Investigating Chaffing and Winnowing: Confidentiality without Encryption

Fleur Kelly

Batchelor of Science in Computer Science with Honours
The University of Bath
May 2008

Investigating Chaffing and Winnowing: Confidentiality without Encryption

Submitted by: Fleur Kelly

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifcally acknowledged, it is the work of the author.

Signed:

Abstract

Chaffing and Winnowing is a new concept to provide data confidentiality, without performing any encryption. It was proposed in response to threats that strong encryption technologies would be outlawed by the U.S. government. Several different Chaffing and Winnowing implementations are presented which incorporate existing ideas and build on them to devise new hybrid methods. Experiments are conducted to compare the schemes implemented with traditional encryption techniques, in terms of execution time and data expansion of the message after encryption. The results demonstrate that Chaffing and Winnowing has the potential to be a feasible alternative to these methods.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my project supervisor, Dr Russell Bradford, for his help and advice throughout the project. I would also like to thank my family and friends for their emotional support, and help with proof reading.

Chapter 1

Introduction

With the dramatic increase in electronic communications over the last few years, people want to ensure that their conversations can remain private. Cryptography provides a solution for achieving confidentiality by making use of the following two main techniques:

- Encryption: the message is transformed into a "ciphertext" using an encryption key. This message is unreadable to anyone who does not possess the secret decryption key which is used to retrieve the original "plaintext". Well-known examples of this include RSA and AES.
- Steganography: the message is hidden in such a way that only the legitimate receiver knows that a message is present. An example of this would be to take an image file and adjust the colour of every 100th pixel to correspond to a letter in the alphabet; a change subtle enough to go unnoticed by anyone who is not actively looking for it.

We wish to investigate a third paradigm for data privacy called Chaffing and Winnowing, proposed by Ronald Rivest (1998a), and determine whether it would be a viable alternative to the above methods of providing confidentiality.

1.1 What is Chaffing and Winnowing?

Chaffing and Winnowing allows confidentiality to be provided without performing any encryption on the data - it is sent 'in the clear'. The message is split up into packets (the wheat), and each wheat packet has a Message Authentication Code (MAC) appended to the end of it, which is calculated based on the packet contents and a secret key known only to the sender and receiver. Random fake packets (the chaff) are then created, given invalid MAC codes, and mixed amongst the wheat. Confidentiality is achieved by the fact that only the legitimate receiver will be able to determine whether a packet is valid, or chaff, because a secret key was used to calculate the MAC. Therefore, in order to 'winnow' the

'wheat' from the 'chaff', the receiver simply recomputes the MAC for each packet, compares it to the MAC value that was sent, and discards any packets where the MACs do not match. The following example, as used by Rivest, illustrates the process using tuples containing sequence number, message and MAC:

The sequence of valid packets ...

(1,Hi Bob,465231)
(2,Meet me at,782290)
(3,7PM,344287)
(4,Love-Alice)

The sequence of packets including chaff ...

(1,Hi Larry,532105)
(1,Hi Bob,465231)
(2,Meet me at,782290)
(2,I'll call you at,793122)
(3,6PM,891231)
(3,7PM,344287)
(4,Yours-Susan,553419)
(4,Love-Alice)

Looking at this example, where there is one chaff packet for every wheat packet, it is obvious that an adversary would not be able to distinguish which packets are valid.

1.2 Why do we need Chaffing and Winnowing?

Rivest's reasons behind proposing this new method are due to the increasing concerns of the U.S. government that encryption technologies are being used to facilitate criminal activity, for example the National Security Agency (NSA) may not be able to decipher messages sent between terrorists and spies if they have been strongly encrypted. For this reason, the government have expressed a desire to have 'back-door' access to encryption and decryption keys (Rivest, 1998b). If encryption techniques were to be banned completely, an alternative way of achieving confidentiality would be required. Rivest argues that the use of keys in a Chaffing and Winnowing scheme is purely to provide authentication, and software that falls into this category is currently approved by the authorities as it does not encrypt any data. A further interesting property of Chaffing and Winnowing is that the process of adding chaff could be performed without the knowledge of either the sender or receiver. For example, if two valid streams of packets are multiplexed together, the recipients will automatically discard packets that have invalid MACs; therefore the wheat of one partic-

ipant acts as chaff for the other (Clayton and Danezis, 2003). In a scenario such as this, the government would have to demand access to the secret authentication key in order to determine which data was sent by which participant. However, the government have long since agreed that they do not want to have access to authentication keys (Rivest, 1998a).

Chaffing and Winnowing can also be used to achieve the property of 'deniable encryption', a concept introduced by Canetti, Dwork, Naor and Ostrovsky (1997). An encryption scheme is *deniable* if, upon a government request to reveal encryption keys, the sender can provide 'plausible' keys such that the authorities are satisfied, but the intial communication remains private. For example, the sender could reveal an authentication key that identifies a wheat subsequence containing an innocuous message, leaving the rest as 'chaff' which would keep the real communication hidden (Rivest, 1998a). A Chaffing and Winnowing scheme named 'Chaffinch', developed by Clayton and Danezis (2003), was explicitly designed to have the property of being able to plausibly deny the existence of certain messages. It was created in response to the UK Regulation of Investigatory Powers Act (UK Stationary Office Ltd, 2000), which states that the authorities can request for protected information to be disclosed in 'an intelligible form'. In Chaffinch, a 'cover message' is sent in order to conceal the contents of the main message. In the face of a legal challenge, the cover message is always revealed first, allowing the sender to deny the existence of any further messages.

1.3 Aims

The aim of this project is to determine whether Chaffing and Winnowing could replace existing encryption technologies, if the US government were to decide that they should be outlawed. In order to do this several different Chaffing and Winnowing schemes are designed and implemented, and compared with popular encryption systems using metrics such as speed of execution.

Chapter 2

Literature Survey

The following section presents an overview of the existing literature surrounding the topic of Chaffing and Winnowing, and other related areas that are required in order to develop our own Chaffing and Winnowing implementations.

2.1 Chaffing and Winnowing schemes

Looking again at the Chaffing and Winnowing example given in section 1.1, an adversary would not be able to determine which packets are wheat and which are chaff without knowledge of the secret key and the algorithm used to calculate the MAC. However in practice, if we send the wheat packets as English sentences, it would be very difficult to create chaff such that it is still indistinguishable from the wheat. The following method takes care of this problem.

2.1.1 Bit-by-Bit

Rivest acknowledged that the method of Chaffing and Winnowing described earlier does not provide a very high level of security. However, Bellare and Boldyreva (2000) state that using bit-by-bit Chaffing and Winnowing provides a provable level of privacy as high as that of standard encryption schemes. This is achieved by creating wheat packets that each contain only a single bit of the original message, and then sending a corresponding chaff packet for each wheat packet that contains the same serial number and the complementary bit. Providing a 'good' MAC algorithm is used, the adversary faces an essentially impossible task of determining the original message. This is illustrated by the following example of packets created using the bit-by-bit scheme, taken from (Rivest, 1998a):

(1,0,351216)
(1,1,895634)

(2,0,452412)
(2,1,534981)
(3,0,639723)
(3,1,905344)
(4,0,321329)
(4,1,978823)
...

Unfortunately, using this approach in practice would be very inefficient, especially if we want to send large messages. For a message of m bits, $2m(1+p+l)$ bits would be transmitted, where p is the length of a counter, and l is the length of the MAC. This data expansion can be reduced by half if we send only one packet per bit, and choose at random whether it should be wheat or chaff (Bellare and Boldyreva, 2000). When the receiver finds packets that have invalid MAC codes, they simply complement the bit that was sent. However this approach would be still be inefficient compared to alternative encryption techniques.

2.1.2 All-Or-Nothing Encryption

Rivest (1997) presented a new mode of operation called *all-or-nothing encryption*, which can be used as a pre-processing step to an existing encryption mode to protect against certain attacks. Rivest (1998a) states that using this technique as part of a Chaffing and Winnowing scheme would greatly improve efficiency compared to the bit-by-bit method. We demonstrate how this is achieved later. There are several candidates which can be used to achieve this 'All-or-Nothing' encryption property; these are known as All-or-Nothing Transforms, or AONTs. Rivest (1997) gave the following definition of the properties that an AONT must posess:

A transformation f mapping a message sequence $m_1, m_2, \ldots, m_s$ into a psuedo-message sequence $m_1^{'}, m_2^{'}, \ldots, m_{s'}^{'}$ is said to be an *all-or-nothing transform* if:

- The transformation f is reversible: given the psuedo-message sequence, one can obtain the original message sequence.
- Both the transformation f and its inverse are efficiently computable (that is, computable in polynomial time).
- It is computationally infeasible to compute any function of any message block if any one of the pseudo-message blocks is unknown.

Additionally, the transformation must be randomised such that a known input does not result in a known pseudo-message. It is also important to note here that an AONT is a keyless operation; the process can be undone by anyone who receives the message, therefore it is not classed as encryption. An AONT could be used simply to ensure that a recipient

will either be able to see all of the message, or none of it.

Using an AONT as part of a Chaffing and Winnowing scheme involves first applying the transform to the input message, then splitting it into blocks and adding MAC codes as before. The use of an AONT means that unless the entire transformed message is received, the individual parts look like random noise, thus allowing more data to be sent in each packet; Rivest suggests 1024 bits. The next step is to create chaff blocks containing 1024 bits of random data, attach random MACs, and mix the chaff packets amongst the wheat. Since an adversary must be able to identify each wheat packet precisely in order to reverse the AONT, it is no longer necessary to send one chaff packet for every wheat packet. The difficulty of separating wheat blocks from chaff will be proportional to the number of ways a subsequence of blocks can be chosen and tested for being wheat, which will be exponential in the total number of blocks, as long as the number of chaff packets does not depend on the length of the plaintext (Rivest, 1998a).

We can see that the use of AONTs reduces the overhead of the number of chaff packets that need to be sent and improves overall efficiency by allowing more data to be sent in each packet, whilst still making it difficult for an adversary to determine which packets are wheat and which are chaff.

2.2 All or Nothing Transforms

We will now look in more detail at the different AONTs that are available.

2.2.1 Package Transform

When Rivest (1997) first presented the concept of *all-or-nothing encryption*, he also proposed a way of implementing an AONT called the 'package transform'. He states that using the package transform increases the time taken to encrypt or decrypt by a factor of three, compared to an ordinary encryption scheme. As stated previously, the AONT does not use a secret key. Instead, a key is chosen at random and used such that it can be easily determined from the output of the transform. Rivest defined the construction of the package transform as follows:

- Let the input message be $m_1, m_2, \ldots, m_s$
- Choose at random a key K' for the package transform block cipher
- Compute the output sequence $m'_1, m'_2, \ldots, m'_{s'}$ for $s' = s + 1$ as follows:
  - Let $m'_i = m_i \oplus E(K', i)$ for i = 1,2,3,...,s

- Let

$$m_{s'}^{'} = K^{'} \oplus h_1 \oplus h_2 \oplus \ldots \oplus h_s,$$

where

$$h_i = E(K_0, m_i^{'} \oplus i) \text{ for } i = 1, 2, \ldots, s,$$

where K_0 is a fixed, publically-known encryption key.

It is then stated that it is not possible to compute $K^{'}$ if any part of the pseudo-message sequence is missing, therefore it is infeasible to compute any block of the original message. However, Bellare and Boldyreva (2000) argue that Rivest's definition of security for an AONT, and hence the package transform, is not very secure. Boyko (1999) points out that Rivest's definition only considers the amount of information that can be learned about a *particular* message block, and does not consider what can be learned about the message as a whole. Boyko's definition of security for an AONT considers exactly how much information about the input can be determined by looking at all but a certain number of bits of the AONT output, and how much effort is required to obtain that information.

2.2.2 CTRT

Desai (2000) proposed a variation on the package transform, called CTRT, which skips one 'pass' altogether providing a more efficient AONT. Where the package transform takes three times as long to encrypt compared with a standard encryption scheme, CTRT only takes twice as long. The construction of CTRT is as follows:

- Compute the output sequence $m_1^{'}, m_2^{'}, \ldots, m_s^{'}$ in the same way as the package transform
- Let the final block be

$$m_{s'}^{'} = K^{'} \oplus m_1^{'} \oplus m_2^{'} \oplus \ldots \oplus m_s^{'}$$

where $K^{'}$ is a key chosen at random as before.

It is pointed out that CTRT is not secure in the strong sense as defined by Boyko (1999). However Desai proves that it is strong enough when used to realise the all-or-nothing encryption paradigm; as long as at least one block of the output is missing, the key remains theoretically hidden.

2.2.3 Optimal Asymmetric Encryption Padding

Optimal Asymmetric Encryption Padding (OAEP) was originally introduced by Bellare and Rogaway (1995) as a means of constructing more efficient public-key encryption schemes that provide semantic security and are plaintext-aware. As stated previously, Boyko (1999), specifies formal definitions for AONT security. He then goes on to prove that these definitions can be met if OAEP is used as the AONT, and he also states that no AONT construction can achieve a substantially better security bound than the one presented. The construction of OAEP is given as follows:

To encrypt $x \in \{0,1\}^n$, choose a random k_0-bit r and set

$$\varepsilon^{G,H}(x) = x \oplus G(r) \;||\; r \oplus H(x \oplus G(r))$$

where '||' denotes concatentation, G is a 'generator' function $G : \{0,1\}^{k_0} \rightarrow \{0,1\}^n$, and H is a 'hash function' $H : \{0,1\}^n \rightarrow \{0,1\}^{k_0}$

The proof of security for using OAEP as an AONT provided by Boyko, assumes that G and H are 'random oracles', as introduced by Bellare and Rogaway (1993). In practice, G and H would be instantiated using any standard cryptographic hash function such as MD5 (Rivest, 1992).

Kuwakado and Tanaka (2005) proposed a 'length-preserving' AONT (LP-AONT), similar to OAEP, but different in the input of G. In their case the input is the exclusive-OR result of the message blocks, however in the case of OAEP, it is an l-bit random string that is independent of the message blocks. This LP-AONT is shown to meet a weaker definition of security than that provided by Boyko.

2.2.4 BEAR

Chaffinch, the Chaffing and Winnowing scheme developed by Clayton and Danezis (2003), uses the BEAR block cipher as its all-or-nothing pre-processing step. BEAR (Anderson and Biham, 1996) is a provably secure block cipher, constructed using two hashes and one stream cipher. The original construction of BEAR uses keys to provide secrecy, however the Chaffinch scheme does not require this property, so BEAR is applied in a keyless manner as follows:

The plaintext is divided into two parts, L and R, where L is the length of the output of the hash function, and R can be of arbitrary length. The encryption is done by:

$$\begin{aligned} L &= L \oplus H(R) \\ R &= R \oplus S(L) \\ L &= L \oplus H(R) \end{aligned}$$

where H can be any cryptographic hash function, and S is a stream cipher.

In order to ensure that the same plaintext message results in a different ciphertext, the Chaffinch scheme precedes each message with some random data.

2.2.5 Exposure-Resilient Functions

Canetti, Dodis, Halevi, Kushilevitz and Sahai (2000) gave the construction of provably secure AONTs that do not make use of 'random oracles'. The key to their approach is a notion which they call an *Exposure-Resilient Function* (ERF) - "a deterministic function whose output appears random even if *almost all* the bits of the input are known". The first construction given looks at the simplest possible one-time private key encryption scheme - the one-time pad, and the second construction can be viewed as a special case of OAEP. Although these definitions meet a strong definition of security, they are somewhat inefficient in practice (Desai, 2000).

2.3 A new Chaffing and Winnowing scheme?

Bellare and Boldyreva (2000) proposed an alternative Chaffing and Winnowing scheme that has low data expansion and is provably secure:

> "Simply apply an AONT to the message and then encrypt the first block of the message and authenticate the remaining part of the message. If the last encryption is done by chaffing and winnowing, say using the bit-by-bit scheme, the whole scheme is also a chaffing-and-winnowing scheme, since the AONT is keyless."

An approach such as this is advantageous as there would be a fixed overhead for each message; the number of bits encrypted is independent of the message length. The security of this scheme relies upon a secure AONT being utilised, such as OAEP.

Now that we have seen the various different ways that a message can be processed as part of a Chaffing and Winnowing scheme, we now move on to look at how the MAC can be constructed.

2.4 Message Authentication Codes

In Chaffing and Winnowing schemes, a Message Authentication Code (MAC) is used to authenticate packets. It is important that the MAC algorithm chosen is indistinguishable from a random function to ensure that no information is 'leaked' about the message being MAC'ed, since that would make it easier for an adversary to distinguish wheat from chaff

(Rivest, 1998a). MACs were traditionally constructed from encryption functions such as the DES block cipher, but Kaliski Jr. and Robshaw (1995) point out that performing authentication in this way could be subject to U.S. export restrictions on encryption functions. Therefore, they proposed a way of using the MD5 hash function as a MAC algorithm. Using a hash function instead of a block cipher is also much more efficient in software implementation. It is for this reason that Rivest suggested the use of a hash function based MAC algorithm as part of a Chaffing and Winnowing scheme, specifically HMAC (Bellare, Krawczyk and Canetti, 1996).

2.4.1 HMAC

HMAC is a secure, efficient, and practical MAC scheme. It can be used with any iterative hash function, such as MD5 (Rivest, 1992) or SHA-1 (Eastlake and Jones, 2001), in such a way that the performance degradation of the hash function is minimal. MD5 has been shown to be vulnerable to collision attacks, however these weaknesses do not compromise the use of MD5 within HMAC. SHA-1 is thought to be the cryptographically stronger function, but MD5 is more suitable for cases where superior performance is required (Krawczyk, Bellare and Canetti, 1997). The construction of HMAC, as given by Bellare et al. (1996) is as follows:

$$HMAC_k(x) = F(\bar{k} \oplus opad, F(\bar{k} \oplus ipad, x))$$

where k is a random string of length l, F is a hash function, $\bar{k}$ is the result of padding k with zeros to a full b-bit block-size, opad (the outer pad) consists of the byte '0x5c' repeated as many times as necessary to get a b-bit block, and ipad (the inner pad) consists of the byte '0x36' repeated as many times as necessary to get a b-bit block.

