 files were padded with zero

bits of $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{3}{4}$ of the test-block size, similar to Figure 21. After running the files through the NIST software, they were analyzed using the second selection function.

Multiple Encrypted Regions Test

The purpose of the final experiment was to determine if the selection function could find multiple encrypted regions within a file. This experiment is the same as the Padding Encryption File test except the number of encrypted regions was increased to three. The three encrypted regions are the same size and are arbitrarily located within the text segment.

Summary

The research methodology for this research is fairly straight forward. The first step was to model the program files. This was done assuming a block of encrypted code would be small (around 320 bits), a modest size (around 3,200 bits), or large (around 16,000 bits). Second, the encryption algorithm is a government approved algorithm mainly AES or DES. Lastly, it was assumed in the first three experiments only one block of encryption would be present in the program file.

Four experiments were conducted to test different hypothesis or parameters selections. The results of the first two experiments led to an improvement in the algorithm for the selection function, and the last two experiments tested the new function to determine if it was better and by how much.

IV. Analysis and Results

Chapter Overview

This chapter discusses the analysis process and present the results of the experiments conducted in this research.

Size Test Results

The first step in analyzing a file for embedded ciphertext is to perform the NIST statistical test for randomness with the file in question as the input. However, the best parameter for the test-block size is not yet known. The purpose of this experiment is to find that parameter. The best test-block is large enough to minimize the false positive error, and small enough to discern the starting and stopping point of the encrypted region.

Six block sizes were selected for testing 32, 160, 320, 640, 1280, and 2048 bits because they were multiples of a 32-bit word. The test files were generated from Adobe Reader 7.0 and the encryption region was encrypted with AES and had a block size of 500 words.

Figure 22 only shows data for block sizes of 1280 and 2048 bits. The NIST tests used in this research (sts-1.5) cannot perform the tests on block sizes less than 1000 bits. The results in Figure 22 are a graphically representation of the average of the 20 trials of each block size with a 95% confidence interval and Table 2 is a numerical representation of those results.

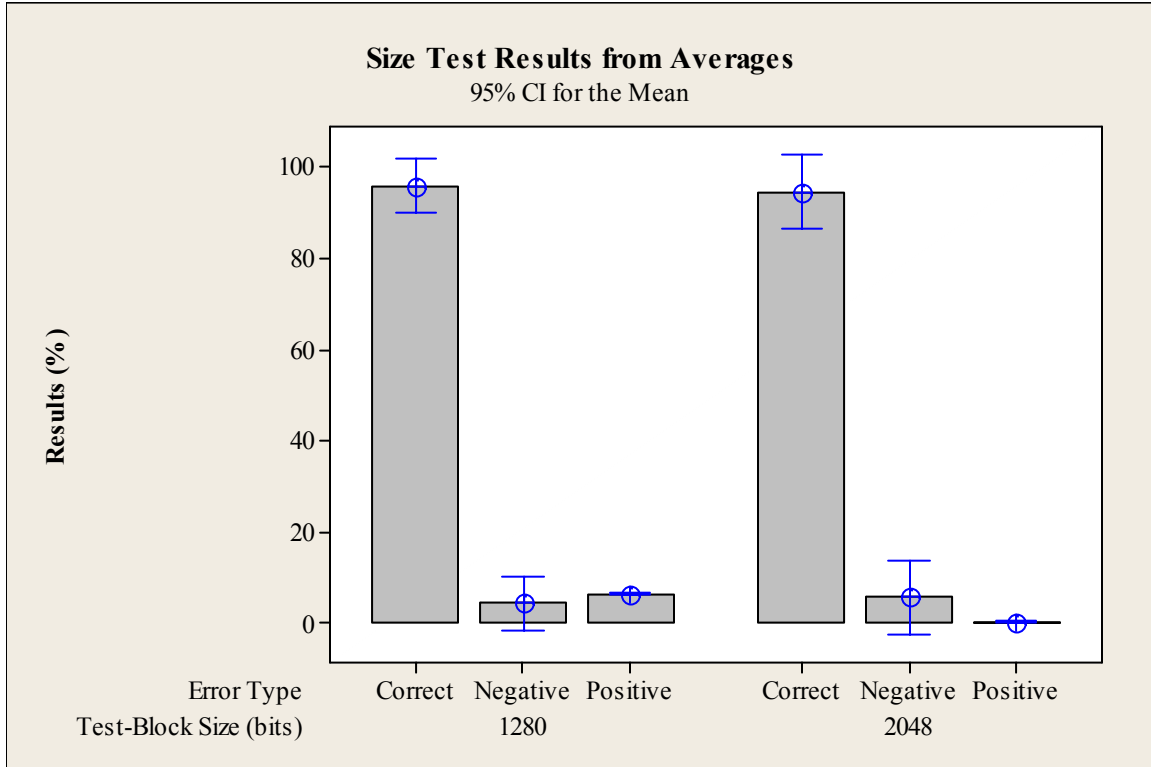


Figure 22. Size Test Results

Table 2. Size Test Average Numerical Results

		1280	2048
Avg	% Correct	95.8%	94.5%
	% Negative	4.2%	5.5%
	% Positive	6.3%	0.2%

It appears that a block size of 1280 bits and 2048 bits performed equally well. The block size 1280 bits performs a bit better with the number of bits found and had less false negatives (the percent correct and percent false negative should sum to 100%), but the performance improvement is only slight. The real deciding factor is the amount of false positives. The percent of false positives is calculated by dividing the number of encryption bits the selection function incorrectly identified as being encrypted when it

was plaintext with size of the actual encryption region. When using a larger input size, the NIST tests become more accurate. Using a smaller input size the NIST test incorrectly identifies a block as random, therefore producing more noise.

