Towards a Secure Communications Protocol Unifying Steganographic and Cryptographic Methods

Russell John Wyatt Ryan
An Extended Essay Submitted in Partial Fulfillment of
International Baccalaureate diploma requirements

North Central High School
Indianapolis, Indiana
United States of America

4000 Words

ACKNOWLEDGEMENTS

First of all I would like to thank my grandfather, Bill Wyatt for sparking my interest in computers when I was five years old. This has influenced my life drastically, and I would not have written this paper if it were not for him. Secondly I would like to thank my mentor, Rick Shadiow, for his support, teachings, and comments during the development of this paper and throughout the past four years of my life. Additionally, I would like to thank my parents and friends for their emotional support throughout the stressful creation of this paper.

ABSTRACT

In a global communication network such as the Internet, maintaining one's privacy is often a daunting task. Few are aware of tools designed to aid in this process, such as cryptography, an information scrambler, and steganography, an information hider. In this paper a hybrid protocol will be designed that uses both cryptographic and steganographic technology so that data may be transmitted securely across an insecure communications channel, such as the Internet. While the purpose of this paper is to design the SCRYPTO protocol, Appendix A lists the source code of a specific implementation of this protocol that I wrote. Please note that this code is the product of many hours of development, and that great efforts were made to eliminate all possibility of code plagiarism. That said, this code is provided as a supplement to the work presented here in order to show that a working implementation is possible. The focus of this paper, however, is the protocol itself, not the example code.

CONTENTS

Introduction

Communication is the transfer of information from one person to another. The desire for privacy is the desire to defend this information transfer from being disclosed to anyone other than the intended party. Communication has changed drastically in the past decade with the rise of a globally connected computer network. The open nature of the Internet is conducive to violation of privacy by willing individuals.

The relevance of privacy is directly related to the value of the information being communicated. In order to protect one's privacy, the method of communication must be altered to combat the open nature of the Internet. Tools, such as cryptography and steganography, are designed to alter reversibly information in a way such that only those intended to receive the message can return it to its original state. This is the equivalent of locking something valuable in a safe and shipping it to the recipient. The privacy factor is controlled by the strength of the safe used. Bruce Schneier, author of Applied Cryptography, writes, "There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files."[1] Although the latter kind is not always necessary, for the sake of privacy, businesses and individuals should be able to protect themselves. The aim of this extended essay is to create a communications protocol featuring cryptographic methods and steganographic[2] methods such that the individual strengths of the methods are maximized while the possibility of exploitation of their respective weaknesses is minimized.

Background Information / Definition of Terminology

To begin the design of this new protocol, the SCRYPTO[3] Protocol, relevant terminology must be identified. First of all, cryptography, literally 'secret-writing,' is the process of transforming information in a way that limits the number of people able to reverse this process and have access to the original information. Cryptography is an overt process in which anyone can see that a message or data exists, but they cannot understand it without reversing the encryption process. On the other hand, steganography, literally 'hidden-writing,' hides information inside other data such that only a limited number of people knowledgeable of the communication may know how to extract the hidden data or that hidden data is even present. The carrier data, or the data in which the sensitive data is hidden, will be called the host data. Steganography is a covert process in that someone who sees the transfer of the host data is not immediately aware of the existence of the hidden data.

[1] Bruce Schneier, Applied Cryptography, 2d ed, Edited by Phil Sutherland, (New York: John Wiley & Sons, Inc, 1996) xix.

[2] Steganography is the process of disguising data within other data

[3] SCRPYPTO : S (tegonographic) CRYPTography, a name I picked in development that I decided to keep for the protocol

A protocol is a strict set of rules and procedures which governs a series of actions.[4] A protocol involving a single person is generally very simple, but can still be formally defined. For example, a protocol can be defined that describes how to make a peanut-butter-and-jelly sandwich, because it can be represented as a series of steps in a process. However simplistic that seems, it is essential that every step of a protocol can be formally defined, for computers cannot perform anything other than definite and clearly defined steps. It is important to note that although the peanut-butter-and-jelly protocol describes one way of making that sandwich, it is not the only way to do so. Similarly, the protocol being developed will attempt to achieve privacy; however, there are many protocols whose goal is privacy, each with their own strengths and weaknesses.

A complex protocol is one whose constituent steps are often made up of less complex protocols. This is a relative term. As the peanut-butter-and-jelly protocol is complex relative to the protocols contained within it (i.e., spreading jelly on a piece of bread, retrieving an item from the refrigerator), so could the peanut-butter-and-jelly protocol be simple relative to a complex protocol, such as a “make lunch” protocol. The cryptographic and steganographic methods presented in this paper are all complex in nature; however, the goal of this exploration is to develop a complex protocol which takes advantage of the strengths afforded by both cryptographic and steganographic protocols in an effort to combine them.

When a protocol involves multiple people, the same rules apply; the scope only becomes larger. All persons involved must be completely clear about the steps they are

[4] Schneier 21.

required to perform and when.[5] Furthermore, all the people in the protocol must abide strictly to its rules. One who purposely does not follow the steps of a protocol in order to gain an advantage is called a "cheater."[6] In privacy protocols, cheating is a serious problem. A well-designed privacy protocol must attempt to deny all access to information in all situations except in that of strict adherence to protocol. When discussing privacy-based communications protocols, generic names for actors in the protocol must be agreed upon. Traditionally, communication is desired between Alice and Bob, two fictitious characters.[7] Eve is traditionally the name of the attacker.[8]

In terms of cryptographic protocols, the message or data that is being communicated will be referred to as the plaintext. Encryption of the plaintext is the process of transforming it into an unintelligible state, while the decryption process returns it to its original form.[9] These are commonly called encryption algorithms or ciphers. The encryption function takes plaintext and converts it into what is known as ciphertext.[10] The decryption function takes ciphertext and converts it back to plaintext.[11] The key to either function is restricted only by the specifications of the protocol being used. The most commonly limited parameter of the key is its length. The reason for this is because a key's length determines in part how long it will take to "bruteforce" an algorithm.[12] All cryptography is breakable in time, one simply must try every single key combination;

---

[5] Additionally, for a protocol to be considered complete, it must specifically provide a course of action for any situation. Schneier 21.

[6] Schneier 27.

[7] Originated from a simplification of Person A and Person B.

[8] There are many other possible characters involved in security protocols, but these are the three most basic.

[9] The encryption and decryption processes, based one or more numerical keys 'k', will be denoted mathematically as functions E_k and D_k, respectively.

[10] Mathematically: $E_k(P) = C$, where P = plaintext and C = ciphertext

[11] Mathematically: $D_k(C) = P$, where P = plaintext and C = ciphertext

[12] To bruteforce an algorithm is to find the plaintext by trying every possible key.

eventually the correct one will be discovered. The key length determines the number of possible keys that exist.[13] Key length varies from algorithm to algorithm and is measured in bits, the fundamental unit of memory for a computer.

In an ideal cryptographic algorithm, the security lies in the key.[14] The individual steps of the protocol need not be kept secret; in fact, they should be public. An algorithm whose security depends on the secrecy of its process, security through obscurity, loses the advantage of peer review. The cryptographic community relies heavily on peer review for improving algorithms. A cryptanalyst is one who analyzes ciphers for statistical patterns and weaknesses.[15] When weaknesses are discovered in an algorithm, then security is less dependent upon the key because a cryptanalyst can use weaknesses in the algorithm to discover information about the key or plaintext.

There are different modes in which ciphers operate. A cipher can run in either block or stream mode. Block mode is where the cipher operates on chunks of data at a time, and stream mode is where the cipher operates on each bit as it is input.[16] The former is good for sets of pre-assembled data. The latter is good for a situation that requires speed where data comes in a flow, and it is unknown how long it would take to fill the block size required for a block mode cipher.

Steganography differs from cryptography in many respects. In steganography the goal is not to protect the plaintext from unauthorized reading, but to prevent Eve from

---

[13] The number of possible key combinations in a key n bits long is 2^n. Thus if one tests one key per second, it will take at most 2^n seconds to find the key. Triple-DES has an effective key length of 168 bits, thus it will take approximately $1.18 * 10^{43}$ years to crack, this is roughly 10 times the estimated age of the universe if the universe is 10 billion years old. Of course, in reality, the rate of key tests is greatly increased, but the time required for simple bruteforcing is still controlled by an exponential function's growth versus a linear function's growth.

[14] Eric Cole, Hiding In Plain Sight: Steganography and the Art of Covert Communication, Edited by Carol Long (Indianapolis: Wiley Publishing, Inc., 2003) 26-27.

[15] Schneier 1.

[16] Schneier 189.

even knowing that there is sensitive data being communicated. Steganography is thus a covert protocol, as opposed to an overt protocol.[17] To accomplish this goal, the plaintext is hidden within another set of data, known as the host data or host file.

Goals of the Protocol

In order to judge the protocol being designed, some basic goals of a security protocol must be outlined, and the relevant ones must be selected for use. First and foremost, a protocol must be able to secure the data which it transmits. This quality, known as confidentiality, is most often controlled by a system such as steganography or cryptography.[18] A protocol needs some method of verification that a person is who they say they are. This is called authentication, and is often verified using digital signatures, passwords, or biometric techniques, such as fingerprint readers or iris-scanners.[19]

Another important characteristic of a security algorithm is integrity. In this case, integrity is the ability of a protocol to detect, prevent, or recover from insertions or substitutions that may occur while the message is in transit.[20] These alterations are often caused by communication errors, but are sometimes intentionally perpetrated by attackers trying to cheat the protocol.

Non-repudiation, another characteristic of a good communication protocols, prevents participants in the protocol from later denying that they transmitted data after having transmitted it.[21] Often the solution to this problem is using time-stamped digital signatures, biometrics, or physical key-ring security cards. Finally, a very commonly

[17] Cole 52.
[18] Cole 19-20.
[19] Schneier 2.
[20] Cole 20.
[21] This is an important and difficult problem on the Internet because one can simply claim that his or her computer had been compromised and that someone else sent it. Schneier 2.

evaluated characteristic of a security protocol is its "availability" or its resistance to denial of service or incorrect input.[22] Cheaters often hurt protocols by providing input of types that protocols are not designed to work with or interpret.

Specific goals of the protocol should be specified to guide the selection of the individual protocols used. The protocol must be very easy to use with as little user interaction as possible, or else many will not use it. The average computer user will feel overwhelmed if a protocol requires too much effort or is very complicated. The ultimate goal is to achieve a private connection through which a user may send messages or data without any extra work. To this end, there should be no key management required on the user's part; the two computers should agree upon a key using an appropriate key negotiation protocol. The message transmitted should not be readable by anyone who is "listening" in on the connection. Ideally, the transmission of the message should not be easily discoverable in the first place. The protocol should also be able to detect or prevent the message from being modified in transit. Availability is not important in this instance because Alice and Bob supply all input, and both intend to communicate with each other. Thus they will not attempt to cheat the protocol. Evaluators such as authentication and non-repudiation are less important in this case because this protocol is meant only to communicate a message securely. The goal is to keep intruders out, not to police the participants in the protocol. If such protection is required, then the specific implementation of the protocol could be easily expanded to include a mandatory digital

[22] Denial of Service is an attack which aims not to break an algorithm but to disable it by starving the host computer of its resources through a flood of input. Cole 21-22.

signature on each message.[23] In short, the protocol will be evaluated according to how well it achieves the goals outlined above.