An important feature of HMAC is that the hash function is used in a 'black box' way. This means that if a security vulnerability is identified in the current hash function, or a new hash function is designed that provides better performance or security, then it is very easy to substitute the old module for the new one. Krawczyk et al. (1997) note that the output of hash-based MAC functions are often truncated. This is an advantage for a Chaffing and Winnowing scheme as it reduces the data expansion of the original message, however it is suggested that the output should not be truncated to less than 80 bits to ensure that security is not compromised. Computing HMAC using SHA-1, with the output truncated to 80 bits, is denoted as HMAC-SHA1-80.

2.5 Existing Encryption techniques

We now need to look at the existing encryption techniques that Chaffing and Winnowing can be compared to.

2.5.1 Symmetric-key Systems

Chaffing and Winnowing can be classed as a symmetric-key 'encryption' scheme, since the same key is used to 'encrypt' and 'decrypt' the message.

The 'Data Encryption Standard' (DES), as defined by National Institute of Standards and Technology (NIST) (1999), is a symmetric-key block-cipher that was selected as an official Federal Information Processing Standard (FIPS) for the United States in 1976, and was widely used by the financial sector as it was considered to be very strong at the time (Piper and Murphy, 2002). To encrypt a message, DES performs 16 iterations of the same operation on blocks that are 64 bits in length. However, DES is now considered to be insecure due to the 56-bit key size being too small; using current technology, DES keys can be found in less than a day (Piper and Murphy, 2002). Therefore DES has now been replaced by Triple-DES (TDES), which applies DES three times using three different 56-bit DES keys.

Although Triple-DES is considered adequately secure, it is very slow when implemented in software. This prompted the National Institute of Standards and Technology of the United States (NIST) to hold a competition to develop a successor to the DES to be known as the Advanced Encryption Standard (AES). The requirements were for a symmetric block cipher capable of using keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. The eventual winner was called Rijndael, developed by two Belgian cryptographers: Joan Daemen and Vincent Rijmen. The official AES standard was published in 2001 (National Institute of Standards and Technology (NIST), 2001), and is currently one of the most widely used algorithms in symmetric key cryptography due to its speed in both software and hardware, and the security level it provides.

The main computation involved in Chaffing and Winnowing is the MAC calculation, which makes use of cryptographic hash functions. As these hash functions are considered to be fast, we would expect Chaffing and Winnowing to be comparable in performance to techniques such as DES and AES. However, as mentioned previously, Chaffing and Winnowing causes a high level of data expansion, whereas AES and DES do not expand the original message at all.

2.5.2 Public-key Systems

The idea of public-key cryptography was first introduced by Diffie and Hellman (1976). In this form of cryptography, each user has a pair of keys; a secret key known only to him, and a public key known to everyone. Messages are encrypted using the public key and the recipient decrypts the messages using their corresponding private key. The two keys are mathematically related in such a way that it is computationally difficult to derive the private key from the public key. The most well-known public-key system is RSA, which was developed by Ronald Rivest, Adi Shamir and Len Adleman in 1978. In RSA, the associated

mathematical problem is the factorisation of large integers, for which there is no known technique that is practicably efficient. Similarly, the ElGamal public-key system makes use of the difficulty of the discrete logarithm problem to provide security. However due to the computational intensity of such techniques, it is common for public-key cryptography to be used in conjunction with symmetric-key techniques, in order to provide a more efficient solution. Such a scheme uses public-key encryption techniques to encrypt a key K, which is then used to encrypt the actual message using symmetric-key encryption techniques. This is known as hybrid encryption (Cramer and Shoup, 2003). Incorporating these hybrid concepts into a Chaffing and Winnowing scheme could potentially increase security and improve efficiency.

2.6 Endianness

Endianness is an attribute of a system that indicates whether integers are represented with the most significant byte stored at the lowest address (big-endian) or at the highest address (little-endian). Both versions are common and used on different architectures; Solaris UNIX on SPARC machines uses big-endian format, whereas Windows on Intel machines uses little-endian format. Endianness can cause problems when sending numbers over a network. For example when storing integer values to a file, if the file is sent to a machine that uses the opposite endianness, the values will be read in reverse and will not make sense. This could potentially be a common occurrence when messages are sent between users of a cryptosystem.

2.7 Conclusion

We have explained the concept of Chaffing and Winnowing, and investigated the underlying components that are required to implement a full Chaffing and Winnowing scheme. We have looked at existing encryption techniques and identified that Chaffing and Winnowing should be comparable in speed to symmetric-key systems, although our schemes will have a higher data expansion rate due to the amount of authentication data required. It was also found that practical cryptographic systems combine the benefits of symmetric-key encryption with public-key cryptography in order to form a hybrid encryption system.

Chapter 3

Requirements

Using the knowledge gained from the Literature Survey, we now outline the requirements that our Chaffing and Winnowing schemes and their underlying components should have. Both functional and non-functional requirements are considered.

3.1 Functional Requirements

3.1.1 MAC Algorithm

The MAC algorithm is a key component of a Chaffing and Winnowing scheme. As identified in the literature survey, the chosen algorithm must be indistinguishable from a random function to ensure that no information about the original message is leaked. It is also necessary for the algorithm to produce completely different digests when given the same input data more than once but with different keys. The MAC algorithm component should be implemented in such a way that it can be easily replaced, in case a more efficient or secure algorithm is developed.

3.1.2 All-Or-Nothing Transform Implementations

Incorporating all-or-nothing transforms into a Chaffing and Winnowing scheme increases efficiency and makes the input data appear random, therefore three different AONTs found from the research will be implemented. These must have the property that it is computationally difficult to compute the original message if any of the output blocks are missing, but if all blocks are present then it must be easy to reproduce the plaintext. Additionally, the transforms must be implemented so that when an AONT is applied to the same input message twice, two different outputs are produced. In order to maximise code reuse, the AONTs should be written in such a way that they can easily be interchanged when incorporated into a full Chaffing and Winnowing scheme.

3.1.3 Hybrid Implementation

It was discovered in the literature survey that cryptosystems commonly combine the methods of public-key and symmetric-key cryptography to produce hybrid encryption techniques. A Chaffing and Winnowing scheme will be designed and implemented that makes use of these techniques.

3.2 Non-Functional Requirements

3.2.1 Performance

In order to determine if Chaffing and Winnowing is a viable alternative to existing encryption techniques, its performance needs to be assessed and compared to similar systems currently in use. Therefore, the implementations produced should be written to 'encrypt' and 'decrypt' messages as efficiently as possible.

3.2.2 Ciphertext Expansion

As discussed previously, Chaffing and Winnowing causes greater ciphertext expansion than traditional encryption methods due to the overhead of adding chaff packets, and the use of MACs. This is not ideal due to possible bandwidth limitations, so the ciphertext expansion in our systems should be minimised as much as possible.

3.2.3 Security

If Chaffing and Winnowing were to be used instead of current encryption technologies, it must be able to provide an acceptable level of security. The security concerns and recommendations found in the literature survey must be considered in our implementations, such as the security of the MAC algorithm and AONTs.

3.2.4 Cross-Platform Compatibility

The platforms available for testing are UNIX and Windows. The schemes implemented must run correctly on each platform, and they must be cross-platform compatible so that a file 'encrypted' on Windows will 'decrypt' successfully on UNIX, and vice versa.

3.2.5 Implementation Language

The programming language chosen to implement the Chaffing and Winnowing schemes needs to meet the following criteria:

- Must be able to perform bitwise operations
- Must support file reading and writing
- It should be efficient, in order to maximise performance
- Must be platform independent
- Implementations of traditional encryption algorithms must be available in the chosen language, so that they can be used to compare against our implementations

3.2.6 Maintainability

The code produced must adhere to the coding standards of the chosen language, and it should be written in a clear and well structured way so that it is easier to maintain if any changes or improvements need to be made.

Chapter 4

Design

The following chapter details how our Chaffing and Winnowing schemes were designed in order to meet the requirements outlined in the previous chapter. We discuss the high-level design of our schemes, and then move on to look at each module in more detail. Finally, we look at how the Chaffing and Winnowing schemes are to be compared against existing encryption techniques.

4.1 Programming Language

The two languages that were considered for this project are C and Java, since they were already familiar to the author, and they met the necessary requirements discussed in section 3.2.5. C encourages machine-independent programming, therefore has good portability, and is also very efficient with low runtime demand on system resources. A further benefit is that lots of cryptographic libraries written in C are available, which could be made use of in our implementation. Java is also a platform-independent language, since its applications can run on any Java Virtual Machine regardless of the underlying computer architecture. Additionally, the Java SE Security platform provides support for a wide range of standard cryptographic algorithms. However, Java is an object-oriented language with automatic memory management (garbage collection). Since our Chaffing and Winnowing schemes are implementing relatively small and concise algorithms, they would not benefit from making use of the object-oriented paradigm. Garbage collection also introduces an unnecessary performance overhead, since program execution is usually suspended to allow garbage collection to occur. This did not suit our requirements, as our Chaffing and Winnowing schemes need to run as efficiently as possible, therefore C was chosen as the implementation language.

4.2 High-Level Design

4.2.1 All-Or-Nothing Transform schemes

The Chaffing and Winnowing 'scattering scheme', presented by Bellare and Boldyreva (2000), applies an AONT to the input message and views the output as a sequence of s wheat blocks, each n bits long. These wheat packets are then interspaced with $s^{'} > 0$ chaff packets, to create the ciphertext, where the positions of the wheat packets are chosen at random from the set $\{1, \ldots, s + s^{'}\}$. Instead of calculating the wheat packet positions, the number of which will vary depending on the size of the input message, our algorithm randomly calculates the chaff packet locations. We fix the value of $s^{'}$ to be 128, as suggested by Bellare and Boldyreva (2000). This means that the number of chaff packets does not depend on the length of the plaintext in any way which is recommended for security, and there is a fixed overhead for the amount of chaff data that is added, which becomes more cost-effective as the size of the plaintext increases. Next, for each block of the ciphertext, if the index is a chaff position a fake packet and fake MAC are written to the output, otherwise a valid MAC is calculated and output along with the corresponding wheat packet. We note here that as a MAC is attached to every block of the output, the ciphertext expansion will increase as the number of output blocks increases. The 'winnowing' process goes through each block of the 'chaffed' message, computes the corresponding MAC, and discards the packets where the computed MAC value does not match the received MAC value. Once this has been completed, the inverse AONT is applied and the original plaintext is output. The following algorithms show the process of 'encryption' and 'decryption' in our AONT Chaffing and Winnowing schemes.

Algorithm 1 AONT Scheme Encryption

Algorithm $\varepsilon^{F(K,\cdot)}(M)$
  $M^{'} \leftarrow \mathsf{AONT}(M)$
  Parse $M^{'}$ as $m_1 \| m_2 \| \ldots \| m_s$ where $|m_i| = n$
  Pick $S \subseteq \{1, \ldots, s + s^{'}\}$ at random subject to $|S| = s$
  $j \leftarrow 0$
  For $i = 1, \ldots, s + s^{'}$ do
    if $i \in S$ then
      $dt[i] \leftarrow^{R} \{0,1\}^n$
      $tg[i] \leftarrow^{R} \{0,1\}^l$
      $\mathrm{Pkt}[i] \leftarrow (dt[i], tg[i])$
    else
      $j \leftarrow j + 1$
      $tg[i] \leftarrow F(K, m_j)$
      $\mathrm{Pkt}[i] \leftarrow (m_j, tg[i])$
    EndIf
  EndFor
  Return Pkt[1],Pkt[2],...,Pkt[$s + s^{'}$]

Algorithm 2 AONT Scheme Decryption

Algorithm $D^{F(K,\cdot)}(\text{Pkt}[1],\ldots,\text{Pkt}[s+s^{'}])$
  For $i = 1,\ldots,s+s^{'}$ do
    Parse Pkt_i as (dt, tg)
    If $F(K, dt) = tg$ then
      $m_i \leftarrow dt$
    EndIf
  EndFor
  $M \leftarrow \text{AONT}^{-1}(m_1 \| m_2 \| \ldots \| m_s)$
  Return M

4.2.2 Hybrid Encryption schemes

As mentioned previously, a hybrid cryptosystem combines public-key and private-key encryption techniques. They are typically constructed as follows:

- Generate a private key, K, at random.
- Encrypt the plaintext using K and a symmetric-key cryptosystem, to produce the ciphertext C.
- Encrypt K using a public-key cryptosystem, and a publicly known key $K^{'}$, to produce the ciphertext $C^{'}$.
- Send C and $C^{'}$ to the intended recipient, who will decrypt $C^{'}$ using their private key which corresponds to $K^{'}$, and then decrypt C using K.

However, in Chaffing and Winnowing there is no encryption key that we are trying to keep secret, it is the location of the chaff packets that we do not want an adversary to discover. Therefore, we can adapt the 'scattering scheme' as follows: apply an AONT to the input message and generate the positions of the chaff packets as before, then use a public-key encryption system to encrypt the location of the chaff packets using a public key, and write the result to file. Next for each block of the output, if the index is a chaff position, a fake block is generated and written to file, otherwise the next wheat block is output. The data to be sent now contains the encrypted chaff locations, along with the transformed and 'chaffed' message. It is no longer necessary to calculate and send MACs for each block, as the recipient is able to determine which packets are wheat and which are chaff by decrypting the chaff packet locations using the public-key encryption system and their corresponding private-key. Once the locations have been decrypted, the 'winnowing' process simply involves going through each block of the message and discarding packets that correspond to chaff positions. This approach greatly improves performance, since we are no longer calculating and sending MACs, and the data expansion of the ciphertext is now constant since the only extra data to be added is the fixed amount of chaff. It could be argued that the use of a public-key encryption system does not make this a Chaffing and

Winnowing scheme, and it could be classed as encryption. However, the original message is still sent 'in the clear', and it is only the chaff locations that are encrypted. The following algorithms show the process of 'encryption' and 'decryption' in our Hybrid Chaffing and Winnowing schemes, where $K^{'}$ is a public-key, K is the corresponding private-key, and P_ke and P_kd are the public-key encryption and decryption functions respectively.

Algorithm 3 Hybrid Scheme Encryption

Algorithm $\varepsilon(M)$
  $M^{'} \leftarrow \mathsf{AONT}(M)$
  Parse $M^{'}$ as $m_1 \| m_2 \| \ldots \| m_s$ where $|m_i| = n$
  Pick $S \subseteq \{1, \ldots, s + s^{'}\}$ at random subject to $|S| = s$
  $S^{'} \leftarrow P_ke(K^{'}, S)$
  $j \leftarrow 0$
  For $i = 1, \ldots, s + s^{'}$ do
    if $i \in S$ then
      $\mathrm{Pkt}[i] \leftarrow^{R} \{0,1\}^n$
    else
      $j \leftarrow j + 1$
      $\mathrm{Pkt}[i] \leftarrow m_j$
    EndIf
  EndFor
  Return $S^{'}$,Pkt[1],Pkt[2],...,Pkt[$s + s^{'}$]

Algorithm 4 Hybrid Scheme Decryption

Algorithm $D(S^{'}$,Pkt[1],...,Pkt[$s + s^{'}$])
  $S \leftarrow P_kd(K, S^{'})$
  $j \leftarrow 0$
  For $i = 1, \ldots, s + s^{'}$ do
    if $i \in S$ then
      $j \leftarrow j + 1$
    else
      $m_i \leftarrow$ Pkt[i]
    EndIf
  EndFor
  $M \leftarrow \mathsf{AONT}^{-1}(m_1 \| m_2 \| \ldots \| m_s)$
  Return M

4.3 Module Design

We now look at the design of the necessary modules for our Chaffing and Winnowing schemes. These are created as C libraries so that they can easily be reused across more than one implementation.

4.3.1 MAC Algorithm

When Rivest (1998a) first presented the concept of Chaffing and Winnowing, he suggested using HMAC as the MAC algorithm, so this is also what we have chosen to use. HMAC makes use of a cryptographic hash function and a secret key; see section 2.4 for the full construction. The hash function is used in such a way that it can easily be replaced if a faster or more secure hash function is developed. To ensure the security of HMAC, it is advisable to perform periodic key refreshement (Bellare et al., 1996). This is especially important within a Chaffing and Winnowing scheme since if an adversary is able to break the MAC algorithm, then they can easily identify which packets are chaff. In our implementations we assume that the participants have previously agreed on their secret authentication key using a standard protocol such as Diffie-Hellman. Sample code for HMAC is provided by Krawczyk et al. (1997), which we have made use of in our implementation.

The cryptographic hash function that we have chosen to use as part of HMAC is MD5. SHA-1 is thought to be a more secure function, however MD5 is more suitable where superior performance is required, as in our case. SHA-1 also has a longer output than MD5, which is undesirable as we wish to minimise ciphertext expansion. The implementation of MD5 that we have used was provided by RSA Data Security, Inc[1].

4.3.2 All-Or-Nothing Transforms

The AONTs that we have chosen to implement are the Package Transform, CTRT, and Optimal Asymmetric Encryption Padding. The Package Transform and CTRT have not been proven to meet the strong definition of AONT security provided by Boyko (1999), unlike OAEP. However, these proofs have concentrated on whether the transform is secure against exhaustive key search which, as stated by Clayton and Danezis (2003), is "not quite the way in which the all-or-nothing transform is being used within a chaffing and winnowing system". Desai (2000) also states that having a weaker characterisation of AONTs allows us to build more efficient constructions that meet it. It was decided that the BEAR block cipher, mentioned in section 2.2.4, would not be implemented since it has not been shown to be an AONT, and it makes use of two hashes and a stream cipher therefore it is likely to be slower. The design of the three different transforms being used is outlined below.