This test was conducted using the first selection function. The second selection function may be able to eliminate some of the noise, but because of the large amount of noise, this is not likely. The analysis calculated the number of encrypted bits to be at least 49,920 bits and there was only 15,872 bits in the encrypted region. When the 2048 bit block size was used the analysis calculated the encrypted region to be no more than 16,384 bits.

Clearly the larger block size is a much better choice for the test-block size. This decreases the granularity of the analysis, however, making it more difficult to locate the exact starting and stopping point of the encryption region.

Algorithm Significance Results

This test determines if one encryption algorithm was easier or more difficult to locate than another. The initial thought was AES and DES would be easy to find because they are known to produce statistically random bit patterns, and that RSA and TEA would be more difficult to find because these algorithms convert plaintext to a number to perform the transformation that would produce an output and would easily map to a integer, thus making it not statistically random. As it turns out, this is not so, and the output of both algorithms (RSA and TEA) appear statistically random within a relatively small block size.

This test was a full factorial test with 480 trials derived from two file types, three block sizes, four encryption algorithms, and 20 trials of each. The results presented in each of the tables and charts are the averages of the 20 trials in each category.

Figure 23 graphically shows the results of the test from the text plaintext file and Table 3 shows the results numerically. From these results a couple of observations arise. First, very few if any of the encrypted regions with a block size of 10 words found.

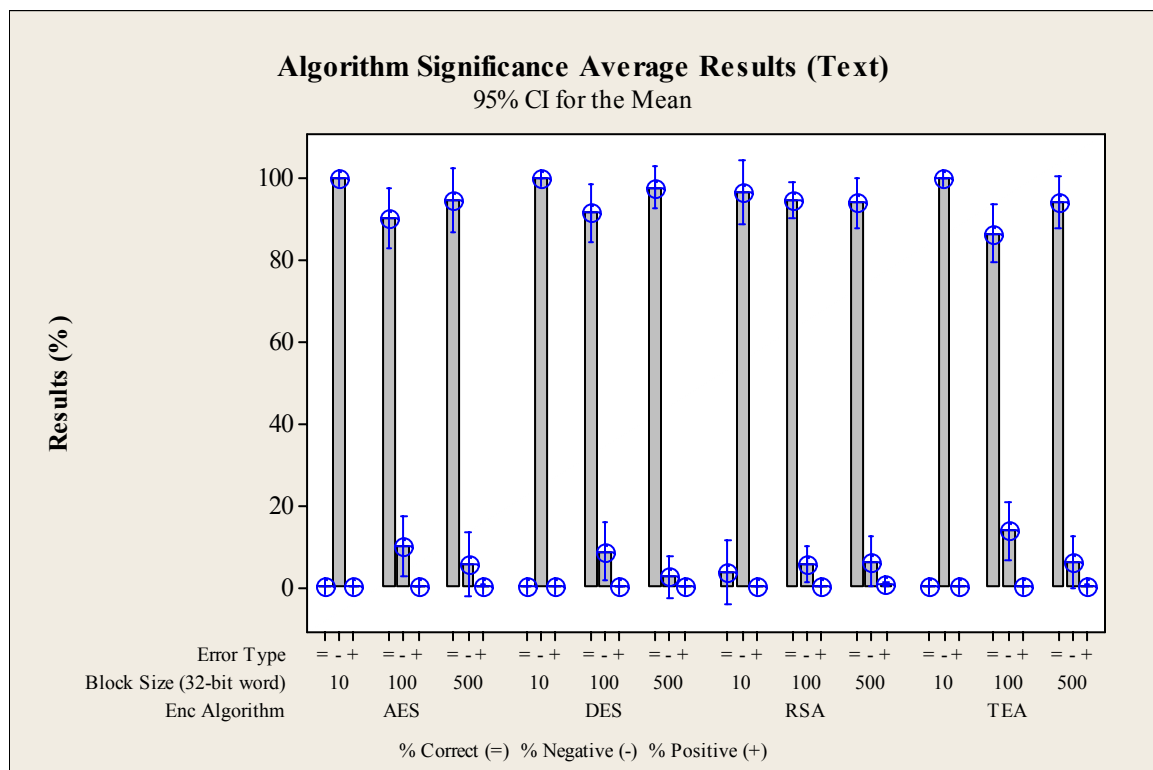


Figure 23. Text Test Results

The next observation is high level of success in finding the encrypted region. However, there is quite a bit of false negatives meaning the encrypted region was not completely captured by the selection function. Even so, this indicates two things,

Table 3. Algorithm Significance Average Numerical Results (Text)

	AES			DES			RSA			TEA		
	10	100	500	10	100	500	10	100	500	10	100	500
% Correct	0.0%	90.0%	94.5%	0.0%	91.3%	97.6%	3.8%	94.5%	93.8%	0.0%	86.4%	94.1%
% Negative	100.0%	10.0%	5.5%	100.0%	8.7%	2.4%	96.3%	5.5%	6.2%	100.0%	13.6%	5.9%
% Positive	0.0%	0.2%	0.3%	0.0%	0.2%	0.2%	0.0%	0.1%	0.6%	0.0%	0.2%	0.3%

first finding embedded ciphertext is possible, and second some improvements needs to be made.