Cryptographic Elements

There are three main types of cryptographic algorithms. The first type is symmetric algorithms, where the same key is used in the encryption process as is used in the decryption process.[24] This is most commonly used where it is possible to generate a common key between the target and recipient. Asymmetric algorithms are the second type where a different key is used in the encryption process than is used in the decryption process.[25] The most common implementation of this second type is the use of a private and public key pair. One distributes the public key and anyone who wishes to privately communicate may encrypt his or her messages with the recipient's public key. Only the recipient's private key can decrypt the message.[26]

Hash functions are the third main type of algorithm. These are designed so that encryption is simple, but decryption is computationally infeasible.[27] This is used often with password authentication; a remote server need not keep a list of all the users' passwords, only a list of hashes. When someone tries to authenticate, the server compares the stored hash with the hash of the entered password. If the two match, then the user is authenticated.

---

[23] A digital signature is commonly a public and private key-pair which is akin to a digital fingerprint that is used to sign documents. A digital signature is valid as long as the private key remains unknown to everyone but the user. This will be discussed later under asymmetric encryption.

[24] Symmetric algorithms have the following mathematical properties: $E_k(P) = C$ and $D_k(C) = P$ where P is the plaintext and C is the ciphertext.

[25] Asymmetric algorithms have the following mathematical properties: $E_{k1}(P) = C$ and $D_{k2}(C) = P$ where P stands for plaintext, C stands for ciphertext, and k1 and k2 are two different keys.

[26] Cole 38.

[27] Mathematically $E_k(P) = C$, but $D_k(C)$

The simplest form of electronic cryptography is Exclusive-OR encryption, a symmetric algorithm. Ironically, this is the only truly secure (in the ideal sense) form of cryptography available.[28] Exclusive-Or, or XOR, is a bit operation where two bits are compared, and a result is given.[29] A plaintext with a length of n bits, requires a key n bits long to be truly secure. The key and plaintext are "XORed" together bit by bit to form the ciphertext. This ciphertext is completely secure.[30] This means that a cryptanalyst will be unable to find any true statistical relation between the bits because they are completely independent of each other.[31] The cryptanalyst must have the key to return the ciphertext to plaintext. There is no other way to learn extra information without making assumptions. This was the encryption scheme used by the infamous "red telephone" link between Washington, D.C. and Moscow during the Cold War.[32]

To facilitate the user not having to deal with keys, asymmetric encryption should not be used. This is because asymmetric encryption has little use without the involvement of a user-managed key pair. Hash encryption is not suitable either because decryption is infeasible. Thus, symmetric encryption is the type of encryption that should be used. The next problem is how the sender and recipient should decide on a key to use. Since the goal is for the user not to need to manage any keys, a key exchange algorithm should be used. The irony of symmetric encryption has always been that if the sender can inform the recipient of the key without anyone else knowing, then why doesn't the sender merely

[28] Provided that the key is equal in length to the plaintext.

[29] XOR is defined such that: A XOR B = C. Where A, B, and C are individual bits. If A and B are both 1 or both 0, then C is 0, if they are not equal, then C is 1.

[30] This type of encryption is known as a one-time pad because once a key is used, it cannot be used again. One-time pads are perfectly secure. Francis Litterio, "Why Are One Time Pads Perfectly Secure?" Cryptography, The WWW Virtual Library, <http://www.vlib.org/>.

[31] This assumes that Eve has no idea what type of information is being transferred. If she knows that it is in English, then she may use statistical occurrences of letters in English mixed with relative occurrences of bits in the ciphertext; however, this is an assumption she must make.

[32] Schneier 16-17

give the recipient the secret message at that point? Furthermore, if the only communication channel between the two is insecure, then how can they agree on a key? A key negotiation algorithm can solve these problems. One of the most prominent of these available today is the Diffie-Hellman key negotiation algorithm, which because there are few differences between key negotiation algorithms, will be used in this protocol due to its simplicity.[33]

As for the encryption algorithm, symmetric encryption narrows the choices considerably. There are many symmetric algorithms, the best of which is not a clear choice. The most commonly used one today is the block-mode Triple-DES algorithm,[34] based on the IBM-developed, NSA-approved[35] Data Encryption Standard protocol developed in the 1970s.[36] The more recent IDEA[37] cipher is another promising candidate that has stood the test of time. However, Bruce Schneier, an expert cryptographer, recently disparaged it for various flaws in its design in an interview posted on SlashDot.org, an online computer news forum,

> "If I needed a secure symmetric algorithm for a design, and performance were not an issue, I would choose triple-DES. No other algorithm has been as well-studied, so nothing can compare in confidence. […] I don't recommend IDEA anymore […] it isn't very fast; […] IDEA is patented, and the terms change regularly. Also,

[33] In this system, the sender and recipient computers each pick two private numbers, and exchange a total of four numbers. From these 5 numbers, each computer can generate an identical number, k, which will be the key. An attacker cannot derive the key with the four numbers communicated; he has to have one of the two private numbers. Schneier 513.

[34] The Triple-DES algorithm is currently one of the four official encryption standards of the United States government. The Advanced Encryption Standard, however, is generally preferred to Triple-DES. United States, National Institute of Standards and Technology, Computer Security Resource Center, Federal Information Processing Standards Publication 46-3, October 25, 1999, 2.

[35] The National Security Agency of the United States of America is in charge of all the government based cryptography research and operations.

[36] The Advanced Encryption Standard (AES) replaced Triple-DES as the National Standard for the United States Federal Government in 2001; however, AES is so new that the chance of weaknesses being discovered soon is still high. It is therefore not going to be considered for this protocol. United States, National Institute of Standards and Technology, Computer Security Resource Center, Federal Information Processing Standards Publication 197, November 26, 2001, 5.

[37] International Data Encryption Algorithm, developed in 1990

> attacks against IDEA have steadily eaten away at the security margin […] There are still no attacks against the full eight-round cipher, and there is no reason to believe that any are possible. Still, since there are algorithms with much better performance, it seems improper to suggest IDEA."[38]

In this protocol, speed is not critical, and it is important that the cryptographic algorithm remain secure. Therefore, Triple-DES will be used over the IDEA cipher in the SCRYPTO protocol.

One problem with the original DES is that it is easily breakable using a dedicated DES cracking machine. Estimates in 1993 indicated that a $1,000,000 machine could crack DES in minutes.[39] For this reason, Triple-DES was created. This is simply DES performed three times with three different keys. The 3 56-bit keys are strung together to create one larger key.[40] Little is known about what the U.S. government can do at this point, but it is naïve to assume that the NSA has not built a machine to crack DES. Whether or not they can easily crack Triple-DES is unknown.

Steganographic Elements

There are three main techniques involved in steganographic algorithms: Insertion techniques, substitution techniques, and language-generation techniques.[41] Insertion-based steganography involves inserting the plaintext into a file in a place that is not easily noticeable; a place that is either no longer used or will never be shown directly to a user. For example, in a Microsoft Word document, plaintext can be inserted into the document

---

[38] Bruce Schneier, "Crypto Guru Bruce Schneier Answers" interview by the users of SlashDot.org, 29 Oct. 1999, http://slashdot.org/interviews/99/10/29/0832246.shtml.

[39] Schneier also believes that the NSA can crack DES in 3 to 15 minutes. Schneier 300.

[40] Triple-DES is a 56-bit block algorithm, and its key is 192 bits long. Although the key is 192 bits long, only 168 bits of these are used as input to the algorithm, the remaining 24 bits are used for integrity checking.

[41] Cole 108-109.

where placeholders are created for its “undo” feature.[42] A user will never see data stored in these sections, which makes them ideal for an insertion technique.

Substitution-based steganography is where the data in the host file is modified slightly according to a pattern to represent the plaintext. This is often accomplished by modifying small bits of data which are the least significant in a file. One such method involves hiding data within image files. Pictures on computers are made up of a grid of dots of color called pixels. Each pixel represents color with a number, generally anywhere from 8 to 32 bits in length. The least significant bit of each pixel is the one which changes the color’s number the least, the binary equivalent of the decimal 1’s place. Data is hidden by changing the least significant bit in each pixel of an image based on each corresponding bit of plaintext.[43] Because only the least significant bit of each color is being changed, the picture will rarely look different. To hide a plaintext message of length n bits, a picture with n pixels is needed.[44]

The major caveat of substitution-based steganography is that is that it often requires a very large amount of host data to hide a relatively smaller amount of plaintext. To resolve this problem, a generation-based technique can be used to generate a host file of the size required to hide the plaintext. This could be a sine wave for an audio file into which a secret message could be stored, a fractal image, or it could be a large enough chunk of randomly generated text, modeled to English grammar rules in order to fool computer scanners.[45]

[42] Cole 111-112.

[43] Cole 114.

[44] Another method similar to this involves editing a document so that every sentence ends in either a period or an exclamation point. A period represents 0 and an exclamation point represents 1. To hide some data that is n bits long, a document with n sentences is required.

[45] Cole 112-113, 135.

One substitution technique which does not suffer as badly from the requirement of a larger host file is one which hides data within the building blocks of the Internet itself. The most basic Internet protocols are defined within the TCP/IP protocol suite, and they are the driving force behind the Internet today. Parts of this suite, particularly the TCP,[46] are no longer used as they were intended. Parts of the network headers, particularly the sequence and IP identification numbers, are no longer actively checked and can be used to hide data.[47] This technique becomes independent of the size constraints on the host file because the host data being transmitted can be generated randomly in real-time.

To prevent Eve from knowing that an encrypted message is even being transmitted, a steganographic algorithm must be selected. The medium or host data is what differentiates most all of the steganographic algorithms. An image can be used to hide data, although this requires that the user pick out an image and give any eavesdroppers good reason to believe that Alice and Bob would be sending images to each other. This can involve weeks, even months, of previous communication in which pictures of classic cars, for example, are exchanged, until the day that one of the pictures slips through undetected with hidden information in it. This sort of user interaction is not permitted as per the goals of this protocol.

Ideally, the steganographic algorithm used must be relatively unaffected by size of host data required because for transmitting larger amounts of data, the host message could potentially become very large. TCP/IP header encoding is unique in that as the message size increases, only the volume of packets sent need be increased. This is good

[46] Transmission Control Protocol, designed the University of Southern California is a base protocol used by the majority of communications between computers on the Internet. Jon Postel, "*RFC-793: Transmission control protocol*," Request for Comments, Information Sciences Institute, University of Southern California, Sept. 1981.

[47] Cole 170.

in that it reduces the amount of host data required because small amounts of randomly generated data can be sent in each packet. However, it is bad because anyone monitoring the volume of packets passing through the network would notice a large number coming from the sender's machine. As with any steganographic algorithm, if Eve is expecting steganography to be used then it is not difficult for her to extract the message. But, to the unsuspecting attacker, most steganographic attacks work beautifully. TCP/IP header encoding is a very recent development, and the sheer volume of traffic flowing through most networks makes it infeasible for steganographically-encoded data to be discovered by network-wide monitoring. Therefore, an attacker must be specifically scanning the sender for the TCP/IP header encoding technique. This is about as good as steganography can get; forcing the attacker to make specifically targeted attacks as opposed to general scans.[48] Because of the lack of size restrictions on TCP/IP header encoding, it will be used as the steganographic element in the SCRYPTO protocol.

Integrity Checking

At this point, the protocol is nearly complete. It contains both the steganographic and cryptographic elements desired and has a method of key exchange; however, thus far there is no integrity checking. A major problem with block mode ciphers is that each block on its own can be encrypted and decrypted independent of all other blocks. If Eve has control of the communication channel, she can insert a block of her own into the transmission or rearrange the blocks sent by Alice or Bob.[49] A group of techniques known as block chaining modes add "feedback" to block ciphers in order to resolve this

[48] Cole 55.