Package Transform

We have used the description of the Package Transform as presented by Rivest (1997); see section 2.2.1 for more details of the algorithm. The Package Transform makes use of a fixed public-key, and a key K' which is chosen at random each time the transform is used. This randomness is important for security as it ensures that performing the transform twice on the same input message will result in two different outputs. We generate values for K'

[1] `http://people.csail.mit.edu/rivest/Md5.c`

which are 128-bits in length, as recommended by Rivest.

The block-cipher that we have chosen to incorporate within the Package Transform is RC5, again as suggested by Rivest, however any other block-cipher could have been chosen instead. The reference implementation of RC5-32/12/16 given in (Rivest, 1996) has been used, which applies RC5 on 64-bit blocks, using 12 rounds and a 128-bit key.

CTRT

We have constructed CTRT using the description given by Desai (2000). As mentioned in section 2.2.2, CTRT is very similar to the Package Transform however the last block is simply the XOR of the key $K^{'}$ with all of the previous output blocks, as opposed to the XOR of $K^{'}$ with a hash of all the previous output blocks. This should mean that CTRT is more efficient than the Package Transform, since one 'pass' is skipped altogether. To retain consistency, we have also used RC5 as the block-cipher in our CTRT implementation. The importance of the randomness of $K^{'}$ is also applicable here.

Optimal Asymmetric Encryption Padding

The design of our OAEP algorithm has been taken from the Public Key Cryptography Standards (PKCS) #1 v2.1 (RSA Security Inc., 2002), which is based on the original definition of OAEP given by Bellare and Rogaway (1995). It makes use of a hash function and a Mask Generation Function (MGF). The MGF outputs a pseudorandom string of a specified length, and given one part of the output but not the input it should be infeasible to predict another part of the output. The MGF we are using is MGF1, again taken from the PKCS, which also makes use of a hash function. Since we have been already been using the MD5 hash function, it is used again here as part of our OAEP implementation.

The encoding process works as follows: first the length of the input message, $mLen$, is checked to ensure that it is not too long. Then a data block DB is constructed as the concatenation of $pHash$ (the hash of a passphrase P), with a variable amount of zero padding PS, the value '01', and the input message M. A *seed* is then generated at random and used to seed the MGF, which outputs a random string that we XOR with the data block to produce a masked data block. This masked data block is then used to seed the MGF, the output of which is XOR'd with the previous random *seed* to produce a masked seed. The output of the algorithm is an encoded message EM, of length $emLen$, which consists of the masked seed concatenated with the masked data block.

The decoding process first checks the structure of the input message and outputs a decoding error if it is invalid. Then the *seed* is recovered by XORing the masked seed with the output of the MGF, which has been seeded with the masked data block. The *seed* is then used again to seed the MGF, and the output is XOR'd with the masked data block to recover the data block DB. We then recalculate $pHash$ and check if it matches the value

received in the data block. If so, the decoding was successful and the original message M is output.

The following algorithms show the full process of OAEP encoding and decoding:

Algorithm 5 OAEP Encoding

Algorithm $\varepsilon^{Hash(\cdot),MGF(seed,length)}(M, P, emLen)$
  If $mLen > emLen - 2hLen - 1$
    output "message too long"
    Return
  EndIf
  $pHash \leftarrow Hash(P)$
  $PS \leftarrow \{0\}^{emLen-mLen-2hLen-1}$
  $DB \leftarrow pHash \| PS \| 01 \| M$
  $seed \leftarrow^R \{0,1\}^{hLen}$
  $dbMask \leftarrow MGF(seed, emLen - hLen)$
  $maskedDB \leftarrow DB \oplus dbMask$
  $seedMask \leftarrow MGF(maskedDB, hLen)$
  $maskedSeed \leftarrow seed \oplus seedMask$
  $EM \leftarrow maskedSeed \| maskedDB$
  Return EM

Algorithm 6 OAEP Decoding

Algorithm $D^{Hash(\cdot),MGF(seed,length)}(M, P, emLen)$
  If $emLen < (2 * hLen) + 1$
    output "decoding error"
    Return
  EndIf
  Parse EM as $maskedSeed \| maskedDB$ where $|maskedSeed| = hLen$ and
  $|maskedDB| = emLen - hLen$
  $seedMask \leftarrow MGF(maskedDB, hLen)$
  $seed \leftarrow maskedSeed \oplus seedMask$
  $dbMask \leftarrow MGF(seed, emLen - hLen)$
  $DB \leftarrow maskedDB \oplus dbMask$
  $pHash \leftarrow Hash(P)$
  Parse DB as $pHash' \| PS \| 01 \| M$ where $|pHash'| = hLen$ and $PS \in 0^*$
  If $pHash \neq pHash'$
    output "decoding error"
    Return
  EndIf
  Return M

4.3.3 Public-key Encryption

The hybrid Chaffing and Winnowing schemes require the use of a public-key encryption system. Our requirements for this were to choose a technique that is common, has a high level of security and performance, and low data expansion. RSA (Rivest, Shamir and Adleman, 1978) is currently the most widely used public-key cryptosystem. Its security is based on the difficulty of the integer factorisation problem. ElGamal (ElGamal, 1985), whose security depends on the difficulty of computing discrete logarithms, is also widely used. Analysis has shown that both systems have similar security for equivalent key lengths, however ElGamal is slower and causes ciphertext expansion by a factor of two. Therefore we have chosen to use RSA. Due to time constraints, an existing implementation of RSA in C has been taken from the open source cryptographic library XySSL[2].

4.4 Experiment Design

We will now discuss what existing encryption techniques our Chaffing and Winnowing schemes are being compared to, and the metrics being used in the comparison.

4.4.1 Comparison Encryption techniques

The OpenPGP standard[3] uses a combination of strong public-key and symmetric-key cryptography to provide security for electronic communications and data storage. The GNU Privacy Guard (GnuPG) is a complete and free implementation of the OpenPGP standard, that provides a command line tool to support encryption and decryption using many popular cryptographic algorithms. By default the algorithms are executed using hybrid encryption, however it is also possible to specify that they are executed using symmetric-key encryption. This suits our purposes as we can compare the hybrid methods against our hybrid Chaffing and Winnowing schemes, and the symmetric methods against our AONT schemes. The GnuPG algorithms are also implemented in C which allows us to make a fairer comparison with our systems. We will be testing our Chaffing and Winnowing schemes against Triple-DES, and AES which were described in section 2.5.1.

4.4.2 Metrics

In order to test the performance of our schemes and the existing encryption algorithms, each one will be executed several times and the average running time will be calculated. We will be timing both encryption and decryption, and a range of different file sizes will be used.

Measuring the ciphertext expansion involves simply recording the file sizes before and after

[2] `http://xyssl.org/code/source/rsa/`
[3] `http://tools.ietf.org/html/rfc4880`

encryption and calculating the difference. Again, a range of file sizes will be used, however this test only requires each algorithm to be run once as the expansion will be the same each time.

4.4.3 Expected outcome

It is expected that the hybrid Chaffing and Winnowing schemes will perform much better than our AONT schemes, as the MAC calculation in the AONT schemes carries a high performance overhead. We predict that all schemes will be comparable in speed for smaller file sizes, however the existing encryption techniques should be noticably faster than the AONT schemes for larger file sizes. This is because a MAC must be calculated for each block of the output, the number of which increases with the size of the file.

We expect that the ciphertext expansion for our AONT schemes will be significantly greater than for our hybrid schemes, and also the existing techniques, due to the amount of authentication data that is added. The hybrid schemes will have a slightly greater ciphertext expansion than AES and 3DES, due to the chaff data that is added, but it will not be dependent on the size of the plaintext.

4.5 Specification of Implementations

The following specification provides a summary of what has been implemented, in relation to the design discussed above. A more detailed description of our implementations is provided in the next chapter.

4.5.1 Libraries

- HMAC, using the MD5 hash function.
- The Package Transform, as an AONT, using the RC5 block cipher.
- CTRT, as an AONT, using the RC5 block cipher.
- OAEP, as an AONT, using the MD5 hash function, and the MGF1 mask generation function.
- RSA, for public-key encryption, using code provided by the open source cryptographic library XySSL.

4.5.2 Chaffing and Winnowing schemes

- AONT scheme, using the Package Transform and HMAC libraries.

- AONT scheme, using the CTRT and HMAC libraries.
- AONT scheme, using the OAEP and HMAC libraries.
- Hybrid scheme, using the Package Transform and RSA libraries.
- Hybrid scheme, using the CTRT and RSA libraries.
- Hybrid scheme, using the OAEP and RSA libraries.

Chapter 5

Implementation and Testing

The following chapter gives a more detailed description of the implementation of our schemes, including any problems that occurred or decisions that were made. Since system testing was carried out after each component was implemented, we have also incorporated details of how that was conducted in this section. Source code listings of the key components can be found in Appendix B, and full source code is provided on the accompanying CD.

5.1 Platforms and Compilers

As mentioned previously, the two platforms available for testing are UNIX and Windows. Full details of each platform, and the compilers being used are as follows:

- Microsoft Windows XP Professional running on AMD Athlon 2.01GHz CPU with 1 GB of RAM.
- Solaris 8.0 running on Sun Ultra2 220 with 2x200Mhz UltraSparc CPUs with 1 GB of RAM.
- The compiler being used is the same on both platforms: GCC version 3.4.3.

5.2 Testing Criteria

The testing criteria that we want our implementations to satisfy are as follows:

- Applying any AONT to the same message twice must result in different outputs.
- Each scheme must be able to successfully 'encrypt' and 'decrypt' on both platforms. This means that applying the 'chaffing' process to a file and then 'winnowing' it must reproduce the original input.

- For each scheme, a file 'encrypted' on Windows must successfully 'decrypt' on UNIX, and vice versa.

5.3 Libraries

In order to create our Chaffing and Winnowing schemes, it was necessary to implement the underlying components first.

5.3.1 Chaffing and Winnowing Library

A library was created to hold the functions that would be required across all of our Chaffing and Winnowing implementations. A function was created to generate a random chaff packet of a specified size. It is important that the chaff packet contents are different from each other and different for each message sent, otherwise it would be easy for an adversary to spot them. In order to ensure this, the C rand() function was used. To make sure that different values are produced each time the program is executed, it was necessary to seed the random number generator. This was done by initialising the seed to be a value representing the second in which the program begins execution. To verify that this function worked as intended, we generated several chaff packets and output the contents to see if they were distinct. This was performed several times in order to confirm that the same packets were not produced upon each execution.

Another function was added to this library to randomly calculate the positions of the chaff packets, depending on the total number of blocks in the output message and the required number of chaff packets. When doing this, it was necessary to ensure that no positions were duplicated, and that the positions were returned in ascending order. To sort the positions, the built-in qsort() function in C was used. The calculateChaffPositions() function was tested on both platforms by running it several times, printing the output, verifying that no duplicates occurred and that the positions were sorted correctly.

As discussed in section 2.6, endianness needs to be considered when integer values are sent from one machine to another. This causes problems for us in our hybrid schemes, since we need to send the chaff packet locations as part of the message, and these are stored as integers. Therefore, we need to ensure that the values are always output in the same byte-order - we have chosen big-endian since that is the standard format for sending information over a network. To meet this requirement, a function was created to determine the endianness of the machine that the program is currently running on. The code for this was found at `http://c-faq.com/misc/endiantest.html`. This was tested on each platform and correctly determined that the UNIX platform was big-endian, and the Windows platform was little-endian. Additionally, a function was added to reverse the ordering of bytes in an integer. The code for this was found at `http://www.ibm.com/developerworks/aix/library/au-endianc/index.html?ca=drs-`. This was also tested

successfully by printing out the value of an integer before and after the byte order had been swapped.

5.3.2 HMAC

HMAC was implemented using the sample code given in (Krawczyk et al., 1997), and the implementation of MD5 provided by RSA Data Security, Inc[1]. The sample code contained the deprecated methods bzero() and bcopy(), so these were simply replaced with memset() and memcpy(). It was also noticed that the initialisation of the inner and outer pads was performed each time a call to HMAC was made. Since our implementations would be calculating MAC codes several times for each input file using the same key, it was not necessary for these values to be re-initialised each time. Therefore, to improve efficiency, an hmac_init() method was created to do this which then only needed to be called once.

In order to test our implementation of HMAC, the test vectors provided in (Krawczyk et al., 1997) were used. These tests passed successfully on both platforms.

5.3.3 Package Transform

As mentioned previously, the Package Transform was implemented using the reference implementation of RC5-32/12/16, which means that the word size is 32-bits, the number of rounds is 12, and the key size is 16 bytes. Rivest (1996) states that this choice of parameters is "nominal" for both speed and security. For a word size of 32-bits, RC5 has 64-bit plaintext and ciphertext block sizes. Therefore the block size of the Package Transform was also chosen to be 64-bits, since the output of the block-cipher must be XOR'd with the input message blocks.

It was necessary to modify RC5 so that it could be used with two different keys, as required by the Package Transform. To do this, we simply removed the global key table array, and modified the RC5_SETUP(), RC5_ENCRYPT() and RC5_DECRYPT() functions to accept a pointer to a key table. This allowed us to create the key tables independently for each key, and pass in a pointer to the table corresponding to the particular key that needed to be used as necessary. The public-key used in the transform was fixed, however another key needed to be generated at random whenever the transform is applied. We achieved this using the same technique to generate a chaff packet that was described above. This randomness ensures that applying the transform twice to the same input will result in different outputs. Computing the last block of the Package Transform involved XOR-ing the 128-bit key with the 64-bit hash of each pseudo-message block. To do this we simply XOR'd the hash of each block with each 'half' of the key to produce the 128-bit final block.

To test that the Package Transform worked correctly, a file was transformed, inverted

[1] http://people.csail.mit.edu/rivest/Md5.c

and then compared to the original to check that they were the same. Then to verify that transforming the same message more than once resulted in different outputs, the UNIX cmp command was used to compare two transformed files. This was performed ten times to ensure that each file appeared to be different. All tests passed on both platforms. Cross-platform testing was not conducted as this falls under the testing of our full Chaffing and Winnowing schemes.

5.3.4 CTRT

The implementation of CTRT was straightforward since it is a simplified version of the Package Transform. The block size was again set to be 64-bits, the only major change made was to compute the last block to be the key XOR'd with all the previous pseudo-message blocks, instead of a hash of the previous psuedo-message blocks.

To check that CTRT worked as expected, the same tests were performed as for the Package Transform. All tests passed on both platforms.

5.3.5 OAEP

OAEP is different from the previous two transforms in that it does not split the input message into blocks, it works on the message as a whole. Figure 5.1 shows how the encoded message is constructed. A further difference is that OAEP takes an extra parameter as input, used to specify the length of the encoded message. As mentioned previously, the Mask Generation Function that we have implemented as part of OAEP is MGF1 (RSA Security Inc., 2002). This works as follows:

1. Let T be the empty octet string
2. For *counter* from 0 to $\lceil maskLen/hLen \rceil - 1$, do the following:
   - Convert *counter* to an octet string C of length 4 octets
   - Let $T = T \parallel \text{Hash}(mgfSeed \parallel C)$
3. Output *mask* as the leading $maskLen$ octets of T

where $hLen$ is the length of the output of the hash function (MD5), $maskLen$ is the desired length of the output for the MGF, and $mgfSeed$ is a seed that is provided as input.

Part of the OAEP algorithm involves generating a random string to be used as a seed for the MGF. This was done in the same way that we generated a key at random for the Package Transform and CTRT. Again, this randomness is important for security since it prevents patterns appearing in the output which an adversary may be able to spot.

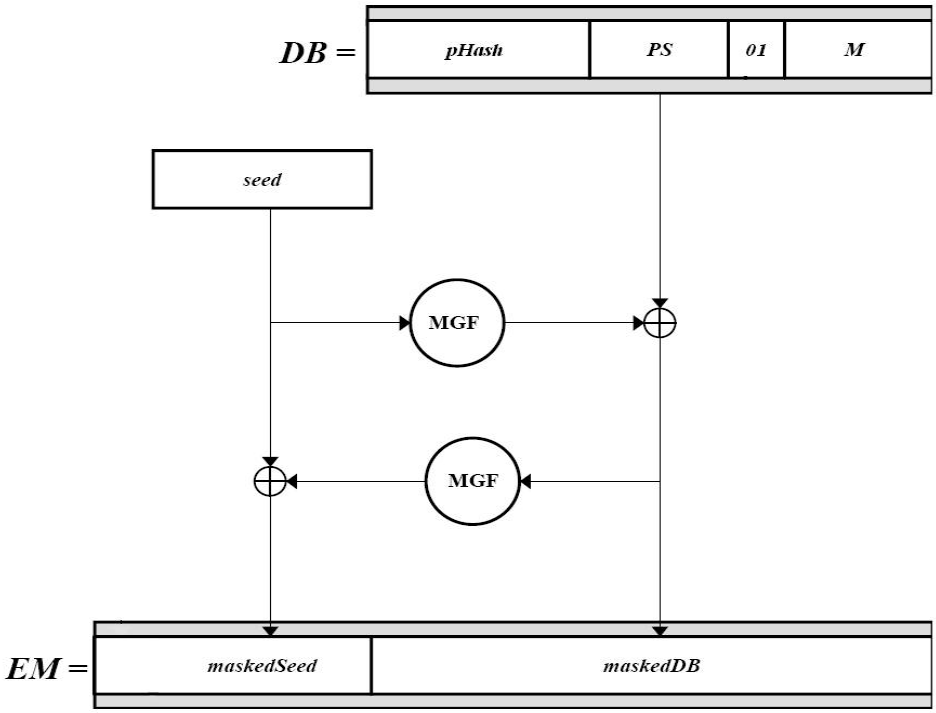

Figure 5.1: Construction of Encoded Message in OAEP

To check that OAEP worked correctly, the same tests were performed as for the previous two transforms. All tests passed on both platforms.