Each algorithm is equally likely to be found by the location function. However, this test used files from a text file. It is very easy to just open the file and observe where the encryption is located as demonstrated in Figure 24. The encrypted section is highlighted for easy recognition. When the encryption is embedded within an executable file, it's not so easy to find.

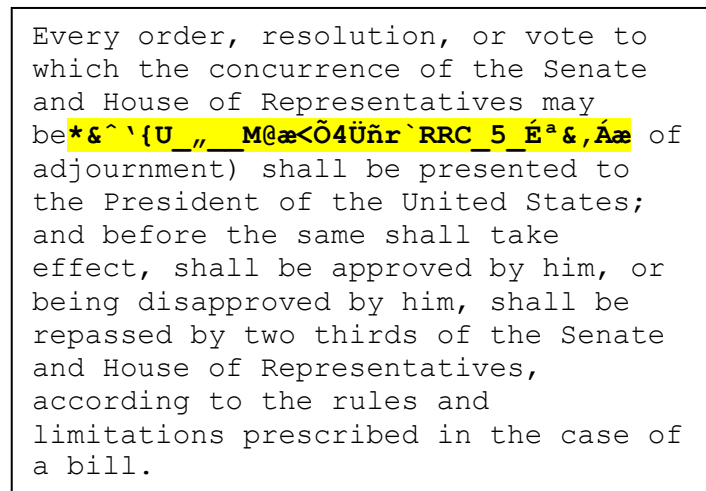


Figure 24. Text Encryption Example

The following two figures show just how difficult it can be to distinguish ciphertext and plaintext in an executable file. Figure 25 is a portion of the disassembled plaintext

file before being encrypted. The italic print extending from the 04 in line 0x4001FFE to the 8B in line 0x40201F is the code segment to be encrypted. The center column and the column furthest to the right is the disassembled op codes. Without the disassembler, the data would be incomprehensible.

0x401FF9:	7503	JNE	0x401FFE	;
0x401FFB:	894808	MOV	DWORD PTR [EAX+0x8],ECX	
0x401FFE:	8B5104	MOV	EDX,DWORD PTR [ECX+0x4];	
0x402001:	807A4400	CMP	BYTE PTR [EDX+0x44],0x0	
0x402005:	8D4104	LEA	EAX,[ECX+0x4]	
0x402008:	8BF1	MOV	ESI,ECX	
0x40200A:	B301	MOV	BL,0x1	
0x40200C:	0F85A5000000	JNE	0x4020B7	;
0x402012:	55	PUSH	EBP	
0x402013:	8B08	MOV	ECX,DWORD PTR [EAX]	;
0x402018:	8B5500	MOV	EDX,DWORD PTR [EBP]	
0x40201B:	3BCA	CMP	ECX,EDX	
0x40201D:	7550	JNE	0x40206F	;
0x40201F:	8B5508	MOV	EDX,DWORD PTR [EBP+0x8]	
0x402022:	807A4400	CMP	BYTE PTR [EDX+0x44],0x0	
0x402026:	7518	JNE	0x402040	;
0x402028:	8B08	MOV	ECX,DWORD PTR [EAX]	;

Figure 25. Program Example (Plaintext)

Figure 26 is the same portion of code encrypted using the AES algorithm. It is highlighted in italic text beginning at the EF in line 0x401FFE and ending at the E5 in line 0x40201E in line. Even though the text is encrypted the disassembler assumes they are valid op codes and disassembles it as if it was not encrypted. Comparing the two figures it's easy to see they are different; however, a would-be reverse engineer would have a very difficult time locating the encrypted portion. In fact, the code can still be executed with unknown side effects when the processor enters the encrypted code segment. Thus, to find embedded ciphertext within an executable file a statistical technique is necessary.

0x401FF9: 7503	JNE	0x401FFE ;
0x401FFB: 894808	MOV	DWORD PTR [EAX+0x8],ECX
0x401FFE: 8B51EF	MOV	EDX,DWORD PTR [ECX-0x11];
0x402001: DF3B	FISTP	QWORD PTR [EBX]
0x402003: BEECA13209	MOV	ESI,0x932A1EC
0x402008: 8CB7346F09F9	MOV	WORD PTR [EDI-0x6F690CC],
0x40200E: D015BC001E8E	RCL	BYTE PTR [0x8E1E00BC],1
0x402014: 7476	JE	0x40208C ;
0x402016: 028176E266FE	ADD	AL,BYTE PTR [ECX-0x1991D8A]
0x40201C: AF	SCASD	
0x40201D: F9	STC	
0x40201E: DBE5		
0x402020: 55	PUSH	EBP
0x402021: 08807A440075	OR	BYTE PTR [EAX+0x7500447A],AL
0x402027: 188B08885944	SBB	BYTE PTR [EBX+0x44598808],CL

Figure 26. Program Example (Ciphertext)

Figure 27 shows the results using the first selection function on executable files and

Table 4 shows the same results numerically.

