Citation for published version:
Larkin, J 2006, Implementation of chaffing and winnowing: Providing confidentiality without encryption. Computer Science Technical Reports, no. CSBU-2006-10, Department of Computer Science, University of Bath.

Publication date:
2006

Department of Computer Science

UNIVERSITY OF BATH

Technical Report

Undergraduate Dissertation: Implementation of Chaffing and Winnowing: providing confidentiality without encryption

John Larkin

Technical Report 2006-10 **June 2006**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Implementation of Chaffing and Winnowing: providing confidentiality without encryption

John Larkin
BSc Computer Science

2006

Implementation of Chaffing and Winnowing: providing confidentiality without encryption

Submitted by John Larkin

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 specifically acknowledged, it is the work of the author.

Signed ....

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed ....

Abstract

Chaffing and winnowing is a new concept for providing data privacy, without encryption. Several chaffing and winnowing schemes are implemented using ideas previously proposed and a new hybrid method proposed here, which incorporates public-key cryptography. Experiments are performed to compare the schemes implemented to traditional encryption algorithms. It is found that chaffing and winnowing is a viable alternative to using these techniques.

Acknowledgements

I would like to thank Dr Russell Bradford for his help and advice, while supervising this project. I would also like to thank Kelly and my family for support throughout the project and proof reading.

Contents

Chapter 1

Introduction

1.1 Cryptography

Cryptography is the art and science of keeping information secret. Cryptography techniques have been around for centuries and have been constantly evolving. Julius Caesar used cryptographic techniques to communicate with his generals and cryptography played a big part during World War II. In the age we live in confidentiality is an increasingly large problem, especially in communication over electronic systems. Companies and individuals need to ensure that important information sent over public networks is kept confidential and is not modified. This is generally achieved through encryption and digital signatures.

1.2 Chaffing and Winnowing

Ronald Rivest has proposed a way to achieve confidentiality without encryption, called “Chaffing and Winnowing”. Rivest is one of the creators of the RSA cryptosystem [Rivest et al., 1978], which one of the most widely used public-key encryption algorithms, he is also the creator of the MD5 one-way hash function and is a respected cryptographer. The RSA cryptosystem is considered “a de facto standard in much of the world” [Schneier, 1993]. He poses the question of whether Governments and Authorities should be able to gain access to encrypted messages and by introducing a scheme that does not encrypt any data, shows that there is a method available that may not be liable to these restrictions. Whether Chaffing and Winnowing would stand up to these claims against legal systems in different parts of the world is debatable.

1.3 Aims

The aims of this project are to investigate and implement Chaffing and Winnowing and to see if it could be a real alternative to using traditional encryption techniques. To do this, several different implementations of the scheme will be produced and will be compared to current standard techniques, based on certain metrics such as data expansion after encryption and execution time. The project will also look to see if any new Chaffing and Winnowing schemes can be produced, that improve on the original ideas.

Chapter 2

Literature Review

2.1 What is Chaffing and Winnowing?

Chaffing and Winnowing is a new type of cryptographic technique introduced by Ronald Rivest in [Rivest, 1998a]. Chaff is a farming term, which describes useless parts of grain and winnowing is “to separate chaff from grain” [Webster’s]. Chaffing and winnowing is different to common approaches of achieving confidentiality. The main techniques for achieving confidentiality are Stenography and Encryption. Stenography is a method of hiding a secret message within another message, such that it is not obvious the secret message is present. Encryption involves transforming a plaintext message into a ciphertext and reversing the process, transforming the ciphertext back into plaintext, known as decryption.

This technique provides confidentiality through authentication. It splits the original message into blocks, appends a ‘Message Authentication Code’ or ‘MAC’ to the block to create a packet and then inter-spaces random packets within the valid packets, which have invalid MAC’s appended to them. A MAC is computed as a function of the contents of the block. The receiver of the message discards the packets with invalid MAC’s and re-assembles the original message. This process is not the same as common encryption techniques because the original message is still present, in the "clear". It is also common to append a serial number to the packet so it has the form: (serial, message block, MAC). We will refer to a packet containing real data with a valid MAC as "wheat" and a packet containing random data with an invalid MAC as "chaff". During this dissertation we may refer to Chaffing and Winnowing as "encrypting" and "decrypting" messages, this is because these are common terms to refer to the transformation of plaintext messages into ciphertext and the reverse. This does not mean that Chaffing and Winnowing is a traditional encryption scheme, although it is comparable in this way.

> "There is a secret key shared by the sender and the receiver to authenticate the origin and contents of each packet—the legitimate receiver, knowing the secret authentication key, can determine that a packet is authentic by re-computing the MAC and comparing it to the received MAC. If the comparison fails, the packet and its MAC are automatically discarded." [Rivest, 1998a]

This method does not provide good confidentiality if the packets contain sentences, for example if a packet contained "Bank Account No: 12345678", an adversary would still be able to see this. The chaff packet would need to contain very similar but false information, in this case the chaffing process would have to be intelligent to be able to construct a similar sentence but even then the adversary would still have access to the correct information. For these reasons, this method is not a feasible option. Instead [Rivest, 1998a] suggests splitting the message into bits, this is discussed in the next section. The MAC algorithm needs to act as a pseudo random function, making it difficult for an adversary to distinguish chaff packets from valid ones. Otherwise the MAC could "leak" information about the message [Rivest, 1998a]. Rivest also notes that adding chaff is a keyless procedure and therefore does not necessarily have to be done by the person creating valid packets from the original message. This is an important point in the argument about whether this method should be considered encryption or not, as someone could merely authenticate blocks of their message and without knowing, another person could add chaff to the message.

2.2 Chaffing and Winnowing Techniques

2.2.1 Bit-by-bit method

The bit-by-bit method of chaffing and winnowing described in [Rivest, 1998a], creates a packet for each bit in the original message, a serial number is appended to the message, a MAC is computed for the packet and appended to the packet. A chaff packet is produced for every valid packet containing the complementary bit, the same serial number as the corresponding packet and an invalid MAC. Packets have the form:

Valid packet: (serial, bit, MAC)
Invalid packet: (serial, complementary bit, invalid MAC)

In this method there must be a chaff packet for every valid packet, otherwise it is easy for the adversary to assume that when there is only one packet for a given serial number, it is valid. When there is a chaff packet for every grain packet this method is very secure, the adversary would essentially have to break the MAC algorithm to distinguish wheat from chaff. The obvious drawback to this method is the size of the message produced after validating the packets and adding chaff.

> "For a message of m bits $2m(1+p+l)$ bits are transmitted, where p is the length of a counter and l is the length of output of F."
> [Bellare and Boldyreva, 2000] (Where F is the MAC function)

So for example, a message 30Kb in size (245,760 bits) with a 32 bit serial number and a 128 bit MAC, after being validated and chaff added to it, grows from 30Kb to approximately 75Mb! This is a considerable size increase and makes transmitting large documents over networks completely impractical. Bleichenbacher suggested that the expansion can be reduced by creating either a chaff or grain packet for each bit, then when winnowing the message if the packet is valid use the bit value and if it is invalid use the complementary bit value, this halves the data expansion [Bellare and

Boldyreva, 2000]. Even so the data expansion would still be impractical when transmitting large messages.

2.2.2 All-Or-Nothing Transform

To make chaffing and winnowing more efficient, it is suggested in [Rivest, 1998a] that an "All-Or-Nothing" transform or "AONT" is used to pre-process the message. This is a keyless, invertible transform, which effectively makes the original message look like random noise. It has the property that inversion of the message is very hard if any block is missing but someone that has all of the blocks can reconstruct the message easily. If the transformed message is then "encrypted" block by block, an adversary cannot find out anything about the message without decrypting all the blocks of ciphertext. There are several candidates for the AONT transforms. As a result of pre-processing the message with an AONT, a chaff packet no longer needs to be added for every wheat packet and the packet can contain more data, [Rivest, 1998a] suggests 1024-bit blocks. This method is much more efficient and seems to provide a very good level of confidentiality, depending on the AONT used. The number of bytes transmitted in this scheme is given by:

$$(max(\lceil m/b \rceil, M) + s') \cdot (b + d)$$

where m is the number of bytes in the message, s' is the number of chaff blocks, b is the block size, M is the minimum number of blocks output by the AONT and d is the number of bytes in the MAC digest. We can see from this that when encrypting a small message the expansion is very large, as there is a fixed overhead of chaff blocks. However the data expansion becomes better as the messages to be encrypted become larger. If a fixed number of chaff blocks is chosen, for example 128, the AONT produces a minimum of 128 blocks (both suggested in [Bellare and Boldyreva, 2000]) and the block size is 128 bytes (1024 bits, as Rivest suggests) then the number of bytes transmitted for a message 30Kb in size is $\sim$ 53Kb and the number of bytes transmitted for a message 500Kb is $\sim$ 581Kb. This is considerably better than the example we considered in Section 2.2.1.

2.3 AONT Candidates

2.3.1 Package Transform

The candidate Rivest proposes for the AONT is called the "package transform" [Rivest, 1997], where the message is split into blocks and a key K randomly chosen. Each block is transformed and an extra block added consisting of the Exclusive-OR of K and a hash of all previous blocks. Then anyone with all of the blocks can reconstruct the message.

> "The legitimate communicants thus pay a penalty of approximately a factor of three in the time it takes them to encrypt or decrypt in all-or-nothing mode, compared to an ordinary separable encryption mode. However, an adversary attempting a brute-force attack pays a penalty of a factor of t, where t is the number of blocks in the ciphertext." [Rivest, 1997]

However [Boyko, 1999] shows that Rivest's definition of AONTs is not provably secure and provides an AONT construction that is provably secure, we would prefer a system that is provably secure and minimises data expansion.

2.3.2 Optimal Asymmetric Encryption Padding

"Optimal Asymmetric Encryption Padding" or "OAEP" is a method described in [Bellare and Rogaway, 1994]. It is a way to provide provable security and efficiency. [Boyko, 1999] presents OAEP as an AONT and shows Rivest's definition of an AONT is not secure. It also shows that no AONT can provide substantially better security than OAEP. OAEP uses a "generator function" $G : \{0,1\}^{k_0} \rightarrow \{0,1\}^n$ and a "hash function"

$H : \{0,1\}^n \rightarrow \{0,1\}^{k_0}$, n is the length of the message and k_0 is the security parameter. The OAEP transform is defined as

$$OAEP^{G,H}(x,r) = x \oplus G(r) \mid\mid r \oplus H(x \oplus G(r))$$

where $||$ denotes concatenation, x is the message and r is a random string chosen from $\{0,1\}^{k_0}$. As with Rivest's AONT, this method provides much greater efficiency than the bit-by-bit method but is provably secure by [Bellare and Boldyreva, 2000]. OAEP is widely used with encryption, RSA crypto-systems [Rivest et al., 1978] are often used with OAEP and it is part of the PKCS (Public Key Cryptography Standards) #1.

2.3.3 BEAR Pre-processing

Using the BEAR [Anderson and Biham, 1996] block cipher as an AONT is the method used in Chaffinch [Clayton and Danezis, 2003], a variant of Chaffing and Winnowing. BEAR is constructed from keyed hash function, although in Chaffinch no key is used, so that the scheme cannot be considered encryption. The message is preceded by some random data each time it is sent, so that transforming a message twice will not lead to the same output.

2.4 "A New Chaffing and Winnowing Scheme"

A new chaffing and winnowing scheme was suggested in [Bellare and Boldyreva, 2000]. It involves applying OAEP as an AONT to the message and then "encrypting" the first block of the message using the original bit-by-bit method. The remaining part of the message is authenticated to make sure the scheme is still considered a chaffing and winnowing scheme. This scheme provides savings in message expansion because only the first block is expanded, so there will only be a fixed overhead, no matter how large the original message is.

2.5 Message Authentication Codes

A "Message Authentication Code" or "MAC" is the method proposed in chaffing and winnowing to authenticate wheat packets. MAC algorithms generally use a secret key to authenticate a message and are used to preserve the integrity of the message. [Rivest, 1998a] notes that the MAC algorithm must be a Pseudo Random Function and suggests using the HMAC (hashed based MAC) construction described in [Krawczyk et al., 1996], which uses a cryptographic hash function.

> "A MAC is a function which takes the secret key k (shared between the parties) and the message m to return a tag $\mathrm{MAC}_k(m)$. The adversary sees a sequence $(m_1,\ a_1)$, $(m_2,\ a_2)$, $\ldots$, $(m_q,\ a_q)$ of pairs of messages and their corresponding tags (that is, $\mathrm{a}_i = \mathrm{MAC}_k(m_i)$) transmitted between the parties." [Krawczyk et al., 1996]

The HMAC algorithm uses cryptographic hash functions as 'black-box', this means that for a MAC implementation the hash function used can be changed very easily, this is useful if it is found that the hash function being used is no longer secure. Popular hash functions to use in HMAC include MD5 [Rivest, 1992] and SHA-1 [SHA]. The MAC algorithm uses an inner pad consisting of the byte 0x5C repeated so that it is as big as the input block b and an outer pad consisting of the byte 0x36 repeated so it is also the size of b. It computes the Exclusive-OR of the key and the inner pad, the hash function is applied to the result of this. The exclusive-or of the key and the outer pad is computed and the hash function is applied to this and the result of the previous step.

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

Where F is the hash function, x is the message and k is the secret key. The main attacks on hash functions are "birthday attacks". This is the method of finding a "collision" in the hash function, having m and m'

such that $f(m) = f(m')$. This is based on the birthday paradox that if there are 23 people in a room there is greater than a 50% chance that 2 people have the same birthday and the probability increases as the number of people increases. Birthday attacks are generally used to forge messages. To find a collision in a hash function you would need to compare around $2^{l/2}$ outputs of the hash function where l is the output length of the hash function. In the case of a MAC you would need to compare around $2^{l/2}$ outputs using the same key. This would require the key owner to generate this amount of messages, as an adversary does not have access to the key. "For values of $l \geq 128$ the attack becomes totally infeasible" [Bellare and Boldyreva, 2000], so in the case of using MD5 in a MAC this sort of attack is infeasible. [Krawczyk et al., 1997] notes that it is good practice to truncate the output of the MAC and output only part of the bits.

> "We recommend the output length t not be less than half the length of the hash output (to match the birthday bound) and not less than 80 bits (a suitably lower bound on the number of bits that need to be predicted by an attacker)." [Krawczyk et al., 1997]

The way to denote a MAC that truncates bits is HMAC-SHA1-80, where SHA1 is the hash function being used and 80 is the number of bits we are truncating to. MACs are widely used in cryptography and [Krawczyk et al., 1996] explains the HMAC in a practical way. There are also other types of MAC such as UMAC (Universal MAC) which is faster than HMAC but more complex [Black et al., 1999] and MACs constructed from block ciphers such as DES-CBC MAC, which is a Federal Standard [FIPS, 1985].

2.6 Why Is Chaffing and Winnowing Needed?

Rivest introduced Chaffing and Winnowing as not being "encryption" in [Rivest, 1998a]. This is the most important feature of Chaffing and Winnowing, Rivest argues that because of the existence of techniques such as Chaffing and Winnowing, efforts to regulate encryption by law must fail, as confidentiality can be efficiently achieved in this way, without encryption. However, McHugh shows that in the refinement of chaffing and winnowing he presents in [McHugh, 2000] is equivalent to encryption and notes that “although the equivalence only appears in the limiting case, its implications for policy makers are unclear”. Law enforcement agencies in the U.S. are pushing to regulate encryption as criminals or terrorists could send messages that the law enforcement agencies would be unable to decipher. There have been proposals that users would be required to register their encryption keys with law enforcement agencies and key-recovery proposals that give government agencies “back-door” access to the keys.

> “In a typical key recovery scheme, an encrypted version of the message encryption key is sent along with each message. An FBI-authorized key-recovery centre can use a master backdoor key to decrypt the message key, which is then used to decrypt the message. In my opinion these, systems would satisfy no one. They are easy to circumvent: spies and criminals could modify the encryption software to disable the key-recovery features or they could simply download alternative software from the Internet” [Rivest, 1998b]

Rivest makes a good point by saying that spies and criminals could remove the “back-door” access to their messages. Rivest also notes that confidence would be lost in the government, as people would not be able to have truly private conversations. This point is highly debatable, on the one hand, as people should have the right to free speech without the authorities being able to read your messages but on the other, is it better that we allow terrorists and criminals to exchange secret information easily, we are not going to try and decide which view is right here.

Another interesting property of Chaffing and Winnowing is that it can easily achieve the concept of “deniable encryption” [Canetti et al., 1997]. This is when it is possible to produce a ciphertext c, where $c = E(m_1, r)$, m_1 is the message and r is random data such that r' can be found so using a fake message m_2, $c = E(m_2, r')$. This means that if an adversary could force the sender to reveal the random bits and the key, they could supply the fake ones and therefore reveal the fake message. In Chaffing and Winnowing this idea can be achieved by using more than one wheat source, authenticated with different keys. In this case wheat packets for one message become the chaff packets for another message as well as some real chaff packets being present.

> “I note that it is possible for a stream of packets to contain more than one subsequence of ‘wheat’ packets, in addition to the chaff packets. Each wheat subsequence would be recognized separately using a different authentication key.” [Rivest, 1998a]

If law enforcement then required to see the deciphered message, the key disclosing a fake or cover message could be used, so that the real message would still be undisclosed. Chaffinch [Clayton and Danezis, 2003] extends this idea and presents a system based on Chaffing and Winnowing that may stand up to the UK’s Regulation of Investigatory Powers Act 2000. Chaffinch always sends a “cover message” and the real message(s) is sent with it. If law enforcement required the message to be deciphered, the “cover message” is always revealed first. Chaffinch also makes some design changes to Rivest’s proposed method. However it could be viewed that not revealing the correct message when asked, is against the law.

2.7 How Does Chaffing and Winnowing Compare To Using Traditional Encryption Techniques?

The Chaffing and Winnowing schemes we have discussed are symmetric schemes, the same key that is used to “encrypt” the message is used to “decrypt” the message, so they should be compared to symmetric encryption schemes. “Data Encryption Standard” or “DES” [FIPS, 2001b] is a symmetric block cipher that has been around for about 30 years and is still widely used. It became a Federal Standard, which has since been withdrawn (but replaced with an updated version). DES encrypts data in 64 bit blocks, 64 bits of plaintext are transformed into 64 bits of ciphertext. 16 rounds of the same operations are performed on each block to encrypt it. DES uses

56 bit keys and has been found to be insecure, it is now generally used as Triple-DES - known as “TDES” or “3DES”, where the message is encrypted with DES 3 times, each time with a different key, TDES replaced the original DES standard.

“Advanced Encryption Standard” or “AES” [FIPS, 2001a] was designed to replace DES. AES has twice the block size - 128 bit, larger keys which can vary in length. It provides better security than DES and is also faster. There were many candidates proposed for AES, Rijndael was the candidate that was chosen. Chaffing and winnowing should be comparable in speed to DES and AES because it is build upon cryptographic hash functions, which are generally very fast. However we would expect the data expansion to be much greater in chaffing and winnowing, because DES and AES do not expand the original message at all and the chaffing and winnowing scheme adds a MAC to each block and a fixed number of chaff packets.

2.8 Symmetric Versus Asymmetric Schemes

The chaffing and winnowing schemes we have described so far are symmetric ones - the same key is used to encrypt the message as to decrypt the message. DES and AES are symmetric encryption schemes, these schemes have benefits in that they are generally very fast and the ciphertext is the same size as the plaintext (not in the case of chaffing and winnowing). In 1976, Diffie and Hellman [Diffie and Hellman, 1976] introduced asymmetric encryption, known as public-key cryptography. In this scheme two keys are used, a public encryption key and a private decryption key, each person has a key pair. Their encryption key is made publicly available and they keep their decryption key private. If a message has been encrypted with your public key, only you will be able to decrypt it. In this scheme it is computationally very difficult to find one key from the other. Public key algorithms are generally very slow because they work modulo very large primes, in the region of a few hundred digits long. They also expand the data they encrypt. However they provide far greater security benefits in exchanging keys. It is not fair to directly compare symmetric and asymmetric schemes to each other, as they are very different. RSA is an example of a public key crypto-system and is widely used. As a result of the benefits and drawbacks of each type of scheme, they are often used in conjunction with one another, in what are called hybrid schemes. In these schemes the message will be symmetrically encrypted with a random session key, using DES for instance, then the session key will be asymmetrically encrypted, perhaps using RSA, which is sent with the encrypted message. To recover the message, the session key is recovered using the correct private key and the message is decrypted using the session key, this is called "digital enveloping". As the bulk of the message is encrypted using a symmetric scheme it is very fast and does not expand the data. An asymmetric scheme is used to recover the session key so it has the desirable key-management properties of a public-key system and because it is only the session key that is pubic-key encrypted (typically ~ 192 bits) the data expansion is minimal.