[49] This sort of attack is commonly known as "block replay" and can be used to take advantage of fixed block size communications, such as bank transactions. Schneier 191-193.

issue.[50] Each block is normally just encrypted with the key; however, in a block-chaining mode, the ciphertext of the current block is also modified so that it depends on every previous block, as well as the key. Therefore, an insertion or rearrangement of the blocks will most likely skew the plaintext considerably so that Bob or Alice realizes something is wrong. The most common types of block chaining are CBC, CFB, and OFB.[51] CFB and OFB are more suited for making a block cipher work as a stream cipher (which in this situation is not necessary because source data is readily available), and CBC has the strictest replay protection.[52] Therefore, CBC will be used.

The integrity checking, however, is not truly complete because if the message is changed the decryption process will simply not decipher the correct plaintext, it will not know that something is amiss. With the addition of a checksum, or CRC check[53], the algorithm will be able to verify that the decrypted plaintext is the same as the one originally sent by re-computing the checksum of the decrypted ciphertext and then comparing the two. A simpler way of using a checksum is to modify the CBC mode used slightly into what is called CBCC.[54] Using CBCC, the last block of plaintext should be

[50] Schneier 193

[51] CBC: Cipher Block Chaining: $C_i = E_k(P_i \text{ XOR } C_{i-1})$, $P_i = C_{i-1} \text{ XOR } D_k(C_i)$
CFB: Cipher Feedback Mode: $C_i = P_i \text{ XOR } E_k(C_{i-1})$, $P_i = C_i \text{ XOR } D_k(C_{i-1})$
OFB: Output Feedback Mode: $C_i = P_i \text{ XOR } S_i$, $S_i = E_k(S_{i-1}) : P_i = C_i \text{ XOR } S_i$, $S_i = E_k(S_{i-1})$
Schneier 192-205.

[52] Schneier 192-205.

[53] A Cyclic Redundancy Check (CRC) check produces a number called a checksum that is computed using a complex algorithm loosely based on the summation of every byte of data in the plaintext.

[54] Cipher Block Chaining with Checksum: In CBCC, each block of plaintext is "XORed" with each previous block of plaintext before being "XORed" with the previous ciphertext block. Using this system, the final block of decrypted plaintext will be affected if any of the original ciphertext has changed. Therefore, if the final block of plaintext is a constant, then it is trivial to verify that the ciphertext has been compromised. Schneier 207-208.

set to some constant, for example “SCRYPTO”.[55] If the last block of decrypted plaintext is not equal to this, then the protocol knows that the message has been violated.

Implementation and Conclusion

Since the steps of the protocol have been decided, an implementation of this protocol can be written within a program. To demonstrate the protocol in action I have written a basic file sender and receiver program that implement the protocol.[56] The protocol now satisfies the security descriptors that were originally selected: integrity and confidentiality. It also meets the requirement of little user activity, and all of the specifications required of a general protocol.

In creating this protocol, my goal was to design a hybrid protocol, combining the advantages of both steganography and cryptography. Having achieved that goal, the question remains: Is it secure? This can only be answered with time and rigorous testing by the security community. The upsetting truth about the Internet today is that the amount of secured and hidden traffic is miniscule compared to the volume of insecure traffic. Many do not know or care that their personal information is being broadcast to anyone who wants it. This is particularly relevant in this time of rampant identity theft, wire fraud, and anti-privacy laws such as the PATRIOT Act in the United States.[57] In other countries, individual privacy is often not even considered a right. Using a protocol such as the one I have developed, individuals can protect themselves from privacy violation on the Internet.

---

[55] The block size of Triple-DES is 64 bits, which is 8 ASCII characters long. For the 8th character, something internal could be selected, or in the case of the implementation of the SCRYPTO protocol provided in Appendix A, the 8th character is the number of bytes padded onto the original plaintext

[56] See Appendix A for code and documentation.

[57] Under the current law, an Internet Service Provider is forbidden to notify its clients that the United States government has ordered a tap of their connection using a Carnivore device. Thus, one’s communications could be under direct surveillance without warning.

APPENDIX A: EXAMPLE SCRYPTO IMPLEMENTATION

The following code is a basic implementation of the SCRYPTO protocol developed in this paper, designed to act as either the sender or receiver of steganographically hidden and cryptographically encoded data.

The code was written for the Linux Operating System in the C++ Programming Language. It compiles successfully using the GNU C++ Compiler. Sample output is provided following the source code.

The program presented here was written completely by myself using these reference documents only: the Linux header file reference, the Federal Information Processeing Standards Publication 46-3 definition of DES and 3DES, the National Institute of Standards and Technology Special Publication 800-17 test vectors for verifying my DES implementation, Eric Cole's description of Diffie-Hellman Key Negotiation in Hiding in Plain Sight, and Craig H. Rowland's "Covert TCP" white paper from 1996.

Great care was taken to prevent any possibility of the code here being taken from someone else's implementation of its sub protocols. I intended this program to be written completely by me using reference documents, so I made certain that I did not look at a single line of another person's code while writing this program, not even for guidance.

The following files are included following this page:

scrypto.cpp	1 page, Page	18
scrypto.h	1 page, Page	19
sender.cpp	4 pages, Pages	20-23
receiver.cpp	4 pages, Pages	24-27
DES.h	6 pages, Pages	28-33
DES.cpp	12 pages, Pages	34-45
diffie.h	1 page, Page	46
diffie.cpp	4 pages, Pages	47-50
rawsocket.h	2 page, Page	51-52
rawsocket.cpp	6 pages, Pages	53-58
error.h	1 page, Page	59
error.cpp	1 page, Page	60

The following files show example input and output for the sender and receiver:

sendlog.txt	1 page, Page 61
send.txt	1 page, Page 62
receivelog.txt	1 page, Page 63
receive.txt	1 page, Page 64

scrypto.cpp

```
// This is the entry point of the SCRYPTO demonstration
//   designed to show an implementation of the stego/crypto protocol
//   developed in this paper.
//
// This file houses the main() procedure and branches either to
//   receiver_main() or sender_main() based on the role chosen
//   by the user.
//

#include "scrypto.h"

int main(int argc, char* argv[])
{
  cout << "Scrypto Protocol Tester by Russell Ryan : Completed January
2005" << endl;

  try { //setup our error detection system, one big try/catch
    while(1) {
      cout << "Function as a sender or receiver? (S/R):";
      char ch;
      cin.read(&ch, 1);

      if(tolower(ch) == 'r') {
        receiver_main();
        break;
      }
      else if(tolower(ch) == 's') {
        sender_main();
        break;
      }

      cout << "\r\nPlease enter R for receiver or S for sender." <<
endl;
    }
  }
  catch(cError &err) {
    err.print();
    return 0;
  }

  return 0;
}
```

scrypto.h

```
//  main include file: quickly includes all relevant headers
//  and defines relevant macros

#define DEBUG false //if true turns on debug printing

//OS and C library specific includes, mostly linux stuff
//  General includes
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//  Linux Includes
#include <endian.h>
#include <linux/types.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
//  Linux Socket Includes
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <linux/ip.h>
#include <linux/tcp.h>

//Use the STL for things like strings
using namespace std;

//name our own bitlength types for little endian processors
#define __int64 long long // long long == 64 bits
#define __int32 long      //      long == 32bits
#define __int16 short     //     short == 16bits
#define __int8 char       //      char == 8bits

//Error Handlers
#include "error.h"

//Connection Portion
#include "rawsocket.h" //TCP Header Encoding

//Encryption Portion
#include "des.h"    //DES and 3DES Cryptography
#include "diffie.h" //Diffie Hellman Key Negotiation

//Helper Functions
int receiver_main(); //Implements a SCRYPTO Receiver
int sender_main(); //Implements a SCRYPTO Sender
```

sender.cpp

```
//  houses the main entry point for the sender
//  arranges a transfer, computes keys, encrypts, and sends a file
//  using the SCRYPTO Protocol by Russell Ryan

#include "scrypto.h"

int sender_main() {
  cRawSocket *rs = new cRawSocket(false);
  cDES *crypt = new cDES(CHAINMODE_CHECKSUMCBC);
  string filename, host;
  bool prompt;
  int port,i,percent;
  FILE *fp;
  char *fileData;
  int fileLength;

  prompt = false;
  while(1) {
    if(prompt)
      cout << "Enter address to connect to:";
    getline(cin, host);

    if(host.length()<=0) {
      prompt = true;
      continue;
    }

    cout << "Enter port:";
    cin >> port;

    if(port<=0 || port>65536) {
      prompt = true;
      continue;
    }

    if(!rs->Connect(host, port)) {
      cout << "Unable to connect." << endl;
      prompt = true;
      continue;
    }

    break;
  }

  prompt = false;
  while(1) {
    if(prompt)
      cout << "Enter filename to send:";
    getline(cin, filename);

    if(filename.length()<=0) {
      prompt = true;
      continue;
    }
    fp = fopen(filename.c_str(), "rb"); //binary mode so it doesn't
smash the \r\n
    if(!fp) {
      prompt = true;
      cout << "Invalid Filename" << endl;
      continue;
    }
    break;
  }

  //read data from file
  fseek(fp, 0, SEEK_END);
  fileLength = ftell(fp);
  fileData = new char[fileLength];

  if(!fileData)
    throw cError("sender_main() : Insufficient Memory");

  fseek(fp, 0, SEEK_SET);
  fread(fileData, 1, fileLength, fp);

  char *ct = NULL;
  int ctlength;

  //This id sent to verify the start of a transmission
  //this would NOT be sent in a production grade
  //implementation of this protocol because it is
  //identifying information that packet filters
  //could monitor for
  rs->Send("SCRY", 4, 4); //"SCRY" w/ IPID 4 signifies start

  cout << "Begin negotiating 196-bit key..." << endl;

  //Perform the key negotiation, act as Alice
  des_key196 key;
  key.int_data[0] = diffieAlice(rs);
  sleep(3); //sleep to make sure our RNG seed changes
  key.int_data[0] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);
  key.int_data[1] = diffieAlice(rs);
  sleep(3);
  key.int_data[1] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);
  key.int_data[2] = diffieAlice(rs);
  sleep(3);
  key.int_data[2] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);

  //do diffie hellman for the IV,
  // it's harmless to transmit the IV plaintext, but there's no reason
not to
  // do it securely
  cout << "Begin computing the initialization vector..." << endl;

  //Agree on an IV using Diffie-Hellman, act as Alice
  des_data196 iv;
  iv.int_data[0] = diffieAlice(rs);
  sleep(3); //sleep to make sure our RNG seed changes
  iv.int_data[0] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);
  iv.int_data[1] = diffieAlice(rs);
  sleep(3);
  iv.int_data[1] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);
  iv.int_data[2] = diffieAlice(rs);
  sleep(3);
  iv.int_data[2] |= ((__int64)(diffieAlice(rs))) << 32;
  sleep(3);
  crypt->setIV(iv);

  ctlength = crypt->encrypt_3des(fileData, fileLength, &ct, key);

  sleep(2);
  //Send the ciphertext length
  rs->Send((char*)&ctlength, 4);

  cout << "Sending " << ctlength << " bytes:..." << endl;

  sleep(4); //make SURE they are ready

  percent = 25;
  //since encrypt_3des returns a buffer with length always divisible by
8,
  // it is safe to assume that the ct length is divisible by 2
  int packets = ctlength/2;
  //it is also safe to assume that for every packet i, ct[i], ct[i+1]
is valid

  for(i=0; i<packets; i++) {

    rs->Send("", 0, *((unsigned short*)(ct+i*2)));

    //Print Progress
    if(percent == 25 && (i*200)/ctlength >= 25) {
      cout << "25%...";
      percent = 50;
    }
    else if(percent == 50 && (i*200)/ctlength >= 50) {
      cout << "50%...";
      percent = 75;
    }
    else if(percent == 75 && (i*200)/ctlength >= 75) {
      cout << "75%...";
      percent = 100;
    }
    else if(percent == 100 && i*2+2 == ctlength)
      cout << "100% Done." << endl;

    //we'll flood the connection layer
    //  if we blast it all through at once
    sleep(2);
  }

  cout << "File transfer complete... Quitting..." << endl;
  //Decryption Test - if we can't decrypt it then they sure can't
  if(DEBUG) {
    ctlength = crypt->decrypt_3des(ct, ctlength, &ct, key);
    for(i=0; i<ctlength; i++)
      cout << ct[i];
    cout << endl;
  }

  return 0;
}
```