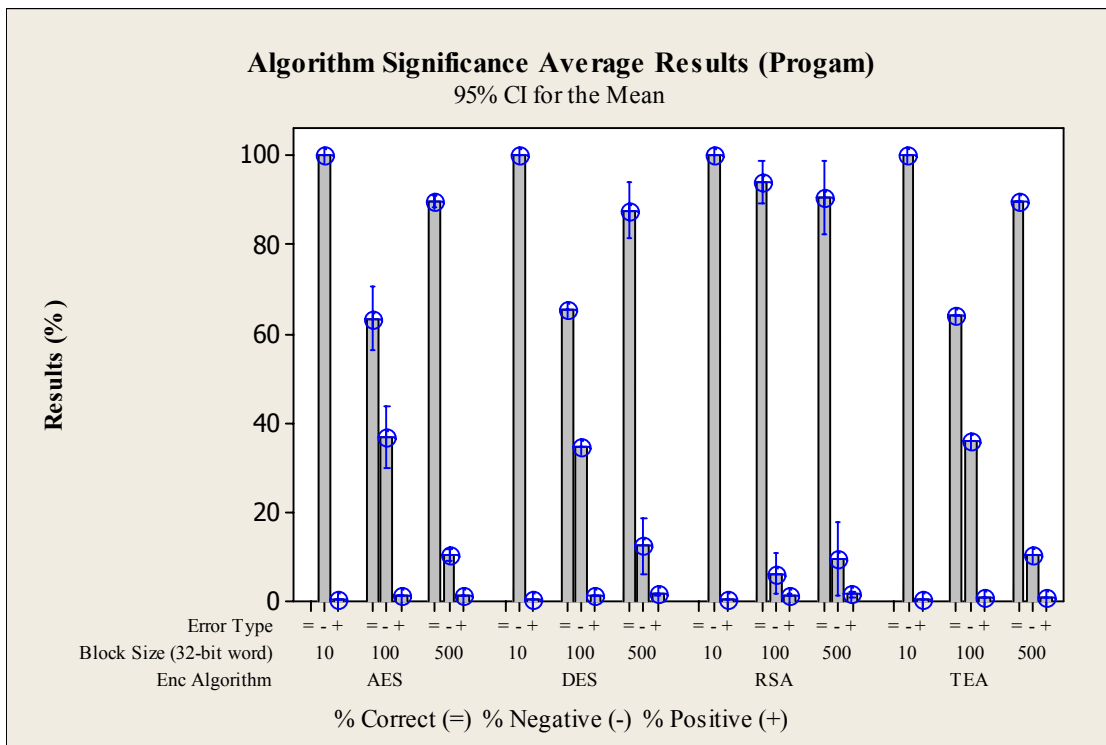


Figure 27. Program Test Results

Table 4. Algorithm Significance Average Numerical Results (Program)

	AES			DES			RSA			TEA		
	10	100	500	10	100	500	10	100	500	10	100	500
% Correct	0.0%	63.3%	89.7%	0.0%	65.3%	87.5%	0.0%	93.8%	90.4%	0.0%	64.0%	89.6%
% Negative	100.0%	36.7%	10.3%	100.0%	34.7%	12.5%	100.0%	6.3%	9.6%	100.0%	36.0%	10.4%
% Positive	0.4%	1.4%	1.4%	0.4%	1.4%	1.5%	0.4%	1.5%	1.7%	0.4%	1.1%	1.1%

The encrypted regions in an executable file are more difficult for the first selection function to locate. Table 5 shows the average error contained in a probable encrypted region. From this data it's evident the probable encrypted region is not catching all of the ciphertext. The negative numbers indicate the number of bits missed (false negatives). The reason for the false negatives is in the last test-block. The ciphertext extends into the

Table 5. Average Bit Error of Encrypted Region

	AES			DES		
	10	100	500	10	100	500
Error Beg	0	0	-102.4	0	0	-409.6
Error End	0	-972.8	-1536	0	-1088	-1497.6
	RSA			TEA		
	10	100	500	10	100	500
Error Beg	0	-460.8	-2406.4	0	-128	-128
Error End	0	51.2	-2304	0	-1024	-1536

last test-block but there are too few ciphertext bits to cause the block to pass enough the NIST statistical tests to be considered random.

Based on this experiment, a few improvements need to be made. The selection function can't assume the largest probable encrypted region will contain all of the ciphertext for two reasons. The file may contain more than one encryption region and the size of a false positive region may be the same size or larger then the probable encryption

region capturing the encryption region. The probable encryption region must be expanded to capture missed ciphertext without significantly increasing false positive error. To overcome these limitations the second selection function was developed.

Padding Encryption File Results

The second selection function as described on page 53 uses a padding technique to decrease the likeliness of selecting a false positive encryption region. This new function was tested using files encrypted with AES and three block sizes. Five trials were conducted across the three block sizes implementing the padding technique. Figure 28 presents the averages of the results with a 95% confidence interval.