It would be an advantage if chaffing and winnowing could incorporate these hybrid concepts to utilise these benefits. Rivest does note that the MACs can be replaced by a special type of digital signatures called "designated verifier signatures" [Jakobsson et al., 1996]. These type of signature allow only the designated verifier to check if a signature is valid, so for use within chaffing and winnowing, only the designated verifier would be able to check which blocks are wheat and which are chaff. This method would slow down the scheme considerably because public key operations would need to be applied to every block of the message. This would not really be a hybrid method because it extensively uses public key cryptography, a better method would allow chaffing and winnowing to use symmetric and asymmetric methods together in an efficient way.

2.9 Technology Considerations

2.9.1 Implementation Language

The programming language considerations for implementation of this project are C, Perl and Java. All of these languages support bit-wise operations, bit-shifts, and file reading and writing. The benefit of C and Java is that the author knows them well, which would better suit the time scale of the project.

As a low-level language, C may run faster than Java. Although Java is generally more portable across platforms. C can be portable if implementation dependant features are not used.

2.9.2 “Endianness” of Target Machines

The implementations of this project should be able to run across UNIX, Mac and Windows platforms, so they will need to take into account the “Endianness” of the machines they are running on. “Endianness” refers to the order in which bytes are stored within a word, on a particular machine. Big-endian refers to when the byte with the highest precedence is stored first and little-endian is when the byte with the lowest precedence is stored first. Solaris UNIX on SPARC machines is big-endian, Windows on Intel machines is little-endian and Mac on Power PC machines is big-endian. Java makes “Endianness” transparent as it stores everything in big-endian order, no matter which machine it is running on.

2.10 Conclusion

In this chapter we have introduced the concept of chaffing and winnowing, described the chaffing and winnowing schemes that have been previously put forward and investigated the underlying concepts needed to implement them. In the next chapter we will discuss what our requirements for the project are, based on the knowledge gained here. Then we will go on to choose which schemes to implement in this project. We will also try to improve these existing methods or devise new ones to create a hybrid chaffing and winnowing scheme. From our initial analysis it seems that chaffing and winnowing should be comparable in speed to traditional encryption methods, although the chaffing and winnowing schemes will carry a greater data expansion overhead. We will see if these remarks are true when the implementations are tested against other methods.

Chapter 3

Requirements

Here we specify the requirements for this project, which have been derived from the initial aims of the project combined with what has been learnt from the literature review. The requirements describe the essential properties of each component and also some general requirements of the project.

3.1 Message Authentication Codes

An implementation of a Message Authentication Code algorithm will be produced, this is one of the underlying pieces of technology that the project requires. The implementation will use a pseudo-random cryptographic hash function, so that no information about the message is leaked by the hash function. The hash function being used must be easily inter-changeable so if at any stage a different hash function needs to be used, it can be changed without changing the function prototypes in the MAC implementation. This makes the hash function transparent to the programs using the MAC implementation. The MAC implementation should have the property that computing the MAC of a particular input with different keys, should produce completely different digests.

3.2 Initial Prototype

An initial prototype will be produced as a "proof-of-concept" to help the author learn about the chaffing and winnowing method and some of the problems associated with implementing it. The scheme that will be implemented as the initial prototype will be the "bit-by-bit" method. We have learnt from the literature review that this is the simplest chaffing and winnowing scheme and it is also the least efficient in terms of data expansion by a long way and because of this it would not be comparable to traditional encryption methods.

3.3 All-Or-Nothing Transform Implementations

Two AONT implementations will be produced. They will be designed so that in a chaffing and winnowing scheme the AONT being used can easily be inter-changed, to enable re-use of code. The AONT implementations will transform a message using some randomly chosen data, so the output of a transformation on the same message will be different each time. The AONT will be able to accept a parameter specifying the minimum amount of blocks to output. The AONT will have the property that if all of the blocks are present, it will be easy to reconstruct the original message and any of the blocks are missing then it will be very difficult to reconstruct the original message.

3.4 Hybrid Implementation

A method will be designed and implemented, that combines the characteristics of using a symmetric key for speed and message expansion benefits and asymmetric keys for key management benefits. From research done in the literature review this is how many techniques are currently implemented, to benefit from these advantages. The only current suggestion to incorporate asymmetric keys into chaffing and winnowing is to use "Designated Verifier Signatures", however from initial analysis it seems that this would slow down the scheme considerably. A new method will need to be devised from the technologies investigated.

3.5 Production Implementations

At least three "production" chaffing and winnowing implementations will be implemented using the AONT and hybrid tools created. They will consist of two AONT implementations and one or more hybrid implementations. These are the implementations that will be fully tested for speed and data expansion and compared with traditional techniques.

3.6 Performance Requirements

The programs produced should be written to run as efficiently as possible. Aside from the fact that it is good practice to write programs to be efficient, one of the aims of this project is to compare chaffing and winnowing to traditional encryption methods and so the implementations need to be able to compete on speed. From initial analysis of the methods chaffing and winnowing should be similar to traditional encryption methods and this may be an area where it can improve on them.

3.7 Data Expansion Requirements

The programs written should minimise data expansion of encrypted messages. From the literature review we have seen that this is an area, which may be less efficient than traditional encryption methods. Due to the nature of chaffing and winnowing, chaff is always going to be added to a message and as a result there will always be some overhead. Also because the scheme is based on authenticating valid packets, authentication data will also always need to be added. So minimise the data expansion, data should always be represented in the smallest form possible.

3.8 Platform Requirements

The implementations produced in this project should be able to run on different platforms, as traditional encryption technologies can. The platforms available for testing in this project are Mac OS X (PowerPC), Windows XP and Solaris 8. The code produced should be portable across these platforms and should allow for "encryption" and "decryption" across platforms. For example if a file was "encrypted" on Mac OS X it should be able to be decrypted on "Windows XP".

3.9 Security Requirements

For chaffing and winnowing to be considered an alternative to traditional techniques it must provide a similar level of security. Security analysis of chaffing and winnowing has been carried out and this was researched in the literature review, so the recommendations must be taken into account.

3.10 Program Requirements

The programs written should adhere to these coding standards:

- The code should be well structured, properly indented and easy to read.
- The code should provide informative comments in appropriate places.
- The code should be easy to debug.
- Implementation dependant features of the language should not be used to provide better portability.
- The programs produced should be very modular to enable re-use of code.

3.11 Implementation Language Requirements

As discussed in the literature review, the language chosen for implementation must be able to perform certain operations and have certain characteristics:

- Must be able to perform bit-wise operations.
- Must have file input/output features.
- Must be portable to the platforms specified.
- Should be a commonly used language to be able to re-use publicly available code.
- Should be a language the author is familiar with to maximise time available.

Chapter 4

Design

In this chapter we discuss the design of the schemes and components that we will implement. Design decisions are discussed and underlying technologies and tools are selected. How the schemes will be compared to traditional techniques is considered and the expected outcomes are stated.

4.1 Language Choice

All of the language candidates selected for this project meet the Implementation Language requirements. However perl is generally considered a slow language these days and as the author is not very familiar with it, it will not be used. Java is a widely used language and is generally fast on most platforms, however Java provides an Object Orientated abstraction, which is not needed in this project that is implementing relatively small algorithms. C is generally a very fast and compact language, which many free encryption tools are implemented in and may be tested against. The test would be more accurate if the programs are written in the same language and for these reasons will be the language used in this project.

4.2 High-level Design

Here we present the high-level design of the chaffing and winnowing schemes that will be implemented in the project.

4.2.1 Prototype

The prototype that will be implemented is the bit-by-bit chaffing and winnowing method. It will implement the well described “bit-by-bit CW” algorithm in [Bellare and Boldyreva, 2000]. To encrypt a message, the message will be read bit-by-bit, a serial number will be concatenated to the bit and the MAC value of this concatenation computed. The serial number, bit value and MAC digest will then be written to a file. A random digest is then computed for the complementary bit, with the same serial number. This is also then written to a file. The key that the MAC will use is a pass-phrase supplied at run-time by the user. As discussed in the literature review, if only one packet is produced for each bit rather than two, the encrypted message size is reduced by a factor of two, this is a significant reduction and so will be done in this prototype. This is done by randomly selecting whether to write a wheat packet or chaff packet for each bit in the original message. When decrypting the message if a wheat block is present for a serial number, the present bit value is used, otherwise the complementary bit is used.

4.2.2 All-Or-Nothing Transform Scheme

This scheme will be implemented as the “scattering scheme” algorithm described in [Bellare and Boldyreva, 2000]. First an AONT is applied to the original message and then this is split into blocks. [Bellare and Boldyreva, 2000] says the next step is to pick at random the positions of the wheat packets, it would however be more efficient to pick the positions of the chaff packets, as this will be a fixed number. The number of wheat packets varies with the size of the original message and when this becomes very large it will take longer to decide the wheat positions and will require more memory to store the indices. Next, for each block in the “chaffed” message, if the index is a chaff position, a chaff block and random MAC digest are computed and written to the output file, otherwise the MAC is computed for the next block of the transformed message and the block contents and MAC digest are written to the output file. To “winnow” the “chaffed” message each packet is read and the MAC computed for it, if the supplied MAC is valid then the packet is kept, otherwise it is thrown away. Once all of the chaff packets have been thrown away the AONT is inverted and the original message recovered. We will specify the security parameters: M the minimum number of blocks output by the AONT and s' the number of chaff blocks. The minimum number of blocks output by the AONT, M will be 128 and $s' = 128$, as suggested in [Bellare and Boldyreva, 2000]. This makes the complexity of guessing a random subset of packets and inverting the transform quite high, specifically it is proportional to $\binom{s+s'}{s}$.

4.2.3 Hybrid Scheme

Here we present a new chaffing and winnowing scheme, which combines symmetric techniques with public-key cryptography to benefit from the speed and key management properties. The scheme works in a similar way to the AONT method from [Bellare and Boldyreva, 2000]. An AONT is applied to the original message, then the chaff positions are chosen as a random subset of the output block indices. An AONT is applied to the chaff positions, which is then encrypted with a public-key algorithm and written to the output file. For each output block, if the index is a chaff position then a chaff block is randomly computed and written to the output file, otherwise the next block of the transformed message is written to the output file. The algorithm is shown in Algorithm 1, which is based on the “scattering scheme” in [Bellare and Boldyreva, 2000]. s is the number of blocks in the message, n is the size of the message block, s' is the number of chaff blocks and *pk_encrypt* is a public key encryption scheme.

An AONT is applied to the chaff positions to make the ciphertext different each time, otherwise an adversary could try and guess the chaff positions, encrypt them with the same public key and see if the ciphertext matches, however using the AONT prevents this happening. This method

Algorithm 1 $\mathcal{E}(M)$

```
  $M' \leftarrow AONT(M)$
  Parse $M'$ as $m_1||m_2||\ldots||m_s$ where $|m_i| = n$
  Pick $S \subseteq 1, \ldots, s + s'$ at random, where $|S| = s'$
  $S' \leftarrow AONT(S)$
  $S'' \leftarrow pk\_encrypt^{pk}(S')$
  $j \leftarrow 0$
  for $i = 1, \ldots, s + s'$ do
    if $i \in S$ then
      $Pkt[i] \leftarrow^R \{0,1\}^n$
    else
      $j \leftarrow j + 1$
      $Pkt[i] \leftarrow m_j$
    end if
  end for
  return $S'', Pkt[1], Pkt[2], \ldots, Pkt[s + s']$
```

provides much more efficiency in terms of data expansion, because there is only a small fixed overhead of the chaff position indices and the chaff packets that is added. A MAC is not added to each block, the blocks are authenticated by these chaff position indices. It is not clear if this scheme can be classified as chaffing and winnowing because there is an encryption step involved, however the original message is still in the clear and it is only the chaff position indices that are encrypted. The chaff position indices are authentication data, like the MACs are and if we were to use the "designated verifier signatures" of [Jakobsson et al., 1996] (as suggested by Rivest), where the digital signature is pubic key encrypted, then authentication data would be encrypted too.

4.3 Module Design

The components specified here will be implemented as modules as they will be used by more than one implementation, they will created as C libraries to enable easy re-use.

4.3.1 HMAC

We will implement the HMAC Message Authentication Code algorithm, because it is the method suggested in [Rivest, 1998a] and is very simple to implement. The HMAC algorithm will be implemented based on the sample code provided in [Krawczyk et al., 1997]. The hash algorithm that will be used in the HMAC is MD5 [Rivest, 1992]. SHA-1 is an alternative and is generally considered to be more secure than MD5 because it produces a 160-bit hash and MD5 only produces a 128-bit hash. However MD5 is secure for use within HMAC [Krawczyk et al., 1997] and is recommended where superior performance is required. In the implementations of this project the HMAC will be called many times and because MD5 has better performance, it will be used in this HMAC implementation. As mentioned in [Krawczyk et al., 1997] we will truncate the MAC to 80 bits, so in effect we will be using HMAC-MD5-80. This may improve security, but more importantly will minimise ciphertext expansion. We will use the "RSA Data Security, Inc. MD5 Message Digest Algorithm"[1] implementation of MD5.

4.3.2 All-Or-Nothing Transforms

The AONTs that have been chosen for implementation in this project are the Package Transform and Optimal Asymmetric Encryption Padding. The reasons for these choices are that they are both explained well and the security of them is analysed in [Bellare and Boldyreva, 2000]. The Package Transform is shown to not be as secure as OAEP, however it does provide reasonable security and will provide a good speed comparison. OAEP is proven to be secure when used as an AONT in chaffing and winnowing.

Package Transform

The Package Transform will be implemented as described in [Rivest, 1997]. It will use the RC5 block cipher as mentioned by Rivest, although any block cipher such as DES or BLOWFISH, could be used. The RC5 code will be

---

[1] Available from `http://theory.lcs.mit.edu/~rivest/md5.c`

based on the reference implementation provided in [Rivest, 1996]. The block size of RC5 is 64-bits (8 bytes) and the recommended key size is 128-bits. As the key needs to be XORed with the hash of each block, each "half" of the key with by XORed with each block. The key will be randomly generated each time the program is run. An argument will be supplied when calling the Package Transform specifying the minimum amount of blocks to be output.

Optimal Asymmetric Encryption Padding

Optimal Asymmetric Encryption Padding will be implemented based on [Bellare and Rogaway, 1994]. The MD5 hash function will be used, as it is the hash function being used throughout this project. The generator function will be constructed as shown in [Bellare and Rogaway, 1994], although using MD5 rather than SHA. The generator function is defined as $\mathrm{H}^{l}_{\sigma}(x)$, where l is the l bit prefix of:

$$\mathrm{MD5}^{128}_{\sigma}(\langle 0\rangle.x) \mid\mid \mathrm{MD5}^{128}_{\sigma}(\langle 1\rangle.x) \mid\mid \mathrm{MD5}^{128}_{\sigma}(\langle 2\rangle.x) \mid\mid \cdots$$

$\mathrm{MD5}^{128}_{\sigma}$ denotes that the initialisation constants of the MD5 algorithm have been set to σ (ABCD $= \sigma$) and we are using the 128-bit prefix of the hash generated. In [Bellare and Rogaway, 1994] the generator function is defined using the 80-bit prefix of the hash generated, however this would increase the execution time of our schemes by around 60%, as this is how much more often the hash function would get called. The generator will produce a hash the length of the message, using a random 128-bit string r, that is produced. The generator value is then XORed with the message, we denote this as y. The random value is XORed with the hash of the y, we denote this as w and $y||w$ is returned.

4.3.3 Public Key Algorithms

The Hybrid scheme requires a public-key encryption algorithm, there are a number of public-key algorithms to consider for use in this project. We need to find one that is easy to implement and is widely used. The most common public-key algorithms are:

RSA

RSA [Rivest et al., 1978] was created by Rivest, Shamir and Adleman and is considered to be the easiest to understand and implement [Schneier, 1993]. It works with numbers modulo very large primes and gets its security from the difficulty of factoring large numbers. RSA Laboratories recommends using a 1024-bit modulus for corporate use [RSA Labs].

ElGamal

ElGamal [Gamal, 1985] is a public-key cryptosystem which gets its security from the difficulty of calculating discrete logarithms in a finite field. The ciphertext produced by ElGamal is twice the length of the plaintext.

Due to the simplicity of RSA and lower data expansion, it will be used in this project.

Tools Needed To Implement Public-Key Algorithms

Generally programming languages can only work with relatively small numbers, being able to work with 1024-bit numbers is not built into most programming languages. There are however many tools that enable you to do this, as we are working with C, we will need to use a C library that can support this kind of functionality. There are many libraries available for this, including GNU Multi-Precision[2], PARI[3] and MIRACL[4] which are all C libraries supporting arbitrary precision numbers. They also include functions to find primes, perform GCD computations and compute exponents modulo other large numbers. The GNU Multi-Precision library seems to be

---

[2] Available from `http://swox.com/gmp/`

[3] Available from `http://pari.math.u-bordeaux.fr/`

[4] Available from `http://indigo.ie/~mscott/`

the most widely used package and is by far the most well documented for both installing and using. For these reasons it will be the library used in this project.

4.4 Specification Of Implementations

Now that we have made decisions for each component, we specify exactly what will be implemented:

4.4.1 Libraries

- HMAC keyed hash function, using the MD5 hash function.
- Package Transform as an AONT, using the RC5 block cipher.
- Optimal Asymmetric Encryption Padding as an AONT, using the MD5 hash function.
- RSA public-key algorithm, using the GNU Multi-Precision library.

4.4.2 Chaffing and Winnowing Schemes

- Initial Prototype, implementing bit-by-bit chaffing and winnowing.
- AONT scheme, using the Package Transform library and the HMAC library.
- AONT scheme, using the OAEP library and the HMAC library.
- Hybrid scheme, using the Package Transform library and the RSA library.
- Hybrid scheme, using the OAEP library and the RSA library.

4.5 Experimental Design

4.5.1 What Will The Implementations Be Compared To?

There are many different cryptographic applications available. OpenPGP [J. Callas and Thayer, 1998] is a standard based on PGP (Pretty Good Privacy), developed by Phil Zimmerman. It provides standard formats for encrypted messages, signatures and certificates for exchanging public keys. There are many organisations that implement the standard, some of which sell their software (PGP Corporation) and some that offer it free of charge (GNU Privacy Guard, Authora - for individuals). PKCS (Public Key Cryptography Standards) is a set of standards developed by RSA Security, although PCKS is more commercially motivated than OpenPGP, so the implementations based on the standard, are generally only commercially available.

GNU Privacy-Guard is an open source application that implements the OpenPGP standard. It implements encryption algorithms such as 3DES, AES, BLOWFISH and ElGamal. It allows you to specify which algorithm to use and for public-key encryption it uses the hybrid methods discussed in the literature review. GNU Privacy-Guard is widely used for e-mail encryption. OpenSSL is an open source package that implements the Secure Sockets Layer Internet Protocol and it uses many of the same encryption algorithms as GNU Privacy-Guard. It provides a command line tool that allows you to run these algorithms individually.

The GNU Privacy-Guard package implements the algorithms we are interested in comparing against, it is easy to install and is freely available. OpenSSL is very similar and is freely available, however its design is more directed to the SSL protocol and so we will use GNU Privacy-Guard in this project.