receiver.cpp

```
//  houses the main entry point for the receiver
//  arranges a transfer, computes keys, decypts, and saves a file
//  sent using the SCRYPTO Protocol by Russell Ryan

#include "scrypto.h"

int receiver_main() {
  cRawSocket *rs = new cRawSocket(true);
  cDES *crypt = new cDES(CHAINMODE_CHECKSUMCBC);
  int port;
  int i, j, percent;
  unsigned short ipid;

  //general purpose input pointer and length
  char *input;
  int length;

  while(1) {
    cout << "Enter a port to listen on:";
    cin >> port;

    if(port<=0 || port>65536)
      continue;

    if(!rs->Bind(port)) {
      cout << "Unable to listen on port " << port << "." << endl;
      continue;
    }
    break;
  }

  char* temp = new char[128];
  int tlen = 128; //arbitrary, enough to hold our start tag
  while(1) {
      rs->Recv(&temp, &tlen, &ipid);

      if(tlen == 4 && temp[0] == 'S' && temp[1] == 'C' && temp[2] ==
'R' && temp[3] == 'Y' && ipid == 4)
        break; //begin the process

      memset(temp, 0, 128);
      tlen = 128;
  }

  cout << "Connected... setting up transfer." << endl;
  cout << "Negotiating 196-bit key..." << endl;

  //Perform the key negotiation, act as Bob
  //sleep between exchanges to make sure our RNG seed changes
  des_key196 key;
  key.int_data[0] = diffieBob(rs);
  sleep(3);
  key.int_data[0] |= ((__int64)(diffieBob(rs))) << 32;
  sleep(3);
  key.int_data[1] = diffieBob(rs);
```

```
sleep(3);
key.int_data[1] |= ((__int64)(diffieBob(rs))) << 32;
sleep(3);
key.int_data[2] = diffieBob(rs);
sleep(3);
key.int_data[2] |= ((__int64)(diffieBob(rs))) << 32;

cout << "Begin Computing Initialization Vector..." << endl;
des_data196 iv;
iv.int_data[0] = diffieBob(rs);
sleep(3);
iv.int_data[0] |= ((__int64)(diffieBob(rs))) << 32;
sleep(3);
iv.int_data[1] = diffieBob(rs);
sleep(3);
iv.int_data[1] |= ((__int64)(diffieBob(rs))) << 32;
sleep(3);
iv.int_data[2] = diffieBob(rs);
sleep(3);
iv.int_data[2] |= ((__int64)(diffieBob(rs))) << 32;
sleep(3);
crypt->setIV(iv);

char *ct = NULL;
int ctlength;

//Receive length of ciphertext
i=0;
length = 4;
while(i<4) {
    if(!rs->Recv(&input, &length))
      break;

    for(j=0; j<length;j++) {
      ((char*)&ctlength)[i] = input[j];
      i++;
    }
    length = 3-i;
    delete input;
}

ct = new char[ctlength];
if(!ct)
  throw cError("receiver_main() - Insufficient Memory");

cout << "Receiving " << ctlength << " bytes:..." << endl;

percent=25;
i=0;
length = 0;
char *ipidptr = (char*)&ipid;
while(i < ctlength) {
    if(!rs->Recv(&input, &length, &ipid ))
      break;

    length = 2; //ipid is a short
    for(j=0; j<length; j++) {
        ct[i] = ipidptr[j];
        i++;
      }

      //Output Progress
      if(percent == 25 && (i*100)/ctlength >= 25) {
        cout << "25%...";
        percent = 50;
      }
      else if(percent == 50 && (i*100)/ctlength >= 50) {
        cout << "50%...";
        percent = 75;
      }
      else if(percent == 75 && (i*100)/ctlength >= 75) {
        cout << "75%...";
        percent = 100;
      }
      else if(percent == 100 && i == ctlength)
        cout << "100% Done.";
  }
  cout << endl;

  //Perform Decryption
  char *pt;
  int ptsize;
  ptsize = crypt->decrypt_3des(ct, ctlength, &pt, key);

  //Now save result to file if everything went ok
  bool prompt = false;
  string filename;
  FILE *fp;
  while(1) {
    if(prompt)
      cout << "Enter filename for received data:";
    getline(cin, filename);

    if(filename.length()<=0) {
      prompt = true;
      continue;
    }

    fp = fopen(filename.c_str(), "wb"); //binary mode so it doesn't
smash the \r\n
    if(!fp) {
      prompt = true;
      cout << "Invalid Filename" << endl;
      continue;
    }
    break;
  }

  fwrite(pt, 1, ptsize, fp);

  cout << "Contents of " << filename << ":" << endl;
  for(i=0;i<ptsize;i++) {
    cout << pt[i];
  }
  cout << endl << "EOF" << endl << "File transfer complete...
Quitting..." << endl;

  return 0;
}
```

DES.h

```
#ifndef _DES_H_
#define _DES_H_

//  houses a class definition for an implementation of the
//  triple DES encryption protocol as per the NIST specifications

//mode defines
#define MODE_EDES 1 //encrypt DES
#define MODE_DDES 2 //decrypt DES
#define MODE_E3DES 3 //encrypt 3DES
#define MODE_D3DES 4 //decrypt 3DES

#define CHAINMODE_ECB 0 // ECB Mode
#define CHAINMODE_CBC 1 // CBC Mode
#define CHAINMODE_CHECKSUMCBC 2 // CBC Checksum Mode

//Bit Manipulation Macros: save lots of typing later
// General Idea Behind this:
//  to test a bit: sd & (1 << bit) will equal true if the bit is set
//  to set a bit: sd |= (1 << bit), will set the bit regardless of its
current value
//  to clear a bit: sd &= ~(1 << bit), will set the bit to 0
//  to toggle a bit: sd ^= (1 << bit) : XOR flips the bit to its
opposite if they are the same

//64 bit operators
#define IS_SET_64(sd, bit) (sd & ((__int64)(1)<<(bit)))
#define FLAG_BIT_64(sd, bit) sd |= ((__int64)(1)<<(bit))
#define CLEAR_BIT_64(sd, bit) sd &= ~((__int64)(1)<<(bit))
#define TOGGLE_BIT_64(sd, bit) sd ^= ((__int64)(1)<<(bit))
#define SET_BIT_64(sd, bit, val) if(val != 0) { FLAG_BIT_64(sd, bit); }
else { CLEAR_BIT_64(sd,bit); }

//56 bit operators
#define IS_SET_56(sd, bit) (sd & ((__int64)(1)<<(bit)))
#define FLAG_BIT_56(sd, bit) sd |= ((__int64)(1)<<(bit))
#define CLEAR_BIT_56(sd, bit) sd &= ~((__int64)(1)<<(bit))
#define TOGGLE_BIT_56(sd, bit) sd ^= ((__int64)(1)<<(bit))
#define SET_BIT_56(sd, bit, val) if(val != 0) { FLAG_BIT_56(sd, bit); }
else { CLEAR_BIT_56(sd,bit); }

//48 bit operators
#define IS_SET_48(sd, bit) (sd & ((__int64)(1)<<(bit)))
#define FLAG_BIT_48(sd, bit) sd |= ((__int64)(1)<<(bit))
#define CLEAR_BIT_48(sd, bit) sd &= ~((__int64)(1)<<(bit))
#define TOGGLE_BIT_48(sd, bit) sd ^= ((__int64)(1)<<(bit))
#define SET_BIT_48(sd, bit, val) if(val != 0) { FLAG_BIT_48(sd, bit); }
else { CLEAR_BIT_48(sd,bit); }

//32 bit operators
#define IS_SET_32(sd, bit) (sd & (__int32(1)<<(bit)))
#define FLAG_BIT_32(sd, bit) sd |= (__int32(1)<<(bit))
#define CLEAR_BIT_32(sd, bit) sd &= ~(__int32(1)<<(bit))
#define TOGGLE_BIT_32(sd, bit) sd ^= (__int32(1)<<(bit))
```

```
#define SET_BIT_32(sd, bit, val) if(val != 0) { FLAG_BIT_32(sd, bit); }
else { CLEAR_BIT_32(sd,bit); }

//28 bit operators
#define IS_SET_28(sd, bit) (sd & (1<<(bit)))
#define FLAG_BIT_28(sd, bit) sd |= (1<<(bit))
#define CLEAR_BIT_28(sd, bit) sd &= ~(1<<(bit))
#define TOGGLE_BIT_28(sd, bit) sd ^= (1<<(bit))
#define SET_BIT_28(sd, bit, val) if(val != 0) { FLAG_BIT_28(sd, bit); }
else { CLEAR_BIT_28(sd,bit); }

//8 bit operators
#define IS_SET_8(sd, bit) (sd & (__int8(1)<<(bit)))
#define FLAG_BIT_8(sd, bit) (sd |= (__int8(1)<<(bit)))
#define CLEAR_BIT_8(sd, bit) (sd &= ~(__int8(1)<<(bit)))
#define TOGGLE_BIT_8(sd, bit) (sd ^= (__int8(1)<<(bit)))
#define SET_BIT_8(sd, bit, val) if(val != 0) { FLAG_BIT_8(sd, bit); }
else { CLEAR_BIT_8(sd,bit); }

//des_data and des_key structures
//  these are simply unions of the prescribed length
//  to provide easy access to chunks of data in different
//  forms, for example, easy access to both all 3 64 bit components
//  of a 196 bit key can be easily accessed via the int_data field,
//  while each of the 24 bytes in that 196 bit key can be accessed
//  individually using the char_data field
//  this is very useful for a bit-manipulation intensive process
//    such as DES or 3DES

union des_data32 {
  unsigned __int32 int_data;
  unsigned char char_data[4];

  des_data32() {
    int_data = 0;
  }
};

union des_data48 {
  unsigned __int64 int_data : 48;
  unsigned char char_data[6];

  des_data48() {
    int_data = 0;
  }
};

union des_data64 {
  unsigned __int64 int_data;
  unsigned char char_data[8];
  char string_data[8];

  des_data64() {
    int_data = 0;
  }

  des_data64(__int64 num) {
    int_data = num;
  }
};

union des_data196 {
  unsigned __int64 int_data[3];
  unsigned char char_data[24];

  des_data196() {
    int_data[0] = 0;
    int_data[1] = 0;
    int_data[2] = 0;
  }
};

union des_key64 {
  unsigned __int64 int_data;
  unsigned char char_data[8];

  des_key64() {
    int_data = 0;
  }

  des_key64(__int64 num) {
    int_data = num;
  }
};

union des_key196 {
  unsigned __int64 int_data[3];
  unsigned char char_data[24];

  des_key196() {
    int_data[0] = 0;
    int_data[1] = 0;
    int_data[2] = 0;
  }
};

union des_key56 {
  unsigned __int64 int_data : 56;
  unsigned char char_data[7];

  des_key56() {
    int_data = 0;
  }
};

union des_key48 {
  unsigned __int64 int_data : 48;
  unsigned char char_data[6];

  des_key48() {
    int_data = 0;
  }
};
```

```
//Tables come from FIPS PUB 46-3 of the US Government
//THESE TABLES AND VALUES HAVE BEEN TRIPLE CHECKED AND COMPARED TO
KNOWN WORKING VALUES

//Key Permutations and Rotations
const unsigned int keyPerm[56] = {57, 49, 41, 33, 25, 17, 9, 1, 58, 50,
42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36,
63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53,
45, 37, 29, 21, 13, 5, 28, 20, 12, 4};
const unsigned int keyShifts[16] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2,
2, 2, 2, 1};
const unsigned int keyCompression[48] = {14, 17, 11, 24, 1, 5, 3, 28,
15, 6, 21, 10, 23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31,
37, 47, 55, 30, 40, 51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50,
36, 29, 32};

//Data Permutations
const unsigned int initialPermutation[64] = {58, 50, 42, 34, 26, 18,
10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6,
64, 56, 48, 40, 32, 24, 16, 8, 57, 49, 41, 33, 25, 17, 9, 1, 59, 51,
43, 35, 27, 19, 11, 3, 61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39,
31, 23, 15, 7};
const unsigned int finalPermutation[64] = {40, 8, 48, 16, 56, 24, 64,
32, 39, 7, 47, 15, 55, 23, 63, 31, 38, 6, 46, 14, 54, 22, 62, 30, 37,
5, 45, 13, 53, 21, 61, 29, 36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43,
11, 51, 19, 59, 27, 34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49,
17, 57, 25};
const unsigned int eBox[] = {32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9,
10, 11, 12, 13, 12, 13, 14, 15, 16, 17, 16, 17, 18, 19, 20, 21, 20, 21,
22, 23, 24, 25, 24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1};
const unsigned int pBox[] = {16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23,
26, 5, 18, 31, 10, 2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11,
4, 25};

//S-Boxes

const unsigned __int8 sBox[8][4][16] ={
  {
    {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
    {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
    {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
    {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}
  },
  {
    {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
    {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
    {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
    {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}
  },
  {
    {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
    {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
    {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
    {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}
  },
  {
    {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
    {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
    {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
    {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}
  },
  {
    {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
    {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
    {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
    {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}
  },
  {
    {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
    {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
    {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
    {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}
  },
  {
    {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
    {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
    {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
    {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}
  },
  {
    {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
    {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
    {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
    {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}
  }
};

//This is essentially a finite state machine
class cDES {

private:
  int m_mode;
  int m_chainmode;
  int m_length;
  des_data196 IV;

  int doDES(char *data, char** output, des_key64 key);
  int do3DES(char *data, char** output, des_key196 key);

  //data functions
  des_data64 processBlock(des_data64 data, des_key56 key); //performs
work
  des_data64 IP(des_data64 data); //initial permutation
  des_data64 FP(des_data64 data); //final permutation

  des_data32 round(des_data32 data, des_key48 key); //1 round
  des_data48 eBoxPermutation(des_data32 data); //expansion permutation
  des_data32 sBoxPermutation(des_data48 data); //substitution
permutation
  des_data32 pBoxPermutation(des_data32 data); //straight permutation

  //key functions
  des_key56 pruneParityBits(des_key64 key);
  des_key48 roundKey(des_key56 key, int round);

public:

  cDES();
  cDES(int mode);

  void testDES();
  void test3DES();

  inline int encrypt_des(char* data, int len, char** output, des_key64
key) {
    m_mode = MODE_EDES;
    m_length = len;
    return doDES(data, output, key);
  }

  inline int decrypt_des(char *data, int len, char** output, des_key64
key) {
    m_mode = MODE_DDES;
    m_length = len;
    return doDES(data, output, key);
  }

  inline int encrypt_3des(char* data, int len, char** output,
des_key196 key) {
    m_mode = MODE_E3DES;
    m_length = len;
    return do3DES(data, output, key);
  }

  inline int decrypt_3des(char* data, int len, char** output,
des_key196 key) {
    m_mode = MODE_D3DES;
    m_length = len;
    return do3DES(data, output, key);
  }

  inline des_data64 getIV() { return (des_data64)IV.int_data[0]; }
  inline des_data196 get3IV() { return IV; }
  inline void setIV(des_data64 newIV) { IV.int_data[0] =
newIV.int_data; }
  inline void setIV(des_data196 newIV) { IV = newIV; }
};

#endif
```