All three AONTs were implemented with the functions transform() and inverse_transform() in order to make them easy to interchange with one another. The only difference in the function prototypes is that the OAEP transform() function requires one extra parameter to specify the length of the encoded message.

5.3.6 RSA

The implementation of RSA made use of source code provided by XySSL. As well as the code for RSA itself, it was also necessary to include the library to support multi precision arithmetic[2], since RSA works with very large numbers. To check that the implementation was set up correctly, the rsa_self_test() function was tested on both platforms. This was successful.

[2] http://xyssl.org/code/source/bignum/

5.4 Chaffing and Winnowing Schemes

Once the underlying components had been created and tested, the implementation of the complete Chaffing and Winnowing schemes was fairly straightforward.

5.4.1 Symmetric Schemes

All three symmetric schemes were implemented in the same way for each AONT. The first step in the 'chaffing' process was to apply the AONT to the input file and store the transformed version in a temporary file. We then determine the number of blocks in the transformed file, and calculate the positions of the chaff packets using the function created in our Chaffing and Winnowing library. Rivest (1998a) suggests splitting the transformed message into 1024-bit blocks, therefore that is the block size that we have used. For each output block, if the index is a chaff position a random chaff packet is generated and written to the output file along with a fake MAC, otherwise we calculate the MAC for the next wheat block and write it to the output file along with the valid packet. The 'winnowing' process simply reads in each block of the input file, checks the MAC for each one, and writes the valid packets to a temporary file. The inverse AONT is then applied to the temporary file and the original message is output.

Once the symmetric schemes had been implemented, each one was tested to verify that 'encrypting' a file and then 'decrypting' it reproduced the original data. Each scheme passed this test on both platforms. Additionally, each implementation was tested for cross-platform compatibility. To do this, a file 'encrypted' on UNIX, was 'decrypted' on Windows, and vice versa. This test was passed successfully by all schemes.

5.4.2 Hybrid Schemes

All three hybrid schemes were also implemented in the same way for each AONT. The first step in the 'chaffing' process was again to apply the AONT to the input file and store the transformed version in a temporary file. Then once the chaff packet locations have been calculated, the isBigEndian() function is called from the Chaffing and Winnowing library. If this returns false, the values of the chaff positions are transformed into big-endian format. Next, the chaff packet locations are encrypted using the RSA library and written to the output file. They are sent before the 'chaffed' message so that they can be easily located by the recipient. Finally, for each output block, if the index is a chaff position a random chaff packet is generated and written to the output file, otherwise the next wheat block is output. The 'winnowing' process first decrypts the chaff packet locations, and converts them back into little-endian format if necessary. Then for each block in the input message, the packets corresponding to chaff locations are discarded and the wheat packets are written to a temporary file. The inverse AONT is then applied to this file to reproduce the original plaintext.

Once the hybrid schemes had been implemented as described above, it was noticed that this technique provides privacy but does not provide any data integrity or authentication. The recipient would not be able to verify that the encrypted chaff positions were really sent by the claimed sender. An adversary could potentially modify the message to contain chaff packet locations that they had generated themselves and encrypted using the sender's public key. In order to solve this problem, our hybrid implementations were modified to calculate a MAC (using HMAC as before) based on the chaff packet positions and then output this in between the encrypted chaff locations, and the 'chaffed' and transformed plaintext. Now when 'winnowing', if the MAC for the chaff positions is not valid, the recipient knows that the data has either been corrupted or tampered with and can request for the message to be sent again.

The hybrid schemes were tested for correctness and cross-platform compatibility in the same way as described for the symmetric schemes. All tests were passed for each scheme.

Chapter 6

Experiments and Results

In this chapter we present and analyse the results of the experiments that were performed to compare our Chaffing and Winnowing schemes both with each other, and with existing encryption techniques.

6.1 Experiments conducted

As discussed previously, the purpose of the experiments carried out was to see how our schemes compare with existing techniques in terms of speed of execution and the data expansion of the ciphertext. We chose to compare our schemes to the GnuPG implementations of Triple-DES and AES, using the hybrid encryption mode when comparing against our hybrid Chaffing and Winnowing schemes. When using GnuPG, there is an option available to compress the file before encryption. This was turned off for our experiments to allow for a fairer comparison. Files of ten different sizes were used in our tests, the approximate values of which were: 10KB, 50KB, 100KB, 250KB, 500KB, 1MB, 7MB, 25MB, 50MB, and 100MB. All tests were carried out on the Windows platform, using the cygwin environment.

6.1.1 Execution time

In order to determine the execution time of our schemes, the UNIX time command was used. The time taken both to encrypt and decrypt was recorded for each implementation, so that we could see if one operation was slower than the other. Files of each size mentioned above were encrypted and decrypted fifteen times for every scheme, and the average execution times were calculated.

6.1.2 Ciphertext expansion

Measuring the ciphertext expansion involved simply encrypting all of the files once for each scheme and recording the size of the output file. This is because the expansion does not change when encrypting the same file more than once.

6.2 Experiment results

We now look at the results that were recorded. The results for our symmetric Chaffing and Winnowing schemes have been grouped with the results for symmetric Triple-DES and AES, and the hybrid Chaffing and Winnowing scheme results have been grouped with the results for hybrid Triple-DES and AES. Each table shows the exact file sizes in bytes.

6.2.1 Execution time

Tables 6.1 and 6.2 show the timings for encryption given to four decimal places. The results for decryption timings can be found in Appendix A.1. It was found that the speeds for decryption were very similar to those for encryption.

Table 6.1: Average encryption times for symmetric schemes

File size (bytes)	CW PT	CW CTRT	CW OAEP	3DES	AES
10752	0.0577	0.0455	0.0569	0.042	0.0407
51595	0.0468	0.0511	0.0659	0.044	0.0413
107985	0.0621	0.062	0.0664	0.048	0.046
256895	0.0892	0.0796	0.1152	0.0613	0.0473
545730	0.1299	0.1267	0.1725	0.0827	0.0627
1244182	0.2683	0.2361	0.3122	0.1487	0.1
7471298	1.3901	1.1796	1.5765	0.688	0.386
27929130	5.1107	4.2356	6.3068	2.4467	1.3407
53263590	9.6943	8.0682	11.4473	4.644	2.5307
103811754	18.4879	15.6133	24.654	8.976	4.832

6.2.2 Ciphertext expansion

Tables 6.3 and 6.4 show the *difference* in bytes between the original file, and the encrypted file for each scheme.

Table 6.2: Average encryption times for hybrid schemes

File size (bytes)	Hybrid PT	Hybrid CTRT	Hybrid OAEP	Hybrid 3DES	Hybrid AES
10752	0.0507	0.0537	0.0499	0.0447	0.0453
51595	0.0517	0.0541	0.0592	0.052	0.0493
107985	0.0612	0.0641	0.0631	0.0553	0.0513
256895	0.0736	0.0675	0.0871	0.07	0.056
545730	0.1123	0.0989	0.1369	0.0887	0.066
1244182	0.2056	0.1508	0.2361	0.1593	0.106
7471298	0.9213	0.7306	1.0733	0.7447	0.3847
27929130	3.3682	2.5555	3.9182	2.6713	1.3313
53263590	6.3744	4.8493	7.467	5.018	2.5073
103811754	12.311	9.749	14.186	9.7473	5.05

Table 6.3: Ciphertext expansion for symmetric schemes

File size (bytes)	CW PT	CW CTRT	CW OAEP	3DES	AES
10752	19776	19776	19920	53	87
51595	24869	24869	25013	48	82
107985	31839	31839	31983	49	81
256895	50545	50545	50545	61	93
545730	86574	86574	86718	57	89
1244182	173930	173930	174074	48	80
7471298	952270	952270	952414	53	85
27929130	3509526	3509526	3509670	53	85
53263590	6676266	6676266	6676410	53	85
103811754	12994854	12994854	12994998	55	87

Table 6.4: Ciphertext expansion for hybrid schemes

File size (bytes)	Hybrid PT	Hybrid CTRT	Hybrid OAEP	Hybrid 3DES	Hybrid AES
10752	19712	19712	19840	337	345
51595	19701	19701	19829	332	340
107985	19631	19631	19759	331	339
256895	19713	19713	19713	343	351
545730	19646	19646	19774	339	347
1244182	19690	19690	19818	330	337
7471298	19646	19646	19774	335	343
27929130	19670	19670	19799	335	343
53263590	19610	19610	19738	335	343
103811754	19670	19670	19798	337	345

6.3 Results analysis

The following section presents an analysis of our results to see if they were as we expected. Several graphs have been produced to allow trends to be seen more easily.

6.3.1 Execution time

Firstly we will look at the symmetric schemes. Figures 6.1 and 6.2 show the execution time results, for small files and large files respectively. The graphs have been scaled appropriately, with the file sizes shown in Kilobytes and the timings shown in seconds. Looking at these graphs, we can see that OAEP was the slowest of the AONT Chaffing and Winnowing schemes, followed by the Package Transform, with CTRT being the fastest. It was expected that this would be the case, since CTRT involves less steps than the other two AONTs. As can be seen in Figure 6.1, for files of approximately 500KB in size, the difference between the execution times of each technique is at most one tenth of a second, and for very small files the difference is minimal. However as the file sizes increase in Figure 6.2, the difference between the speed of the Chaffing and Winnowing schemes compared to Triple-DES and AES increases greatly - up to as much as several seconds. This was expected, since in the Chaffing and Winnowing implementations, the larger the input file is, the more MAC calculations that are needed, which increases the performance overhead.

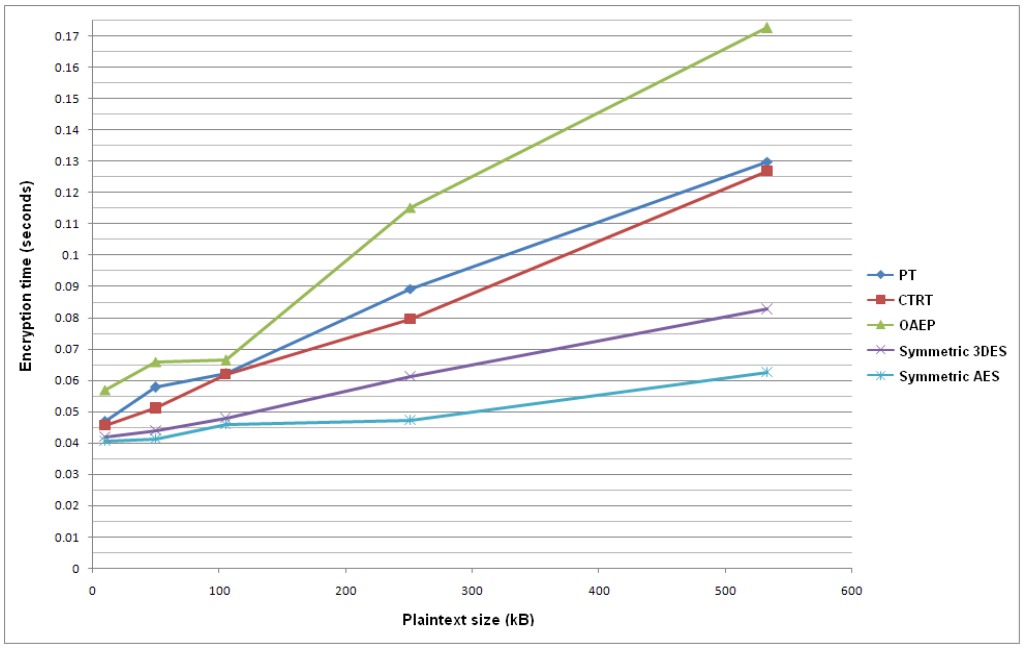

Figure 6.1: Symmetric scheme encryption times for small files

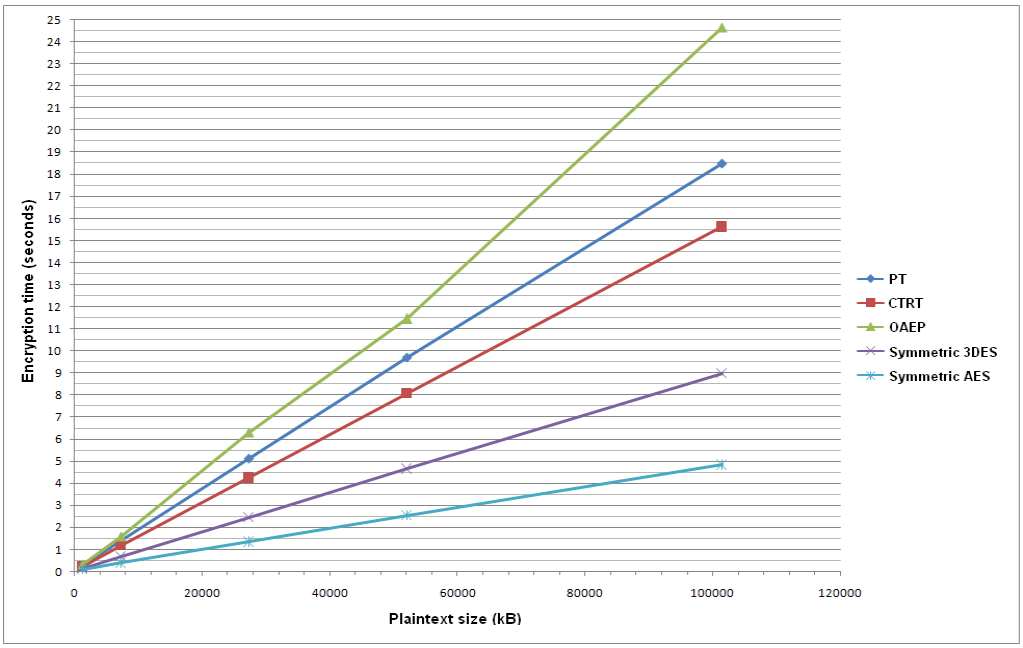

Figure 6.2: Symmetric scheme encryption times for large files

Figures 6.3 and 6.4 show the encryption times of the hybrid schemes, again for small and large file sizes respectively. Firstly we notice that the hybrid Chaffing and Winnowing schemes are significantly faster than the symmetric Chaffing and Winnowing schemes. This was expected since the hybrid schemes do not require a MAC to be calculated for each output block. Looking at the graphs, we can see that again the hybrid OAEP implementation is the slowest of the hybrid Chaffing and Winnowing schemes, with hybrid CTRT being the fastest. However the main thing we notice here is that the difference in execution times between the existing encryption techniques and the Chaffing and Winnowing implementations is much less than for the symmetric schemes. Again this is due to the fact that the number of MAC calculations required for the symmetric schemes increases with the size of the input file. Looking at Figure 6.3, for files of size 500KB or less the difference in speed between all techniques is minimal - no more than a few hundredths of a second. We can see in Figure 6.4 that for the larger file sizes, the encryption times for our hybrid CTRT scheme are almost identical to those for hybrid Triple-DES, with the hybrid Package Transform scheme also performing at a similar level. For files of more than 10MB in size hybrid AES is noticably faster than the other techniques, which was expected since AES is known to be fast.