4.5.2 Metrics To Compare

As already discussed, the main things we are looking to compare our schemes against are execution time and data expansion. Security is also something that needs to be compared, this has already been done by cryptographic experts and will be discussed in the conclusion. In order to get an accurate execution time for each algorithm, they will be tested many times and the mean time will be taken. Each algorithm will also be tested on a variety of different file sizes.

To measure the data expansion of each algorithm, we will encrypt the same file and check the size of the file after encryption to see how much the file has been expanded. This will be done with files of different sizes for each algorithm because with the symmetric chaffing and winnowing schemes, we know that the expansion is not constant for different file sizes, with some of the schemes the efficiency should better as the file size grows. This will only need to be done once for each file size and each algorithm as the data expansion for a particular file size and algorithm will not change.

4.5.3 Expected Outcome

As we have discussed already, the execution times of the schemes we produce should be similar to traditional techniques. We expect the data expansion of the symmetric chaffing and winnowing schemes to be much higher than traditional methods, because the amount of data added grows as the file size grows. The data expansion of the hybrid chaffing and winnowing schemes we expect to be a fixed amount, although it will still be greater than that of the encryption techniques.

Chapter 5

Implementation

In this chapter we discuss how the schemes were implemented and some of the decisions made during implementation. The listings of the source code produced can be found in sections C1-C9 and the full source can be found on the CD provided with this document.

5.1 Development Tools

The code for this project will be developed in Xcode, which is Apple's IDE. Although it will not be used as an IDE, it will simply be used as a tool to edit the code, providing the useful features of syntax colouring and line numbering. So that the code can be easily compiled across platforms, a UNIX shell script will be produced that builds the libraries and applications. The GCC compiler will be used across platforms because it is available on each platform and is very widely used. On the Windows machine the MinGW[1] (Minimalist GNU for Windows) compiler will be used, which is a pre-built GCC compiler for Windows. Also on the Windows machine MSYS[2] (Minimal SYStem) will be used, which provides a Bourne shell, allowing the same build script to be used as the one on Mac and Solaris. The `ar` and `ranlib` tools will be used to create libraries. GDB (GNU Project Debugger) will be

---

[1] Available from `http://www.mingw.org/`

[2] Available from `http://www.mingw.org/`

used to help debug the code if necessary. As the size of the algorithms that will be produced is quite small, no source code control system will be used in the project.

5.2 Prototype

5.2.1 Implementing The Prototype

Implementing the prototype involved firstly building the HMAC. This was implemented based on the sample code in [Krawczyk et al., 1997] except that the functions `bzero()` and `bcopy()` are now depreciated, so `memset()` and `memcopy()` respectively, were used instead. Problems involved in implementing the bit-by-bit scheme included:

- Functions to get and set bits of a `char` had to be created.
- Generating a fake MAC digest, that was different each time. Using `rand()`, each time the program is run the same sequence of numbers is produced, this would allow an adversary to learn about the validity of packets because the chaff packets would always be the same. To prevent this `srand(time())` was used to initialise the random state with the current time.
- Output was written to a file in the smallest possible format, which is a character.
- Outputting either the real bit and real MAC or the complementary bit and fake MAC, to half the size of the "encrypted" message.

5.2.2 What Was Learnt From The Prototype

Implementing the prototype allowed the author to gain a greater understanding of the chaffing and winnowing concept. It provided experience of using bit-wise operators, file input/output features and brought awareness of the problems generating random numbers. It was found after doing more research, that there is a C standard library function `arc4random()`, which uses the ARC4 cipher key stream generator and is specifically designed for cryptographic applications. This is the random number generator that will be used throughout the rest of the project. It was also found through a simple test that the `fread()` and `fwrite()` C standard library functions were much faster than using a loop with `getc()` or `putc()` in it.

5.3 Libraries

5.3.1 Chaffing and Winnowing Library

After implementing the prototype it seemed that there would be some functions that would need to be used by more than one implementation, so the best thing to do would be to create a library that carried out common tasks related to chaffing and winnowing. The library includes functions to check which byte-order the machine is using, creating a chaff packet and randomly generating the chaff packet positions. The Solaris and Windows C implementations we were using did not support the `arc4random()` function but did define the symbolic constants `sun` and `WIN32`, so a check was done in this library and if these constants were defined by the C implementation then the function `arc4random()` was defined as `random()` and `rand()` on Solaris and Windows respectively. The library includes a macro, which swaps the byte order of a 32-bit integer for any byte swapping that needs to be done.

5.3.2 HMAC

Implementing the HMAC library was trivial as the code produced in the prototype was used. One change was made however, because the HMAC is called many times with the same pass-phrase throughout execution there is no need to initialise the inner and outer pad each time the `hmac()` function is called. So a `hmac_init()` function was created, which initialised the inner and outer pads and the functionality was taken out of the `hmac()` function.

This should improve the speed slightly.

5.3.3 Package Transform

The Package Transform was implemented with a block size of 8 bytes because this is the block size of RC5 on a 32 bit machine and the RC5 block needs to be XORed with the input block. The RC5 implementation was the RSA Data Security Inc. "Reference implementation of RC5-32/12/16", meaning that the word size is 32 bit, the number of rounds is 12 and the key size is 16 bytes. These are parameters that can change but 12 rounds is recommended as the minimum and the key can become "weak" if they are any shorter than this [Rivest, 1996]. The RC5 implementation was modified because a time consuming part of RC5 is the initialisation, in the Package Transform two RC5 keys need to be used - a publicly known one and a secret one. The reference implementation used a global array for the key table, meaning each time you use a different key, the key table needed to be initialised again. To improve this, the global array for the key table was removed and the `RC5_SETUP()` function was given an extra argument, a pointer to the key table and the encrypt/decrypt functions were also given this argument. As a result you can initialise two key tables and pass a pointer to the encrypt/decrypt functions, depending on which key you want to encrypt with. This change vastly improved the speed of the package transform.

As the key size is twice the size of the blocks the hash of each block was XORed with each "half" of the key. The RC5 key is randomly generated each time the Package Transform is run. The final block consisting of the key XORed with the hash of every block, needed to be put at the end of the "outer" block (the chaffing and winnowing block), because the transformed message will be padded and we needed to make sure that the final block will always be at the very end of the message.

The main functions of the Package Transform are `transform()` and `inverse_transform()`. The transform and inverse transform processes are very similar, the only difference is that when inverting the transform the RC5 key needs to be recovered from the message and not generated. So to take advantage of this the `transform()` function was modified to accept a key and the `inverse_transform()` function simply recovers the key and passes it to the `transform()` function.

5.3.4 Optimal Asymmetric Encryption Padding

Implementation of OAEP was fairly simple. The MD5 hash function was used to implement it as described in [Bellare and Rogaway, 1994]. Even though OAEP does not have "blocks" as such, the message was read in as 128-byte blocks. This is the same size blocks that the chaffing and winnowing will use. The only difference to [Bellare and Rogaway, 1994] was that the XOR of the random string and the hash of the transformed message was put at the beginning of the output rather than the end, this was to ensure that it could always be found. The output needed to be padded so that no padding has to be added by the chaffing and winnowing process, this is because a hash of the message is used to recover the random string. If the chaffing and winnowing process hands back a message with padding, a different hash would be produced and the random string would not be recovered and therefore the message would not be recovered. An alternative to this would be to put the original message length in the output, so that you know how much of the transformed message to look at.

The OAEP library was designed to be very similar to the Package Transform, in the sense that they could be substituted for each other very easily. The OAEP has the same functions as the Package Transform - `transform()` and `inverse_transform()`, with the only difference being that the `transform()` function accepts one less argument than the Package Transform function.

5.3.5 RSA

The implementation of RSA was straightforward using the GNU Multi-Precision library. The most difficult part was transforming the message into an integer and reversing the process. This was achieved by printing each input character into a string in hexadecimal notation. Once the message had been encrypted/decrypted the result was written to a string in hexadecimal format and each two hexadecimal numbers were converted to a character for output. The message was split into blocks, each block had to be represented as an integer less than the modulus n. If the encrypted or decrypted block was less than the block size, it was padded with zeros at the most significant end of the integer, this ensured that the integer would be correctly recovered when it was read in.

The modulus size used in the implementation was 1024-bits, this is a recommended size for RSA to provide good security currently, although often a 2048-bit key is used. 1024-bit is the modulus size we will be using with GNU Privacy Guard and this is the main concern, to be able to provide a fair comparison. The public exponent we are using is 65537 ($2^{16} + 1$), it is a commonly used public exponent, because it is a Fermat prime - it provides enough security against a low-exponent attack and has the property that the modular exponent can be calculated very quickly.

The RSA implementation was not produced according to a standard such as PCKS or OpenPGP but simply so that public-key encryption could be provided in our scheme. The implementation provides the generation of keys, which it writes to key files in the same directory that it is run in. In practice we would need to make sure the private key is more secure but that is not something that is in the scope of our project.

5.4 Chaffing and Winnowing Schemes

Once the libraries were built, the implementation of the schemes was made easier, because they rely on the underlying functionality of the libraries.

5.4.1 Symmetric Package Transform Scheme

This scheme was simple to implement, the original file is transformed using the package transform and put into a temporary file, which is then split into blocks. The indices of the chaff packets are chosen using the `set_chaff_positions()` function from the chaffing and winnowing library. Then for each output packet if the index belongs to a chaff packet, a chaff packet is produced using the `generate_chaff_packet` function from the chaffing and winnowing library, otherwise the MAC of the next valid packet is computed and both are written to the output file. The winnowing process simply reads in each block and computes its MAC, if it matches the supplied MAC it is written to a temporary file otherwise it is discarded. Once the whole input file has been processed, the inverse AONT is applied to the temporary file, recovering the original message.

5.4.2 Symmetric OAEP Scheme

This scheme was implemented very much the same as the Symmetric Package Transform scheme, except the OAEP header file was included and the library linked to, instead of the PackageTransform header and library. Also one of the `transform()` arguments was removed as the OAEP library accepts one less.

5.4.3 Hybrid Package Transform Scheme

This scheme was more difficult to implement than the symmetric ones. The original file is transformed using the package transform and written to a temporary file, the indices of the chaff packets are then produced using the chaffing and winnowing library. A check is then performed to see if the machine running the program is big-endian or little-endian. If the machine is little-endian the chaff position indices were copied and byte-swapped. This ensures that everything is stored in big-endian byte order and if a little-endian machine runs the program it will need to swap the bytes of certain data. The reason everything is stored in big-endian order is that two big-endian machines were being used and only one little-endian machine. An alternative to this would have been to put a bit in front of the indices specifying which endian-order they are stored in, this would require the same amount of swapping to be done across machines but less on the little-endian machine however would add extra data to the output. The next step is to write the chaff indices to a temporary file (in big-endian order) and transform the file using the package transform. The result of this is then encrypted using RSA and written to the output file. The rest of the process is the same as the symmetric one, except no MAC's are calculated or written to the output file.

The winnowing process calculates the size of the encrypted chaff packet indices by looking at the size of the modulus being used and the size of the message that would be produced after transformation. Once it has this information, it can read in the encrypted chaff packet indices, decrypt them and invert the transform. The indices are read in as 32-bit integers and if the machine is little-endian the bytes are swapped. The packets with valid indices are then written to a temporary file and the packets with invalid indices are discarded. The inverse AONT is then applied to the temporary file, recovering the original file.

5.4.4 Hybrid OAEP Scheme

This was implemented in the same way as the Hybrid Package Transform scheme, except that the OAEP library was used rather than the Package Transform library.

5.5 Implementation Notes

5.5.1 All-Or-Nothing Transforms

At first it was thought that after the AONT implementations had been created, there would need to be some byte swapping done, so that they could run cross-platform. However, after initial tests it was seen that this was not the case. The reason for this is that even though bytes are stored in a different order on different machines, C still treats them the same when using the left and right bit-shift operators (<< and >>). So for example, the number 0x12345678 would be stored in the byte order 12 34 56 78 on a big-endian machine and stored as the byte order 78 56 34 12 on a little-endian machine, but performing a ">> 24" (right shift by 24 bits) on the number on both machines gives the result 0x12. Also for the package transform, as the RC5 block cipher produces the same results on big and little endian machines, no byte swapping needed to be done there either.

Chapter 6

Testing

In this chapter we describe how the implementations we produced were tested. The tests performed were to make sure that the libraries and schemes were working correctly. Testing was done in parallel with implementation. This is due to the design, the underlying functionality needed to be tested before it could be used in the Chaffing and Winnowing schemes.

6.1 Platforms and Compilers

The platforms being used to test the programs and compilers used to build the programs on each platform are:

- Platform: Mac OS X 10.4.5 running on Mac Powerbook, PowerPC G4, 1.5Ghz Processor, 512Mb RAM. Compiler: GCC 4.0.0 (Apple build).
- Platform: Windows XP Home Service Pack 2 running on Dell Inspiron 5100, Intel Pentium 4, 2.4Ghz Processor, 256Mb RAM. Compiler: GCC 3.2.3 (MinGW build).
- Platform: Solaris 8 running on SUN Ultra E450, UltraSPARC-II 4 x 296MHz Processors, 2176 Mb RAM. Compiler: GCC 2.95.3.

6.2 Library Testing

These tests were produced using a `main` function in each library, which could be easily defined and undefined using symbolic constants. The `main` function was defined for these tests and undefined for use in the chaffing and winnowing schemes.

6.2.1 HMAC

The HMAC library was tested with the test vectors given in [Krawczyk et al., 1997], the output digests were checked against the digests given and the tests were repeated on each platform. All of the tests passed, the digests were the same for each test vector on each platform and matched the given digests.

6.2.2 Chaffing and Winnowing Library Testing

The CW library was tested to make sure that each function within it worked correctly. It is important that this library works correctly because a lot of the security of the chaffing and winnowing schemes rely on how well these functions work. The tests were run 10 times on each machine. There is a check to make sure that the `WIN32` symbolic constant is defined on the Windows platform and the `sun` symbolic constant is defined on the Solaris platform. This test succeeded. The `is_big_endian()` function should always return 0 if the machine is big-endian and return 1 otherwise, so the byte-order of the machine the code is run on should be correctly printed out. This test succeeded with "Big-endian" being printed out on the Mac and Solaris platforms and "Little-endian" being printed out on the Windows platform.

A test was performed to check that a "random" chaff packet is produced each time the `generate_chaff_packet()` function is called. This test passed, with the chaff packet being completely different each time it was called. It is important to note that on the Solaris and Windows platforms, because they are not using the `arc4random()` function, when they are not initialised with `srand()` or `srandom()` they produce exactly the same chaff packet each time, which would make it very easy for an adversary to distinguish chaff from grain. It is also important to note that the seed should only be initialised with the current time once at the start of the program, because the programs execute very quickly, initialising the seed again with the same time will produce the same sequence of numbers. Ideally `arc4random()` would be used on every platform if it was available to us, because it randomly initialises itself each time it is called and is specifically designed for cryptographic applications.

A test was performed on the `set_chaff_positions()` function, which picks a random subset of the output packet indices for chaff packets to be placed in. This test should ensure that the chaff packet positions are different each time the program is run, the indices should be in ascending order and there must be no duplicates. This test originally failed, sometimes there was a duplicate because the first position in the array was not being checked for duplicates, this was easily rectified. After this change, the library was re-tested and passed. The final test checked that the `SWAP` macro worked correctly, it should swap the bytes of a 32-bit integer and when applied again recover the original number. This test was passed.

6.2.3 Package Transform Library Testing

The Package Transform library was tested by transforming a file and inverting the transform. The transform and inverse transform were done on the same platform, as the library will get tested cross-platform when the chaffing and winnowing schemes are tested. A file was transformed and inverse transformed 10 times on each platform, it was checked to see that once the file had been transformed back it was the same as the original and that the transformed file was different every time, as a random key is chosen each time it is run. The test was performed specifying that there must be a minimum of 128 blocks output. To see if the transformed file was different each time the UNIX command `diff` was used. These tests passed and the library will be tested further when the chaffing and winnowing schemes are tested.

6.2.4 OAEP Library Testing

The OAEP library was tested in the same way as the Package Transform library, as they essentially do the same things. A file was transformed and inverse transformed 10 times on each platform, each time making sure that the file was transformed back correctly and that the transformed file was different each time. These tests were passed.

6.2.5 RSA Library Testing

The RSA library was tested by encrypting and decrypting a file, using a different key pair each time. With a particular key, a file only needs to be encrypted and decrypted once because, this implementation of RSA produces the same ciphertext each time for a given plaintext and so will decrypt exactly the same each time, whereas the AONTs transform the plaintext into a different ciphertext each time. The RSA library was tested 20 times on each platform, using a new key pair each time, checking that the decrypted file was exactly the same as the original file using the `diff` UNIX command.

This test initially failed for some keys, a bug was found where an incorrect calculation was being done. The GNU Multi-Precision library transforms a number into hexadecimal but without leading zeros, this was being taken into account, however the calculation was not being performed properly, this was changed and re-tested. After the change, the library passed on all platforms.

6.3 Chaffing and Winnowing Scheme Testing

Here we test the chaffing and winnowing schemes produced in this project.

6.3.1 Correctness Testing

There is no way to tell if a chaffing and winnowing scheme is "correct", because there are no official implementations of it. Here we test that the schemes work in the way they should - add chaff to a message and winnow it to recover the original message. Each chaffing and winnowing scheme was tested by "chaffing" a file and "winnowing" it to reveal the original file. As the chaffing and winnowing implementations produced will always generate a different ciphertext for the same plaintext, it is not enough chaff and winnow a file once, there could be a situation in which a particular ciphertext will not "winnow" back to the original file. So the tests were performed 200 times for each implementation, on each platform. This is a lot of tests to perform manually, so a UNIX shell script was written to perform the tests automatically, the script is listed in Appendix B.1. The test script used the `diff` UNIX command to compare the "winnowed" file to the original file, although because padding is added to the message the `diff` command will say that the files are different, even if the "winnowed" file has been recovered correctly. So instead we chaffed and winnowed a file, confirmed it had been recovered correctly and used this to compare against the other "winnowed" files. The test script was run on each platform, to make sure there were no platform specific issues. The file that was "chaffed" and "winnowed" was a simple text file. The hybrid methods generated a new key pair every 50 tests, to ensure the schemes worked correctly with different keys. Table B.1 shows the results of these tests.

Table 6.1: Correctness Test Results

	Platform		
Implementation	**Mac**	**Windows**	**Solaris**
AONT Package Transform	Pass	Pass	Pass
AONT OAEP	Pass	Pass	Pass
Hybrid Package Transform	Pass	Pass	Pass
Hybrid OAEP	Pass	Pass	Pass

After we had made sure the schemes worked correctly after being run multiple times, the chaffing and winnowing schemes were also tested using different types of files on each platform, to make sure they recovered the original file correctly after winnowing. Different types of file were used because many different types of file are encrypted and decrypted using traditional techniques and we need to make sure that our chaffing and winnowing can provide this. The chaffing and winnowing implementations also add some padding if the input file is not an exact multiple of the block size. So we "chaffed" and "winnowed" a Microsoft Powerpoint Presentation, a bit-map image and a PDF document, then checked that the winnowed file opened in their respective applications correctly. All the implementations passed these tests.

6.3.2 Platform Testing

The chaffing and winnowing implementations were tested to make sure that when a file has been "chaffed" on one platform, it can be "winnowed" on either of the other platforms, as this is what the implementations were required to do. This test used a text file, which was "chaffed" by a scheme on one platform and was then "winnowed" by the same scheme on the remaining platforms. This was done for every scheme, on every platform. The results are shown in table 6.2, each scheme had the same result.

Table 6.2: Platform Test Results

	Winnowed		
Chaffed	**Mac**	**Windows**	**Solaris**
Mac	Pass	Pass	Pass
Windows	Pass	Pass	Pass
Solaris	Pass	Pass	Pass

We have tested the schemes to make sure they work correctly and as we specified they should in Chapter 4. In the next Chapter we will go onto see how the schemes compare to traditional encryption methods.