DES.cpp

```
//  houses a class that implements DES and 3DES encryption
//  according to Federal Information Processing Standard 46-3
//  from the US National Institute of Standards and Technology
// supported chaining modes:
//   ECB: electronic code book
//   CBC: cipher block chaining
//   CBC Checksum: CBC with a checksum for integrity verification

//cDES is a finite state machine, so all that is required
//  is to set its modes, then call its main functions

#include "scrypto.h"

//Display a buffer one byte at a time in hexadecimal
void disp_hex(unsigned char *data, int n ) {
  int i;

  for (i=0; i<n; i++) {
    //use printf for its hex printing abilities
    printf("%02X ", data[i]);
  }
  printf("\n");
}

//Helper Functions for Printing des_data and des_key types
void disp_desdata64(des_data64 *data) {
  disp_hex((unsigned char*)&data->char_data, 8);
}

void disp_deskey64(des_key64 *data) {
  disp_hex((unsigned char*)&data->char_data, 8);
}

void disp_deskey48(des_key48 *data) {
  disp_hex((unsigned char*)&data->char_data, 6);
}

cDES::cDES() {
  m_mode = 0;
  m_chainmode = CHAINMODE_ECB;
  m_length = 0;
}

cDES::cDES(int mode) {
  m_mode = 0;
  m_chainmode = mode;
  m_length = 0;
}


//This function was used to test compliance to
// FIPS PUB 46-3 using the NIST Document 800-17
// which has test vectors for a DES encryption unit
void cDES::testDES() {
  int i,j;
  des_data64 data;
  des_key64 key;
  des_key56 prunedKey;
  unsigned char work;

  //NIST Document 800-17 : Test Vectors for DES
  unsigned char test[9][8]= {
    //NIST Example Round
    //CONFIRMED: we can do this one
    {0x10,0x31,0x6E,0x02,0x8C,0x8F,0x3B,0x4A}, //KEY
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, //PLAINTEXT
    {0x82,0xDC,0xBA,0xFB,0xDE,0xAB,0x66,0x02}, //GOAL CIPHERTEXT

    //NIST Variable Plaintext Known Answer Test
    //CONFIRMED: we can do this one
    {0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01},
    {0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
    {0x95,0xF8,0xA5,0xE5,0xDD,0x31,0xD9,0x00},

    //NIST Variable Key Known Answer Test
    //CONFIRMED: we can do this one
    {0x80,0x01,0x01,0x01,0x01,0x01,0x01,0x01},
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
    {0x95,0xA8,0xD7,0x28,0x13,0xDA,0xA9,0x4D}
  };

  //fill the data and key
  for(i=0; i<8; i++) {
    key.char_data[i] = test[0][i];
    data.char_data[i] = test[1][i];
  }

  cout << "Key       =";
  disp_deskey64(&key);
  cout << "Plaintext =";
  disp_desdata64(&data);

  //reverse key and data byte order b/c the government uses big endian
computers
  for(i=0;i<8;i++) {
    work = 0;
    for(j=0;j<8; j++) {
      if(IS_SET_8(data.char_data[i], j)) {
        work |= (1 << (7-j));
      }
    }
    data.char_data[i] = work;
  }

  for(i=0;i<8;i++) {
    work = 0;
    for(j=0;j<8; j++) {
      if(IS_SET_8(key.char_data[i], j)) {
        work |= (1 << (7-j));
      }
    }
```

```
      key.char_data[i] = work;
    }

    m_mode = MODE_EDES;

    prunedKey = pruneParityBits(key);
    data = processBlock(data, prunedKey);

    //reverse again for printing
    for(i=0;i<8;i++) {
      work = 0;
      for(j=0;j<8; j++) {
        if(IS_SET_8(data.char_data[i], j)) {
          work |= (1 << (7-j));
        }
      }
      data.char_data[i] = work;
    }

    cout << "Ciphertext=";
    disp_desdata64(&data);
    cout << "Should Be =";
    disp_hex(test[2], 8);

  }

  //Main entry point for 3DES Encryption,
  //  sets up the key and IV, padds the data, adds a checksum
  //  then calls DES three times
  int cDES::do3DES(char *data, char **output, des_key196 key) {
    char *r1, *r2, *r3, *paddedPT;
    des_data196 tempIV;
    des_key64 k1, k2, k3;
    des_data64 i1, i2, i3;
    char pad;

    r1 = r2 = r3 = paddedPT = NULL;

    //Pad the data even if we arn't in checksum mode
    //  it lets us know how much we had to pad it
    if(m_mode == MODE_E3DES) {
      //Postpend the identification tag and the padding

      //We want the data to be a multiple of 8 bytes long
      pad = 8 - m_length%8;
      paddedPT = new char[m_length + pad + 8]; //our final block is 8
bytes

      if(!paddedPT)
        throw cError("cDES::do3DES() - Insufficient Memory for padding");

      //clear and copy the data into the new buffer
      memset(paddedPT, 0, m_length+pad+8);
      memcpy(paddedPT, data, m_length);

      m_length = m_length+pad+8;
      //Pad the plaintext with the checksum tag
      //The tag is of this format "SCRYPTOX"
      //  where X is how many bytes of data were added
      //  to the original data, this way the receiver
      //  upon verifying that SCRYPTO is intact,
      //  can find the true length of the data transmitted
      //its faster and more accurate to unroll the loop
      paddedPT[m_length-1] = pad;
      paddedPT[m_length-2] = 'O';
      paddedPT[m_length-3] = 'T';
      paddedPT[m_length-4] = 'P';
      paddedPT[m_length-5] = 'Y';
      paddedPT[m_length-6] = 'R';
      paddedPT[m_length-7] = 'C';
      paddedPT[m_length-8] = 'S';
    }
    else
      paddedPT = data;

    tempIV = IV; //backup the IV
    if(m_mode == MODE_E3DES) {
      k1.int_data = key.int_data[0];
      k2.int_data = key.int_data[1];
      k3.int_data = key.int_data[2];
      i1 = IV.int_data[0];
      i2 = IV.int_data[1];
      i3 = IV.int_data[2];
    }
    else if(m_mode == MODE_D3DES) {
      k1.int_data = key.int_data[2];
      k2.int_data = key.int_data[1];
      k3.int_data = key.int_data[0];
      i1 = IV.int_data[2];
      i2 = IV.int_data[1];
      i3 = IV.int_data[0];
    }

    //3DES = DES(DES(DES(PT, k1), k2), k3)
    IV.int_data[0] = i1.int_data;
    doDES(paddedPT, &r1, k1);
    IV.int_data[0] = i2.int_data;
    doDES(r1, &r2, k2);
    IV.int_data[0] = i3.int_data;
    doDES(r2, &r3, k3);

    IV = tempIV; //reset the IV

    //Confirm and strip the checksum pad
    if(m_mode == MODE_D3DES) {
      pad = r3[m_length-1]; //the last byte is the pad amount
      if(strncmp(r3+m_length-8, "SCRYPTO", 7) != 0)
        throw cError("cDES::do3DES() - Checksum is incorrect; Transfer
  failed.");

      m_length = m_length - pad - 8;
    }
    char *final = new char[m_length];

    if(!final)
      throw cError("cDES::do3DES() - Insufficient Memory for 3DES");

    memcpy(final, r3, m_length);
    *output = final; //pass the result back to the caller

    //free up our allocated memory
    delete r3;
    delete r2;
    delete r1;
    if(m_mode == MODE_E3DES)
      delete paddedPT;
    r3 = r2 = r1 = paddedPT = NULL; //for safety

    //return the length of *output to the caller
    return m_length;
  }

  int cDES::doDES(char *data, char **output, des_key64 key) {
    des_data64 *blocks;
    des_data64 work, work2, total;
    des_key56 prunedKey;
    unsigned int i,k;
    int j;
    char *result, *paddedPT;
    char pad;

    unsigned int numBlocks;

    //Postpend the identification tag and the padding
    //  this is more or less identical to the 3DES version
    if(m_mode == MODE_EDES) {

      pad = 8-m_length % 8;
      paddedPT = new char[m_length + pad + 8]; //our final block is 8
  bytes

      if(!paddedPT)
        throw cError("cDES::doDES() - Insufficient Memory for padding");

      //clear and fill the buffer
      memset(paddedPT, 0, m_length+pad+8);
      memcpy(paddedPT, data, m_length);

      m_length = m_length+pad+8;

      //Pad the plaintext with the checksum tag
      //its faster and more accurate to unroll the loop
      paddedPT[m_length-1] = pad;
      paddedPT[m_length-2] = 'O';
      paddedPT[m_length-3] = 'T';
      paddedPT[m_length-4] = 'P';
      paddedPT[m_length-5] = 'Y';
      paddedPT[m_length-6] = 'R';
      paddedPT[m_length-7] = 'C';
      paddedPT[m_length-8] = 'S';
    }
    else
      paddedPT = data;

    numBlocks = m_length/8; //we padded it so this should divide evenly
    blocks = new des_data64[numBlocks];

    if(!blocks)
      throw cError("cDES::doDES() - Insufficient Memory for DES");

    memset(blocks, 0, sizeof(des_data64)*numBlocks);

    prunedKey = pruneParityBits(key);

    //set the initialization vector
    work.int_data = IV.int_data[0];

    //for every block
    for(i=0, j=0; i<numBlocks; i++) {
      //copy each block into the block structure
      for(k=0; j<m_length && k<8; k++, j++) {
        blocks[i].char_data[k] = paddedPT[j];
      }

      //now perform the chain mode processing
      if(m_mode == MODE_EDES || m_mode == MODE_E3DES) {
        //for ECB, just process the block
        if(m_chainmode == CHAINMODE_ECB) {
          blocks[i] = processBlock(blocks[i], prunedKey);
        }
        //for CBC, each PT block is XOR'd with the previous CT block
  before encryption
        else if(m_chainmode == CHAINMODE_CBC) {
          blocks[i].int_data ^= work.int_data; //xor with previous (CBC)
          blocks[i] = processBlock(blocks[i], prunedKey);