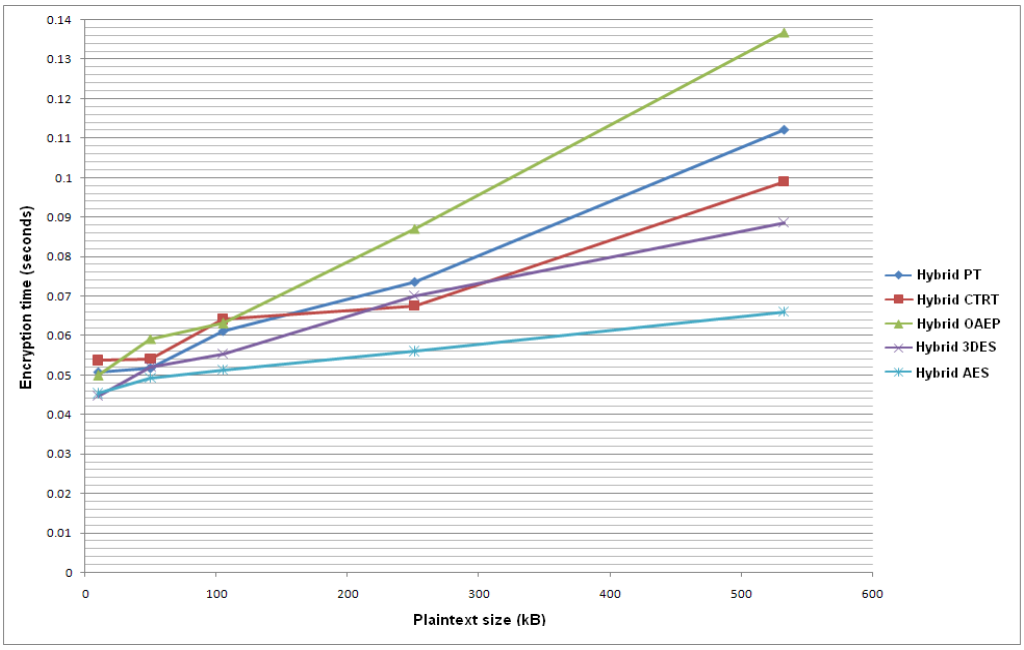

Figure 6.3: Hybrid scheme encryption times for small files

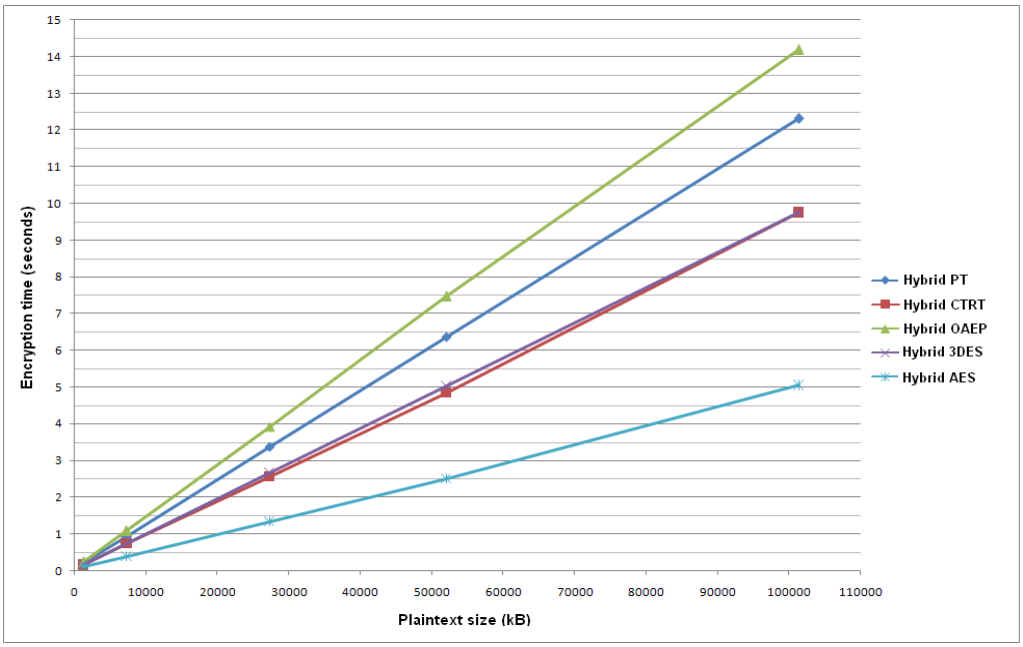

Figure 6.4: Hybrid scheme encryption times for large files

6.3.2 Ciphertext expansion

Figures 6.5 and 6.6 show the *percentage* increase in the size of the ciphertext compared to the original file size. Tables containing this data can be found in Appendix A.2. Since the ciphertext expansion for each AONT Chaffing and Winnowing scheme was roughly the same, and the expansion for each hybrid Chaffing and Winnowing scheme was roughly the same, only one of each has been represented in our graphs. In Figure 6.5 we can see that for smaller files, the percentage increase in the ciphertext size is substantially larger for the Chaffing and Winnowing schemes compared to the existing encryption techniques. This is due to the fact that a fixed amount of chaff data is added no matter what size the input file is. Figure 6.6 shows that as the file size gets larger, the data expansion for the symmetric Chaffing and Winnowing schemes converges to an increase of approximately 12%, whilst the percentage increase for the hybrid Chaffing and Winnowing schemes continues to fall until the values are roughly the same as for the existing encryption techniques. This is as expected, since the hybrid schemes have a fixed ciphertext expansion, whereas the symmetric schemes require more authentication data as the size of the input message increases. All of the existing encryption techniques have a fixed data expansion overhead, however for the Chaffing and Winnowing schemes, we can see that the overhead becomes more cost effective as the input message gets larger.

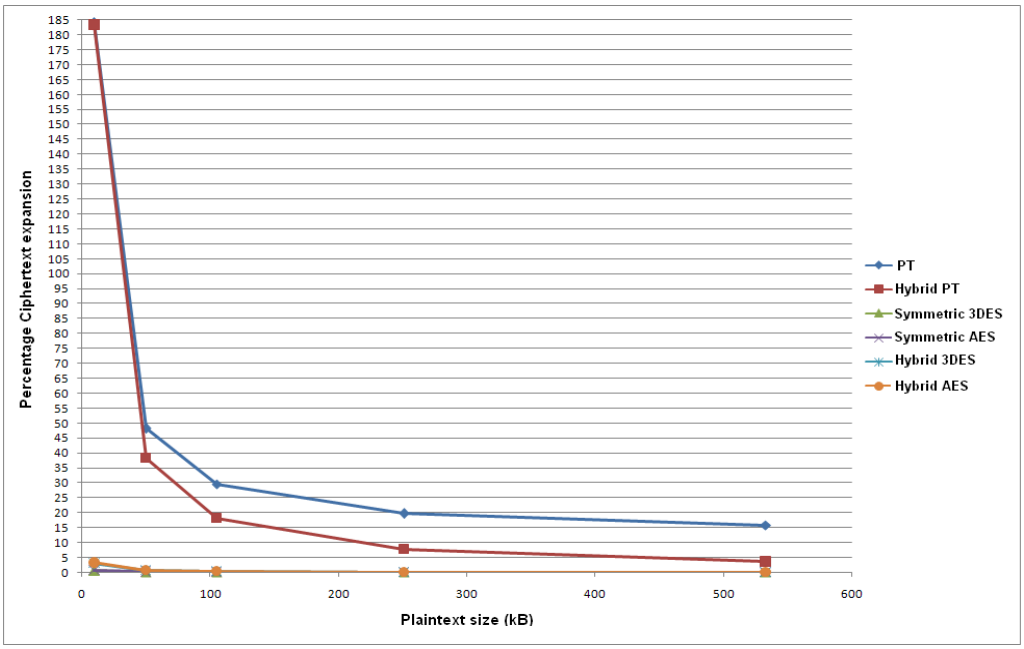

Figure 6.5: Percentage ciphertext expansion for small files

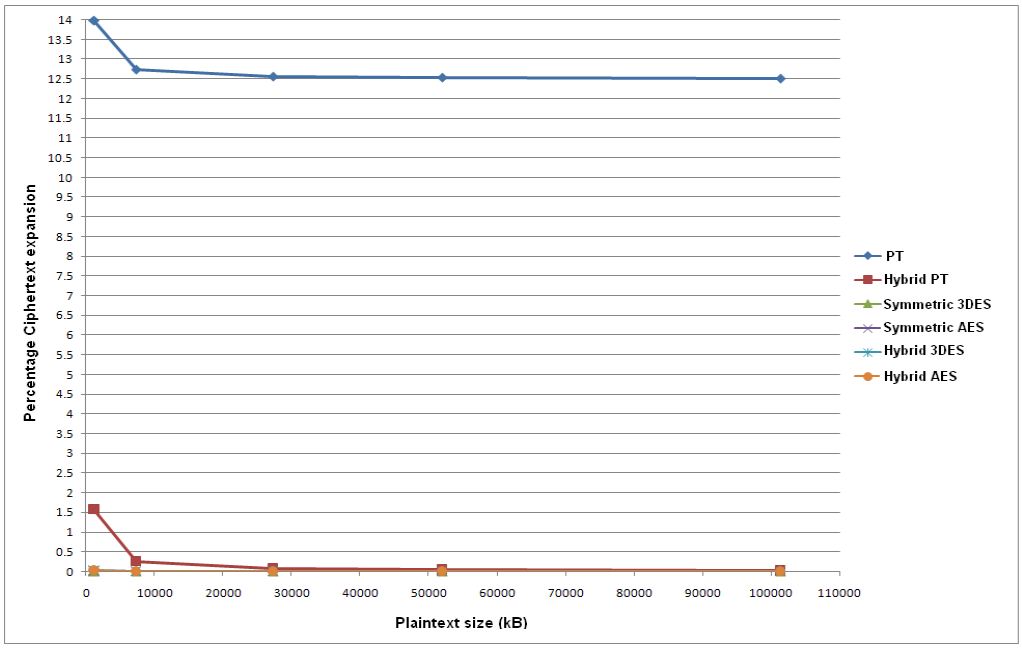

Figure 6.6: Percentage ciphertext expansion for large files

Chapter 7

Conclusion

This chapter provides an overall conclusion of the project, identifying the achievements made, areas that could be improved on, and any future work that could be carried out.

7.1 Project Conclusion

This project has produced three symmetric Chaffing and Winnowing schemes using methods that have previously been presented, and it has devised and implemented three new hybrid Chaffing and Winnowing schemes. These were then compared against encryption technologies that are currently in use, in an attempt to determine if Chaffing and Winnowing could be a viable alternative. When analysing our results it was found that in terms of ciphertext expansion, the hybrid schemes introduced a fixed overhead of approximately 1.5 Kilobytes, which could be considered acceptable, especially for larger file sizes. However the symmetric schemes considerably increased the size of the plaintext after 'encryption', adding an overhead of at least twelve percent of the original file size, which is undesirable due to possible bandwidth limitations.

In terms of execution time, for files of size 500KB or less the symmetric schemes were only slightly slower than the existing encryption methods, and the difference between the speed of the hybrid schemes and the existing techniques was negligible. For larger file sizes, up to 100MB, the symmetric schemes became more noticably slower than Triple-DES and AES, with the differences reaching as much as several seconds. The hybrid schemes became noticably slower than AES, however the hybrid Package Transform and CTRT schemes continued to perform comparably to Triple-DES. Since CTRT skips a whole step in comparison to the Package Transform, it was expected that there would be a noticeable difference in their execution times. This turned out not to be the case, therefore it would be more beneficial to use the Package Transform over CTRT due to the higher level of security it provides. The larger file sizes were included in our experiments in order to see if there was a significant deterioration in performance as the sizes increased, however in practice it would

be uncommon for a cryptosystem to need to encrypt such large files. A study conducted at Berkeley University in 2003[1] found that the average size of an HTML web page was 8KB, and the average size of email was approximately 59KB. Therefore, our Chaffing and Winnowing schemes could be a feasible alternative for encryption in an email application for example, especially our hybrid schemes due to the smaller ciphertext expansion.

The hybrid Chaffing and Winnowing implementations moved away from the original concept of using a MAC as a means of authenticating packets and allowing the recipient to determine if a packet is wheat or chaff. However a MAC was still required to authenticate the chaff packet locations, otherwise an adversary could tamper with them without the knowledge of the recipient. This is a disadvantage since the MAC calculation requires the sender and receiver to share a secret key, which is not necessary for standard hybrid cryptosystems, and introduces an extra key management overhead since the participants must have a secure way of agreeing on and distributing this key. It is also unclear whether our hybrid implementations can be classed as true Chaffing and Winnowing schemes, since they make use of public-key encryption. However it could be argued that the original data is still sent 'in the clear', since it is only the chaff packet locations that are encrypted.

The original motivation for the concept of Chaffing and Winnowing was the threat that strong encryption technologies could be outlawed, therefore another method would be required to provide confidentiality. We have seen that if Chaffing and Winnowing was not classed as encryption then it could be used as a practicable alternative. However we have not looked at whether it meets the same levels of security that traditional encryption technologies provide. The underlying components that were used, such as the MAC algorithm and AONTs, have been shown to provide varying levels of security, however putting 'secure' components together does not necessarily result in a secure cryptosystem. Considerations of how this issue could be addressed are given below.

7.2 Critical Evaluation

This project was successful in exploring the idea of Chaffing and Winnowing and producing several implementations to be compared with existing encryption techniques. Our experiments made use of a wide range of file sizes which allowed us to gain a better understanding of how well our schemes performed in comparison. The fact that the execution times were similar in many cases was a good achievement. Writing the code in a modular fashion made it much easier to substitute components for one another and would make it easier for any changes to be made in the future, for example if a new AONT is developed.

The use of a hash function was required extensively in our implementations, therefore the performance of the one chosen had a large impact on the execution time of our schemes as a whole. The security level of the hash function was also important in many cases,

[1] `http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/execsum.htm`

so it would have been beneficial if different hashing algorithms had been researched and compared in terms of these two factors, before it was decided which one would be used. Unfortunately time constraints prevented this.

The importance of randomness for the security of our schemes has been mentioned several times, such as when generating a key to be used as part of the Package Transform. Our implementations made use of the C `rand()` function which is a pseudorandom number generator. However, cryptosystems are recommended to make use of cryptographically secure pseudorandom number generators, which have certain properties to ensure that the outputs appear to be truly random, and are thus harder to break. This property would be highly beneficial for the security of our Chaffing and Winnowing schemes. As mentioned in section 6.1 the GnuPG system allows input messages to be compressed before encryption, which we chose to turn off for our experiments. Compressing the input data for our Chaffing and Winnowing schemes would improve performance since the input message would be smaller, meaning that the number of MAC calculations would be reduced. If a specialised cryptographic algorithm is used, compression can also increase security since it helps remove patterns in the message, making the output appear more random.

7.3 Future Work

We now look at some of the future work that could be carried out into the investigation of Chaffing and Winnowing.

Optimisation

The performance of our Chaffing and Winnowing implementations depends largely on the performance of the AONT that is used, which depends on the speed of the underlying components such as a hash function or block cipher. Therefore it may be possible to optimise the overall execution time by using more efficient components, or by implementing lower level code. To further maximise optimisation, Rivest (1997) points out that the Package Transform can be parallelised, and as mentioned above compressing the data prior to encryption would also decrease the execution times. A further benefit of compression is that it would reduce the ciphertext expansion of our symmetric Chaffing and Winnowing schemes, since less authentication data is required if the input is smaller.

Security

In order to be able to fully determine if Chaffing and Winnowing could replace traditional encryption methods, the security of our schemes would need to be tested, and possibly increased. This would involve the following:

- As mentioned above, our implementations would need to be changed to use a cryptographically strong pseudorandom number generator.
- One of our requirements was to ensure that applying an AONT to the same message twice results in two different outputs. For increased security, we would want to check that there is no other way for an adversary to identify that the two ciphertexts originated from the same plaintext. This can done by looking for similarities or patterns in the outputs.
- A possible attack on our schemes, assuming that an adversary knows the number of chaff packets and the block size used, would be to try every different combination of chaff packet positions and invert the AONT to see if an intelligible message is produced. It would be necessary to determine how computationally difficult this attack is, and perhaps consider increasing or periodically changing the number of chaff packets that is used to make the attack harder.
- The security of the symmetric Chaffing and Winnowing schemes relies on the difficulty of breaking the MAC algorithm. Therefore this would need to be tested, and if found to be insecure a stronger hash function could be utilised instead.

Complete Cryptosystem

The Chaffing and Winnowing schemes presented here form merely the underlying algorithms that would be required for secure communication. A complete cryptosystem, for example to support the sending of secure email, usually provides a user interface and has built in techniques for secure key management. Such a system would also typically have built in techniques for re-transmission of lost packets. If this occurs in a Chaffing and Winnowing scheme, it would be necessary to re-send all of the original packets, including the chaff, to prevent an adversary from gaining any information about the message.

New Schemes and Components

There may be potential for new Chaffing and Winnowing algorithms to be devised that incorporate different ideas from other popular encryption techniques. It would be beneficial for an AONT to be designed specifically for use in Chaffing and Winnowing, unlike the ones that we have implemented. This may allow more efficient and secure schemes to be produced.

7.4 Personal Reflection

A great deal was learnt throughout this project, both about the concept of Chaffing and Winnowing and cryptography in general. It was extremely interesting to carry out, and

we hope that it may encourage further investigation in the area, especially looking into the security levels that a Chaffing and Winnowing scheme could provide in practice.

Bibliography

Anderson, R. and Biham, E. (1996), 'Two Practical and Provably Secure Block Ciphers: BEAR and LION', *IWFSE: International Workshop on Fast Software Encryption, LNCS* .

Bellare, M. and Boldyreva, A. (2000), 'The Security of Chaffing and Winnowing', *Lecture Notes in Computer Science* **1976**, 517–530.

Bellare, M., Krawczyk, H. and Canetti, R. (1996), 'Keying hash functions for message authentication', *Lecture Notes In Computer Science, Advances in Cryptography - CRYPTO '96* **1109**. Springer-Verlag, 1996.

Bellare, M. and Rogaway, P. (1993), 'Random oracles are practical: A paradigm for designing efficient protocols', *Proceedings of the First Annual Conference on Computer and Communications Security* . ACM, 1993. Latest version can be obtained from `http://www-cse.ucsd.edu/users/mihir/papers/ro.html`.

Bellare, M. and Rogaway, P. (1995), 'Optimal Asymmetric Encryption - How to encrypt with RSA', *Lecture Notes in Computer Science* **950**, 92–111.

Boyko, V. (1999), 'On the Security Properties of OAEP as an All-or-Nothing Transform', *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference* **Springer-Verlag**, 503–518.

Canetti, R., Dodis, Y., Halevi, S., Kushilevitz, E. and Sahai, A. (2000), 'Exposure-Resilient Functions and All-or-Nothing Transforms', *Lecture Notes in Computer Science* **1807**, 453–469.

Canetti, R., Dwork, C., Naor, M. and Ostrovsky, R. (1997), 'Deniable Encryption', *Lecture Notes In Computer Science, Advances In Cryptography - CRYPTO '97* **1294**.

Clayton, R. and Danezis, G. (2003), 'Chaffinch: Confidentiality in the Face of Legal Threats', *Revised Papers from the 5th International Workshop on Information Hiding* pp. 70–86.

Cramer, R. and Shoup, V. (2003), 'Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack', *SIAM Journal of Computing* **33**, 167–226.

Desai, A. (2000), 'The Security of All-or-Nothing Encryption: Protecting against Exhaustive Key Search', *CRYPTO '00: Proceedings of the 20th International Cryptology Conference on Advances in Cryptology* **Springer-Verlag**, 359–375.