Chapter 7

Experimental Results and Analysis

In this chapter, we show how the experiments were carried out on the schemes implemented in the project and on traditional encryption techniques. We then use various methods to analyse our findings and comment on the results. The full statistical results can be found in Appendix A and on the provided CD.

7.1 Experiments Performed

Two types of experiment were performed on each scheme, execution time analysis and ciphertext expansion analysis. The purpose of the experiments was not only to compare the different schemes against each other but to compare them against traditional encryption techniques. As discussed earlier, GNU Privacy Guard (GPG) is the implementation of traditional schemes that we will use to compare our schemes with and the schemes implemented in GPG underwent the same tests as our schemes. The execution time analysis was to see how the speed of the schemes compared to traditional techniques and the ciphertext expansion was to see how much the message expanded after “encryption”.

7.1.1 Execution Time Experiments

In this project, 4 chaffing and winnowing schemes were implemented and they will be compared to 4 schemes implemented by GPG. The schemes being tested using GPG are symmetric Triple-DES, symmetric AES, hybrid Triple-DES and hybrid AES. This is because we have produced 2 symmetric schemes and 2 hybrid schemes, we will compare the symmetric chaffing and winnowing schemes to the symmetric GPG schemes and the hybrid chaffing and winnowing schemes to the hybrid GPG schemes.

The schemes were timed using the UNIX command `time` which records execution times to one thousandth of a second. We recorded 75 execution times of each scheme, on 8 different file sizes. The times were recorded using a shell script, which redirected the "user" process execution time into a file. The script is listed in Appendix B.2. The user process time given by the `time` command is how long it has taken to execute the user process. The file sizes that the schemes were tested on were approximately 20Kb, 50Kb, 100Kb, 250Kb, 500Kb, 1Mb, 5Mb and 20Mb, the exact sizes can be seen in the results. The experiments were all carried out on the Mac machine described in the last chapter, which had been rebooted and had minimal applications running.

The Hybrid schemes we implemented were using a 1024-bit RSA key and the GNU Privacy Guard was set up to use a 1024-bit ElGamal key. GPG defaults to using a high level of compression, this may seem like a disadvantage in speed testing but once a file has been compressed, there could be a lot less data to encrypt, making the whole process faster. This would also distort the results of the ciphertext expansion experiments, as the original file was compressed before encryption. We also wanted GPG to behave the same as our schemes - to just "encrypt" the file - so the compression was turned off in GPG.

We omitted the first 2 results of each execution time experiment when calculating the means, to remove outliers. The first and second times were generally much higher than the remaining times and it is assumed that this is because the application is being read from disk into RAM and after the second execution, the file is stored and read from RAM, which is a lot faster than reading from disk. Leaving these results in would distort the results of the experiments. The chaffing and winnowing implementations were compiled using the "-O3" flag for maximum optimisation and the "mcpu=powerpc" flag to produce optimal code for the architecture being tested on.

7.1.2 Ciphertext Expansion Experiments

The ciphertext expansion experiments were carried out in a similar way to the execution time experiments, however each experiment only needed to be run once because the expansion does not vary each time the program is run on a certain file. Each implementation was run once on each of the files used in the execution time experiments and the size of the resulting file was recorded.

7.2 Experimental Results

7.2.1 Execution Time Results

Table 7.1 shows the mean execution times of the symmetric schemes in seconds for encryption, on each file size.

Table 7.1: Symmetric Execution Times

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	0.0041	0.0050	0.0110	0.0080
51808	0.0080	0.0099	0.0140	0.0100
102677	0.0130	0.0170	0.0189	0.0130
274106	0.0300	0.0390	0.0350	0.0235
475136	0.0502	0.0658	0.0536	0.0350
1080054	0.1117	0.1469	0.1100	0.0700
5030281	0.5132	0.6743	0.4812	0.3016
20948069	2.1208	2.7995	1.9765	1.2319

Table 7.2 shows the mean execution times of the hybrid schemes in seconds for encryption, on each file size.

Table 7.2: Hybrid Execution Times

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	0.0060	0.0070	0.0228	0.0218
51808	0.0090	0.0100	0.0260	0.0237
102677	0.0120	0.0160	0.0311	0.0267
274106	0.0240	0.0320	0.0490	0.0369
475136	0.0370	0.0520	0.0699	0.0485
1080054	0.0781	0.1120	0.1323	0.0840
5030281	0.3461	0.5058	0.5408	0.3158
20948069	1.4271	2.1747	2.1877	1.2451

7.2.2 Ciphertext Expansion Results

Table 7.3 shows the ciphertext expansion results for the symmetric schemes, the table shows how much extra data (in bytes) has been added to each file.

Table 7.3: Symmetric Ciphertext Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	19360	19360	49	83
51808	21746	21746	49	83
102677	25801	25801	54	86
274106	39154	39154	49	81
475136	54922	54922	56	88
1080054	102192	102192	64	96
5030281	410783	410783	62	94
20948069	1654261	1654261	48	80

The amount of data added for the chaffing and winnowing schemes increases with the file size, this is as expected and was discussed in the literature review. The GPG schemes add a fixed amount of data, probably information such as how the file was encrypted and the size of the original file. The GPG results vary very slightly for each file size, probably because the files are different multiples of the block sizes being used and so different amounts of padding are added.

Table 7.4 shows the ciphertext expansion results for the hybrid schemes, the table shows how much extra data (in bytes) has been added to each file. The expansion for all of the schemes in Table 7.4 is a fixed amount. In the

Table 7.4: Hybrid Ciphertext Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	17252	17252	341	333
51808	17188	17188	341	333
102677	17263	17263	344	336
274106	17226	17226	331	339
475136	17284	17284	338	346
1080054	17294	17294	346	354
5030281	17275	17275	344	352
20948069	17183	17183	330	338

table it is clear that for each file size the amount varies very slightly, this is due to the amount of padding added. The padding changes depending on how close the file size is to a multiple of the block size.

7.3 Analysis Of Results

7.3.1 Execution Times

Here we use statistical methods to analyse the results. To analyse the execution times of the schemes we performed Analysis Of Variance (ANOVA) on the means of the symmetric schemes and of the hybrid schemes. This tells us if the results are significantly different or not. The ANOVA was not performed on the ciphertext expansion because it is clear to see the results in these tests and they are fixed values, they are not mean results.

Hypothesis

Here we define the hypothesis for the experiments, it is assumed that the data follows a normal distribution. We define a null hypothesis H_0 and an alternative hypothesis H_1 as follows:

H_0 : There exists $\mu_i \neq \mu_j$ $(i \neq j)$

$H_1 : \mu_1 = \mu_2 = \ldots = \mu_n$

Where μ_i is the mean execution time of scheme i. Here the null hypothesis says that the means of at least two schemes are different and the alternative hypothesis says that the means are the same.

Analysis Of Variance

The analysis of variance was performed on the means of the execution times, using tools provided in Microsoft Excel. The analysis performed is called a two-factor full factorial design without replications. This means that there are two factors we are varying - the scheme and the size of file being encrypted, with each factor having multiple levels - 4 different schemes and 8 different file sizes. The ANOVAs were calculated to a 5% significance level. The ANOVA for the symmetric schemes is shown in Table 7.5.

Table 7.5: ANOVA For Symmetric Schemes

Component	Sum Of Squares	Degrees Of Freedom	Mean Square	F-Computed	F-Table
File Size	13.7712	7	1.9673	39.5047	2.4876
Scheme	0.2681	3	0.0894	1.7947	3.0725
Errors	1.0458	21	0.0498		

The F-Computed value is the value computed for our data and the F-Table is the value from the F-Table of $F_{[0.95,7,21]}$ for the file size and $F_{[0.95,3,21]}$ for the scheme. If the computed F value is less than the value from the table then there is not enough evidence to suggest the results are different. Table 7.5 shows that the computed F value for the scheme (1.7947) is less than the table value (3.0725), so there is enough evidence at this level to reject the null hypothesis and accept the alternative hypothesis.

Table 7.6: ANOVA For Hybrid Schemes

Component	Sum Of Squares	Degrees Of Freedom	Mean Square	F-Computed	F-Table
File Size	10.2150	7	1.4593	49.8461	2.4876
Scheme	0.1576	3	0.0525	1.7944	3.0725
Errors	0.6148	21	0.0293		

For the Hybrid schemes we get the same result, shown in Table 7.6. The computed F value (1.7944) is less than the F value from the table (3.0725), so we have enough evidence to reject the null hypothesis.

Graphical Analysis

We have determined that there is not enough evidence to show that the execution times are significantly different but now we will analyse the results in graphical form to spot any possible trends in the results. The execution time graphs are specifically **scaled** to make the results more distinguishable from each other, when shown on an equally scaled graph the differences would not be noticeable. The file sizes on the execution time graphs are shown in mega-bytes.

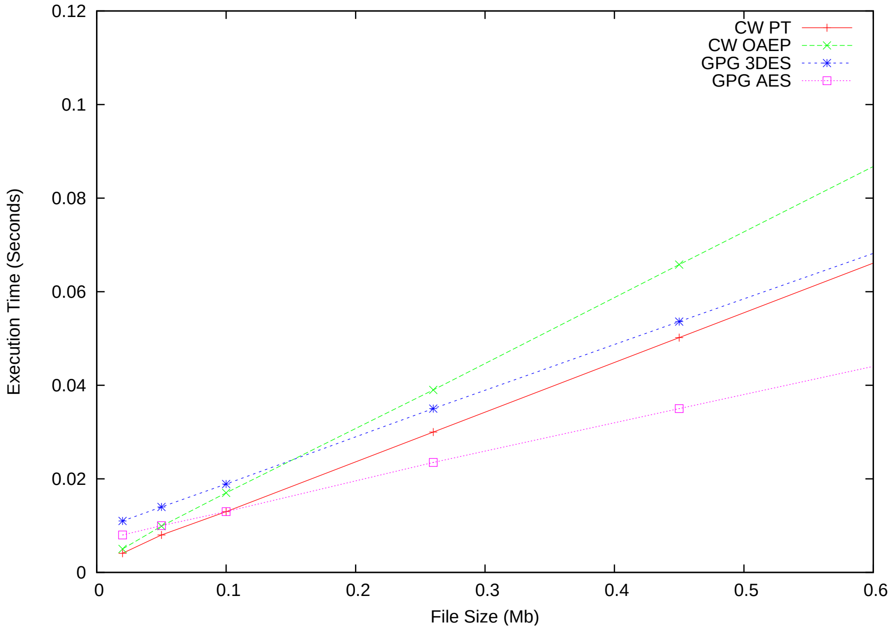

Figure 7.1: Symmetric Execution Times (Small Files)

Figures 7.1 and 7.2 show the execution times of the symmetric schemes. Figure 7.1 shows the smaller files in greater detail and Figure 7.2 shows the larger files as well. The lines between the points are an estimation of how the schemes would perform on the intermediate file sizes. Figure 7.1 indicates that the symmetric chaffing and winnowing schemes are marginally quicker for small files, than the GPG schemes. As the file size gets larger, the times cross over and the GPG schemes seem to become faster. Figure 7.2 indicates that as the file sizes get even larger, AES will be the fastest and the OAEP chaffing and winnowing scheme will be the slowest. The Package Transform chaffing and winnowing scheme and the Triple-DES seem to be very similar and the graph indicates that they will continue to perform at similar speeds as the file size increases.

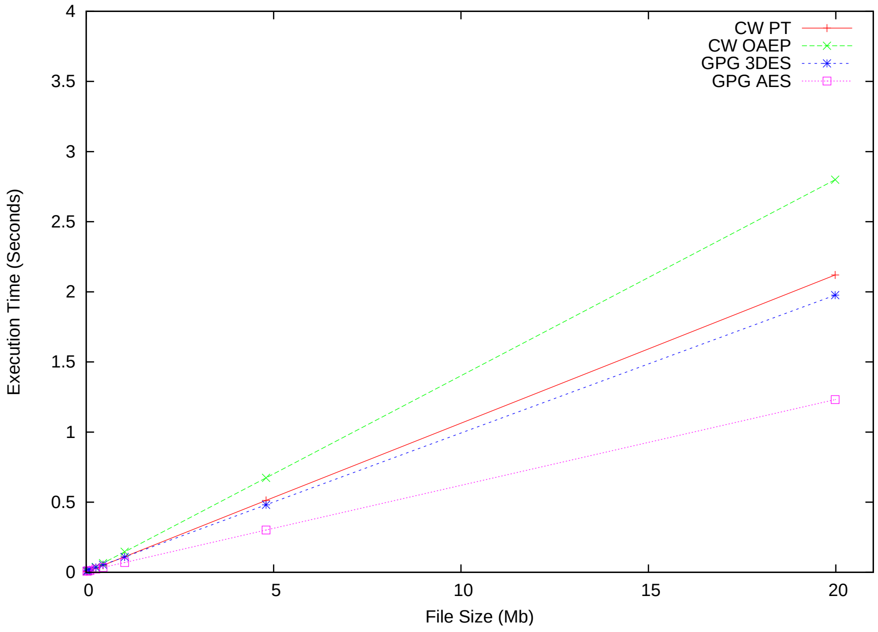

Figure 7.2: Symmetric Execution Times (Large Files)

Figures 7.3 and 7.4 show the execution times of the hybrid schemes. Figure 7.3 shows the smaller files in greater detail and Figure 7.4 shows the larger files as well. Again, the lines between the points are an estimation of how the schemes would perform on intermediate file sizes. Figure 7.3 shows that for files up to around 400Kb, the hybrid chaffing and winnowing schemes are faster than the hybrid GPG schemes. After this point the results become very similar. Figure 7.4 shows that the Triple-DES and OAEP chaffing and winnowing schemes are almost identical for the file sizes we tested. It also shows that the chaffing and winnowing Package Transform scheme is very similar to AES and that all of these schemes generally have very similar execution speeds.

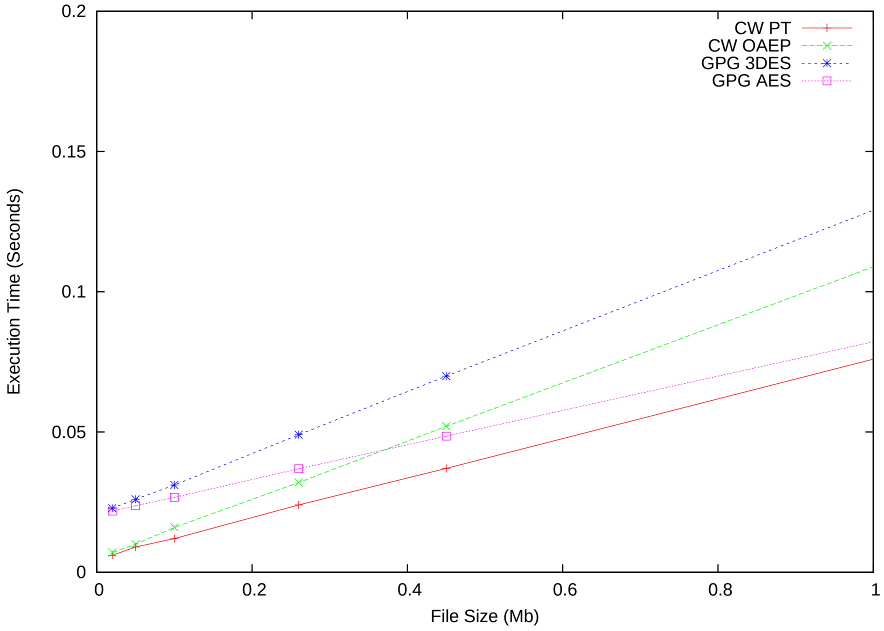

Figure 7.3: Hybrid Execution Times (Small Files)

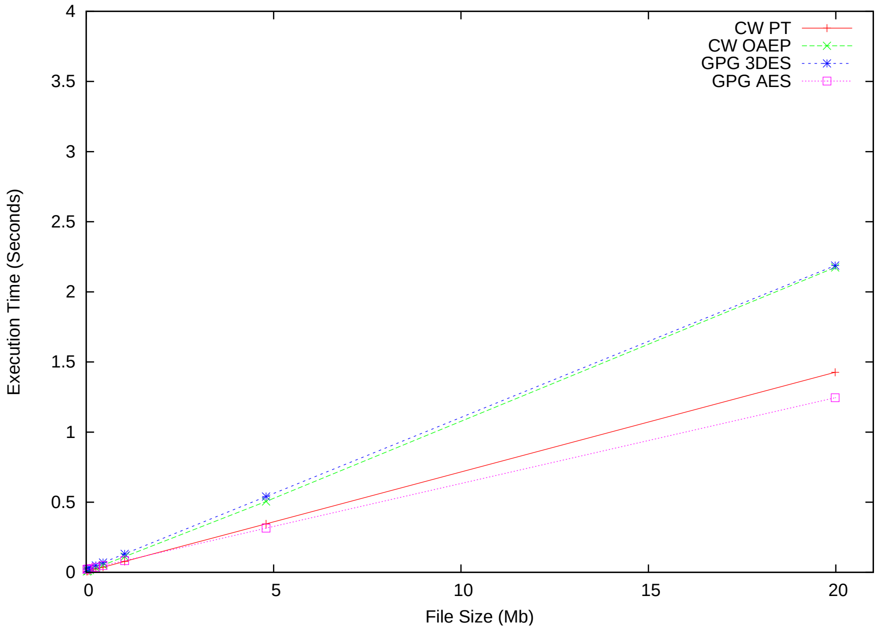

Figure 7.4: Hybrid Execution Times (Large Files)

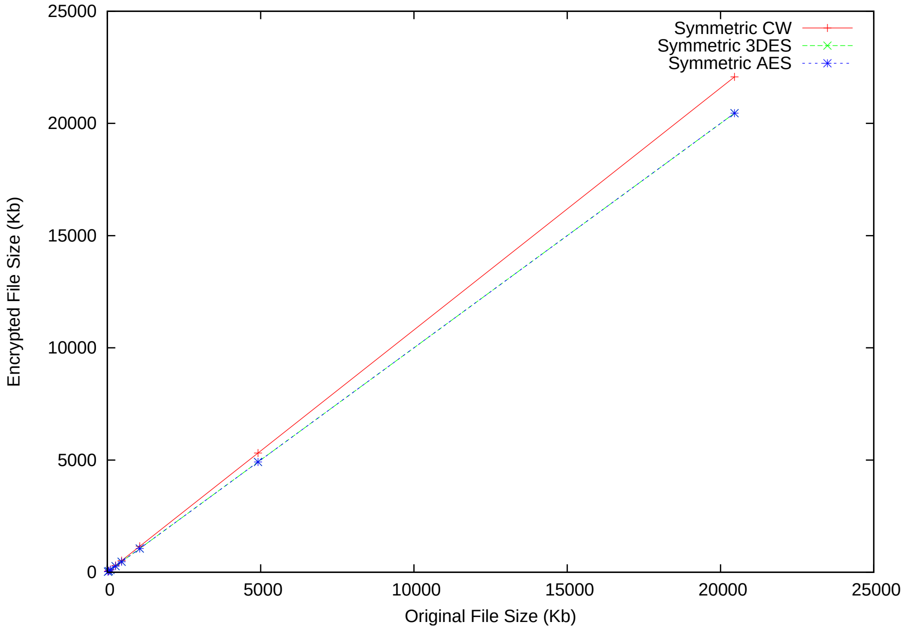

Figure 7.5: Symmetric Data Expansion

Figure 7.5 shows the encrypted file size against the original file size for the symmetric schemes, the CW schemes are shown as one because their results were exactly the same. Only one of the GPG schemes is visible even though they are both plotted, this is because they are very similar. The lines between points are an estimate of how much data would be added for intermediate file sizes. We can see from this graph that as the size of the file being encrypted increases, more data is added by the chaffing and winnowing schemes than the GPG schemes and this will continue to happen as the file size grows.