          work.int_data = blocks[i].int_data;
        }
        //for CBC Checksum, each PT block is XOR'd with the previous CT
  block
        // and the final block is XORed with a running XOR of all the PT
  blocks
        else if(m_chainmode == CHAINMODE_CHECKSUMCBC) {
          if(i==0) //keep a running total XOR
            total.int_data = blocks[i].int_data;
          else if(i<numBlocks-1)
            total.int_data ^= blocks[i].int_data;
          else if(i==numBlocks-1)
            blocks[i].int_data ^= total.int_data;

          blocks[i].int_data ^= work.int_data; //xor with previous (CBC)

          blocks[i] = processBlock(blocks[i], prunedKey);
          work.int_data = blocks[i].int_data;
        }
      }
      else if (m_mode == MODE_DDES || m_mode == MODE_D3DES) {
        //for ECB just process the block
        if(m_chainmode == CHAINMODE_ECB) {
          blocks[i] = processBlock(blocks[i], prunedKey);
        }
        //for CBC each block is decrypted then XOR'd with the previous CT
  block
        else if(m_chainmode == CHAINMODE_CBC) {
          work2.int_data = blocks[i].int_data; //save CT for next block
          blocks[i] = processBlock(blocks[i], prunedKey);
          blocks[i].int_data ^= work.int_data; //XOR with previous CT
  block or IV
          work.int_data = work2.int_data;
        }
        else if(m_chainmode == CHAINMODE_CHECKSUMCBC) {
          work2.int_data = blocks[i].int_data;
          blocks[i] = processBlock(blocks[i], prunedKey);
          blocks[i].int_data ^= work.int_data;
          work.int_data = work2.int_data;

          if(i==0) //keep a running total XOR
            total.int_data = blocks[i].int_data;
          else if(i<numBlocks-1)
            total.int_data ^= blocks[i].int_data;
          else if(i==numBlocks-1)
            blocks[i].int_data ^= total.int_data;

        }
      }
    }

    //Check and strip the checksum pad
    if(m_mode == MODE_DDES) {
      pad = blocks[numBlocks-1].char_data[7];
      if(strncmp((char*)blocks[numBlocks-1].char_data, "SCRYPTO", 7) !=
  0)
        throw cError("cDES::doDES() - Checksum is incorrect; Transfer
  failed.");
      m_length = m_length - pad - 8;
    }

    //Compose Result
    result = new char[m_length];
    if(!result)
      throw cError("cDES::doDES() - Insufficient Memory for DES");
    memset(result, 0, m_length);

    //could do a memset, but this is more clear
    for(i=0,j=0; i<numBlocks; i++) {
      for(k=0;k<8 && j<m_length;k++,j++){
        result[j] = blocks[i].char_data[k];
      }
    }

    //return the result in *output
    *output = result;

    //free our memory
    if(m_mode == MODE_EDES)
      delete paddedPT;
    delete blocks;

    //return the length of *output to the caller
    return m_length;
  }

  //This performs the key permutation and strips out parity bits
  des_key56 cDES::pruneParityBits(des_key64 key) {
    int i;
    des_key56 pKey;

    for(i=0; i<56; i++) {
      SET_BIT_56(pKey.int_data, i, IS_SET_64(key.int_data, (keyPerm[i]-
  1)));
    }

    return pKey;
  }

  //Performs the initial permutation
  des_data64 cDES::IP(des_data64 data) {
    int i;
    des_data64 result;

    for(i=0; i<64; i++) {
      SET_BIT_64(result.int_data, i, IS_SET_64(data.int_data,
  (initialPermutation[i]-1)));
    }

    return result;
  }

  //Performs the final permutation (inverse of IP)
  des_data64 cDES::FP(des_data64 data) {
    int i;
    des_data64 result;

    for(i=0; i<64; i++) {
      SET_BIT_64(result.int_data, (initialPermutation[i]-1),
  IS_SET_64(data.int_data, i));
    }

    return result;
  }

  //Main processing block function for DES
  des_data64 cDES::processBlock(des_data64 data, des_key56 key) {
    int i;
    des_data32 li, lip;
    des_data32 ri, rip;
    des_key48 ki;
    des_data64 result;

    data = IP(data); //Perform Initial Permutation
    //split the 64 bit block into two 32 bit segments
    // left : bits 0-31 right : bits 32 - 63
    lip.int_data = __int32(data.int_data &
  ((__int64)(0x00000000FFFFFFFFLL)));
    rip.int_data = __int32((data.int_data &
  ((__int64)(0xFFFFFFFF00000000LL))) >> 32);

    //Perform Round Work
    //For rounds i from 0 to 15
    for(i=0; i<16; i++, lip = li, rip = ri) {
      //Get the key for this round
      ki = roundKey(key, i);

      //Li = Ri-1
      li = rip;
      //Ri = Li-1 XOR round(Ri-1, Ki)
      ri.int_data = lip.int_data ^ round(rip, ki).int_data;

      if(DEBUG)
        cout << "Round #" << i << " Left: " << li.int_data << " Right: "
  << ri.int_data << endl;
    }

    //Rejoin the two chunks, right left rather than left right
    result.int_data = ri.int_data;
    result.int_data |= (((__int64)(li.int_data)) << 32);

    result = FP(result); //Perform Final Permutation

    return result;
  }

  //returns the permuted 48 bit key based on the original 56 bit key
  des_key48 cDES::roundKey(des_key56 key, int round) {
    des_key48 result;
    des_key56 rotatedKey;
    int shift, i;

    //if decrypting, deliver keys in opposite order
    if(m_mode == MODE_DDES || m_mode == MODE_D3DES)
      round = 15-round;

    for(i=0, shift=0; i<=round; i++) {
      shift += keyShifts[i];
    }

    //don't split into two 28 bit halves, not necessary
    //circularly rotate each set of 28 bits in the 56 bit key by the
  shift
    //intense bit manipulation here: basically shifts everything around

    rotatedKey.int_data = key.int_data;
    for(i=0; i<shift; i++) {
      //Circular Left Shift
      //rotatedKey.int_data = ((rotatedKey.int_data << 1) &
  ((__int64)(0xFFFFFFEFFFFFFELLU))) | ((rotatedKey.int_data >> 27) &
  ((__int64)(0x00000010000001LLU)));
```

```
    //Circular Right Shift - the government says left but means right
    rotatedKey.int_data = ((rotatedKey.int_data >> 1) &
((__int64)(0x7FFFFFF7FFFFFFLLU))) | ((rotatedKey.int_data << 27) &
((__int64)(0x80000008000000LLU)));
  }

  if(DEBUG) {
    cout << "Original 56bit Key: " << key.int_data << endl;
    cout << "Rotated 56bit Key: " << rotatedKey.int_data << endl;
  }

  //now compress the 56 bit key to the 48 using the compression table
  for(i=0; i<48; i++) {
    SET_BIT_48(result.int_data, i, IS_SET_56(rotatedKey.int_data,
(keyCompression[i]-1)));
  }

  if(DEBUG)
    cout << "Result Compressed Key: " << result.int_data << endl;

  return result;
}

//expand 32 bit data to 48 bits according to the expansion box
des_data48 cDES::eBoxPermutation(des_data32 data) {
  int i;
  des_data48 result;

  for(i=0; i<48; i++) {
    SET_BIT_48(result.int_data, i, IS_SET_32(data.int_data, (eBox[i]-
1)));
  }

  return result;
}


//SBox bit assignments
//S1: 1-6 S2: 7-12 S3: 13-18 S4: 19-24 S5: 25-30 S6: 31-36 S7: 37-42
S8: 43-48

//Perform the SBox substitutions
des_data32 cDES::sBoxPermutation(des_data48 data) {
  int row, col;
  des_data32 result;
  unsigned __int32 sBoxInputs[8];
  unsigned __int64 sval;
  int i;

  //this basically chops the 48 bit input into 8 6-bit chunks
  for(i=0; i<8; i++) {
    sBoxInputs[i] = __int32 (  ( (((unsigned __int64)(0x3FLLU)) <<
(i*6)) & data.int_data ) >> (i*6) );
  }

  for(i=0; i<8; i++) {
    //Little Endian Byte Order
    //row = (sBoxInputs[i] & 0x01) | ((sBoxInputs[i] & 0x20) >> 4);
//bits 1 and 6 are the row
    //col = (sBoxInputs[i] & 0x1E) >> 1; //bits 2-5 are the column

    //Big Endian Byte Order
    //The SBoxes were designed by the government so flip byte order
    row = (sBoxInputs[i] & 0x01) << 1 | ((sBoxInputs[i] & 0x20) >> 5);
//bits 1 and 6 are the row
    col = ((sBoxInputs[i] & 0x10) >> 4) | ((sBoxInputs[i] & 0x08) >> 2)
| ((sBoxInputs[i] & 0x04)) | ((sBoxInputs[i] & 0x02) << 2);

    //also flip the value of the sbox, because the table is listed in
big endian byte order
    sval = (__int64)(sBox[i][row][col]);
    sval = ((sval & 0x01) << 3) | ((sval & 0x02) << 1) | ((sval & 0x04)
>> 1) | ((sval & 0x08) >> 3);

    result.int_data |= (sval << (4*i)); //pack the sbox values in every
4 bits
  }

  return result;
}

//Perform the pBox permutation
des_data32 cDES::pBoxPermutation(des_data32 data) {
  des_data32 result;
  int i;

  for(i=0; i<32; i++) {
    SET_BIT_32(result.int_data, i, IS_SET_32(data.int_data, (pBox[i]-
1)));
  }

  return result;
}

//round function, all the round work goes on in here
des_data32 cDES::round(des_data32 data, des_key48 key) {
  //we are given the 32 bit data from the round and the 48 bit round
key
  des_data48 expandedData;
  des_data48 sBoxInput;
  des_data32 result;

  //Perform the Expansion Permutation
  expandedData = eBoxPermutation(data);

  //XOR Expansion Permutation with the Round Key
  sBoxInput.int_data = key.int_data ^ expandedData.int_data;

  //Perform the S-Box Substitution
  result = sBoxPermutation(sBoxInput);

  //Perform the P-Box Permutation
  result = pBoxPermutation(result);
  //Return the result
  return result;
}
```