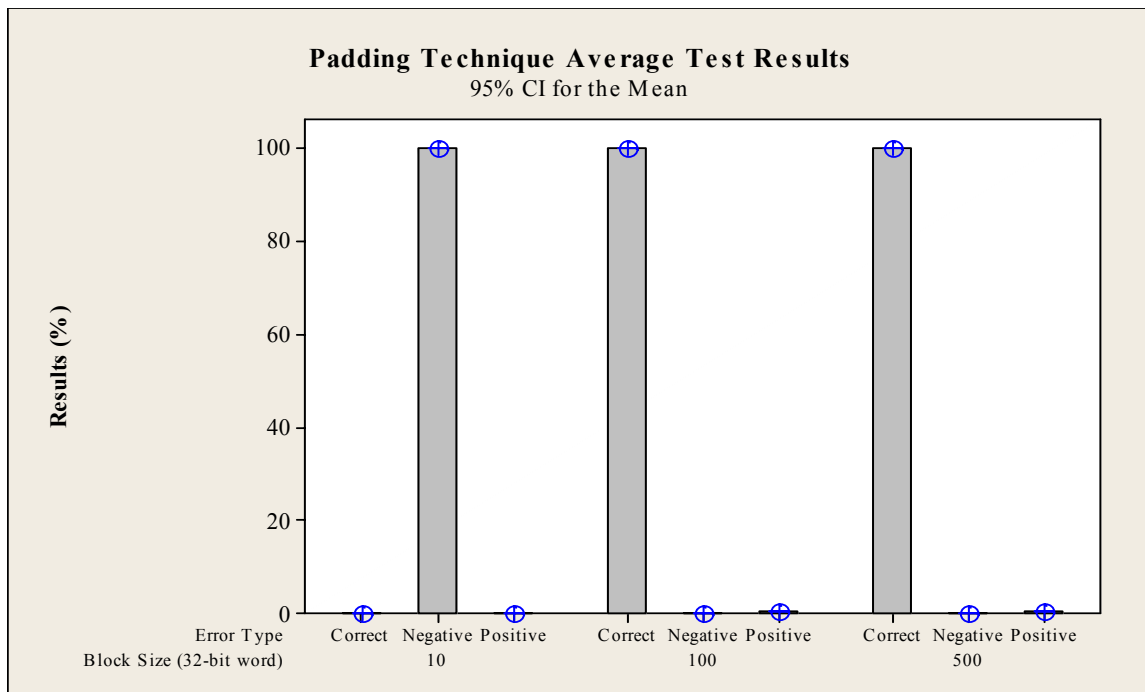


Figure 28. Padding technique Test Results

With the exception of the 10 word block, the entire encrypted region was captured by the probable encrypted region with no false negative error. The merge function within the selection function combines like probable encryption regions by setting the encryption starting bit to the smallest of the two and the encryption stopping bit to the largest of the two regions. This produced more false positives, but eliminated the noise.

Figure 29 directly compares the positive error of the first and second selection function. The amount of false positive found with the first selection function is significantly more than that of the second. Therefore, the improvements due to the second selection function is statistically significant.