Figures 7.6 and 7.7 show the encrypted file size against the original file size for the hybrid schemes, again the CW schemes are shown as one because their results were exactly the same. Figure 7.6 shows the graph at a higher scale, to be able to see the difference between the schemes, which even at high scale appears to be very small. Both the hybrid GPG schemes are plotted but only one can be seen because the results were very similar. Figure 7.7 shows the results at normal scale, here a difference cannot be seen in the results.

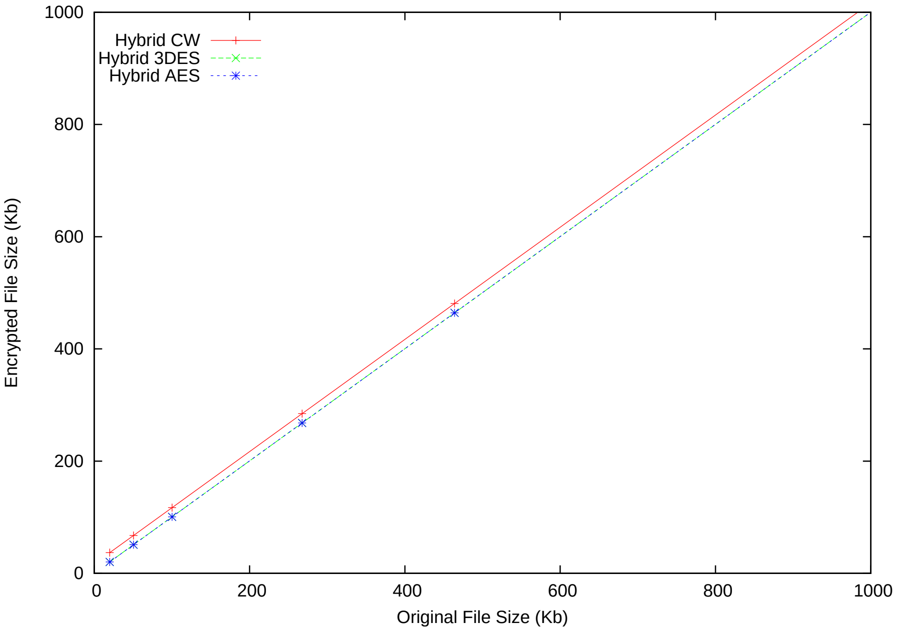

Figure 7.6: Hybrid Data Expansion (Large Scale)

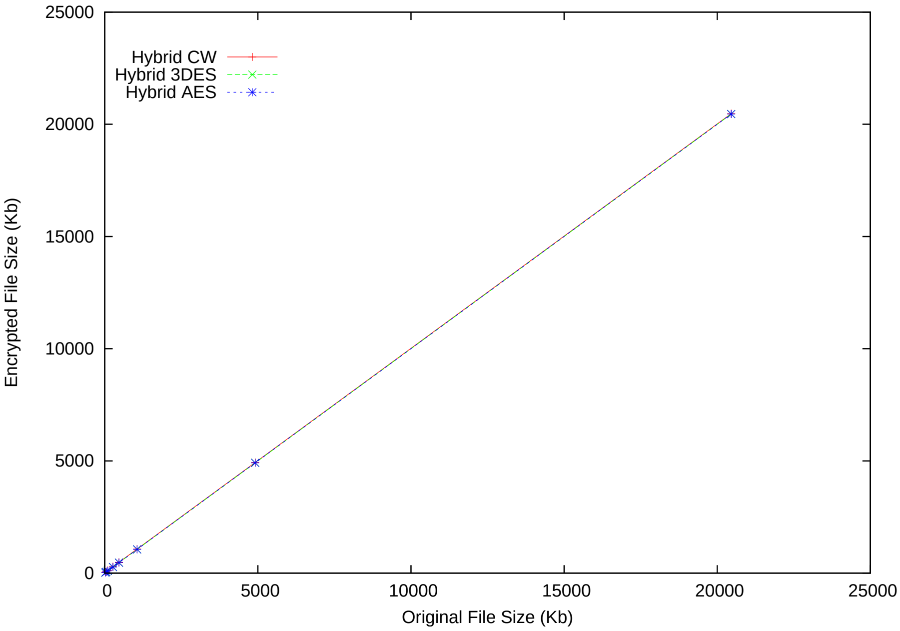

Figure 7.7: Hybird Data Expansion (Normal Scale)

7.4 Experimental Conclusions

We have analysed the data collected from the experiments performed. As we thought in Section 4.5.3, the execution times of the symmetric and hybrid chaffing and winnowing schemes are comparable to traditional encryption methods, from the Analysis Of Variance. The graphical analysis indicates that some further work needs to be done on our schemes to be as fast as AES for large files. Although for small files, the chaffing and winnowing schemes seem to be faster. Files of around this size (> 400Kb) are a typical size of files that would be "encrypted" during normal use of an encryption application. The graphical analysis also indicates that the CW OAEP schemes are slightly slower than the CW Package Transform schemes, this must be down to the speed of the All-Or-Nothing Transformations because the chaffing and winnowing algorithms using OAEP and the Package Transform are almost identical. This could be down to the speed of the MD5 implementation, which is used extensively by OAEP and is something which could be improved upon.

The data expansion results show that the symmetric chaffing and winnowing schemes increase the expansion as the input file gets larger, whereas the symmetric GPG schemes add a negligible amount of data. This was expected and was discussed earlier. The amount of data added by the hybrid chaffing and winnowing schemes is higher than the hybrid GPG schemes but it is still only a fixed amount (around 17Kb compared to around 0.3Kb) and we can see from the graphs that this difference is very small. Having a fixed amount is a substantial increase on having a variable amount and as the input file gets larger, the scheme becomes much more efficient.

Chapter 8

Conclusion

8.1 Project Conclusion

During this project we implemented two different types of chaffing and winnowing scheme. One type of scheme had been proposed previously and one was created during the project. We have shown that these schemes are a real alternative to using traditional encryption techniques, although some are more suitable than others. We used the Package Transform all-or-nothing transformation, even though there were some security concerns, which were discussed in the literature review, and Optimal Asymmetric Encryption Padding as an AONT, which is provably secure. The Package Transform appeared to be the faster of the two AONTs in the experiments. However, the results were shown not to be significantly different and for the fractional time penalty it would be more suitable to use the OAEP schemes for the security benefits.

We compared the implementations that were produced in the project to traditional encryption methods, with good results. From the statistical analysis done in Chapter 7, we showed that our schemes had similar execution speeds to the widely used encryption algorithms Triple-DES and AES, implemented in GNU-Privacy Guard. From the analysis of cipher-text expansion, we showed that our hybrid schemes produce a fixed overhead but that the symmetric schemes create more overhead as the file size increases. However although the symmetric schemes do become more efficient as the input file size grows, the hybrid schemes become much more efficient. The hybrid method therefore would be very practical for real use, as it runs at a very similar speed to traditional techniques and has a fixed data-expansion overhead. The symmetric schemes would not be so practical in situations where large files are being used and any significant increase in file size is not acceptable.

The new hybrid chaffing and winnowing scheme we have introduced is important for the concept of chaffing and winnowing because most encryption techniques used today are hybrid methods, providing the speed benefits of symmetric encryption with the key management properties of public-key cryptography. In creating the new scheme, not only did we incorporate hybrid methods but we reduced the amount of work that needed to be done by the algorithm and removed the variable cipher-text expansion overhead. However, whether it will be considered by all to be a true "chaffing and winnowing" method remains to be seen. Although the Message Authentication Codes were removed and some encryption is done in the new scheme, the packets are still authenticated by their serial number. Chaff is still interspaced between the valid packets and it is authentication information that is encrypted, which would have been done if we had used the "Designated Verifier Signatures" [Jakobsson et al., 1996], which Rivest suggested could replace the MACs. An alternative hybrid scheme could take a hash of each of the chaff packets, combine these and then public-key encrypt them. This would be very similar to how digital signatures work and may be viewed as a scheme that only uses authentication techniques.

The schemes we produced worked as we required them to do in Chapter 3. This is a good achievement, especially because there were cross-platform complexities involved. We benefited from the modular design when implementing the chaffing and winnowing schemes because the AONTs were very easy to change. We tested the implementations rigorously and as a result found some small bugs, which could have been overlooked with minimal testing.

The question has to be asked, why would someone decide to use chaffing and winnowing when there are already provably secure encryption techniques that are widely used. Unless the methods could stand up to laws regarding encryption, then there would not be much incentive to use them. However we have shown that if there was a need to use them, they would be a good alternative. We have at least shown that chaffing and winnowing is technically a real alternative to traditional encryption techniques. If chaffing and winnowing was not regarded by authorities as "encryption" and if a new law was proposed that vastly restricted traditional techniques, then an alternative would be available.

8.2 Critical Evaluation

This project was successful in what it set out to achieve, it investigated the new concept of chaffing and winnowing, implemented it and created a new scheme improving on the initial proposals. The knowledge gained from the research done into the initial proposals and into current encryption technologies allowed us to produce a new scheme. However, the new scheme was created by someone that is not an expert cryptographer and therefore cannot be proven to be secure here. The choice of using C as an implementation language was a good one, it proved to be fast, the code written was portable and the tools needed for the project GPG and GNU Multi-Precision library, were implemented in C. The way in which the schemes were designed was good too, the modularity made implementation easier and would make it easy to make changes in the future, for instance using a different AONT.

We could have compared the data expansion of the schemes when the original file had been compressed before "encryption", this is what GPG does by default but was turned off for the experiments. However, the results for how much extra data was added to the file would have been the same, at least for the GPG schemes and the hybrid chaffing and winnowing scheme, because they add a fixed overhead regardless of the original file size. The symmetric chaffing and winnowing schemes would have performed better, they would still add the same proportion of extra data but because the original file is smaller, there would be less data added.

Experiments were not carried out that measured the decryption speed of the schemes. We would expect the execution times to be very similar to the times we found, as the "decryption" process for all the schemes is very similar to the "encryption" process. However we would need to test this to be sure. It also may have been an advantage to test different implementations of hash functions for speed, before implementing OAEP, because in it the hash function is called once for every 16 bytes of data in the file. This may also have slightly speeded up the symmetric chaffing and winnowing schemes.

8.3 Future Work

Future work that could be performed in chaffing and winnowing:

Further Optimisation - The code we produced could be optimised further, to make it perform even better. Lower level code could be written for some components of the schemes and a better performing hash function could be used in OAEP to improve its speed.

Compression - The original file could be compressed before it is "encrypted". This is what GNU-Privacy Guard does and it has many benefits. The file is compressed so it is much smaller than the original file, because the file is smaller, it can be encrypted much faster and also compressed files generally contain less redundant data which makes breaking a scheme more difficult [Schneier, 1993].

Complete Application - The implementations produced here are far from a completely usable and practical application. Cryptographic applications usually provide features such as: a user interface - making it easy for anyone to use, key-management - a key database listing recipients public keys or connection to a key server to retrieve public keys and the ability to sign messages. Cryptographic applications are also generally built on a standard such as OpenPGP or PCKS, so that keys are stored in a specified format and the application can tell which algorithm a file was encrypted with. This means that a file encrypted with one application can be decrypted with another application that uses the same standard.

New Schemes - There is scope in this area to devise new and more efficient chaffing and winnowing schemes, incorporating more ideas from commonly used confidentiality techniques. Also different All-Or-Nothing Transforms could be used and new ones developed.

8.4 Personal Remarks

This project was very interesting and a great deal was learnt throughout it. A lot was learnt about chaffing and winnowing, why it was proposed and about cryptography in general. The author also learnt about how cryptography is used in practice. The author's programming skills were improved, including knowledge about cross-platform development and using external libraries for extra functionality.

Bibliography

Ross Anderson and Eli Biham. Two practical and provably secure block ciphers: BEAR and LION. In *IWFSE: International Workshop on Fast Software Encryption, LNCS*, 1996.

M. Bellare and A. Boldyreva. The Security of Chaffing and Winnowing. In T. Okamodo, editor, *Lecture Notes In Computer Science, Advances In Cryptography - ASIACRYPT '00*, volume 1976. Springer-Verlag, 2000.

M. Bellare and P. Rogaway. Optimal Asymmetric Encryption - How to Encrypt with RSA. In A. De Santis, editor, *Lecture Notes In Computer Science, Advances In Cryptography - EUROCRYPT '94*, volume 950. Springer-Verlag, 1994.

J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Lecture Notes in Computer Science*, volume 1666, pages 216–233, 1999.

Victor Boyko. On the Security Properties of OAEP as an All-or-Nothing Transform. In Michael Wiener, editor, *Lecture Notes In Computer Science, Advances In Cryptography - CRYPTO '99*, volume 1666. Springer-Verlag, 1999.

Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable Encryption. In Burton S. Kaliski Jr., editor, *Lecture Notes In Computer Science, Advances In Cryptography - CRYPTO '97*, volume 1294. Springer-Verlag, 1997.

Richard Clayton and George Danezis. Chaffinch: Confidentiality in the Face of Legal Threats. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 70–86. Springer-Verlag, 2003.

Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

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

FIPS. *Data Encryption Standard.* National Institute for Standards and Technology, 2001b. URL `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

FIPS. *Computer Data Authentication.* National Institute for Standards and Technology, 1985.

Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 0-387-15658-5.

H. Finney J. Callas, L Donnerhacke and R. Thayer. OpenPGP Message Format, 1998. URL `http://www.ietf.org/rfc/rfc2440.txt`. RFC: 2440.

Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. *Lecture Notes in Computer Science*, 1070: 143–154, 1996.

H. Krawczyk, M. Bellare, and R. Canetti. Keying Hash Functions for Message Authentication. In N. Koblitz, editor, *Lecture Notes In Computer Science, Advances In Cryptography - CRYPTO '96*, volume 1109. Springer-Verlag, 1996.

H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication, 1997. RFC: 2104.

John McHugh. Chaffing at the bit: Thoughts on a note by ronald rivest. In *IH '99: Proceedings of the Third International Workshop on Information Hiding*, pages 395–404, London, UK, 2000. Springer-Verlag. ISBN 3-540-67182-X.

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

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

Ronald L. Rivest. The RC5 encryption algorithm, from dr. dobb's journal, january, 1995. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996*, 1996. URL `http://theory.lcs.mit.edu/~rivest/Rivest-rc5.pdf`.

Ronald L. Rivest. All-or-Nothing Encryption and The Package Transform. In *Proceedings of the 1997 Fast Software Encryption Conference, Springer Lecture Notes in Computer Science*, volume 1267, pages 210–218. Springer-Verlag, 1997.

Ronald L. Rivest. Chaffing and Winnowing: Confidentiality without Encryption. *CryptoBytes (RSA Laboratories)*, 4(1):12–17, 1998a. URL `http://theory.lcs.mit.edu/~rivest/chaffing.txt`.

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

RSA Labs. Crypto FAQ. Web Reference, 2004. URL `http://www.rsasecurity.com/rsalabs/node.asp?id=2218`. 3.1.5 How large a key should be used in the RSA cryptosystem?

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0471597562.

SHA. Secure Hash Standard, 1999. FIPS 180-1, National Institute of Standards of Technology, US Department of Commerce.

Appendix A

Statistical Results

Here we detail the full statistical results, a spreadsheet will the full execution time results in, can be found on the provided CD, named Analysis.xls.

A.1 Analysis Of Variance

In this section we give the full analysis of variance results.

A.1.1 ANOVA Symmetric Execution Time Results

Here is the Analysis of Variance the execution times of the symmetric schemes. Table A.1 shows the results and Table A.2 shows the ANOVA table computed from the results.

A.1.2 ANOVA Hybrid Execution Time Results

Here is the Analysis of Variance the execution times of the symmetric schemes. Table A.3 shows the results and Table A.4 shows the ANOVA table computed from the results.

Table A.1: Symmetric Execution Time Analysis

File Size (bytes)	CW PT	CW OAEP	GPG Triple-DES	GPG AES	Row Sum	Row Mean
20384	0.0041	0.0050	0.0110	0.0080	0.0281	0.0070
51808	0.0080	0.0099	0.0140	0.0100	0.0419	0.0105
102677	0.0130	0.0170	0.0189	0.0130	0.0619	0.0155
274106	0.0300	0.0390	0.0350	0.0235	0.1275	0.0319
475136	0.0502	0.0658	0.0536	0.0350	0.2047	0.0512
1080054	0.1117	0.1469	0.1100	0.0700	0.4387	0.1097
5030281	0.5132	0.6743	0.4812	0.3016	1.9703	0.4926
20948069	2.1208	2.7995	1.9765	1.2319	8.1287	2.0322
Column Sum	2.8510	3.7575	2.7003	1.6930		
Column Mean	0.3564	0.4697	0.3375	0.2116		

Table A.2: ANOVA For Symmetric Schemes

Component	Sum Of Squares	Degrees Of Freedom	Mean Square	F-Computed	F-Table
File Size	13.7712	7	1.9673	39.5047	2.4876
Scheme	0.2681	3	0.0894	1.7947	3.0725
Errors	1.0458	21	0.0498		

Table A.3: Hybrid Execution Time Analysis

File Size (bytes)	CW PT	CW OAEP	GPG Triple-DES	GPG AES	Row Sum	Row Mean
20384	0.0060	0.0070	0.0228	0.0218	0.0576	0.0144
51808	0.0090	0.0100	0.0260	0.0237	0.0687	0.0172
102677	0.0120	0.0160	0.0311	0.0267	0.0858	0.0214
274106	0.0240	0.0320	0.0490	0.0369	0.1419	0.0355
475136	0.0370	0.0520	0.0699	0.0485	0.2074	0.0518
1080054	0.0781	0.1120	0.1323	0.0840	0.4063	0.1016
5030281	0.3461	0.5058	0.5408	0.3158	1.7085	0.4271
20948069	1.4271	2.1747	2.1877	1.2451	7.0345	1.7586
Column Sum	1.9392	2.9095	3.0595	1.8025		
Column Mean	0.2424	0.3637	0.3824	0.2253		

Table A.4: ANOVA For Hybrid Schemes

Component	Sum Of Squares	Degrees Of Freedom	Mean Square	F-Computed	F-Table
File Size	10.2150	7	1.4593	49.8461	2.4876
Scheme	0.1576	3	0.0525	1.7944	3.0725
Errors	0.6148	21	0.0293		

A.2 Data Expansion

Here we give the full data expansion results for the schemes.

A.2.1 Symmetric Schemes

Table A.5 shows how much extra data was added to each file after "encryption" by the symmetric schemes. Table A.6 shows the file size after "encryption".

Table A.5: Symmetric Cipher-text Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	19360	19360	49	83
51808	21746	21746	49	83
102677	25801	25801	54	86
274106	39154	39154	49	81
475136	54922	54922	56	88
1080054	102192	102192	64	96
5030281	410783	410783	62	94
20948069	1654261	1654261	48	80

Table A.6: Symmetric Cipher-text Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	39744	39744	20433	20467
51808	73554	73554	51857	51891
102677	128478	128478	102731	102763
274106	313260	313260	274155	274187
475136	530058	530058	475192	475224
1080054	1182246	1182246	1080118	1080150
5030281	5441064	5441064	5030343	5030375
20948069	22602330	22602330	20948117	20948149

A.2.2 Hybrid Schemes

Table A.7 shows how much extra data was added to each file after "encryption" by the hybrid schemes. Table A.8 shows the file size after "encryption".

Table A.7: Hybrid Cipher-text Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	17252	17252	341	333
51808	17188	17188	341	333
102677	17263	17263	344	336
274106	17226	17226	331	339
475136	17284	17284	338	346
1080054	17294	17294	346	354
5030281	17275	17275	344	352
20948069	17183	17183	330	338

Table A.8: Hybrid Cipher-text Expansion Results

File Size (bytes)	CW Package Transform	CW OAEP	GPG Triple-DES	GPG AES
20384	37636	37636	20717	20725
51808	68996	68996	52141	52149
102677	119940	119940	103013	103021
274106	291332	291332	274437	274445
475136	492420	492420	475474	475482
1080054	1097348	1097348	1080400	1080408
5030281	5047556	5047556	5030625	5030633
20948069	20965252	20965252	20948399	20948407

Appendix B

Test Scripts

Here are listings of the shell scripts that were used to test the implementations.