diffie.h

```
#ifndef _DIFFIE_H_
#define _DIFFIE_H_

//  Function prototypes for Diffie Hellman Key Negotiation
int diffieAlice(cRawSocket *socket);
int diffieBob(cRawSocket *socket);

#endif
```

diffie.cpp

```
//  houses an implementation of Diffie Hellman Key Negotiation
//  tailored to work with cRawSocket

//Diffie Hellman Key Negotiation
// Alice            Bob
// =======          =======
// Pick xa          Pick xb
// Pick n           Pick g
// Send n           Send g
// Recv g           Recv n
// Calc ya          Calc yb
// Send ya          Send yb
// Recv yb          Recv ya
// Calc k           Calc k'
//          k = k'

//since the exponents in calculating ya and yb will pretty much
// overflow a 64 bit int, even a double: use this property of modular
// division to keep numbers nice and small
// to calculate ya = pow(g, xa) % n :
//for(int i=0, ya=1; i<xa; i++) {
//  ya *= g;
//  ya = ya % n;
//}

#include "scrypto.h"

int diffieAlice(cRawSocket *socket) {
  unsigned int i, j;
  int length;
  unsigned int xa, ya, yb;
  unsigned int n, g, k;
  char *input;

  //Seed the RNG
  srand(getpid()*time(NULL));

  //Pick xa
  //To keep variables from overflowing, limit xa's size
  //real implementations should use high-precision variables
  xa = rand() % 255 + 2; //xa > 1 b/c its weak as 1 or 0
  //Pick n
  n = rand() % 255 + 2; //n > 1 because 0 div by zero and 1 is weak
(mod n)

  sleep(1);

  //Send n
  socket->Send((char*)&n, 4);

  //Get g
  i=0;
  length = 4;
  while(i<4) {
    if(!socket->Recv(&input, &length))
```

```
      break;

    for(j=0; j<length; j++) {
      ((char*)&g)[i] = input[j];
      i++;
    }
    length = 3-i;
    delete input;
  }

  //Calculate ya
  //ya = g^xa mod n;
  for(i=0, ya=1; i<xa; i++) {
    ya *= g;
    ya = ya % n;
  }

  sleep(1);
  //Send ya
  socket->Send((char*)&ya, 4);

  //Get yb
  i=0;
  length = 4;
  while(i < 4) {
    if(!socket->Recv(&input, &length))
      break;

    for(j=0; j<length; j++) {
      ((char*)&yb)[i] = input[j];
      i++;
    }
    length = 3-i;
    delete input;
  }

  //Calculate k
  //k=yb ^ xa mod n
  for(i=0,k=1; i<xa; i++) {
    k *= yb;
    k = k % n;
  }

  if(DEBUG)
    cout << "DiffieAlice xa:" << xa << " g:" << g << " n:" << n << "
ya:" << ya << " yb:" << yb << " k:" << k << endl;

  return k;
}

int diffieBob(cRawSocket *socket) {
  unsigned int i, j;
  int length;
  char *input;
  unsigned int xb, yb, ya;
  unsigned int n, g, k;
```

```
//Seed the RNG
srand(getpid()*time(NULL));

//Pick xb
//To keep variables from overflowing, limit xa's size
//real implementations should use high-precision variables
xb = rand()%255+2;
//Pick g
g = rand()%255+2;

//Get n
i=0;
length = 4;
while(i < 4) {
  if(!socket->Recv(&input, &length))
    break;

  for(j=0; j<length; j++) {
    ((char*)&n)[i] = input[j];
    i++;
  }
  length = 3-i;
  delete input;
}

sleep(1);

//Send g
socket->Send((char*)&g, 4);

//Get ya
i=0;
length = 4;
while(i < 4) {
  if(!socket->Recv(&input, &length))
    break;

  for(j=0; j<length; j++) {
    ((char*)&ya)[i] = input[j];
    i++;
  }
  length = 3-i;
  delete input;
}

//Calculate yb
for(i=0, yb=1; i<xb; i++) {
  yb *= g;
  yb = yb % n;
}

sleep(1);
//Send yb
socket->Send((char*)&yb, 4);

//Calculate k
// k = ya ^ xb mod n
```

```
  for(i=0, k=1; i<xb; i++) {
    k *= ya;
    k = k % n;
  }

  if(DEBUG)
    cout << "DiffieBob xb:" << xb << " g:" << g  << " n:" << n << "
yb:" << yb << " ya:" << ya << " k:" << k << endl;

  return k;
}
```

rawsocket.h

```
#ifndef _RAWSOCKET_H_
#define _RAWSOCKET_H_

//  a class definition a for raw linux socket
//  used to send data covertly through the IPID
//  using TCP Header Encoding

// Credit for the idea of TCP Header Encoding is attributed to
// Craig H. Rowland - Circa 1996 (crowland@psionic.com)

#define MAX_RECEIVE 4096
#define MAX_SEND 4096

struct packet {
  iphdr ip; //20bytes
  tcphdr tcp; //20bytes
  char data[MAX_SEND];
};

//used to compute the TCP checksum
//  take the checksum of the source address,
//  dest_address, placeholder, protocol, tcp_length, the TCP packet
itself,
//  and the data being sent
struct tcphdr_checksum {
  unsigned int source_address;
  unsigned int dest_address;
  unsigned char placeholder;
  unsigned char protocol;
  unsigned short tcp_length;
  struct tcphdr tcp;
  char data[MAX_SEND];
};

class cRawSocket {
private:
  hostent* m_lpHostEntry;

  int m_pInSock, m_pOutSock; //holds the SOCKET ids for
sending/receiving
  struct sockaddr_in m_inAddr; //input socket data structure
  struct sockaddr_in m_outAddr; //output socket data structure

  int m_retVal;
  bool m_isServer;
  int m_iPort;

public:
  cRawSocket(bool server);
  ~cRawSocket();

  //these two neither bind() nor connect()
  // they only set our structures up properly
  bool Bind(int port);
  bool Connect(string address, int port);

  bool Send(char* data, int len);
  bool Send(char* data, int len, unsigned short ipid);

  bool Recv(char **output, int *len);
  bool Recv(char **output, int *len, unsigned short *ipid);

  bool CheckError();
  inline bool isServer() { return m_isServer; }
};

#endif
```