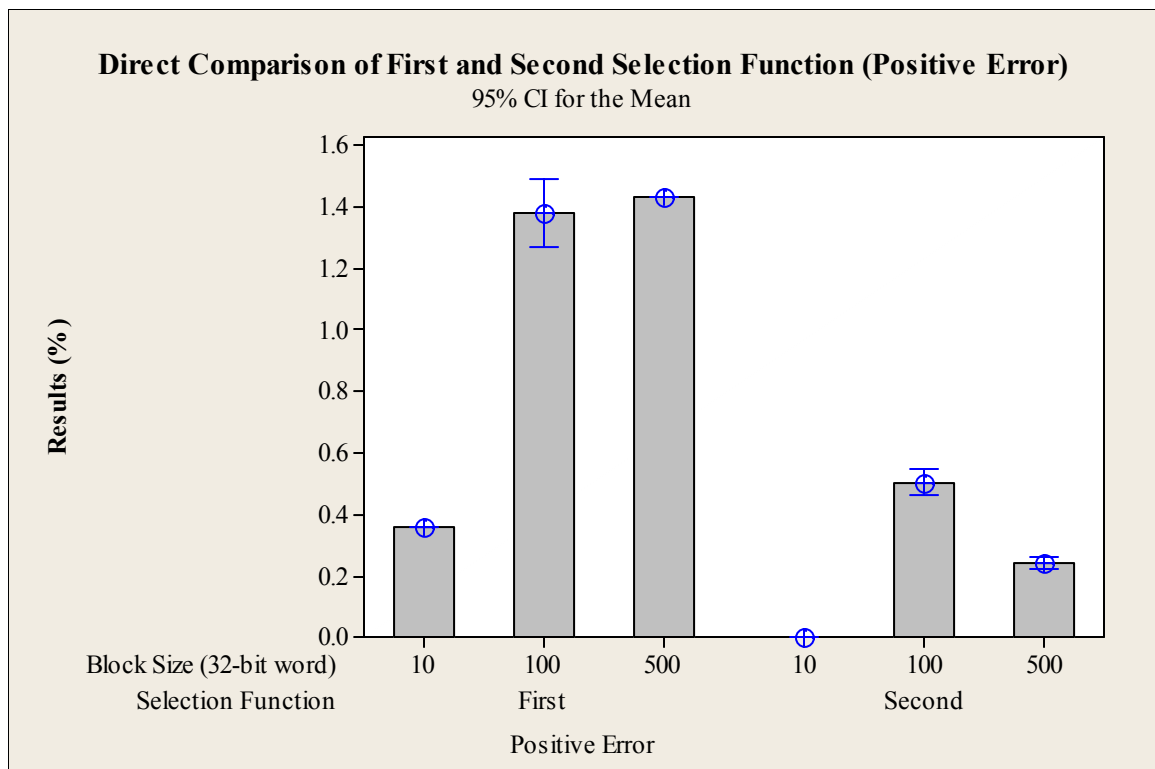


Figure 29. Selection Function Comparison

Multiple Encrypted Regions Results

The Multiple Encrypted Region tests are similar to the padding technique tests. The only real difference is the location and numbers of encryption regions. These tests were conducted with three separated encryption regions all within the text segment of the executable file. The purpose was to determine the benefits of the second selection function. Figure 30 present the results of this experiment graphically, while Table 6 presents them numerically.

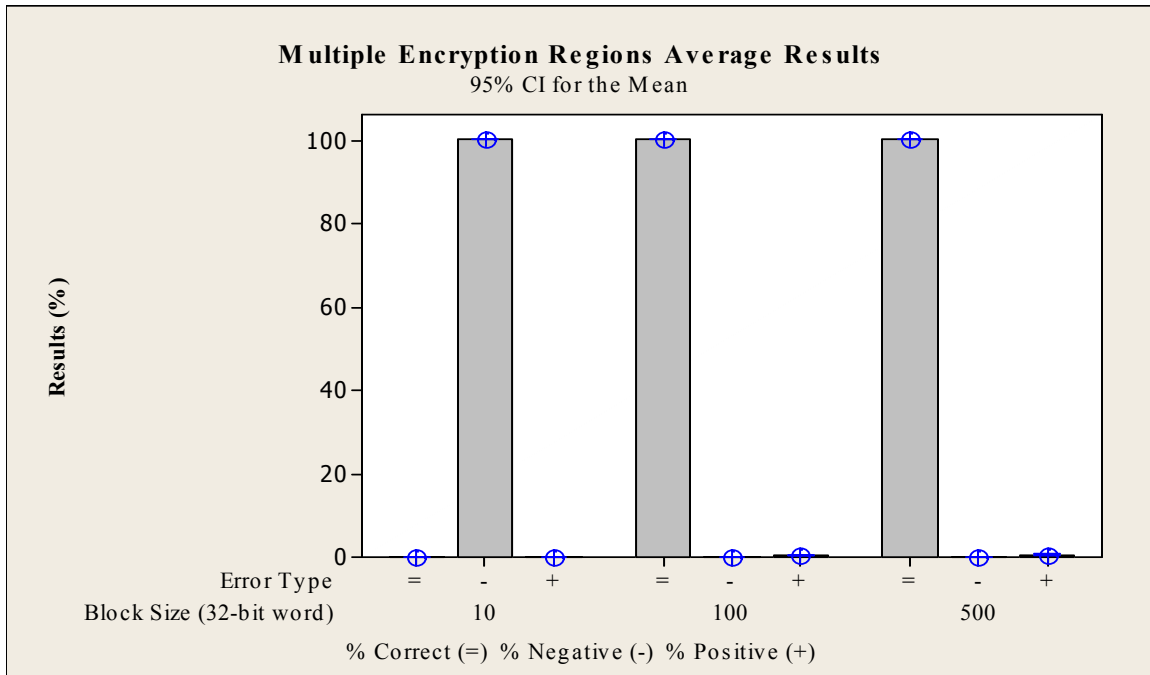


Figure 30. Multiple Encryption Regions Test Results

Table 6. Multiple Encrypted Regions Average Numerical Results

		10	100	500
Avg	% Correct	0.0%	100.0%	100.0%
	% Negative	100.0%	0.0%	0.0%
	% Positive	0.0%	0.5%	0.5%

The results of the multiple region tests look very promising. There was a slight increase in the false positive error, but that was expected since there are two more encryption regions. The fact that false positives did not increase for the 100 word block is not expected. Figure 31 is a portion of the output from the Encryption Location System for the 500 word block and each of the 5 trials. The actual location of the three encryption regions are at bit 43008 – 58880, 65536 – 81408, 98304 – 114176. All of the encryption regions are captured by the probable encryption regions with false positives on both sides of most regions.

```

500
1: the bit stream between bit 43008 and bit 59328 appears to be random.
   the bit stream between bit 65472 and bit 81856 appears to be random.
   the bit stream between bit 98240 and bit 116544 appears to be random.

2: the bit stream between bit 43008 and bit 59328 appears to be random.
   the bit stream between bit 65536 and bit 81856 appears to be random.
   the bit stream between bit 98240 and bit 116544 appears to be random.

3: the bit stream between bit 43008 and bit 59328 appears to be random.
   the bit stream between bit 65472 and bit 81856 appears to be random.
   the bit stream between bit 98304 and bit 114688 appears to be random.

4: the bit stream between bit 43008 and bit 59328 appears to be random.
   the bit stream between bit 65536 and bit 81856 appears to be random.
   the bit stream between bit 98240 and bit 116544 appears to be random.

5: the bit stream between bit 43008 and bit 59328 appears to be random.
   the bit stream between bit 65536 and bit 81920 appears to be random.
   the bit stream between bit 98240 and bit 114560 appears to be random.