Diffie, W. and Hellman, M. (1976), 'New Directions in Cryptography', *IEEE Transactions on Information Theory* **IT-22**(6), 644–654.

Eastlake, D. and Jones, P. (2001), 'US Secure Hash Algorithm (SHA-1)', *Internet Engineering Task Force* .

ElGamal, T. (1985), 'A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms', *IEEE Transactions on Information Theory* **IT-31**(4), 469–472.

Kaliski Jr., B. and Robshaw, M. (1995), 'Message authentication with MD5', *RSA Laboratories CryptoBytes* **1**(1), 5–8.

Krawczyk, H., Bellare, M. and Canetti, R. (1997), 'HMAC: Keyed-hashing for message authentication'. RFC: 2104.

Kuwakado, H. and Tanaka, H. (2005), 'Secure Length-Preserving All-or-Nothing Transform', *IPSJ Digital Courier* **1**, 304–312.

National Institute of Standards and Technology (NIST) (1999), 'FIPS Publication 46-3: Data Encryption Standard (DES)'. URL `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

National Institute of Standards and Technology (NIST) (2001), 'FIPS Publication 197: Advanced Encryption Standard (AES)'. URL `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

Piper, F. and Murphy, S. (2002), *CRYPTOGRAPHY A Very Short Introduction*, Oxford University Press.

Rivest, R. (1992), 'The MD5 message digest algorithm'. URL `http://theory.lcs.mit.edu/~rivest/Rivest-MD5/txt`. RFC: 1321.

Rivest, R. (1996), 'The RC5 Encryption Algorithm', *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press* . URL `http://theory.lcs.mit.edu/~rivest/Rivest-rc5.pdf`.

Rivest, R. (1997), 'All-or-Nothing Encryption and The Package Transform', *Proceedings of the 1997 Fast Software Encryption Conference, Springer Lecture Notes in Computer Science* **1267**, 210–218. Springer-Verlag, 1997.

Rivest, R. (1998a), 'Chaffing and Winnowing: Confidentiality without Encryption', *RSA Laboratories CryptoBytes* **4**(1), 12–17.

Rivest, R. (1998b), 'The Case Against Regulating Encryption Technology', *Scientific American* **279**(4), 116. URL `http://theory.lcs.mit.edu/~rivest/sciam98.txt`.

Rivest, R., Shamir, A. and Adleman, L. (1978), 'A Method for Obtaining Digital Signatures and Public-Key Cryptosystems', *Communications of the ACM* **21**(2), 120–126.

RSA Security Inc. (2002), 'PKCS #1 v2.1: RSA Cryptography Standard'.

UK Stationary Office Ltd (2000), 'Regulation of Investigatory Powers Act'. ISBN 0-10-54230-9.

Appendix A

Further Experiment Results

This section includes some further results from our experiments that were not included in the main document.

A.1 Decryption timings

The following tables show the decryption timings for each scheme. We found that the decryption times were very similar to the encryption times, but slightly slower for the symmetric Chaffing and Winnowing schemes. This is because more MAC calculations are needed in order to 'winnow' the wheat from the chaff.

Table A.1: Decryption timings for symmetric schemes

File size (bytes)	CW PT	CW CTRT	CW OAEP	3DES	AES
10752	0.0563	0.0462	0.0558	0.0353	0.0327
51595	0.0568	0.0517	0.068	0.038	0.0387
107985	0.0735	0.056	0.0621	0.046	0.042
256895	0.1004	0.1087	0.1077	0.0653	0.4933
545730	0.1539	0.1361	0.2039	0.092	0.076
1244182	0.2859	0.2621	0.3497	0.1707	0.1227
7471298	1.5703	1.3515	1.5547	0.8807	0.572
27929130	5.7548	4.9015	5.8507	3.218	2.0593
53263590	10.9088	9.1733	11.2712	6.398	3.8993
103811754	20.813	17.9129	25.436	12.534	7.646

Table A.2: Decryption timings for hybrid schemes

File size (bytes)	Hybrid PT	Hybrid CTRT	Hybrid OAEP	Hybrid 3DES	Hybrid AES
10752	0.1539	0.1467	0.1487	0.038	0.04
51595	0.1562	0.1539	0.1539	0.0447	0.0427
107985	0.16	0.156	0.1655	0.0528	0.0427
256895	0.1839	0.1767	0.1714	0.0714	0.056
545730	0.2247	0.2184	0.2205	0.1285	0.0847
1244182	0.3288	0.2892	0.3208	0.1936	0.132
7471298	1.2223	0.992	1.16	0.9236	0.59
27929130	4.0586	3.263	3.9338	3.4757	2.104
53263590	7.6889	6.1744	7.311	6.2407	4.0333
103811754	14.639	11.686	13.905	12.3536	7.8733

A.2 Percentage ciphertext expansion

The following tables show the *percentage* ciphertext expansion for each scheme. Graphs representing this data were shown in section 6.3.2.

Table A.3: Percentage ciphertext expansion for symmetric schemes

File size (bytes)	CW PT	CW CTRT	CW OAEP	3DES	AES
10752	183.93	183.93	185.27	0.4929	0.8091
51595	48.2	48.2	48.48	0.093	0.1589
107985	29.48	29.48	29.61	0.0453	0.075
256895	19.68	19.68	19.68	0.0237	0.0362
545730	15.86	15.86	15.89	0.0104	0.0163
1244182	13.98	13.98	13.99	0.0038	0.0064
7471298	12.75	12.75	12.75	0.0007	0.0011
27929130	12.57	12.57	12.57	0.00019	0.0003
53263590	12.53	12.53	12.53	0.00009	0.00016
103811754	12.51	12.51	12.52	0.00005	0.00008

Table A.4: Percentage ciphertext expansion for hybrid schemes

File size (bytes)	Hybrid PT	Hybrid CTRT	Hybrid OAEP	Hybrid 3DES	Hybrid AES
10752	183.33	183.33	184.52	3.13	3.21
51595	38.18	38.18	38.43	0.64	0.66
107985	18.18	18.18	18.29	0.307	0.31
256895	7.67	7.67	7.67	0.134	0.13
545730	3.59	3.59	3.62	0.062	0.063
1244182	1.58	1.58	1.59	0.027	0.027
7471298	0.2629	0.2629	0.264	0.004	0.004
27929130	0.0704	0.0704	0.0708	0.001	0.001
53263590	0.0368	0.0368	0.037	0.0006	0.0006
103811754	0.0189	0.0189	0.019	0.0003	0.0003

Appendix B

Source Code

Here we provide source code for the main components in our implementation:

- Chaffing and Winnowing global library
- HMAC module
- Package Transform module
- CTRT module
- OAEP module
- Symmetric Chaffing and Winnowing scheme using the Package Transform
- Hybrid Chaffing and Winnowing scheme using the Package Transform

We have not included the code for every full Chaffing and Winnowing scheme here, since they are implemented in the same way but making use of different AONT libraries. The full source code can be found on the accompanying CD.

B.1 File: cwin_lib.c

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "cwin_lib.h"

// taken from http://www.cplusplus.com/reference/
//              clibrary/cstdlib/qsort.html
// used in qsort to compare two integers
int compare(const void *a, const void *b)
{
        return ( *(int*)a - *(int*)b );
}

/* calculates the positions of chaff packets according
*  to the number of total blocks that there needs to
*  be. It makes sure that no values are duplicated, and
*  the positions are put into order using qsort.
*/
void calculateChaffPositions(int *chaffPositions, int
    numChaffBlocks, int numBlocks)
{
        int i, j, n, totalBlocks;

        totalBlocks = numChaffBlocks + numBlocks;

        for(i = 0; i < numChaffBlocks; i++)
        {
                n = rand() % totalBlocks;
                // check that n hasn't been used yet
                for(j = 0; j < i; j++)
                {
                        if(chaffPositions[j] == n)
                        {
                                n = rand() % totalBlocks;
                                j = -1;
                        }
                }
                chaffPositions[i] = n;
        }
        qsort(chaffPositions, numChaffBlocks,
            sizeof(int), compare);
}

/* Generate a random chaff packet of a certain size */
void generateChaffPacket(unsigned char *chaffPacket, int
    size)
{
        int i;
        for(i = 0; i < size; i++)
        {
                chaffPacket[i] = rand() % 255;
        }
}

// taken from:
// http://c-faq.com/misc/endiantest.html
// tests if the machine is big or little endian
int isBigEndian()
{
        int x = 1;
        if(*(char *)&x == 1)
        {
                // little endian
                return 0;
        }
        else
        {
                // big endian
                return 1;
        }
}

// taken from: http://www.ibm.com/developerworks/aix/
//              library/au-endianc/index.html?ca=drs-
// reverses the byte order of an integer
int reverseInt (int i)
{
    unsigned char c1, c2, c3, c4;
    c1 = i & 255;
    c2 = (i >> 8) & 255;
    c3 = (i >> 16) & 255;
    c4 = (i >> 24) & 255;

    return ((int)c1 << 24) + ((int)c2 << 16) + ((int)c3
        << 8) + c4;
}
```

```
//#define LIB_TEST
#ifdef LIB_TEST
void main()
{
        int i;
        int chaff[50];
        calculateChaffPositions(chaff, 50, 50);

        // seed the random number generator
        srand(time(NULL));

        // print out results so we can check for
            duplicates
        // and check positions are correctly ordered
        for(i = 0; i < 50; i++)
        {
                printf("%d\n", chaff[i]);
        }

        // check endianness test works
        if(isBigEndian())
        {
                printf("Big endian\n");
        }
        else
        {
                printf("Little endian\n");
        }

        // check reverseInt works correctly
        i = 0x12345678
        printf("%08lx\n", i);
        i = reverseInt(i);
        printf("%08lx\n", i);

}
#endif
```

B.2 File: hmac.c

```
/*Implementation of HMAC-MD5 taken from
* http://www.faqs.org/rfcs/rfc2104.html. */

/*
** Function: hmac_md5
*unsigned char*  text;          pointer to data stream
*int             text_len;      length of data stream
*unsigned char*  key;           pointer to authentication
    key
*int             key_len;       length of authentication
    key
*unsigned char*  digest;        caller digest to be
    filled in
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hmac.h"
#include "md5.h"

// inner padding - key XORd with ipad
unsigned char k_ipad[65];

// outer padding - key XORd with opad
unsigned char k_opad[65];

/* Init() function added to initialise the
* inner and outer padding values according to the key */
void hmac_init(unsigned char* key, int key_len)
{
        MD5_CTX context;

        unsigned char tk[16];
        int i;
        /* if key is longer than 64 bytes reset it to
            key=MD5(key) */
        if (key_len > 64) {

                MD5_CTX      tctx;

                MD5Init(&tctx);
                MD5Update(&tctx, key, key_len);
                MD5Final(tk, &tctx);

                key = tk;
                key_len = 16;
```

```
        }

        /*
        * the HMAC_MD5 transform looks like:
        *
        * MD5(K XOR opad, MD5(K XOR ipad, text))
        *
        * where K is an n byte key
        * ipad is the byte 0x36 repeated 64 times

        * opad is the byte 0x5c repeated 64 times
        * and text is the data being protected
        */

        /* start out by storing key in pads */
        memset(k_ipad, 0, sizeof k_ipad);
        memset(k_opad, 0, sizeof k_opad);
        memcpy(k_ipad, key, key_len);
        memcpy(k_opad, key, key_len);

        /* XOR key with ipad and opad values */
        for (i=0; i<64; i++) {
                k_ipad[i] ^= 0x36;
                k_opad[i] ^= 0x5c;
        }
}

void hmac_md5(unsigned char* text, int text_len, unsigned
    char digest[16])
{
        MD5_CTX context;

        /*
        * perform inner MD5
        */

        // init context for 1st pass
        MD5Init(&context);
        // start with inner pad
        MD5Update(&context, k_ipad, 64);
        // then text of datagram
        MD5Update(&context, text, text_len);
        // finish up 1st pass
        MD5Final(digest, &context);

        /*
        * perform outer MD5
        */

        // init context for 2nd pass
        MD5Init(&context);
        // start with outer pad
        MD5Update(&context, k_opad, 64);
        // then results of first hash
        MD5Update(&context, digest, 16);
        // finish up 2nd pass
        MD5Final(digest, &context);
}


//#define HMAC_TEST
#ifdef HMAC_TEST
int main ()
{
        unsigned char digest[16], key1[16], text[50],
            key2[16];
        unsigned char key[] = {0xab, 0xdd, 0x34, 0x56,
            0xff, 0xcc, 0x12, 0x38, 0xdb, 0x5b, 0x4a,
            0x67, 0xfb, 0x9c, 0x31, 0xe4};
        char f;
        char p[1];
        int i;
        for(i = 0; i < 16; i++)
        {
                key1[i] = 0x0b;
        }

        hmac_init(key1, 16);
        hmac_md5("Hi There", 8, digest);
        printf("Hi There: ");
        MDPrint(digest);*/

        hmac_init("Jefe", 4);
        hmac_md5("what do ya want for nothing?", 28,
            digest);
        printf("\nwhat do ya want for nothing?: ");
        MDPrint(digest);

        for(i = 0; i < 50; i++)
        {
```

```
                text[i] = 0xdd;
        }

        for(i = 0; i < 16; i++)
        {
                key2[i] = 0xaa;
        }

        hmac_init(key2, 16);
        hmac_md5(text, 50, digest);
        printf("\noxdd 50 times: ");
        MDPrint(digest);
}
#endif
```

B.3 File: aont_pt.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "RC5REF.H"
#include "aont_pt.h"

#define KEY_SIZE 16 // 128-bit Key size
#define BLOCK_SIZE 8 // 64-bit Block size

/* public key chosen at random for RC5 block cipher */
unsigned char public_key[] = {0xab, 0xdd, 0x34, 0x56,
    0xff, 0xcc, 0x12, 0x38, 0xdb, 0x5b, 0x4a, 0x67,
    0xfb, 0x9c, 0x31, 0xe4};

/* method to generate a random key of size KEY_SIZE */
void generateRandomKey(unsigned char *key)
{
        int i;
        for(i = 0; i < KEY_SIZE; i++)
        {
                key[i] = (rand() % 255);
        }
}

/* Method to perform the package transform on an input
* file ifp, and store the output in the file ofp
*/
void transform(FILE *ifp, FILE *ofp)
{
        unsigned char private_key[KEY_SIZE],
            currentBlock[BLOCK_SIZE],
            outputBlock[BLOCK_SIZE], lastBlock[KEY_SIZE];
        WORD pt[2], ct[2] = {0, 0};
        WORD S_public[t], S_private[t]; // key tables
            for the RC5 block cipher
        int count, i, j;

        // generate a key to be used as the private key
        generateRandomKey(private_key);

        // set up the key tables
        RC5_SETUP(public_key, S_public);
        RC5_SETUP(private_key, S_private);

        // store the private key in the last block, so
        // it can be xor'd with the hashes later
        memcpy(lastBlock, private_key, KEY_SIZE);

        count = 1;
        // loop until we reach the end of the file
        while(!feof(ifp))
        {
                memset(currentBlock, 0, BLOCK_SIZE);
                // read in the next block
                fread(currentBlock, sizeof(char),
                    BLOCK_SIZE, ifp);

                pt[0] = count;
                pt[1] = 0;

                // encrypt counter with private key
                RC5_ENCRYPT(pt, ct, S_private);

                // need to xor currentBlock with ct[0]
                // and ct[1] to get outputBlock
                for(i = 0, j = 24; i < 4; i++)
                {
                        outputBlock[i] = currentBlock[i]
                            ^ ((ct[0] >> j) & 0xFF);
                        outputBlock[i + 4] =
                            currentBlock[i + 4] ^
```

```
                ((ct[1] >> j) & 0xFF);
        j -= 8;
}

//write output block to file
fwrite(outputBlock, sizeof(char),
    BLOCK_SIZE, ofp);

// calculate the last block as we go
// along
pt[0] = 0;
pt[1] = 0;

// set pt[0] to be first half of
// outputBlock xor'd with counter,
// and pt[1] to be output Block
for(i = 0, j = 24; i < 4; i++)
{
        pt[0] = pt[0] | ((outputBlock[i]
            ^ ((count >> j) & 0xFF)) <<
            j);
        pt[1] = pt[1] | (outputBlock[i +
            4] << j);
        j -= 8;
}

// reset ct
ct[0] = 0;
ct[1] = 0;

// encrypt pt with public key
RC5_ENCRYPT(pt, ct, S_public);

// last block needs to be xor of all
// previous hashes. Last block is
    128-bits.
for(i = 0, j = 24; i < 4; i++)
{
        lastBlock[i] = lastBlock[i] ^
            ct[0] >> j;
        lastBlock[i + 8] = lastBlock[i +
            8] ^ ct[0] >> j;
        j -= 8;
}
for(i = 4, j = 24; i < 8; i++)
{
        lastBlock[i] = lastBlock[i] ^
            ct[1] >> j;
        lastBlock[i + 8] = lastBlock[i +
            8] ^ ct[1] >> j;
        j -= 8;
}

        count++;
    }

    // write last block to output file
    fwrite(lastBlock, sizeof(char), KEY_SIZE, ofp);
}

/* This method takes an input file ifp, performs the
* inverse of the package transform, and stores the
* output in ofp
*/
void inverse_transform(FILE *ifp, FILE *ofp)
{
    unsigned char private_key[KEY_SIZE],
        currentBlock[BLOCK_SIZE],
        hashBlock[BLOCK_SIZE],
        outputBlock[BLOCK_SIZE], lastBlock[KEY_SIZE];
    WORD pt[2], ct[2] = {0, 0};
    WORD S_public[t], S_private[t]; // key tables
        for RC5
    int filesize, numBlocks, i, j, count;

    // calculate the size of the input file
    fseek (ifp , 0 , SEEK_END);
    filesize = ftell (ifp);
    rewind (ifp);

    // calculate the no of blocks in the input file
    numBlocks = (int)floor((double)filesize /
        BLOCK_SIZE);

    memset(private_key, 0, KEY_SIZE);
    // setup the public key table
    RC5_SETUP(public_key, S_public);

    count = 1;
    // loop until we reach the last block
```

```
while(count <= (numBlocks -
    (KEY_SIZE/BLOCK_SIZE)))
{
        memset(currentBlock, 0, BLOCK_SIZE);
        fread(currentBlock, sizeof(char),
            BLOCK_SIZE, ifp);

        pt[0] = 0;
        pt[1] = 0;

        // create hash
        for(i = 0, j = 24; i < 4; i++)
        {
                pt[0] = pt[0] |
                    ((currentBlock[i] ^ ((count
                    >> j) & 0xFF)) << j);
                pt[1] = pt[1] | (currentBlock[i
                    + 4] << j);
                j -= 8;
        }

        ct[0] = 0;
        ct[1] = 0;

        // encrypt pt with the public key
        RC5_ENCRYPT(pt, ct, S_public);

        for(i = 0, j = 24; i < 4; i++)
        {
                private_key[i] = private_key[i]
                    ^ ct[0] >> j;
                private_key[i + 8] =
                    private_key[i + 8] ^ ct[0]
                    >> j;
                j -= 8;
        }
        for(i = 4, j = 24; i < 8; i++)
        {
                private_key[i] = private_key[i]
                    ^ ct[1] >> j;
                private_key[i + 8] =
                    private_key[i + 8] ^ ct[1]
                    >> j;
                j -= 8;
        }
        count++;
}

// read in the last block from input
memset(lastBlock, 0, KEY_SIZE);
fread(lastBlock, sizeof(char), KEY_SIZE, ifp);

// the private key is the XOR of all the hashes
// and the last block
for(i = 0; i < KEY_SIZE; i++)
{
        private_key[i] = private_key[i] ^
            lastBlock[i];
}

/* Now that we have the key, we can go back to
* the start of the file and invert the transform
*/

rewind (ifp);

RC5_SETUP(private_key, S_private);

count = 1;
// loop until we reach the last block
while(count <= (numBlocks -
    (KEY_SIZE/BLOCK_SIZE)))
{
        memset(currentBlock, 0, BLOCK_SIZE);
        fread(currentBlock, sizeof(char),
            BLOCK_SIZE, ifp);

        pt[0] = count;
        pt[1] = 0;

        ct[0] = 0;
        ct[1] = 0;

        // encrypt counter with the private key
        RC5_ENCRYPT(pt, ct, S_private);

        // need to xor currentBlock with ct[0]
        // and ct[1] to get outputBlock
        for(i = 0, j = 24; i < 4; i++)
        {
```

```
                        outputBlock[i] = currentBlock[i]
                            ^ ((ct[0] >> j) & 0xFF);
                        outputBlock[i + 4] =
                            currentBlock[i + 4] ^
                            ((ct[1] >> j) & 0xFF);
                        j -= 8;
                }

                //write output block to file
                fwrite(outputBlock, sizeof(char),
                    BLOCK_SIZE, ofp);
                count++;
        }
}

//#define PT_TEST
#ifdef PT_TEST
void main()
{
        FILE *ifp, *ofp;

        ifp = fopen("FleurKellyProjProposal.pdf", "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                exit(1);
        }

        ofp = fopen("transformed", "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                exit(2);
        }

        // seed the random number generator
        srand(time(NULL));

        // transform the file
        printf("Performing package transform\n");
        transform(ifp, ofp);

        fclose(ifp);
        fclose(ofp);

        ifp = fopen("transformed", "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                exit(1);
        }

        ofp = fopen("inverse", "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                exit(2);
        }

        printf("Inverting package transform\n");
        inverse_transform(ifp, ofp);

        fclose(ifp);
        fclose(ofp);
}
#endif
```