B.1 Correctness Testing

```
#!/bin/sh

# corr_test.sh
#
#
# Created by John Larkin on 21/04/2006.
#set −x

EXE=
TYPE=‘uname −m‘
if [ ”$TYPE” == ”Power Macintosh” ]
then
    ROOT=$HOME/Project/Builds/Production
elif [ ”$TYPE” == ”i686” ]
then
    ROOT=/c/Project/Production
    EXE=.exe
elif [ ”$TYPE” == ”sun4u” ]
then
    ROOT=$HOME/Project/Production
```

```
else
    echo "System not supported"
    exit 1
fi

FILE=file.txt
OUTFILE=output.txt
REC_FILE=file2.txt
TEST_FILE1=file_test1.txt
TEST_FILE2=file_test1.txt
TEST_FILE3=file_test1.txt
TEST_FILE4=file_test1.txt

TESTING=$ROOT/Testing
TESTS=200
KEY_CHANGE=20

COUNTER=0
while [ $COUNTER -lt $TESTS ]
do
    command $ROOT/AONT_PT/cw_aont_pt$EXE -c $FILE \
                    $OUTFILE test_key
    if [ $? -ne 0 ]; then echo "CW_AONT_PT Chaffing failed"; exit 1; fi
    command $ROOT/AONT_PT/cw_aont_pt$EXE -w $OUTFILE \
                    $REC_FILE test_key
    if [ $? -ne 0 ]; then echo "CW_AONT_PT Winnowing failed"; exit 1; fi

    diff $REC_FILE $TEST_FILE1
    if [ $? -ne 0 ]
    then
        echo "CW_AONT_PT Files don't match, iteration $COUNTER"
        exit 1
    fi

    COUNTER=`expr $COUNTER + 1`
done

echo "CW_AONT_PT  $COUNTER tests, 0 failures"

COUNTER=0
while [ $COUNTER -lt $TESTS ]
do
    command $ROOT/AONT_OAEP/cw_aont_oaep$EXE -c $FILE \
                    $OUTFILE test_key
    if [ $? -ne 0 ]; then echo "CW_AONT_OAEP Chaffing failed"; exit 1; fi
    command $ROOT/AONT_OAEP/cw_aont_oaep$EXE -w $OUTFILE \
```

```
                        $REC_FILE test_key
    if [ $? -ne 0 ]; then echo "CW_AONT_OAEP Winnowing failed"; exit 1; fi

    diff $REC_FILE $TEST_FILE2
    if [ $? -ne 0 ]
    then
        echo "CW_AONT_OAEP Files don't match, iteration $COUNTER"
        exit 1
    fi

    COUNTER=`expr $COUNTER + 1`
done

echo "CW_AONT_OAEP   $COUNTER tests, 0 failures"

COUNTER=0
while [ $COUNTER -lt $TESTS ]
do
    if [ `expr $COUNTER % $KEY_CHANGE` -eq 0 ]
    then
        command $ROOT/PubKey_PT/cw_pk_pt$EXE -g
        if [ $? -ne 0 ]; then echo "CW_PK_PT key generation failed"; exit 1; fi
    fi

    command $ROOT/PubKey_PT/cw_pk_pt$EXE -c $FILE $OUTFILE pub.key
    if [ $? -ne 0 ]; then echo "CW_PK_PT Chaffing failed"; exit 1; fi
    command $ROOT/PubKey_PT/cw_pk_pt$EXE -w $OUTFILE $REC_FILE prv.key
    if [ $? -ne 0 ]; then echo "CW_PK_PT Winnowing failed"; exit 1; fi

    diff $REC_FILE $TEST_FILE3
    if [ $? -ne 0 ]
    then
        echo "CW_PK_PT Files don't match, iteration $COUNTER"
        exit 1
    fi

    COUNTER=`expr $COUNTER + 1`
done

echo "CW_PK_PT $COUNTER tests, 0 failures"

COUNTER=0
while [ $COUNTER -lt $TESTS ]
do
    if [ `expr $COUNTER % $KEY_CHANGE` -eq 0 ]
    then
```

```
        command $ROOT/PubKey_OAEP/cw_pk_oaep$EXE -g
        if [ $? -ne 0 ]; then echo "CW_PK_OAEP key generation failed"; exit 1; fi
    fi

    command $ROOT/PubKey_OAEP/cw_pk_oaep$EXE -c $FILE $OUTFILE pub.key
    if [ $? -ne 0 ]; then echo "CW_PK_OAEP Chaffing failed"; exit 1; fi
    command $ROOT/PubKey_OAEP/cw_pk_oaep$EXE -w $OUTFILE $REC_FILE prv.key
    if [ $? -ne 0 ]; then echo "CW_PK_OAEP Winnowing failed"; exit 1; fi

    diff $REC_FILE $TEST_FILE4
    if [ $? -ne 0 ]
    then
        echo "CW_PK_OAEP Files don't match, iteration $COUNTER"
        exit 1
    fi

    COUNTER=`expr $COUNTER + 1`
done

echo "CW_PK_OAEP  $COUNTER tests, 0 failures"
```

B.2 Speed Testing

```
#!/bin/sh

# speed_test.sh
#
#
# Created by John Larkin on 21/04/2006.
#
#set -x

TYPE=`uname -m`
if [ "$TYPE" == "Power Macintosh" ]
then
    ROOT=$HOME/Project/Builds/Production
elif [ "$TYPE" == "i686" ]
then
    ROOT=/c/Project/Production
elif [ "$TYPE" == "sun4u" ]
then
    ROOT=$HOME/Project/Production
else
    echo "System not supported"
    exit 1
fi

if [ $# -ne 3 ]
then
    echo "Usage SCHEME [-e|-d] file"
    exit 1
fi

RECIP=cs2jmal@bath.ac.uk
TMP=/tmp/`basename $0`$$.tmp
PATH=$PATH:/usr/local/bin
FILE=$3
TESTING=$ROOT/Testing
KEYS=$TESTING/Keys
TESTS=75

if [ "$1" == "CW_AONT_PT" ]
then
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.cw
        SCHEME="$ROOT/AONT_PT/cw_aont_pt -c $FILE $OUTFILE test_key"
```

```
    elif [ ”$2” == ”-d” ]
    then
        OUTFILE_NAME=‘basename -s .cw $3 | \
            awk ’ { split($1,a,”.”); print a[1]”_2.”a[2]; }’‘
        OUTFILE_DIR=‘dirname $3‘
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME=”$ROOT/AONT_PT/cw_aont_pt -w $FILE $OUTFILE test_key”
    else
        echo ”Invalid option $2”
        exit 1
    fi
elif [ ”$1” == ”CW_AONT_OAEP” ]
then
    if [ ”$2” == ”-e” ]
    then
        OUTFILE=$3.cw
        SCHEME=”$ROOT/AONT_OAEP/cw_aont_oaep -c $FILE $OUTFILE test_key”
    elif [ ”$2” == ”-d” ]
    then
        OUTFILE_NAME=‘basename -s .cw $3 | \
                            awk ’ { split($1,a,”.”); print a[1]”_2.”a[2]; }’‘
        OUTFILE_DIR=‘dirname $3‘
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME=”$ROOT/AONT_OAEP/cw_aont_oaep -w $FILE $OUTFILE test_key”
    else
        echo ”Invalid option $2”
        exit 1
    fi
elif [ ”$1” == ”CW_PK_PT” ]
then
    if [ ”$2” == ”-e” ]
    then
        OUTFILE=$3.cw
        SCHEME=”$ROOT/Pubkey_PT/cw_pk_pt -c $FILE $OUTFILE $KEYS/pub.key”
    elif [ ”$2” == ”-d” ]
    then
        OUTFILE_NAME=‘basename -s .cw $3 |
                awk ’ { split($1,a,”.”); print a[1]”_2.”a[2]; }’‘
        OUTFILE_DIR=‘dirname $3‘
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME=”$ROOT/Pubkey_PT/cw_pk_pt -w $FILE $OUTFILE $KEYS/prv.key”
    else
        echo ”Invalid option $2”
        exit 1
    fi
elif [ ”$1” == ”CW_PK_OAEP” ]
```

```
then
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.cw
        SCHEME="$ROOT/Pubkey_OAEP/cw_pk_oaep -c $FILE $OUTFILE $KEYS/pub.key"
    elif [ "$2" == "-d" ]
    then
        OUTFILE_NAME=`basename -s .cw $3 | \
                    awk '{ split($1,a,"."); print a[1]"_2."a[2]; }'`
        OUTFILE_DIR=`dirname $3`
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME="$ROOT/Pubkey_OAEP/cw_pk_oaep -w $FILE $OUTFILE $KEYS/prv.key"
    else
        echo "Invalid option $2"
        exit 1
    fi
elif [ "$1" == "GPG_3DES_SYM" ]
then
    CAT="cat $KEYS/pp.txt"
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.gpg
        SCHEME="gpg -c -z 0 --passphrase-fd 0 --cipher-algo 3DES -o $OUTFILE $FILE"
    elif [ "$2" == "-d" ]
    then
        OUTFILE_NAME=`basename -s .gpg $3 | \
                    awk '{ split($1,a,"."); print a[1]"_2."a[2]; }'`
        OUTFILE_DIR=`dirname $3`
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME="gpg -d --passphrase-fd 0 -o $OUTFILE $FILE"
    else
        echo "Invalid option $2"
        exit 1
    fi
elif [ "$1" == "GPG_AES_SYM" ]
then
    CAT="cat $KEYS/pp.txt"
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.gpg
        SCHEME="gpg -c -z 0 --passphrase-fd 0 --cipher-algo AES -o $OUTFILE $FILE"
    elif [ "$2" == "-d" ]
    then
        OUTFILE_NAME=`basename -s .gpg $3 | \
                    awk '{ split($1,a,"."); print a[1]"_2."a[2]; }'`
        OUTFILE_DIR=`dirname $3`
```

```
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME="gpg -d --passphrase-fd 0 -o $OUTFILE $FILE"
    else
        echo "Invalid option $2"
        exit 1
    fi
elif [ "$1" == "GPG_3DES_PK" ]
then
    CAT="cat $KEYS/pp.txt"
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.gpg
        SCHEME="gpg -e -z 0 --passphrase-fd 0 --cipher-algo 3DES \
                --recipient $RECIP -o $OUTFILE $FILE "
    elif [ "$2" == "-d" ]
    then
        OUTFILE_NAME=`basename -s .gpg $3 | \
                awk '{ split($1,a,"."); print a[1]"_2."a[2]; }'`
        OUTFILE_DIR=`dirname $3`
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME="gpg -d --passphrase-fd 0 -o $OUTFILE $FILE"
    else
        echo "Invalid option $2"
        exit 1
    fi
elif [ "$1" == "GPG_AES_PK" ]
then
    CAT="cat $KEYS/pp.txt"
    if [ "$2" == "-e" ]
    then
        OUTFILE=$3.gpg
        SCHEME="gpg -e -z 0 --passphrase-fd 0 --cipher-algo AES \
                    --recipient $RECIP -o $OUTFILE $FILE"
    elif [ "$2" == "-d" ]
    then
        OUTFILE_NAME=`basename -s .gpg $3 | \
                awk '{ split($1,a,"."); print a[1]"_2."a[2]; }'`
        OUTFILE_DIR=`dirname $3`
        OUTFILE=$OUTFILE_DIR/$OUTFILE_NAME
        SCHEME="gpg -d --passphrase-fd 0 -o $OUTFILE $FILE"
    else
        echo "Invalid option $2"
        exit 1
    fi
fi

```

```

COUNTER=0
while [ $COUNTER -lt $TESTS ]
do
    if [ `echo $1 | grep GPG | wc -l | awk '{print $1}'` -gt 0 ]
    then
        rm -f $OUTFILE
        (time $CAT | $SCHEME) 2>> $TMP
    else
        (time $SCHEME) 2>> $TMP
    fi

    COUNTER=`expr $COUNTER + 1`
done
#echo $SCHEME
#cat $TMP

echo "$1 $2 Results, $TESTS executions on $3"
cat $TMP | grep -v real | grep -v sys | grep -v ^$ | awk '{print $2}' | cut -c3-7
rm -f $TMP
```

Appendix C

Source Code

Here we list the C source code that was produced in the project. We do not list the implementations of MD5 and RC5 that we used but these can be found on the supplied CD.

C.1 hmac.c

This is the source code for the HMAC that was produced, which was based on a reference implementation.

```
/*
 *  hmac.c
 *  Created by John Larkin.
 *  Based on sample code presented in RFC 2104
 *
 */

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

unsigned char k_ipad[65];
unsigned char k_opad[65];
```

```

/* initialise hmac */
void hmac_init(unsigned char *key, int key_len) {

    int i;

    if(key_len > 64) {

        MD5_CTX tctx;

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

        key = tctx.digest;
        key_len = 16;
    }

    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);

    for (i=0; i<64; i++) {
        k_ipad[i] ^= 0x36;
        k_opad[i] ^= 0x5c;
    }

}

/* compute hmac */
void hmac(unsigned char* text, int text_len, unsigned char digest[DIGEST_LENGTH])
{
    MD5_CTX context;
    int i;

    MD5Init(&context);

    MD5Update(&context,k_ipad,64);
    MD5Update(&context,text,text_len);
    MD5Final(&context);

    MD5Init(&context);

    MD5Update(&context,k_opad,64);
    MD5Update(&context,context.digest,ORIG_DIGEST_LENGTH);
```

```
    MD5Final(&context);

    for(i=0; i<DIGEST_LENGTH; i++)
        digest[i] = context.digest[i];
}

/*#define TEST*/
#ifdef TEST
/* main method checks the test vectors from RFC 2104 */
int main (int argc, char **argv)
{

    unsigned char checksum[DIGEST_LENGTH], key1[16], *key2, *data, data2[50];
    int i;

    memset(key1,0x0b,16);
    memset(k_ipad,0,sizeof(k_ipad));
    memset(k_opad,0,sizeof(k_opad));
    memcpy(k_ipad,key1,16);
    memcpy(k_opad,key1,16);

    for (i=0; i<64; i++) {
        k_ipad[i] ^= 0x36;
        k_opad[i] ^= 0x5c;
    }

    data = "Hi There";
    hmac(data,8,checksum);

    printf("Data: %s",data);
    printf("\t\t\t\tKey: ");
    for (i = 0; i < 16; i++) {
        printf ("%02x", key1[i]);
    }
    printf("\tDigest: ");

    for (i = 0; i < DIGEST_LENGTH; i++) {
        printf ("%02x", (unsigned int) checksum[i]);
    }
    printf ("\n");

    key2 = "Jefe";
    hmac_init(key2,strlen(key2));
    data = "what do ya want for nothing?";
    hmac(data,strlen(data),checksum);
    printf("Data: %s\tKey: %s\t\t\t\tDigest: ",data,key2);
```

```

    for (i = 0; i < DIGEST_LENGTH; i++) {
        printf ("%02x", (unsigned int) checksum[i]);
    }
    printf ("\n");

    memset(key1,0xaa,16);
    memset(data2,0xdd,50);

    memset(k_ipad,0,sizeof(k_ipad));
    memset(k_opad,0,sizeof(k_opad));
    memcpy(k_ipad,key1,16);
    memcpy(k_opad,key1,16);

    for (i=0; i<64; i++) {
        k_ipad[i] ^= 0x36;
        k_opad[i] ^= 0x5c;
    }

    hmac(data2,50,checksum);

    printf("Data: ");
    for (i = 0; i < 5; i++) {
        printf ("%02x", data2[i]);
    }
    printf("...(50 bytes)\t\tKey: ");
    for (i = 0; i < 16; i++) {
        printf ("%02x", key1[i]);
    }
    printf("\tDigest: ");
    for (i = 0; i < DIGEST_LENGTH; i++) {
        printf ("%02x", (unsigned int) checksum[i]);
    }
    printf ("\n");

    return 0;
}

#endif
```

C.2 cw_lib.c

This is the source code for the Chaffing and Winnowing library that was produced.

```c
/*
 *  cw_lib.c
 *
 *  Created by John Larkin.
 *  Provides common functions
 *  required by chaffing and winnowing.
 *
 */

#include <stdlib.h>
#include "cw_lib.h"

/* if Windows or Mac, don't use arc4random() */
#ifdef WIN32
    #define arc4random() rand()
#endif
#ifdef sun
    #define arc4random() random()
#endif

/* check if machine is big-endian */
int is_big_endian(void) {

    long int i=0x12345678;
    unsigned char* c_ptr= (unsigned char*) &i;

    if(*c_ptr == 0x12) {
        return 0;
    } else {
        return 1;
    }

}

/* generate a random "chaff" packet */
void generate_chaff_packet(unsigned char *chaff_packet, int size) {

    int i;
    for(i=0; i<size; i++) {
        chaff_packet[i] = arc4random() % 256;
```

```
   }

}

/* comparison function used by qsort */
int int_comp(const void *a, const void *b) {

   return *(int *)a - *(int *)b;
}

/* calculate random subset of indices, for chaff packets to be placed in
   return list of indices in ascending order, without replications */
void set_chaff_positions(unsigned long int *positions,
                         int chaff_blocks, int blocks){

   int i, j, total_blocks;
   unsigned long int n;

   total_blocks = blocks + chaff_blocks;

   for(i=0; i<chaff_blocks; i++) {
      n = rand() % total_blocks;
      for(j=0; j<i; j++) {
         if(positions[j] == n) {
            n = arc4random() % total_blocks;
            j=-1;
         }
      }
      positions[i]=n;
   }
   qsort((void *)positions,chaff_blocks,sizeof(long int),int_comp);
}

/*#define TEST*/
#ifdef TEST
#include <stdio.h>
#include <time.h>
int main(void) {

   unsigned char chaff_packet[128];
   unsigned long int positions[128];
   int i, last, duplicate=0;

   /* test if symbolic constants are defined */
   #ifdef WIN32
      printf("WIN32 symbolic constant defined\n");
```

```
        srand(time(NULL));
    #endif
    #ifdef sun
        printf("sun symbolic constant defined\n");
        srandom(time(NULL));
    #endif

    printf("Machine is %s\n", is_big_endian() ? "Little Endian" : "Big Endian");

    /* generate some test chaff packets */
    generate_chaff_packet(chaff_packet,128);
    printf("Chaff packet: ");
    for(i=0;i<128;i++)
        printf("%02x",chaff_packet[i]);
    printf("\n");

    generate_chaff_packet(chaff_packet,128);
    printf("Chaff packet: ");
    for(i=0;i<128;i++)
        printf("%02x",chaff_packet[i]);
    printf("\n");

    /* generate some test chaff positions */
    set_chaff_positions(positions,128,128);
    printf("Chaff packet positions: ");
    for(i=0;i<128;i++)
        printf("%i ",positions[i]);
    printf("\n");

    /* check there are no duplicates in the chaff positions */
    last=-1;
    for(i=0;i<128;i++){
        if(positions[i] == last) {
            printf("Index: %i, %i duplicate!\n");
            duplicate = 1;
        }
        last = positions[i];
    }
    if(duplicate == 0)
        printf("No duplicates found in chaff packet positions\n");

    /* check SWAP macro works correctly */
    i=0x12345678;
    printf("i = %08lx\n",i);
    SWAP(i);
    printf("i = %08lx\n",i);
```

```
    SWAP(i);
    printf("i = %08lx\n", i);

    return 0;
}
#endif
```