```

Figure 31. Encryption Location System Output

The second selection function accurately locates multiple encryption regions of at least 100 words with no false negatives, no noise, and a small amount of false positives.

Summary

The Encryption Location System has demonstrated its ability to accurately locate embedded ciphertext using the second selection function. The tests demonstrated its ability to capture all of the ciphertext within its probable encryption region with minimal error.

The first experiment searched for the best test-block size to use for the NIST tests. The experiment determined a test-block size of 2048 bits eliminated the most false positive error while maintain its ability to locate the ciphertext.

The next experiment determined if any of the four algorithms was harder to detect than any others and which block size was able to be detected. Each of the four algorithms were just as detectable as the next; however, a block size of 10 32-bit words was too small to accurately detect.

The final two experiments tested the padding technique to improve in the detection of ciphertext and the elimination of errors. Both did very well. The final experiment was to find multiple encryption regions within a single file.

V. Conclusions and Recommendations

Chapter Overview

This chapter provides an overview of this research, its findings and significance. Recommendations for future research and action to be taken by anyone relying on embedded ciphertext for security purposes are suggested.

Conclusions of Research

This research developed a method of using the NIST Statistical Test for Randomness to accurately locate embedded ciphertext within a text or executable file. This method located all of the ciphertext within a file while minimizing the false positives assuming the file contains one or more blocks of ciphertext of at least 100 32-bit words (3,200 bits in length). The ciphertext must be produced by an encryption algorithm whose output is statistically random such as AES, DES, TEA, or RSA.

This technique cannot determine the difference between ciphertext produced by an encryption algorithm or another random bit generator. It relies on the output of the NIST test; however, it may be successful with other statistical tests for randomness as long as the program can test individual blocks of data and produce an output file similar to the NIST tests. The size test may need to be reaccomplished to calibrate the statistical tests to meet the same parameters used in this research.

Significance of Research

Any system relying on the difficulty of detecting embedded ciphertext in a binary file as part of its security should use this technique against their files to ensure their system will not be compromised. If the key and initialization vector (if used) is kept separate from the file, the security is still dependent on the strength of the encryption algorithm. Otherwise the security is dependent on the difficulty of finding those two items. Depending on the size of the file, a brute force attack could still be possible.

Recommendations for Action

The Encryption Location System should be used to ensure the security of a system is maintained. Some recommendation are:

- Develop techniques to defeat the NIST test.
 - Add non-random data to the ciphertext in a reversible manner, thus helping to camouflage the encryption region.
 - Add random data throughout the file to essentially defeat the selection function by causing it to falsely report more encryption regions than actually exist.
- Keep the key separate from the file to keep it from being compromised

- If the key must be hidden in the file, doubly encrypt the ciphertext. If possible, use two different encryption algorithms, forcing the attacker to locate two separated keys.

Recommendations for Future Research

There are several possible routes to take for future research. The original intent of this research was to complete the first of several steps to crack the encryption software protection technique described in Chapter 1. The steps are to locate the ciphertext, identify the encryption algorithm, locate the key and initialization vector, and use that information to compromise the protection scheme. The research partially completes step one. Techniques to accomplish the other steps still need to be developed.

Future research could also focus on improving the technique described in this paper. This technique cannot completely isolate the ciphertext from plaintext data on either end of the encryption region in close proximity. Another improvement would be the ability to locate smaller regions. This technique was not able to detect regions smaller than 100 32-bit words.

Lastly, research to defeat this technique, similar to the ones described in the Recommendation for Action, could be tested and developed.

Summary

This chapter presents the conclusions of this research and the encryption location technique developed. It provides recommendations to developers of systems that rely on

the difficulty of detecting embedded ciphertext as a part of the security of their systems.

This chapter also presented topics for future research.

Bibliography

- [And03] Andem, Vikram Reddy, *A Cryptanalysis of the Tiny Encryption Algorithm*, A Masters Thesis from The University of Alabama, 2003.
- [DaR99] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available at: www.csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf/.
- [DES06] DES Key Schedule web page: <http://eee.ucc.ie/staff/marnanel/Files/handoutee5251/lec1-deskey.pdf#search=%22DES%20key%20schedule%22>, accessed on 14 Oct 06.
- [Fei06] The Feistel Network web page: <http://cs-exhibitions.uni-klu.ac.at/index.php?id=261>, accessed on 14 Oct 06.
- [Fil02] Eric Filiol, *A New Statistical Testing for Symmetric Ciphers and Hash Functions*, Proceedings of the 4th International Conference on Information and Communications Security, Lecture Notes in Computer Science, Springer Verlag, 2002.
- [Gla06] B. Gladman's AES related home page: http://fp.gladman.plus.com/cryptography_technology/, accessed on 10 Oct 06.
- [KSW97] John Kelsey; Bruce Schneier and David Wagner, *Related-Key Cryptanalysis of 3-Way, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA*, Lecture Notes In Computer Science; Vol. 1334, Pages: 233 – 246, 1997.
- [MVV97] Menezes, A.J.; van Oorschot, P.C. & Vanstone, S.A., *Handbook Of Applied Cryptography*, CRC Press LLC, 1997.
- [NIS01] NIST Special Publication 800-22B, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, National Institute of Standards and Technology, 15 May 2001.
- [NIS01a] NIST Special Publication 800-38A, *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, National Institute of Standards and Technology, December 2001.
- [NIS01b] NIST Special Publication, *Federal Information Processing Standards Publication 197*, National Institute of Standards and Technology, November 26, 2001.