B.4 File: aont_ctrt.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "RC5REF.H"
#include "aont_ctrt.h"

#define KEY_SIZE 16 // 128-bit Key size
#define BLOCK_SIZE 8 // 64-bit Block size

/* method to generate a random key of size KEY_SIZE */
void generateRandomKey(unsigned char *key)
{
        int i;
        for(i = 0; i < KEY_SIZE; i++)
        {
                key[i] = (rand() % 256);
        }
```

```
}

/* method to calculate key2 = key1 mod 2^k */
void calculateKey(unsigned char *key1, unsigned char
    *key2)
{
        int i, j;
        j = pow(2, KEY_SIZE);
        for(i = 0; i < KEY_SIZE; i++)
        {
                key2[i] = key1[i] % j;
        }
}

void transform(FILE *ifp, FILE *ofp)
{
        unsigned char key1[KEY_SIZE], key2[KEY_SIZE],
            currentBlock[BLOCK_SIZE],
            outputBlock[BLOCK_SIZE], lastBlock[KEY_SIZE];
        WORD pt[2], ct[2] = {0, 0};
        WORD S_key2[t]; // key table for RC5
        int count, i, j;

        // generate a random key, and calculate key2
        // based on it
        generateRandomKey(key1);
        calculateKey(key1, key2);

        RC5_SETUP(key2, S_key2);

        // copy the key into the last block so we can
        // XOR it with each intermediate block
        memcpy(lastBlock, key1, KEY_SIZE);

        count = 1;
        // loop until the end of the file
        while(!feof(ifp))
        {
                memset(currentBlock, 0, BLOCK_SIZE);
                // read in the next block
                fread(currentBlock, sizeof(char),
                    BLOCK_SIZE, ifp);

                pt[0] = count;
                pt[1] = 0;

                // encrypt the counter with key2
                RC5_ENCRYPT(pt, ct, S_key2);

                // need to xor currentBlock with ct[0]
                // and ct[1] to get outputBlock
                for(i = 0, j = 24; i < 4; i++)
                {
                        outputBlock[i] = currentBlock[i]
                            ^ ((ct[0] >> j) & 0xFF);
                        outputBlock[i + 4] =
                            currentBlock[i + 4] ^
                            ((ct[1] >> j) & 0xFF);
                        j -= 8;
                }

                // want to write output block to file
                fwrite(outputBlock, sizeof(char),
                    BLOCK_SIZE, ofp);

                // last block is key1 XOR'd with all
                // other output blocks
                for(i = 0; i < 8; i++)
                {
                        lastBlock[i] = lastBlock[i] ^
                            outputBlock[i];
                        lastBlock[i + 8] = lastBlock[i +
                            8] ^ outputBlock[i];
                }
                count++;
        }
        // write last block to file
        fwrite(lastBlock, sizeof(char), KEY_SIZE, ofp);
}

void inverse_transform(FILE *ifp, FILE *ofp)
{
        unsigned char key1[KEY_SIZE], key2[KEY_SIZE],
            currentBlock[BLOCK_SIZE],
            outputBlock[BLOCK_SIZE], lastBlock[KEY_SIZE];
        WORD pt[2], ct[2] = {0, 0};
        WORD S_key2[t];
        int count, i, j, filesize, numBlocks;

        // calculate the size of the input file
```

```
fseek (ifp , 0 , SEEK_END);
filesize = ftell (ifp);
rewind (ifp);

// calculate the number of blocks
numBlocks = (int)floor((double)filesize /
    BLOCK_SIZE);

memset(key1, 0, KEY_SIZE);

// loop until we reach the last block
// XOR each block so we can get the key
count = 1;
while(count <= (numBlocks -
    (KEY_SIZE/BLOCK_SIZE)))
{
        memset(currentBlock, 0, BLOCK_SIZE);
        fread(currentBlock, sizeof(char),
            BLOCK_SIZE, ifp);

        for(i = 0; i < 8; i++)
        {
                key1[i] = key1[i] ^
                    currentBlock[i];
                key1[i + 8] = key1[i + 8] ^
                    currentBlock[i];
        }
        count++;
}

// read in the last block from input
memset(lastBlock, 0, KEY_SIZE);
fread(lastBlock, sizeof(char), KEY_SIZE, ifp);

// the key is the XOR of all intermediate blocks
// and the last block
for(i = 0; i < KEY_SIZE; i++)
{
        key1[i] = key1[i] ^ lastBlock[i];
}

// calculate the key we need to reverse
// the transform
calculateKey(key1, key2);

RC5_SETUP(key2, S_key2);

// now that we have the key, we can go back
// to the start of the input file
rewind(ifp);
count = 1;

while(count <= (numBlocks -
    (KEY_SIZE/BLOCK_SIZE)))
{
        memset(currentBlock, 0, BLOCK_SIZE);
        fread(currentBlock, sizeof(char),
            BLOCK_SIZE, ifp);

        pt[0] = count;
        pt[1] = 0;

        // encrypt the counter with key2
        RC5_ENCRYPT(pt, ct, S_key2);

        // need to xor currentBlock with ct[0]
        // and ct[1] to get outputBlock
        for(i = 0, j = 24; i < 4; i++)
        {
                outputBlock[i] = currentBlock[i]
                    ^ ((ct[0] >> j) & 0xFF);
                outputBlock[i + 4] =
                    currentBlock[i + 4] ^
                    ((ct[1] >> j) & 0xFF);
                j -= 8;
        }

        // want to write output block to file
        fwrite(outputBlock, sizeof(char),
            BLOCK_SIZE, ofp);
        count++;
}

}

//#define CTRT_TEST
#ifdef CTRT_TEST
void main()
{
        FILE *ifp, *ofp;
```

```
ifp = fopen("FleurKellyProjProposal.pdf", "rb");

if(ifp == NULL)
{
        printf("Unable to open input file");
        exit(1);
}

ofp = fopen("transformed", "wb");
if(ofp == NULL)
{
        printf("Unable to open output file");
        exit(2);
}

// seed the random number generator
srand(time(NULL));

// transform the file
printf("Performing CTRT transform\n");
transform(ifp, ofp);

fclose(ifp);
fclose(ofp);

ifp = fopen("transformed", "rb");

if(ifp == NULL)
{
        printf("Unable to open input file");
        exit(1);
}

ofp = fopen("inverse", "wb");
if(ofp == NULL)
{
        printf("Unable to open output file");
        exit(2);
}

printf("Inverting CTRT transform\n");
inverse_transform(ifp, ofp);

fclose(ifp);
        fclose(ofp);
}
#endif
```

B.5 File: aont_oaep.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "aont_oaep.h"
#include "md5.h"

#define HASH_LEN 16 // length of hash function output

// encoding parameter
char *encoding_param = "oaep_encoding";

void MGF1(char *T, unsigned char mgfSeed[16], int
    maskLen)
{
        int i, outLen = 0;
        unsigned char counter[4], digest[16];
        MD5_CTX context;

        MD5Init(&context);

        // loop until output reaches the
        // desired mask length
        for(i = 0; outLen < maskLen; i++)
        {
                // convert counter to an octet string
                // C of length 4 octets
                // C = I2OSP(counter, 4)
                counter[0] = (unsigned char)((i << 24) &
                    255);
                counter[1] = (unsigned char)((i << 16) &
                    255);
                counter[2] = (unsigned char)((i << 8) &
                    255);
                counter[3] = (unsigned char)(i & 255);

                // calculate and concatenate the hash
```

```
                // of mgfSeed and C
                MD5Update(&context, mgfSeed, HASH_LEN);
                MD5Update(&context, counter, 4);
                MD5Final(digest, &context);

                // concatenate the hash of the seed
                // mgfSeed and C to the octet string
                // T: T = T || Hash(mgfSeed || C)
                if((outLen + HASH_LEN) < maskLen)
                {
                        memcpy(T + outLen, digest,
                            HASH_LEN);
                        outLen += HASH_LEN;
                }
                else
                {
                        memcpy(T + outLen, digest,
                            maskLen - outLen);
                        outLen = maskLen;
                }
        }
        // T now contains the leading maskLen octets as
        // the octet string mask
}

// a function to generate a random string of
// length HASH_LEN
void generateRandomString(char *seed)
{
        int i;
        for(i = 0; i < HASH_LEN; i++)
        {
                seed[i] = (rand() % 255);
        }
}

void transform(FILE *ifp, FILE *ofp, int emLen)
{
        int filesize, i, j, paddingSize;
        unsigned char *PS, *message, pHash[HASH_LEN],
            *DB, seed[HASH_LEN], maskedSeed[HASH_LEN],
            *dbMask, *maskedDB, seedMask[HASH_LEN];
        MD5_CTX context;

        // calculate file size
        fseek (ifp , 0 , SEEK_END);
        filesize = ftell (ifp);
        rewind (ifp);

        // if mLen (filesize) > emLen - 2hLen - 1 output
        // msg too long and stop
        if(filesize > emLen - (2 * HASH_LEN) - 1)
        {
                printf("Error in OAEP - transform(): 
                    Input message is too long\n");
                exit(1);
        }

        // read in the message
        message = malloc(sizeof(char) * filesize);
        fread(message, sizeof(char), filesize, ifp);

        // let pHash = Hash(P) where P is
        // the encoding parameter
        MD5Init(&context);
        MD5Update(&context, encoding_param,
            strlen(encoding_param));
        MD5Final(pHash, &context);

        // generate an octet string PS consisting of
        // emLen - mLen - 2hLen - 1 zero octets.
        // Length of ps may be 0
        paddingSize = emLen - filesize - (2 * HASH_LEN)
            - 1;
        PS = malloc(sizeof(char) * paddingSize);

        for(i = 0; i < paddingSize; i++)
        {
                PS[i] = '0x00';
        }

        // concatenate pHash, PS, the message M, '01'
        // and other padding to form a data block DB:
        // DB = pHash||PS||01||M
        DB = malloc(sizeof(char) * (emLen - HASH_LEN));
        memset(DB, 0, emLen - HASH_LEN);

        // concatenate DB with pHash
        for(i = 0; i < HASH_LEN; i++)
        {
```

```
        DB[i] |= pHash[i];
}
// concatenate DB with PS
for(i = HASH_LEN, j = 0; i < HASH_LEN +
    paddingSize; i++)
{
        DB[i] |= PS[j];
        j++;
}
// concatenate DB with 1
DB[HASH_LEN + paddingSize] = 255;

// concatenate DB with M
for(i = HASH_LEN + paddingSize + 1, j = 0; i <
    filesize + HASH_LEN + paddingSize + 1; i++)
{
        DB[i] |= message[j];
        j++;
}

// generate a random octet string seed of length
// hLen to seed MGF
generateRandomString(seed);

// let dbMask = MGF(seed, emLen - hLen)
dbMask = malloc(sizeof(char) * (emLen -
    HASH_LEN));
MGF1(dbMask, seed, emLen - HASH_LEN);

// let maskedDB = DB XOR dbMask
maskedDB = malloc(sizeof(char) * (emLen -
    HASH_LEN));
for(i = 0; i < emLen - HASH_LEN; i++)
{
        maskedDB[i] = DB[i] ^ dbMask[i];
}

// Let seedMask = MGF(maskedDB, hLen)
MGF1(seedMask, maskedDB, HASH_LEN);

// Let maskedSeed = seed XOR seedMask
for(i = 0; i < HASH_LEN; i++)
{
        maskedSeed[i] = seed[i] ^ seedMask[i];
}

// output EM = maskedSeed||maskedDB
fwrite(maskedSeed, sizeof(char), HASH_LEN, ofp);
fwrite(maskedDB, sizeof(char), emLen - HASH_LEN,
    ofp);

// free memory
free(message);
free(DB);
free(maskedDB);
free(dbMask);
free(PS);
}

void inverse_transform(FILE *ifp, FILE *ofp)
{
        int emLen, i, padding, j, mLen;
        unsigned char *message, pHash[HASH_LEN],
            pHashPrime[HASH_LEN], *DB, seed[HASH_LEN],
            maskedSeed[HASH_LEN], *dbMask, *maskedDB,
            seedMask[HASH_LEN];
        MD5_CTX context;

        // calculate file size
        fseek (ifp , 0 , SEEK_END);
        emLen = ftell (ifp);
        rewind (ifp);

        // if emLen < 2hLen + 1 output decoding error
        // and stop
        if(emLen < (2 * HASH_LEN) + 1)
        {
                printf("Error in OAEP -
                    inverse_transform(): Decoding
                    error\n");
                exit(2);
        }

        // let maskedSeed be the first hLen octets of EM
        // and let maskedDB be the remaining
        // emLen - hLen octets
        fread(maskedSeed, sizeof(char), HASH_LEN, ifp);

        maskedDB = malloc(sizeof(char) * (emLen -
            HASH_LEN));
```

```
fread(maskedDB, sizeof(char), emLen - HASH_LEN,
    ifp);

// let seedMask = MGF(maskedDB, hLen)
MGF1(seedMask, maskedDB, HASH_LEN);

// let seed = maskedSeed XOR seedMask
for(i = 0; i < HASH_LEN; i++)
{
        seed[i] = maskedSeed[i] ^ seedMask[i];
}

// let dbMask = MGF(seed, emLen - hLen)
dbMask = malloc(sizeof(char) * (emLen -
    HASH_LEN));
MGF1(dbMask, seed, emLen - HASH_LEN);

// let DB = maskedDB XOR dbMask
DB = malloc(sizeof(char) * (emLen - HASH_LEN));
for(i = 0; i < emLen - HASH_LEN; i++)
{
        DB[i] = maskedDB[i] ^ dbMask[i];
}

// let pHash = Hash(P)
MD5Init(&context);
MD5Update(&context, encoding_param,
    strlen(encoding_param));
MD5Final(pHash, &context);

// separate DB into an octet string pHash'
// consisting of the first hLen octets of DB,
// a (Possibly empty) octet string PS
// consisting of consecutive zero octets
// following pHash', and a message M as
// DB = pHash'||PS||01||M

for(i = 0; i < HASH_LEN; i++)
{
        pHashPrime[i] = DB[i];
}

// if pHash' does not equal pHash, output
// decoding error and stop
if(memcmp(pHash, pHashPrime, HASH_LEN) != 0)
{
        printf("Error in OAEP -
            inverse_transform(): Decoding error,
            pHash != pHash'\n");
        exit(3);
}

i = HASH_LEN;
// work out how much padding there is
padding = 0;
while((DB[i] != 255) && (i < emLen - HASH_LEN))
{
        padding++;
        i++;
}

// if there is no 01 octet to separate PS from
// M, output decoding error and stop
if(DB[i] != 255)
{
        printf("Error in OAEP -
            inverse_transform(): Decoding error,
            no 01 octet present'\n");
        exit(4);
}

// size of message = emLen - HASH_LEN -
// padding - 1
mLen = emLen - HASH_LEN - padding - 1;

// retrieve the message from DB
message = malloc(sizeof(char) * mLen);
for(i = i + 1, j = 0; i < emLen - HASH_LEN; i++)
{
        message[j] = DB[i];
        j++;
}
// output M
fwrite(message, sizeof(char), mLen, ofp);

// free memory
free(maskedDB);
free(dbMask);
free(DB);
free(message);
```

```
}

//#define OAEP_TEST
#ifdef OAEP_TEST
void main()
{
        FILE *ifp;
        FILE *ofp;
        int filesize;

        ifp = fopen("FleurKellyProjProposal.pdf", "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                exit(1);
        }

        ofp = fopen("transformed", "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                exit(2);
        }

        srand(time(NULL));

        fseek (ifp , 0 , SEEK_END);
        filesize = ftell (ifp);
        rewind (ifp);

        // transform the file
        printf("Performing OAEP transform\n");
        transform(ifp, ofp, filesize + 35);

        fclose(ifp);
        fclose(ofp);

        ifp = fopen("transformed", "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                exit(1);
        }

        ofp = fopen("inverse", "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                exit(2);
        }

        printf("Inverting OAEP transform\n");
        inverse_transform(ifp, ofp);

        fclose(ifp);
        fclose(ofp);

}
#endif
```