rawsocket.cpp

```
// Houses the definition for the class cRawSocket, which outlines ways
the Unix Sockets
//  library can be used to send data through a covert channel (in this
implementation, the ipid)
//all of the API calls here are standard Unix (Berkeley) Sockets
functions, nothing special

#include "scrypto.h"

//This function computes a 32 bit checksum,
//  a standard library function originally implemented
//  in the ICMP_ECHO code of Unix Sockets. It is released
//  under a General Public License, and is reproduced
//  here with copyright notice for accurate computations,
//  since calculating checksums falls beyond the scope of this paper

/* Copyright (c)1987 Regents of the University of California.
* All rights reserved.
*
* Redistribution and use in source and binary forms are permitted
* provided that the above copyright notice and this paragraph are
* dupliated in all such forms and that any documentation, advertising
* materials, and other materials related to such distribution and use
* acknowledge that the software was developed by the University of
* California, Berkeley. The name of the University may not be used
* to endorse or promote products derived from this software without
* specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS
* IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING,
* WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHATIBILITY AND
* FITNESS FOR A PARTICULAR PURPOSE
*/

unsigned short checksum(unsigned short *ptr, int nbytes)
{
  register long   sum;    /* assumes long == 32 bits */
  u_short     oddbyte;
  register u_short  answer;   /* assumes u_short == 16 bits */

  /*
  * Our algorithm is simple, using a 32-bit accumulator (sum),
  * we add sequential 16-bit words to it, and at the end, fold back
  * all the carry bits from the top 16 bits into the lower 16 bits.
  */

  sum = 0;
  while (nbytes > 1)  {
    sum += *ptr++;
    nbytes -= 2;
  }

  /* mop up an odd byte, if necessary */
  if (nbytes == 1) {
    oddbyte = 0;    /* make sure top half is zero */
    *((u_char *) &oddbyte) = *(u_char *)ptr;   /* one byte only */
    sum += oddbyte;
  }

  /*
  * Add back carry outs from top 16 bits to low 16 bits.
  */

  sum  = (sum >> 16) + (sum & 0xffff);  /* add high-16 to low-16 */
  sum += (sum >> 16);      /* add carry */
  answer = ~sum;    /* ones-complement, then truncate to 16 bits */
  return(answer);
} /* end checksum() */

cRawSocket::cRawSocket(bool server) {
  m_lpHostEntry = NULL;
  m_pInSock = m_pOutSock = 0;
  m_retVal = 0;
  m_isServer = server;
  m_iPort = 0;

  //clear the connection info buffers
  memset(&m_inAddr, 0, sizeof(struct sockaddr_in));
  memset(&m_outAddr, 0, sizeof(struct sockaddr_in));

  m_inAddr.sin_family = AF_INET;
  m_inAddr.sin_addr.s_addr = INADDR_ANY;
  m_inAddr.sin_port = 0;

  m_outAddr.sin_family = AF_INET;
  m_outAddr.sin_addr.s_addr = INADDR_ANY;
  m_outAddr.sin_port = 0;
}

cRawSocket::~cRawSocket() {
  //the sockets are closed in Send and Recv, nothing really to do here
}

//don't actually call bind(), just setup the structure information
bool cRawSocket::Bind(int port) {

  if(!isServer())
    return false;

  m_iPort = port;

  m_inAddr.sin_port = htons(port);
  m_outAddr.sin_port = htons(port+1);

  return true;
}

bool cRawSocket::Connect(string address, int port) {
  unsigned int ip;

  if(address.length() <= 0 || port <= 0 || port > 65536)
    return false;
  m_iPort = port;
  m_lpHostEntry = gethostbyname(address.c_str());

  if (!m_lpHostEntry) {
    ip = inet_addr(address.c_str());
  }
  if(!m_lpHostEntry && ip == INADDR_NONE) {
    return false;
  }

  m_outAddr.sin_family = AF_INET;
  if(m_lpHostEntry)
    memcpy(&m_outAddr.sin_addr, m_lpHostEntry->h_addr, m_lpHostEntry-
>h_length);
  else
    m_outAddr.sin_addr.s_addr = ip;

  m_inAddr.sin_family = AF_INET;
  m_inAddr.sin_addr.s_addr = m_outAddr.sin_addr.s_addr;

  if(m_isServer) {
    m_inAddr.sin_port = htons(m_iPort);
    m_outAddr.sin_port = htons(m_iPort+1);
  }
  else {
    m_inAddr.sin_port = htons(m_iPort+1);
    m_outAddr.sin_port = htons(m_iPort);
  }

  return true;
}


bool cRawSocket::Send(char *data, int len) {
  return Send(data, len, 0);
}

//return false if ok, throw cError on error
bool cRawSocket::CheckError() {

  if(m_retVal < 0) //socket functions return <0 on error
    throw cError("cRawSocket: Connection Error");

  return false;
}

//we name this Recv vs. recv to not conflict with sockets library
bool cRawSocket::Recv(char **output, int *len) {
  unsigned short ipid = 0;
  return Recv(output, len, &ipid);
}

bool cRawSocket::Send(char *data, int len, unsigned short ipid) {
  packet send;

  //Fill the IP Header, most of these values are standard for IP
  send.ip.ihl = 5;
  send.ip.version = 4;
  send.ip.tos = 0;
  send.ip.tot_len = htons(40); // ipheader: 20 + tcpheader: 20

  send.ip.id = htons(ipid); //encode the ipid here (htons switches the
byte order for ANSI compliance

  send.ip.frag_off = 0;
  send.ip.ttl = 128; //arbitrary time to live
  send.ip.protocol = IPPROTO_TCP;
  send.ip.check = 0;
  send.ip.saddr = 0; //this doesn't matter because raw sockets are
connectionless
  send.ip.daddr = m_outAddr.sin_addr.s_addr;

  //now make perform the checksum of the ipheader
  send.ip.check = checksum ((unsigned short *)&send.ip, 20);

  //Now fill the TCP Header

  if(m_isServer) {
    send.tcp.source = 0;
    send.tcp.dest = htons(m_iPort+1);
  }
  else {
    send.tcp.dest = htons(m_iPort);
    send.tcp.source = 0;
  }

  //we do this for a reason: an ipid of zero is automatically replaced
  // by the linux kernel with some random value, so, we can't send ipid
0
  // if both bytes happen to be zero, set the sequence number,
  // which is unused in raw sockets, to 1234 if ipid == 0
  // the recv function will just have to check for this
  if(ipid == 0)
    send.tcp.seq = 1234;
  else
    send.tcp.seq = 1; //arbitrary

  send.tcp.ack_seq = 0;
  send.tcp.res1 = 0;
  send.tcp.doff = 5;
  send.tcp.fin = 0;
  //set the SYN flag for easy identification,
  //  and so routers don't auto-filter the packet
  send.tcp.syn = 1;
  send.tcp.rst = 0;
  send.tcp.psh = 0;
  send.tcp.ack = 0;
  send.tcp.urg = 0;
  send.tcp.cwr = 0;
  send.tcp.ece = 0;
  send.tcp.window = htons(512);
  send.tcp.check = 0;
  send.tcp.urg_ptr = 0;
  //Now compute the TCP Header Checksum
  tcphdr_checksum tcpck;
  tcpck.source_address = send.ip.saddr;
  tcpck.dest_address = send.ip.daddr;
  tcpck.placeholder = 0;
  tcpck.protocol = IPPROTO_TCP;
  tcpck.tcp_length = htons(20+len); //tcphdr + len

  memcpy((char*)&tcpck.tcp, (char*)&send.tcp, 20);
  memcpy(tcpck.data, data, len); //include data in checksum
  send.tcp.check = checksum((unsigned short *)&tcpck, 32+len);

  //header is setup, now copy data, then send
  memcpy(send.data, data, len);

  //open a raw socket once per packet so the kernel doesn't get
confused
  m_pOutSock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
  m_retVal = sendto(m_pOutSock, (char*)&send, 40+len, 0,
(sockaddr*)&m_outAddr, sizeof(struct sockaddr_in));
  close(m_pOutSock);

  if(CheckError())
    return false;

  //for debugging purposes
  if(DEBUG)
    cout << "Send: Size:" << m_retVal << " IPID: " << int(ipid) <<
endl;

  return true;
}

bool cRawSocket::Recv(char **data, int *len, unsigned short *ipid) {
  int fromlen = sizeof(struct sockaddr_in);
  char recbuf[MAX_RECEIVE];
  packet *recv; // = new packet;

  recv = (packet*)recbuf;

  //Blocking Input: wait for a suitable packet
  while(1) {
    memset(recbuf, 0, MAX_RECEIVE);

    m_pInSock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if(*len > 0)
      m_retVal = recvfrom(m_pInSock,recbuf,*len+40,0,(struct
sockaddr*)&m_inAddr, (socklen_t*)&fromlen);
    else
      m_retVal = recvfrom(m_pInSock,recbuf,MAX_RECEIVE,0,(struct
sockaddr*)&m_inAddr, (socklen_t*)&fromlen);
    close(m_pInSock);

    if(CheckError())
      return false;

    if(recv->tcp.syn == 1) {//process the packet if its ours
```

```
      if(m_isServer && ntohs(recv->tcp.dest) == m_iPort)
        break;
      if(!m_isServer && ntohs(recv->tcp.dest) == m_iPort+1)
        break;
    }
  }

  if(m_retVal < 40 && m_retVal > 0)
    throw cError("cRawSocket::Recv - something's wrong, <40 bytes of
data, not enough for a packet header");

  //extract the IPID
  if(recv->tcp.seq == 1234)
    *ipid = 0;
  else
    *ipid = ntohs(recv->ip.id);

  //for debugging purposes
  if(DEBUG)
    cout << "Recv: Size:" << m_retVal << " IPID:" << *ipid << endl;

  //Return the received data, this is why we have a double pointer
  *data = new char[m_retVal];
  *len = m_retVal-40;
  memcpy(*data, recv->data, m_retVal-40);

  return true;
}
```

error.h

```
#ifndef _ERROR_H_
#define _ERROR_H_

//  class definition for a generic error handling class

class cError {
private:
  string description;

public:
  cError(string desc);
  cError();

  void print();
};
```

#endif

error.cpp

```
//definition for a generic error handling class

#include "scrypto.h"

cError::cError(string desc) {
  description = desc;
}

cError::cError() {
  description = "Unspecified Error";
}

void cError::print() {
  cout << endl;
  cout << "Exception Caught: " << description;
  cout << endl << "Aborting..." << endl;
}
```

sendlog.txt – output of the program in send mode sending file: send.txt

```
[root@atrius.ath.cx]# ./scrypto
Scrypto Protocol Tester by Russell Ryan : Completed January 2005
Function as a sender or receiver? (S/R):s
Enter address to connect to:localhost
Enter port:4000
Enter filename to send:send.txt
Begin negotiating 196-bit key...
Begin computing the initialization vector...
Sending 416 bytes:...
25%...50%...75%...100% Done.
File transfer complete... Quitting...
[root@atrius.ath.cx]#
```

send.txt – file sent in sendlog.txt

BEGIN

```
This is a test of the SCRYPTO Example Protocol
Implementation included with Russell Ryan's
International Baccalaureate Extended Essay.

If you are reading this right now, then the transfer worked.
The following list of numbers is the first 5 terms
of an arithmetic series with constant interval 5,
starting at 1. This is an easy method of verification:
1 6 11 16 21

This is the last line of data.
```

EOF

receivelog.txt – output of the program in receive mode saving data as receive.txt

```
[root@atrius.ath.cx]# ./scrypto
Scrypto Protocol Tester by Russell Ryan : Completed January 2005
Function as a sender or receiver? (S/R):r
Enter a port to listen on:4000
Connected... setting up transfer.
Negotiating 196-bit key...
Begin Computing Initialization Vector...
Receiving 416 bytes:...
25%...50%...75%...100% Done.
Enter filename for received data:receive.txt
Contents of receive.txt:
This is a test of the SCRYPTO Example Protocol
Implementation included with Russell Ryan's
International Baccalaureate Extended Essay.

If you are reading this right now, then the transfer worked.
The following list of numbers is the first 5 terms
of an arithmetic series with constant interval 5,
starting at 1. This is an easy method of verification:
1 6 11 16 21

This is the last line of data.
EOF
File transfer complete... Quitting...
[root@atrius.ath.cx]#
```

receive.txt – file received in receivelog.txt

BEGIN
```
This is a test of the SCRYPTO Example Protocol
Implementation included with Russell Ryan's
International Baccalaureate Extended Essay.

If you are reading this right now, then the transfer worked.
The following list of numbers is the first 5 terms
of an arithmetic series with constant interval 5,
starting at 1. This is an easy method of verification:
1 6 11 16 21

This is the last line of data.
```
EOF

Annotated Bibliography

Cole, Eric. <u>Hiding In Plain Sight: Steganography and the Art of Covert Communication.</u> Edited by Carol Long. Indianapolis: Wiley Publishing, Inc., 2003.

This book is one of the first about steganography, as its existence as a field in the electronic world has just recently emerged. It is an in depth introduction to the field complete with plentiful amounts of sample code demonstrating steganographic techniques.

Litterio, Francis. "Why Are One Time Pads Perfectly Secure?" Cryptography. The WWW Virtual Library. <http://www.vlib.org/>.

This essay was particularly useful in verifying that encryption via a one time pad is secure because this was unclear in both Cole and Schneier's books.

Postel, Jon. "*RFC-793: Transmission control protocol*." Request for Comments. Information Sciences Institute. University of Southern California. Sept. 1981.

This Request for Comments publication was written when the protocols governing the Internet were first designed, describing exactly how the TCP was to work. This was useful because it describes the TCP headers in detail, some features of which are no longer used.

Rowland, Craig H. "Covert Channels in the TCP/IP Protocol Suite." <u>First Monday: Peer-Reviewed Journal on the Internet.</u> Published 1996. <http://www.firstmonday.org/issues/issue2_5/rowland/>.

This white-paper, published in 1996 is the first recorded attempt at using portions of the TCP/IP headers to transmit data steganographically. This reference document was used in the development of the example implementation of the SCRYPTO Protocol in Appendix A.

Schneier, Bruce. <u>Applied Cryptography.</u> Second Edition. Edited by Phil Sutherland. New York: John Wiley & Sons, Inc., 1996.

This book is the main resource used for information concerning cryptographic problems. Bruce Schneier is a well-known expert in the field and this book is considered a seminal work in the cryptography community. Despite its publishing date, it is flush with in depth discussion on algorithms which are still in some cases considered industry standards today.

_____________. "Crypto Guru Bruce Schneier Answers." Interview by the users of SlashDot.org. 29 Oct. 1999. <http://slashdot.org/interviews/99/10/29/0832246.shtml>.

This interview provides insight into how Schneier's opinions have changed, particularly in relation the IDEA cipher. In Applied Cryptography, he states his admiration for it, but in this interview, expresses that this has changed over the three years since the books publishing. This was useful in selecting a cryptographic algorithm for use in the protocol.

United States. National Institute of Standards and Technology. Computer Security Resource Center. Federal Information Processing Standards Publication 46-3. October 25, 1999, 2.

The FIPS 46-3 document describes in detail the specifications of the Triple-DES algorithm and that it is approved by the United States government as one of four standard algorithms used to protect data of the federal government. This document was useful for information about the algorithm's processes in writing the example programs in Appendix A. The standard information was useful while defining Triple-DES in the Cryptographic Elements section of the paper.

United States. National Institute of Standards and Technology. Computer Security Resource Center. Federal Information Processing Standards Publication 197. November 26, 2001, 5.

The FIPS 197 document describes the specifications of the Advanced Encryption Standard, which describes basic information about the AES algorithm. This was useful in examining the processes of the AES algorithm and deciding to wait before using it in a public protocol due to its lack of age.