C.3 aont.c

This is the source code for the Package Transform that was produced.

```

 /*
 *  aont.c
 *
 *
 *  Created by John Larkin.
 *  Implementation of Ronald Rivest's Package Transform
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "aont.h"
#include "RC5REF.h"

#define KEY_SIZE              16
#define BLOCK_SIZE            8
#define TRANSFORM             1
#define INVERSE_TRANSFORM     2
#define OUTPUT_BLOCKS         128
#define OUTPUT_BLOCK_SIZE     128

#define t                     26   /* size of key table S = 2*(r+1) words */

#ifdef WIN32
   #define arc4random() rand()
#endif
#ifdef sun
   #define arc4random() random()
#endif

/* randomly chosen public key */
unsigned char public_key[KEY_SIZE] = { '\x52','\x69','\xF1','\x49',
                                       '\xD4','\x1B','\xA0','\x15',
                                       '\x24','\x97','\x57','\x4D',
                                       '\x7F','\x15','\x31','\x25'};

WORD S_pub[t];      /* expanded key table for public key   */
```

```
WORD S_prv[t];      /* expanded key table for private key  */
size_t length = BLOCK_SIZE, size = sizeof(char);

/* generate a random key */
void generate_key(unsigned char key[KEY_SIZE]) {

    unsigned char c;
    int i;
    for(i=0; i<KEY_SIZE; i++) {
        c = arc4random() % 256;
        key[i] = c;
    }

}

void transform(FILE *in, FILE *out, unsigned char k[KEY_SIZE], int blocks) {

    int             i, j, counter = 0, mode, BLOCK_COUNT;
    long int        file_size;
    unsigned char   key[KEY_SIZE];
    unsigned char   final_block[KEY_SIZE];
    unsigned char   input_block[BLOCK_SIZE];
    unsigned char   output_block[BLOCK_SIZE];
    WORD            pt[2]  ={0,0};
    WORD            ct[2]  ={0,0};

    BLOCK_COUNT = ((blocks * OUTPUT_BLOCK_SIZE)  / BLOCK_SIZE) - 1;

    /* get input file size */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    /* if a key has been passed, we are inverting a transform
        otherwise generate one  */
    if(k == NULL) {
        generate_key(key);
        mode = TRANSFORM;
    } else {
        memcpy(key,k,KEY_SIZE);
        mode = INVERSE_TRANSFORM;
    }

    /* setup RC5 key tables */
    RC5_SETUP(key, S_prv);
    RC5_SETUP(public_key, S_pub);
```

```

    memcpy(final_block,key,KEY_SIZE);

    while(!feof(in) || counter <= (BLOCK_COUNT-2) ||
                          (counter > BLOCK_COUNT && (counter % 16 != 14))) {

        if(mode == INVERSE_TRANSFORM && (counter*BLOCK_SIZE) >= (file_size-KEY_SIZE))
            break;

        /* read next input block */
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,size,length,in);

        pt[0] = 0, pt[1] = 0;
        pt[0] = counter;

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

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

        /* XOR cipher text with input block */
        for(i=0, j=24; i<4; i++) {
            output_block[i] = input_block[i] ^ ((ct[0] >> j) & 0xFF);

            if(mode == TRANSFORM) {
                pt[0] |= ((output_block[i] ^ ((counter >> j) & 0xFF)) << j);
            }
            j = j-8;
        }

        for(i=4, j=24; i<8; i++) {
            output_block[i] = input_block[i] ^ ((ct[1] >> j) & 0xFF);

            if(mode == TRANSFORM) {
                pt[1] |= (output_block[i] << j);
            }
            j = j-8;
        }

        /* write output to file */
        fwrite(output_block,size,length,out);

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

        /* create hash of current block */
```

```
        if(mode == TRANSFORM) {
            RC5_ENCRYPT(pt, ct, S_pub);

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

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

        counter++;
    }

    /* if we are transforming, write final block to output file */
    if(mode == TRANSFORM) {
        fwrite(final_block,size,KEY_SIZE,out);
    }

}

void inverse_transform(FILE *in, FILE *out) {

    long int        file_size;
    long int        counter=0;
    int             i, j, blocks;
    int             key_blocks = KEY_SIZE / BLOCK_SIZE;
    WORD            pt[2] ={0,0};
    WORD            ct[2] ={0,0};
    unsigned char   key[KEY_SIZE];
    unsigned char   final_block[KEY_SIZE];
    unsigned char   input_block[BLOCK_SIZE];
    unsigned char   hash_block[BLOCK_SIZE];

    /* get input file size */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    blocks = (int)floor((double)file_size / BLOCK_SIZE);

```

```
    /* setup RC5 key table */
    RC5_SETUP(public_key, S_pub);

    memset(key,0,KEY_SIZE);

    while(!feof(in) && counter < (blocks-key_blocks)) {

        /* read next input block */
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,size,length,in);

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

        /* create hash of each input block, to recover key */
        for(i=0, j=24; i<4; i++) {
            hash_block[i] = input_block[i] ^ ((counter >> j) & 0xFF);
            pt[0] = pt[0] | (hash_block[i] << j);
            j = j-8;
        }

        for(i=4, j=24; i<8; i++) {
            hash_block[i] = input_block[i];
            pt[1] = pt[1] | (hash_block[i] << j);
            j = j-8;
        }

        RC5_ENCRYPT(pt,ct,S_pub);

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

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

        counter++;

    }

    fread(final_block,size,KEY_SIZE,in);

```

```
   /* recover key by XORing with XOR of the hash of all blocks */
   for(i=0; i<KEY_SIZE; i++) {
      key[i] = key[i] ^ final_block[i];
   }

   /* transform back */
   fseek(in,0,SEEK_SET);
   transform(in, out, key, 0);

}

/*#define TEST*/
#ifdef TEST

int main(int argc, char **argv)
{
   FILE *in = NULL, *out = NULL;

   #ifdef WIN32
      srand(time(NULL));
   #endif
   #ifdef sun
      srandom(time(NULL));
   #endif

   if(argc == 4) {
      in = fopen(argv[2],”rb”);
      if(in == NULL) {
         printf(”Error opening file %s\n”, argv[2]);
         return 1;
      }

      out = fopen(argv[3],”wb”);
      if(out == NULL) {
         printf(”Error opening file %s\n”, argv[3]);
         return 1;
      }

      if((strcmp(argv[1],”−t”)) == 0) {
         printf(”Transforming file\n”);
         transform(in, out, NULL, 128);
      } else if((strcmp(argv[1],”−i”)) == 0) {
         printf(”Inverting transformed file\n”);
         inverse_transform(in, out);
      } else {
         printf(”Invalid option: %s\n”,argv[1]);
```

```
            return 1;
        }

    } else {
        printf("Usage: [-t|-i] <infile> <outfile>");
        return 1;
    }

    return 0;
}

#endif
```

C.4 oaep.c

This is the source code for Optimal Asymmetric Encryption Padding that was produced.

```c
/*
 *  oaep.c
 *
 *
 *  Created by John Larkin.
 *  Implementation of Optimal Asymmetric
 *  Encryption Padding.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include "md5.h"
#include "oaep.h"

#define BLOCK_SIZE                128
#define DIGEST_LENGTH             16
#define TRUNCATED_DIGEST_LENGTH   16

#ifdef WIN32
    #define arc4random() rand()
#endif
#ifdef sun
    #define arc4random() random()
#endif

char *desc = "OAEP using MD5";

void generator(unsigned char *gen, int hashes, unsigned char r[DIGEST_LENGTH],
                              unsigned char d2[DIGEST_LENGTH]) {

    long int i, j;
    unsigned char buffer[DIGEST_LENGTH+sizeof(long int)];
    MD5_CTX ctx;

    /* initialise MD5 variables with given digest */
```

```
    MD5Init(&ctx);
    for(i=0,j=0;i<4;i++) {
        ctx.buf[i] |= (d2[j] << 24);
        ctx.buf[i] |= (d2[j+1] << 16);
        ctx.buf[i] |= (d2[j+2] << 8);
        ctx.buf[i] |= d2[j+3];
        j+=4;
    }

    /* copy random digest into buffer */
    for(i=sizeof(long int),j=0;i<(DIGEST_LENGTH+sizeof(long int));i++,j++)
        buffer[i] = r[j];

    /* create generator string */
    for(i=0;i<hashes;i++) {
        buffer[0] = (i >> 24) & 0xFF;
        buffer[1] = (i >> 16) & 0xFF;
        buffer[2] = (i >> 8) & 0xFF;
        buffer[3] = i & 0xFF;
        MD5Update(&ctx, buffer, 20);
        MD5Final(&ctx);
        for(j=0;j<TRUNCATED_DIGEST_LENGTH;j++) {
            *gen = ctx.digest[j];
            gen++;
        }
    }
}

/* create random string */
void random_string(unsigned char rs[DIGEST_LENGTH]) {

    int i;

    for(i=0;i<DIGEST_LENGTH;i++) {
        rs[i] = arc4random() % CHAR_MAX;
    }

}

void transform(FILE *in, FILE *out, int min_blocks) {

    int          file_size;
    int          orig_file_size;
    int          generator_length;
    int          blocks, i, j, counter=0;
    char         *num2 = "00000002";
```

```
   char           *num3 = "00000003";
   unsigned char  rand_digest[DIGEST_LENGTH];
   unsigned char  *gen;
   unsigned char  *x_bar_p;
   unsigned char  input_block[BLOCK_SIZE];
   MD5_CTX        context;
   MD5_CTX        context2;
   MD5_CTX        context3;

   /* get size of input file */
   fseek(in,0,SEEK_END);
   file_size = ftell(in);
   fseek(in,0,SEEK_SET);

   orig_file_size = file_size;

   /* find number of blocks in input file */
   blocks = (int)ceil((double)file_size / BLOCK_SIZE);

   /* if minimum number of blocks set, udjust sizes accordingly */
   if(min_blocks >= 0 && blocks < min_blocks) {
      blocks = min_blocks;
      generator_length = (int)ceil((double)(blocks * BLOCK_SIZE)
                                    / TRUNCATED_DIGEST_LENGTH);
      file_size = blocks * BLOCK_SIZE;
   } else {
      generator_length = (int)ceil((double)(blocks * BLOCK_SIZE)
                                    / TRUNCATED_DIGEST_LENGTH);
   }

   /* allocate memory for generator string */
   gen = malloc(sizeof(char) * ((generator_length * TRUNCATED_DIGEST_LENGTH)
                                     +(BLOCK_SIZE-DIGEST_LENGTH)));

   /* compute random string */
   random_string(rand_digest);

   /* initialise 1st context and compute digest*/
   MD5Init(&context);
   MD5Update(&context, desc, strlen(desc));
   MD5Final(&context);

   /* initialise 2nd context and compute digest */
   MD5Init(&context2);
   for(i=0,j=0;i<4;i++) {
      context2.buf[i] |= (context.digest[j] << 24);
```

```
        context2.buf[i] |= (context.digest[j+1] << 16);
        context2.buf[i] |= (context.digest[j+2] << 8);
        context2.buf[i] |= context.digest[j+3];
        j+=4;
    }
    MD5Update(&context2,num2,strlen(num2));
    MD5Final(&context2);

    /* create generator string */
    generator(gen,generator_length,rand_digest,context2.digest);
    x_bar_p = gen;

    while(counter < (blocks-1)) {
        /* zero input block and read next block */
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),BLOCK_SIZE,in);

        /* XOR input block with generator string */
        for(i=0;i<BLOCK_SIZE;i++) {
            *gen = *gen ^ input_block[i];
            gen++;
        }
        counter += 1;
    }

    counter *= BLOCK_SIZE;

    /* check how much of input file left to read and decide how much
        more padding needs to be added */
    if((orig_file_size-ftell(in)) > (BLOCK_SIZE-DIGEST_LENGTH)) {

        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),BLOCK_SIZE,in);
        for(i=0;i<BLOCK_SIZE;i++) {
            *gen = *gen ^ input_block[i];
            gen++;
        }
        counter += BLOCK_SIZE;

        for(i=0;i<(BLOCK_SIZE-DIGEST_LENGTH);i++) {
            *gen = *gen ^ 0;
            gen++;
        }
        counter += (BLOCK_SIZE-DIGEST_LENGTH);

    } else {
```

```
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),(BLOCK_SIZE-DIGEST_LENGTH),in);
        for(i=0;i<(BLOCK_SIZE-DIGEST_LENGTH);i++) {
            *gen = *gen ^ input_block[i];
            gen++;
        }
        counter += BLOCK_SIZE-DIGEST_LENGTH;
    }

    /* initialise context3 and compute digest */
    MD5Init(&context3);
    for(i=0,j=0;i<4;i++) {
        context3.buf[i] |= (context.digest[j] << 24);
        context3.buf[i] |= (context.digest[j+1] << 16);
        context3.buf[i] |= (context.digest[j+2] << 8);
        context3.buf[i] |= context.digest[j+3];
        j+=4;
    }
    MD5Update(&context3,num3,strlen(num3));
    MD5Update(&context3,x_bar_p,counter);
    MD5Final(&context3);

    /* XOR context 3 with rand string block */
    for(i=0;i<DIGEST_LENGTH;i++)
        rand_digest[i] ^= context3.digest[i];

    /* write everything to output file */
    gen = x_bar_p;
    fwrite(rand_digest,sizeof(char),DIGEST_LENGTH,out);
    fwrite(x_bar_p,sizeof(char),counter,out);

    /* free memory we allocated */
    free(gen);

}

void inverse_transform(FILE *in, FILE *out) {

    int            file_size, generator_length;
    int            blocks, i, j, message_length;
    char           *num2 = "00000002";
    char           *num3 = "00000003";
    unsigned char  rand_digest[DIGEST_LENGTH];
    unsigned char  *gen, *gen_p;
    unsigned char  *s, *s_ptr;
    MD5_CTX        context;
```

```
    MD5_CTX         context2;
    MD5_CTX         context3;

    /* get input file size */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    /* read random string */
    fread(rand_digest,sizeof(char),DIGEST_LENGTH,in);

    /* set size variables */
    message_length = file_size - DIGEST_LENGTH;
    blocks = file_size / BLOCK_SIZE;
    generator_length = (int)ceil((double)(blocks * BLOCK_SIZE)
                                        / TRUNCATED_DIGEST_LENGTH);

    /* initialise 1st context and compute digest */
    MD5Init(&context);
    MD5Update(&context,desc,strlen(desc));
    MD5Final(&context);

    /* allocate memory */
    gen = malloc(sizeof(char) * ((generator_length * TRUNCATED_DIGEST_LENGTH)
                                            +(BLOCK_SIZE-DIGEST_LENGTH)));
    s = malloc(sizeof(char) * message_length);

    /* store starting positions of allocated memory */
    s_ptr = s;
    gen_p = gen;

    /* read in rest of message */
    fread(s,sizeof(char),message_length,in);

    /* initialise 3rd context and compute digest */
    MD5Init(&context3);
    for(i=0,j=0;i<4;i++) {
        context3.buf[i] |= (context.digest[j] << 24);
        context3.buf[i] |= (context.digest[j+1] << 16);
        context3.buf[i] |= (context.digest[j+2] << 8);
        context3.buf[i] |= context.digest[j+3];
        j+=4;
    }
    MD5Update(&context3,num3,strlen(num3));
    MD5Update(&context3,s_ptr,message_length);
    MD5Final(&context3);
```

```

    /* XOR rand digest with context 3 digest */
    for(i=0; i<DIGEST_LENGTH; i++)
        rand_digest[i] ^= context3.digest[i];

    /* initialise 3rd context and compute digest */
    MD5Init(&context2);
    for(i=0,j=0;i<4;i++) {
        context2.buf[i] |= (context.digest[j] << 24);
        context2.buf[i] |= (context.digest[j+1] << 16);
        context2.buf[i] |= (context.digest[j+2] << 8);
        context2.buf[i] |= context.digest[j+3];
        j+=4;
    }
    MD5Update(&context2,num2,strlen(num2));
    MD5Final(&context2);

    /* create generator string */
    generator(gen,generator_length,rand_digest,context2.digest);

    /* recover original message, by XORing input with generator string */
    s = s_ptr;
    for(i=0; i<message_length; i++) {
        *s = *s ^ *gen;
        gen++;
        s++;
    }

    /* write original message to output */
    s = s_ptr;
    fwrite(s_ptr,sizeof(char),message_length,out);

    /* free memory we allocated */
    free(gen_p);
    free(s);

}

/*#define TEST*/
#ifdef TEST
int main(int argc, char **argv) {

    FILE *in, *out;

    #ifdef WIN32
        srand(time(NULL));
```

```c
    #endif
    #ifdef sun
        srandom(time(NULL));
    #endif

    if(argc == 4) {
        in = fopen(argv[2],"rb");
        if(in == NULL) {
            printf("Error opening file %s\n", argv[2]);
            return 1;
        }

        out = fopen(argv[3],"wb");
        if(out == NULL) {
            printf("Error opening file %s\n", argv[3]);
            return 1;
        }

        if((strcmp(argv[1],"-t")) == 0) {
            printf("Transforming file\n");
            transform(in, out, 128);
        } else if((strcmp(argv[1],"-i")) == 0) {
            printf("Inverting transformed file\n");
            inverse_transform(in, out);
        } else {
            printf("Invalid option: %s\n",argv[1]);
            return 1;
        }

    } else {
        printf("Usage: [-t|-i] infile outfile");
        return 1;
    }

    return 0;
}
#endif
```

C.5 rsa.c

This is the source code for RSA that was produced.

```c
/*
 *  rsa.c
 *
 *  Implementation of RSA using the
 *  GNU Multi Precision Library.
 *  Created by John Larkin on 22/03/2006.
 *
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <gmp.h>
#include ”rsa.h”
#include ”cw_lib.h”

void generate_key_pair(mpz_t *ep, mpz_t *dp, mpz_t *np) {

    mpz_t e, d, n, p, ps, q, qs, u_bound, l_bound, mod, gcd;
    gmp_randstate_t state;
    FILE *pub, *prv;

    pub = fopen(”pub.key”,”wb+”);
    prv = fopen(”prv.key”,”wb+”);

    /* initialise random state */
    gmp_randinit_default(state);
    gmp_randseed_ui(state, time(NULL));

    /* Initialise integers */
    mpz_init(q);
    mpz_init(qs);
    mpz_init(p);
    mpz_init(ps);
    mpz_init(e);
    mpz_init(d);
    mpz_init(n);
    mpz_init(u_bound);
```

```
    mpz_init(l_bound);
    mpz_init(gcd);
    mpz_init(mod);

    /* set bound on size of primes */
    mpz_ui_pow_ui(u_bound,2,512);
    mpz_ui_pow_ui(l_bound,2,504);

    /* set public exponent to (2^16)+1 */
    mpz_init_set_ui(e,65537);

    /* choose primes at random ~512 bits long */
    mpz_urandomm(p,state,u_bound);
    while(mpz_cmp(p,l_bound) < 0) {
        mpz_urandomm(p,state,u_bound);
    }

    /* generate prime p, make sure (p-1) is relatively prime to e*/
    mpz_nextprime(p,p);
    mpz_sub_ui(ps,p,1);
    mpz_gcd(gcd,e,ps);
    while(mpz_cmp_ui(gcd,1) != 0) {
        mpz_nextprime(p,p);
        mpz_sub_ui(ps,p,1);
        mpz_gcd(gcd,e,ps);
    }

    mpz_urandomm(q,state,u_bound);
    while(mpz_cmp(q,l_bound) < 0) {
        mpz_urandomm(q,state,u_bound);
    }

    /* generate prime q, make sure (q-1) is relatively prime to e*/
    mpz_nextprime(q,q);
    mpz_sub_ui(qs,q,1);
    mpz_gcd(gcd,e,qs);
    while(mpz_cmp_ui(gcd,1) != 0) {
        mpz_nextprime(q,q);
        mpz_sub_ui(qs,q,1);
        mpz_gcd(gcd,e,qs);
    }

    /* compute (p-1)(q-1) */
    mpz_mul(mod,ps,qs);

    mpz_mul(n, p, q);
```

```

    /* find private key from public key */
    mpz_invert(d,e,mod);

    /* write key to file */
    mpz_out_str(pub,16,e);
    putc('\n',pub);
    mpz_out_str(pub,16,n);

    mpz_out_str(prv,16,d);
    putc('\n',prv);
    mpz_out_str(prv,16,n);

    mpz_init_set(*ep,e);
    mpz_init_set(*dp,d);
    mpz_init_set(*np,n);

    /* free large integers */
    mpz_clear(q);
    mpz_clear(qs);
    mpz_clear(p);
    mpz_clear(ps);
    mpz_clear(e);
    mpz_clear(d);
    mpz_clear(n);
    mpz_clear(u_bound);
    mpz_clear(l_bound);
    mpz_clear(gcd);
    mpz_clear(mod);

}

void rsa_encrypt(FILE *in, FILE *out, mpz_t *e, mpz_t *n) {

    int             i, j, diff;
    int             size_n = mpz_sizeinbase(*n,16);
    int             out_size = (int)ceil((double)size_n / 2);
    long int        file_size;
    unsigned char   input_block[RSA_BLOCK_SIZE], output_block[out_size];
    char            *buffer,*buffer_p, *hex, *str,*str_p, out_buf[3];
    mpz_t           block, cipher;

    /* make size of n even */
    if(size_n % 2 == 1)
        size_n += 1;