B.6 File: cwin_pt.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "hmac.h"
#include "cwin_lib.h"
#include "aont_pt.h"

#define NUM_CHAFF_BLOCKS 128
#define DIGEST_LENGTH 16
#define BLOCK_SIZE 128

// key to be used in HMAC, value read in from the user
unsigned char *MACkey;

void chaff(FILE *ifp, FILE *ofp)
{
        FILE *temp;
        int chaffPositions[NUM_CHAFF_BLOCKS];
        int filesize, numBlocks, i, chaffIndex;
        unsigned char currentBlock[BLOCK_SIZE],
            digest[DIGEST_LENGTH],
            chaffPacket[BLOCK_SIZE + DIGEST_LENGTH];
```

```
// apply package transform to input file
// and write to temp file
temp = tmpfile();
transform(ifp, temp);

// calculate size of transformed file
fseek (temp , 0 , SEEK_END);
filesize = ftell (temp);
rewind (temp);

// calculate the number of blocks
numBlocks = (int)floor((double)filesize /
    BLOCK_SIZE);

// calculate the position of the chaff packets
calculateChaffPositions(chaffPositions,
    NUM_CHAFF_BLOCKS, numBlocks);

// initialise HMAC
hmac_init(MACkey, 16);

chaffIndex = 0;
for(i = 0; i < numBlocks + NUM_CHAFF_BLOCKS; i++)
{
        // if the current index should be a
        // chaff block, generate a fake block
        // and MAC and write it to file
        if(i == chaffPositions[chaffIndex])
        {
                generateChaffPacket(chaffPacket,
                    BLOCK_SIZE + DIGEST_LENGTH);
                fwrite(chaffPacket,
                    sizeof(char), BLOCK_SIZE +
                    DIGEST_LENGTH, ofp);
                chaffIndex++;
        }
        // otherwise read the next block in from
        // the file, calculate it's MAC
        // and write them both to file
        else
        {
                memset(currentBlock, 0,
                    BLOCK_SIZE);
                fread(currentBlock,
                    sizeof(char), BLOCK_SIZE,
                    temp);
                hmac_md5(currentBlock,
                    BLOCK_SIZE, digest);

                fwrite(currentBlock,
                    sizeof(char), BLOCK_SIZE,
                    ofp);
                fwrite(digest, sizeof(char),
                    DIGEST_LENGTH, ofp);
        }
    }
}

void winnow(FILE *ifp, FILE *ofp)
{
        FILE *temp;
        unsigned char currentBlock[BLOCK_SIZE],
            currentMac[DIGEST_LENGTH],
            digest[DIGEST_LENGTH];
        temp = tmpfile();

        // initialise HMAC
        hmac_init(MACkey, 16);

        // loop until we reach the end of the file
        while(!feof(ifp))
        {
                // read in the next block and MAC
                memset(currentBlock, 0, BLOCK_SIZE);
                fread(currentBlock, sizeof(char),
                    BLOCK_SIZE, ifp);
                memset(currentMac, 0, DIGEST_LENGTH);
                fread(currentMac, sizeof(char),
                    DIGEST_LENGTH, ifp);

                // calculate the MAC for the
                // current block
                hmac_md5(currentBlock, BLOCK_SIZE,
                    digest);

                // compare the calculated MAC to the one
                // that was sent
                if(memcmp(digest, currentMac,
                    DIGEST_LENGTH) == 0)
                {
```

```
                        // if they match, write current
                        // block to temp file
                        fwrite(currentBlock,
                            sizeof(char), BLOCK_SIZE,
                            temp);
                }
        }
        // go back to the start of the file
        rewind(temp);
        // invert the AONT
        inverse_transform(temp, ofp);
}

#define CWIN_PT_TEST
#ifdef CWIN_PT_TEST
int main(int argc, char *argv[])
{
        FILE *ifp;
        FILE *ofp;

        // seed the random number generator
        srand(time(NULL));

        ifp = fopen(argv[2], "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                return 1;
        }

        ofp = fopen(argv[3], "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                return 2;
        }

        MACkey = argv[4];

        if(strcmp(argv[1], "e") == 0)
        {
                // encrypt
                printf("Encrypting\n");
                chaff(ifp, ofp);
        }
        else if(strcmp(argv[1], "d") == 0)
        {
                // decrypt
                printf("Decrypting\n");
                winnow(ifp, ofp);
        }

        fclose(ifp);
        fclose(ofp);

}
#endif
```

B.7 File: cwin_hybrid_pt.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "rsa.h"
#include "aont_pt.h"
#include "hmac.h"

#define NUM_CHAFF_BLOCKS 132
#define PT_LEN  24
#define KEY_LEN 128
#define BLOCK_SIZE 128
#define DIGEST_LENGTH 16

#define RSA_N   "9292758453063D803DD603D5E777D788" \
                "8ED1D5BF35786190FA2F23EBC0848AEA" \
                "DDA92CA6C3D80B32C4D109BE0F36D6AE" \
                "7130B9CED7ACDF54CFC7555AC14EEBAB" \
                "93A89813FBF3C4F8066D2D800F7C38A8" \
                "1AE31942917403FF4946B0A83D3D3E05" \
                "EE57C6F5F5606FB5D4BC6CD34EE0801A" \
                "5E94BB77B07507233A0BC7BAC8F90F79"

#define RSA_E   "10001"

#define RSA_D   "24BF6185468786FDD303083D25E64EFC" \
                "66CA472BC44D253102F8B4A9D3BFA750" \
```

```
                "91386C0077937FE33FA3252D28855837" \
                "AE1B484A8A9A45F7EE8C0C634F99E8CD" \
                "DF79C5CE07EE72C7F123142198164234" \
                "CABB724CF78B8173B9F880FC86322407" \
                "AF1FEDFDDE2BEB674CA15F3E81A1521E" \
                "071513A1E85B5DFA031F21ECAE91A34D"

#define RSA_P   "C36D0EB7FCD285223CFB5AABA5BDA3D8" \
                "2C01CAD19EA484A87EA4377637E75500" \
                "FCB2005C5C7DD6EC4AC023CDA285D796" \
                "C3D9E75E1EFC42488BB4F1D13AC30A57"

#define RSA_Q   "C000DF51A7C77AE8D7C7370C1FF55B69" \
                "E211C2B9E5DB1ED0BF61D0D9899620F4" \
                "910E4168387E3C30AA1E00C339A79508" \
                "8452DD96A9A5EA5D9DCA68DA636032AF"

#define RSA_DP  "C1ACF567564274FB07A0BBAD5D26E298" \
                "3C94D22288ACD763FD8E5600ED4A702D" \
                "F84198A5F06C2E72236AE490C93F07F8" \
                "3CC559CD27BC2D1CA488811730BB5725"

#define RSA_DQ  "4959CBF6F8FEF750AEE6977C155579C7" \
                "D8AAEA56749EA28623272E4F7D0592AF" \
                "7C1F1313CAC9471B5C523BFE592F517B" \
                "407A1BD76C164B93DA2D32A383E58357"

#define RSA_QP  "9AE7FBC99546432DF71896FC239EADAE" \
                "F38D18D2B2F0E2DD275AA977E2BF4411" \
                "F5A3B2A5D33605AEBBCCBA7FEB9F2D2F" \
                "A74206CEC169D74BF5A8C50D6F48EA08"

rsa_context rsaContext;
unsigned char *MACkey;

void initRSA()
{
        memset( &rsaContext, 0, sizeof( rsa_context ) );

    rsaContext.len = KEY_LEN;
    mpi_read_string( &rsaContext.N , 16, RSA_N  );
    mpi_read_string( &rsaContext.E , 16, RSA_E  );
    mpi_read_string( &rsaContext.D , 16, RSA_D  );
    mpi_read_string( &rsaContext.P , 16, RSA_P  );
    mpi_read_string( &rsaContext.Q , 16, RSA_Q  );
    mpi_read_string( &rsaContext.DP, 16, RSA_DP );
    mpi_read_string( &rsaContext.DQ, 16, RSA_DQ );
    mpi_read_string( &rsaContext.QP, 16, RSA_QP );

}

void chaff(FILE *ifp, FILE *ofp)
{
        FILE *temp, *chaffFile;
        unsigned char currentBlock[BLOCK_SIZE],
            chaffPacket[BLOCK_SIZE];
        unsigned char rsa_plaintext[PT_LEN],
            rsa_ciphertext[KEY_LEN],
            digest[DIGEST_LENGTH];
        int chaffPositions[NUM_CHAFF_BLOCKS],
            reversedChaffPositions[NUM_CHAFF_BLOCKS];
        int filesize, numBlocks, i, chaffIndex, test;

        // apply package transform to input file
        // and write to temp file
        temp = tmpfile();
        transform(ifp, temp);

        // calculate size of transformed file
        fseek (temp , 0 , SEEK_END);
        filesize = ftell (temp);
        rewind (temp);

        // calculate the number of blocks
        numBlocks = (int)floor((double)filesize /
            BLOCK_SIZE);

        // calculate the positions of the chaff packets
        calculateChaffPositions(chaffPositions,
            NUM_CHAFF_BLOCKS, numBlocks);

        hmac_init(MACkey, 16);
        // calculate MAC for chaff positions
        hmac_md5(chaffPositions, NUM_CHAFF_BLOCKS,
            digest);

        chaffFile = tmpfile();
        // if the machine is little endian, transform
        // the chaff positions to big endian order
        // and write them to the file
```

```
if(!isBigEndian())
{
        for(i = 0; i < NUM_CHAFF_BLOCKS; i++)
        {
                reversedChaffPositions[i] =
                    reverseInt(chaffPositions[i]);
        }
        fwrite(reversedChaffPositions,
            sizeof(int), NUM_CHAFF_BLOCKS,
            chaffFile);
}
else
// write the chaff positions to file as they are
{
        fwrite(chaffPositions, sizeof(int),
            NUM_CHAFF_BLOCKS, chaffFile);
}

rewind(chaffFile);

// encrypt chaff positions using RSA and
// write to file
for(i = 0; i < (NUM_CHAFF_BLOCKS * 4) / PT_LEN;
    i++)
{
        memset(rsa_plaintext, 0, PT_LEN);
        fread(rsa_plaintext, sizeof(char),
            PT_LEN, chaffFile);

        test = rsa_pkcs1_encrypt(&rsaContext,
            RSA_PUBLIC, PT_LEN, rsa_plaintext,
            rsa_ciphertext);
        if(test != 0)
        {
                printf("RSA Encryption
                    failed\n");
                exit(1);
        }
        fwrite(rsa_ciphertext, sizeof(char),
            KEY_LEN, ofp);
}

// write the MAC of the chaff positions to file
fwrite(digest, sizeof(char), DIGEST_LENGTH, ofp);
        chaffIndex = 0;
        for(i = 0; i < numBlocks + NUM_CHAFF_BLOCKS; i++)
        {
                // if the current index should be a
                // chaff block, generate
                // a fake block and write it to file
                if(i == chaffPositions[chaffIndex])
                {
                        generateChaffPacket(chaffPacket,
                            BLOCK_SIZE);
                        fwrite(chaffPacket,
                            sizeof(char), BLOCK_SIZE,
                            ofp);
                        chaffIndex++;
                }
                // otherwise read the wheat block and
                // write it to the output file
                else
                {
                        memset(currentBlock, 0,
                            BLOCK_SIZE);
                        fread(currentBlock,
                            sizeof(char), BLOCK_SIZE,
                            temp);
                        fwrite(currentBlock,
                            sizeof(char), BLOCK_SIZE,
                            ofp);
                }
        }
}

void winnow(FILE *ifp, FILE *ofp)
{
        FILE *temp, *chaffFile;
        unsigned char currentBlock[BLOCK_SIZE],
            chaffPacket[BLOCK_SIZE];
        unsigned char rsa_decrypted[PT_LEN],
            rsa_ciphertext[KEY_LEN],
            chaffPosMAC[DIGEST_LENGTH],
            digest[DIGEST_LENGTH];
        int chaffPositions[NUM_CHAFF_BLOCKS];
        int filesize, numBlocks, i, chaffIndex, test,
            len, fileLocation;
```

```
chaffFile = tmpfile();

// calculate size of input file
fseek (ifp , 0 , SEEK_END);
filesize = ftell (ifp);
rewind (ifp);

// read in the encrypted chaff packet positions,
// decrypt the data, and write the output to a
// temporary file
for(i = 0; i < (NUM_CHAFF_BLOCKS * 4) / PT_LEN;
    i++)
{
        fread(rsa_ciphertext, sizeof(char),
            KEY_LEN, ifp);

        test = rsa_pkcs1_decrypt(&rsaContext,
            RSA_PRIVATE, &len, rsa_ciphertext,
            rsa_decrypted);
        if(test != 0)
        {
                printf("RSA Decryption
                    failed\n");
                exit(2);
        }
        fwrite(rsa_decrypted, sizeof(char),
            PT_LEN, chaffFile);
}

// read the MAC for the chaff positions
fread(chaffPosMAC, sizeof(char), DIGEST_LENGTH,
    ifp);

// get current position in the file
fileLocation = ftell(ifp);

rewind(chaffFile);

// read in the decrypted chaff positions
fread(chaffPositions, sizeof(int),
    NUM_CHAFF_BLOCKS, chaffFile);

// if the machine is little endian, need to
// transform the order of the chaff postiions
if(!isBigEndian())
{
        for(i = 0; i < NUM_CHAFF_BLOCKS; i++)
        {
                chaffPositions[i] =
                    reverseInt(chaffPositions[i]);
        }
}

// check the MAC for the chaffPositions
hmac_init(MACkey, 16);
hmac_md5(chaffPositions, NUM_CHAFF_BLOCKS,
    digest);
if(memcmp(digest, chaffPosMAC, DIGEST_LENGTH) !=
    0)
{
        // MACs did not match, exit
        printf("Error: MAC of chaff positions
            was invalid\n");
        exit(4);
}

// calculate the number of blocks
numBlocks = (int)floor((double)(filesize -
    fileLocation) / BLOCK_SIZE);

chaffIndex = 0;
temp = tmpfile();
for(i = 0; i < numBlocks; i++)
{
        memset(currentBlock, 0, BLOCK_SIZE);
        // read in the next block
        fread(currentBlock, sizeof(char),
            BLOCK_SIZE, ifp);

        // if the index is a chaff positions,
        // move on to the next block
        if(i == chaffPositions[chaffIndex])
        {
                chaffIndex++;
        }
        // else write the block to a temp file
        else
        {
                fwrite(currentBlock,
                    sizeof(char), BLOCK_SIZE,
```

```
                        temp);
                }
        }

        // perform inverse tranform on temp file
        rewind(temp);
        inverse_transform(temp, ofp);
}

#define HYBRID_PT_TEST
#ifdef HYBRID_PT_TEST
int main(int argc, char *argv[])
{
        FILE *ifp;
        FILE *ofp;

        // seed the random number generator
        srand(time(NULL));

        // initialise RSA
        initRSA();

        ifp = fopen(argv[2], "rb");

        if(ifp == NULL)
        {
                printf("Unable to open input file");
                return 1;
        }

        ofp = fopen(argv[3], "wb");
        if(ofp == NULL)
        {
                printf("Unable to open output file");
                return 2;
        }

        MACkey = argv[4];

        if(strcmp(argv[1], "e") == 0)
        {
                // encrypt
                printf("Encrypting");
                chaff(ifp, ofp);
        }
        else if(strcmp(argv[1], "d") == 0)
        {
                // decrypt
                printf("Decrypting");
                winnow(ifp, ofp);
        }

        fclose(ifp);
        fclose(ofp);

}
#endif
```