- [NIS06] Commerce Department Announces Winner of Global Information Security Competition, available at:
http://www.nist.gov/public_affairs/releases/g00-176.htm, accessed on 10 Oct 06.
- [NIS99] NIST Special Publication, *Federal Information Processing Standards Publication 46-3*, National Institute of Standards and Technology, October 25, 1999.
- [PGP05] PGP White Paper, The OpenPGP Standard & PGP Products, May 2005, PGP Corporation, available at:
<http://www.pgp.com/downloads/whitepapers/index.html#encryptionplatform>.
- [RSA78] R. Rivest, A. Shamir, L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, Vol. 21 (2), pp.120–126. 1978.
- [RSA93] PKCS #7, *Cryptographic Message Syntax Standard*, RSA Laboratories, Version 1.5, Revised November 1, 1993.
- [RSA99] PKCS #5, *Password-Based Encryption Standard*, RSA Laboratories, Version 2.0, March 1999.
- [Sha49] Shannon, Claude. *Communication Theory of Secrecy Systems*, *Bell System Technical Journal*, vol.28 (4), page 656–715, 1949.
- [Sot99] Juan Soto, *Randomness Testing of the AES Candidates Algorithms*, National Institute of Standards & Technology, 1999.
- [Sot99a] Juan Soto, *Statistical Testing of Random Number Generators*, National Institute of Standards & Technology, October 1999.
- [Squ06] Square block cipher home page:
<http://homes.esat.kuleuven.be/~rijmen/square/>, accessed on 10 Oct 06.
- [WhN94] D. Wheeler and R. Needham, *TEA, a Tiny Encryption Algorithm*, November 1994, available at:
<http://www.simonshpherd.supanet.com/tea.htm>.
- [Wik06] The image is public domain available at:
<http://en.wikipedia.org/wiki/Image:TEA.png>, accessed on 16 Oct 06.
- [Wik06a] The image is public domain available at:
http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation, accessed on 26 October 2006.

[XML02] XML Encryption Syntax and Processing W3C Recommendation 10 December 2002, found at: <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/>.

Vita

Captain Walter J. Hayden graduated from Marshalltown High School in January of 1993, a semester ahead of his peers. After graduation he enlisted in the Air Force and served as a Space System Operator. After seven years of enlisted service, Capt. Hayden was awarded the Airman Scholarship Commission Program scholarship. He used his scholarship at Iowa State University where he graduated with Distinction with a Bachelor of Science degree in Computer Science. In December of 2002, he received his commission through AFROTC Detachment 250.

His first assignment was at Hill AFB as a communication officer with the 367th Training Support Squadron. In October 2004, he was assigned to the 729th Air Support Squadron where he served as the Deputy Network Support Flight Commander. In August 2005, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to the Air Force Test Operational Test and Evaluation Center Detachment 3.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 22-03-2007		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) August 2005 – March 2007	
4. TITLE AND SUBTITLE Locating Encrypted Data Hidden Among Non-Encrypted Data Using Statistical Tools				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Hayden, Walter J., Captain, USAF				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/07-06	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Christopher E Reuter, Civ, AFRL/SNTA (937) 320-9068 x163, Christopher.Reuter2@wpafb.af.mil 2241 Avionics Circle WPAFB, OH 4433-7320				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This research tests the security of software protection techniques that use encryption to protect code segments containing critical algorithm implementation to prevent reverse engineering. Using the National Institute of Standards and Technology (NIST) Tests for Randomness encrypted regions hidden among non-encrypted bits of a binary executable file are located. The location of ciphertext from four encryption algorithms (AES, DES, RSA, and TEA) and three block sizes (10, 100, and 500 32-bit words) were tested during the development of the techniques described in this research. The test files were generated from the Win32 binary executable file of Adobe's Acrobat Reader version 7.0.9. The culmination of this effort developed a technique capable of locating 100% of the encryption regions with no false negative error and minimal false positive error with a 95% confidence. The encrypted region must be encrypted with a strong encryption algorithm whose ciphertext appears statistically random to the NIST Tests for Randomness, and the size of the encrypted region must be at least 100 32-bit words (3,200 bits).					
15. SUBJECT TERMS Encryption, Location, Statistical, Analysis					
16. SECURITY CLASSIFICATION OF: UNCLASSIFIED			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 92	19a. NAME OF RESPONSIBLE PERSON Rusty O. Baldwin, Civ, AFIT/ENG
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-6565, ext 4445 (Rusty.Baldwin@afit.edu)