```

```
    /* get input file size */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    /* make file size big endian and write to file*/
    if(is_big_endian() != 0)
        SWAP(file_size);

    fwrite(&file_size,sizeof(long int),1,out);

    /* allocate memory for variables */
    buffer = malloc(sizeof(char) * size_n);
    str = malloc(sizeof(char) * size_n);
    hex = malloc(sizeof(char) * 4);

    buffer_p = buffer;
    str_p = str;

    /* initialise integers */
    mpz_init(block);
    mpz_init(cipher);

    while(!feof(in)) {

        buffer = buffer_p;
        str = str_p;
        *buffer = ’\0’;
        *str=’\0’;

        /* read next block */
        memset(input_block,0,RSA_BLOCK_SIZE);
        fread(input_block,sizeof(char),RSA_BLOCK_SIZE,in);

        /* convert block to hexadecimal */
        for(i=0;i<RSA_BLOCK_SIZE;i++) {
            sprintf(hex,”%02x”,input_block[i]);
            buffer = strcat(buffer,hex);
        }

        /* convert string to integer and encrypt */
        mpz_set_str(block,buffer,16);
        mpz_powm(cipher,block,*e,*n);

        buffer = mpz_get_str(NULL,16,cipher);

```

```
        out_buf[2] = '\0';
        diff = size_n - mpz_sizeinbase(cipher,16);

        /* pad output string if necessary */
        if(diff > 0) {
            for(i=0;i<diff;i++)
                str = strcat(str,"0");
            buffer = strcat(str,buffer);
        }

        /* convert hexadecimal number to chars */
        for(j=0;j<out_size; j++) {
            out_buf[0] = *buffer;
            buffer++;
            out_buf[1] = *buffer;
            buffer++;

            output_block[j] = (char)strtol(out_buf,NULL,16);
        }

        /* write encrypted block to file */
        fwrite(output_block,sizeof(char),out_size,out);
    }

    /* free large integers */
    mpz_clear(block);
    mpz_clear(cipher);

}

void rsa_decrypt(FILE *in, FILE *out, mpz_t *d, mpz_t *n) {

    int            i, j, diff, orig_file_size;
    int            file_size, write_counter=0, counter=0;
    int            size_n = mpz_sizeinbase(*n,16);
    int            in_size = (int)ceil((double)size_n / 2);
    unsigned char  input_block[in_size], output_block[RSA_BLOCK_SIZE];
    char           *buffer, *buffer_p, *hex, *str, *str_p, out_buf[3];
    mpz_t          block, cipher;

    /* make size of n even */
    if(size_n % 2 == 1)
        size_n += 1;

    /* allocate memory for variables */
    buffer = malloc(sizeof(char) * size_n);
```

```
    str = malloc(sizeof(char) * size_n);
    hex = malloc(sizeof(char) * 4);

    buffer_p = buffer;
    str_p = str;

    /* read original file size */
    fread(&orig_file_size,sizeof(int),1,in);

    if(is_big_endian() != 0)
       SWAP(orig_file_size);

    /* get size of input file */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,sizeof(int),SEEK_SET);
    file_size -= sizeof(long int);

    /* initialise big ints */
    mpz_init(block);
    mpz_init(cipher);

    while(counter < file_size) {

        /* read next input block */
        memset(input_block,0,in_size);
        fread(input_block,sizeof(char),in_size,in);

        buffer = buffer_p;
        str = str_p;
        *buffer='\0';
        *str='\0';

        /* convert block to hexadecimal */
        for(i=0;i<in_size;i++) {
            sprintf(hex,”%02x”,input_block[i]);
            buffer = strcat(buffer,hex);
        }

        /* convert string to integer and decrypt */
        mpz_set_str(cipher,buffer,16);
        mpz_powm(block,cipher,*d,*n);
        buffer = mpz_get_str(NULL,16,block);

        out_buf[2] = '\0';
        diff = (RSA_BLOCK_SIZE*2) - mpz_sizeinbase(block,16);
```

```

        /* pad decrypted string if necessary */
        if(diff > 0) {
            for(i=0;i<diff;i++)
                str = strcat(str,"0");
            buffer = strcat(str,buffer);
        }

        memset(output_block,0,RSA_BLOCK_SIZE);

        /* convert hexadecimal number to chars */
        for(j=0;j<RSA_BLOCK_SIZE;j++) {
            out_buf[0] = *buffer;
            buffer++;
            out_buf[1] = *buffer;
            buffer++;
            output_block[j] = (char)strtol(out_buf,NULL,16);
        }

        /* write decrypted block to output file */
        if(write_counter < (orig_file_size-RSA_BLOCK_SIZE))
            fwrite(output_block,sizeof(char),RSA_BLOCK_SIZE,out);
        else
            fwrite(output_block,sizeof(char),(orig_file_size-write_counter),out);

        write_counter += RSA_BLOCK_SIZE;
        counter += in_size;
    }

    /* free big ints */
    mpz_clear(block);
    mpz_clear(cipher);

}

void read_key(FILE *key, mpz_t *k, mpz_t *n) {

    int ch, i=0;
    char input_num[257];

    /* read in encryption/decryption exponent */
    ch = getc(key);
    while(ch != '\n') {
        input_num[i] = ch;
        i++;
        ch = getc(key);
```

```
    }
    input_num[i] = '\0';

    mpz_init(*k);
    mpz_set_str(*k,input_num,16);

    /* read in modulus */
    i=0;
    memset(input_num,0,257);
    while((ch = getc(key)) != EOF) {
        input_num[i] = ch;
        i++;
    }
    input_num[i] = '\0';
    mpz_init(*n);
    mpz_set_str(*n,input_num,16);

}

/*#define TEST*/
#ifdef TEST
int main(int argc, char **argv) {

    mpz_t e, d, n, cipher, block;
    FILE *in = NULL, *out = NULL, *key = NULL;

    /* check input arguments */
    if(argc == 5) {
        in = fopen(argv[2],"rb");
        if(in == NULL) {
            printf("Error opening file %s\n", argv[2]);
            return 1;
        }

        out = fopen(argv[3],"wb");
        if(out == NULL) {
            printf("Error opening file %s\n", argv[3]);
            return 1;
        }

        key = fopen(argv[4],"rb");
        if(key == NULL) {
            printf("Error opening key file %s\n", argv[4]);
            return 1;
        }

```

```
        if((strcmp(argv[1],"-e")) == 0) {
            printf("Encrypting file\n");
            read_key(key,&e,&n);
            rsa_encrypt(in, out, &e, &n);
        } else if((strcmp(argv[1],"-d")) == 0) {
            printf("Decrypting file\n");
            read_key(key,&d,&n);
            rsa_decrypt(in, out, &d, &n);
        }

    } else if(argc == 2 && (strcmp(argv[1],"-g") == 0)) {
        generate_key_pair(&e,&d,&n);
        printf("e: limbs=%i",mpz_size(e));
        mpz_out_str(NULL,16,e);
        printf("\n");
        printf("d: limbs=%i ",mpz_size(d));
        mpz_out_str(NULL,16,d);
        printf("\n");
        printf("n: limbs=%i length=%i ",mpz_size(n),mpz_sizeinbase(n,16));
        mpz_out_str(NULL,16,n);
        printf("\n");

    } else {
        printf("Usage: [-e|-d] <infile> <outfile> <keyfile>\n");
        return 1;
    }

    return 0;

}
#endif
```

C.6 cw_aont_pt.c

This is the source code for the symmetric Chaffing and Winnowing Package Transform scheme that was produced.

```
/*
 *  cw_aont_pt.c
 *
 *  Created by John Larkin.
 *  Chaffing and Winnowing implementation
 *  using the "package transform"
 *  pre-processing method.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "hmac.h"
#include "aont.h"
#include "cw_lib.h"

#define BLOCK_SIZE       128
#define CHAFF_BLOCKS     128
#define MIN_BLOCKS       128

size_t chaff_length = DIGEST_LENGTH + BLOCK_SIZE;

void usage(void) {
    printf("Usage: cw [-c|-w] <input file> \
            <output file> <passkey>\n");
}

void addchaff(FILE *in, FILE *out, char *passphrase) {

    unsigned char       checksum[DIGEST_LENGTH];
    unsigned char       input_block[BLOCK_SIZE];
    unsigned char       chaff_packet[BLOCK_SIZE+DIGEST_LENGTH];
    unsigned long int   chaff_positions[CHAFF_BLOCKS];
    int                 j;
    int                 counter=0;
    int                 chaff_pointer=0;
    int                 total_blocks;
```

```
    long int            file_size;
    FILE                *tmp;

    /* create temporary file */
    tmp = tmpfile();

    /* apply AONT to input file */
    transform(in, tmp, NULL, MIN_BLOCKS);

    /* calculate size of transformed file */
    fseek(tmp,0,SEEK_END);
    file_size = ftell(tmp);
    fseek(tmp,0,SEEK_SET);

    /* create random subset of chaff positions */
    set_chaff_positions(chaff_positions,CHAFF_BLOCKS,(file_size/BLOCK_SIZE));
    total_blocks = CHAFF_BLOCKS + (file_size/BLOCK_SIZE);

    /* setup hmac with the supplied passphrase */
    hmac_init((unsigned char*)passphrase,strlen(passphrase));

    for(j=0; j<total_blocks; j++) {

        if(j == chaff_positions[chaff_pointer]) {

            /* create chaff block and write to output file */
            generate_chaff_packet(chaff_packet,BLOCK_SIZE+DIGEST_LENGTH);
            chaff_pointer += 1;

            fwrite(chaff_packet,sizeof(char),chaff_length,out);

        } else {

            /* zero input array and read next block */
            memset(input_block,0,BLOCK_SIZE);
            fread(input_block,sizeof(char),BLOCK_SIZE,tmp);

            /* compute the hmac of the input block */
            hmac(input_block,BLOCK_SIZE,checksum);

            /* write next block of transformed file to output */
            fwrite(input_block,sizeof(char),BLOCK_SIZE,out);

            /* write MAC of current block to output */
            fwrite(checksum,sizeof(char),DIGEST_LENGTH,out);
        }
```

```

        counter++;
    } /* end of loop */

} /* end of addchaff function */

void winnow(FILE *in, FILE *out, char *passphrase) {

    int             counter=0;
    unsigned char   checksum[DIGEST_LENGTH];
    unsigned char   input_checksum[DIGEST_LENGTH];
    unsigned char   input_block[BLOCK_SIZE];
    long int        file_size;
    FILE            *tmp;

    /* get size of input file */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    /* create temp file */
    tmp = tmpfile();

    /* initialise hmac */
    hmac_init((unsigned char*)passphrase,strlen(passphrase));

    while(((counter*(BLOCK_SIZE+DIGEST_LENGTH))+1) <= file_size) {

        /* zero input array and read next block */
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),BLOCK_SIZE,in);

        /* zero MAC array and read next MAC */
        memset(input_checksum,0,DIGEST_LENGTH);
        fread(input_checksum,sizeof(char),DIGEST_LENGTH,in);

        /* compute the hmac of the input block */
        hmac(input_block,BLOCK_SIZE,checksum);

        /* check if block is valid */
        if((memcmp(checksum,input_checksum,DIGEST_LENGTH) == 0)) {
            fwrite(input_block,sizeof(char),BLOCK_SIZE,tmp);
        }

        counter++;
    } /* end of loop */
```

```

    /* find start of temp file and invert transformation */
    fseek(tmp,0,SEEK_SET);
    inverse_transform(tmp,out);

} /* end of winnow function */

int main(int argc, char **argv)
{
    char *passphrase;
    FILE *in = NULL, *out = NULL;

    /* if Windows or Mac, initialise random functions */
    #ifdef WIN32
        srand(time(NULL));
    #endif
    #ifdef sun
        srandom(time(NULL));
    #endif

    /* check input arguments */
    if(argc == 5) {
        in = fopen(argv[2],"rb");
        if(in == NULL) {
            printf("Error opening file %s\n", argv[2]);
            exit(1);
        }

        out = fopen(argv[3],"wb");
        if(out == NULL) {
            printf("Error opening file %s\n", argv[3]);
            return 1;
        }

        passphrase = argv[4];

        if((strcmp(argv[1],"-c")) == 0) {
            addchaff(in, out, passphrase);
        } else if((strcmp(argv[1],"-w")) == 0) {
            winnow(in, out, passphrase);
        } else {
            usage();
            return 1;
        }
    } else {
        usage();
```

```
        exit(1);
    }

    return 0;
}
```

C.7 cw_aont_oaep.c

This is the source code for the symmetric Chaffing and Winnowing OAEP scheme that was produced.

```c
/*
 *  cw_aont_oaep.c
 *
 *  Created by John Larkin.
 *  Chaffing and Winnowing implementation
 *  using the ”optimal asymmetric encryption
 *  padding” pre-processing method.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include ”hmac.h”
#include ”oaep.h”
#include ”cw_lib.h”

#define BLOCK_SIZE          128
#define CHAFF_BLOCKS        128
#define MIN_BLOCKS          128

size_t chaff_length = DIGEST_LENGTH + BLOCK_SIZE;

void usage(void) {
    printf(”Usage: cw [-c|-w] <input file> \
                <output file> <passkey>\n”);
}

void addchaff(FILE *in, FILE *out, char *passphrase) {

    unsigned char        checksum[DIGEST_LENGTH];
    unsigned char        input_block[BLOCK_SIZE];
    unsigned char        chaff_packet[BLOCK_SIZE+DIGEST_LENGTH];
    unsigned long int    chaff_positions[CHAFF_BLOCKS];
    int                  j;
    int                  counter=0;
    int                  chaff_pointer=0;
    int                  total_blocks;
```

```
    long int            file_size;
    FILE                *tmp;

    /* create temporary file */
    tmp = tmpfile();

    /* apply AONT to input file */
    transform(in, tmp, MIN_BLOCKS);

    /* calculate size of transformed file */
    fseek(tmp,0,SEEK_END);
    file_size = ftell(tmp);
    fseek(tmp,0,SEEK_SET);

    /* create random subset of chaff positions */
    set_chaff_positions(chaff_positions,CHAFF_BLOCKS,(file_size/BLOCK_SIZE));
    total_blocks = CHAFF_BLOCKS + (int)ceil((double)file_size/BLOCK_SIZE);

    /* setup hmac with the supplied passphrase */
    hmac_init((unsigned char*)passphrase,strlen(passphrase));

    for(j=0; j<total_blocks; j++) {

        if(j == chaff_positions[chaff_pointer]) {

            /* create chaff block and write to output file */
            generate_chaff_packet(chaff_packet,BLOCK_SIZE+DIGEST_LENGTH);
            chaff_pointer += 1;

            fwrite(chaff_packet,sizeof(char),chaff_length,out);

        } else {

            /* zero input array and read next block */
            memset(input_block,0,BLOCK_SIZE);
            fread(input_block,sizeof(char),BLOCK_SIZE,tmp);

            /* compute the hmac of the input block */
            hmac(input_block,BLOCK_SIZE,checksum);

            /* write next block of transformed file to output */
            fwrite(input_block,sizeof(char),BLOCK_SIZE,out);

            /* write MAC of current block to output */
            fwrite(checksum,sizeof(char),DIGEST_LENGTH,out);
        }
```

```

        counter++;
    } /* end of loop */

} /* end of addchaff function */

void winnow(FILE *in, FILE *out, char *passphrase) {

    int             counter=0;
    unsigned char   checksum[DIGEST_LENGTH];
    unsigned char   input_checksum[DIGEST_LENGTH];
    unsigned char   input_block[BLOCK_SIZE];
    long int        file_size;
    FILE            *tmp;

    /* create temporary file */
    tmp = tmpfile();

    /* calculate size of input file */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,0,SEEK_SET);

    /* setup hmac */
    hmac_init((unsigned char*)passphrase,strlen(passphrase));

    while(((counter*(BLOCK_SIZE+DIGEST_LENGTH))+1) <= file_size) {

        /* zero input array and read next block */
        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),BLOCK_SIZE,in);

        /* zero MAC array and read next MAC */
        memset(input_checksum,0,DIGEST_LENGTH);
        fread(input_checksum,sizeof(char),DIGEST_LENGTH,in);

        /* compute the hmac of the input block */
        hmac(input_block,BLOCK_SIZE,checksum);

        /* check if block is valid */
        if((memcmp(checksum,input_checksum,DIGEST_LENGTH) == 0)) {
            fwrite(input_block,sizeof(char),BLOCK_SIZE,tmp);
        }

        counter++;

```

```
    } /* end of loop */

    /* find start of temp file and invert transformation */
    fseek(tmp,0,SEEK_SET);
    inverse_transform(tmp,out);

} /* end of winnow function */

int main(int argc, char **argv) {

    char *passphrase;
    FILE *in = NULL, *out = NULL;

    /* if Windows or Mac, initialise random functions */
    #ifdef WIN32
        srand(time(NULL));
    #endif
    #ifdef sun
        srandom(time(NULL));
    #endif

    /* check input arguments */
    if(argc == 5) {
        in = fopen(argv[2],”rb”);
        if(in == NULL) {
            printf(”Error opening file %s\n”, argv[2]);
            return 1;
        }

        out = fopen(argv[3],”wb”);
        if(out == NULL) {
            printf(”Error opening file %s\n”, argv[3]);
            return 1;
        }

        passphrase = argv[4];

        if((strcmp(argv[1],”−c”)) == 0) {
            addchaff(in, out, passphrase);
        } else if((strcmp(argv[1],”–w”)) == 0) {
            winnow(in, out, passphrase);
        } else {
            usage();
            return 1;
        }
    } else {
```

```
        usage();
        return 1;
    }

    return 0;
}
```

C.8 cw_pk_pt.c

This is the source code for the hybrid Chaffing and Winnowing Package Transform scheme that was produced.

```
/*
 *  cw_pk_pt.c
 *
 *
 *  Created by John Larkin.
 *  Chaffing and Winnowing implementation
 *  using the ”package transform” pre-processing method
 *  and RSA to hide the chaff packet positions
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <gmp.h>
#include ”rsa.h”
#include ”aont.h”
#include ”cw_lib.h”

#define BLOCK_SIZE      128
#define CHAFF_BLOCKS    128
#define MIN_BLOCKS      128

void usage(void) {
    printf(”Usage: cw [-c|-w] <input file> <output file> <keyfile>\n”);
}

void addchaff(FILE *in, FILE *out, mpz_t *e, mpz_t *n) {

    unsigned char       input_block[BLOCK_SIZE];
    unsigned char       chaff_packet[BLOCK_SIZE];
    unsigned long int   chaff_positions[CHAFF_BLOCKS];
    unsigned long int   swapped_chaff_positions[CHAFF_BLOCKS];
    int                 i, j, counter=0, chaff_pointer=0;
    long int            file_size, total_blocks;
    FILE                *tmp, *tmp_aont,*tmp_rsa;

    /* create temp files */
```

```
    tmp = tmpfile();
    tmp_aont = tmpfile();
    tmp_rsa = tmpfile();

    /* apply AONT to input file */
    transform(in, tmp, NULL, MIN_BLOCKS);

    /* get size of transformed file */
    fseek(tmp,0,SEEK_END);
    file_size = ftell(tmp);
    fseek(tmp,0,SEEK_SET);

    /* create chaff packet indices */
    set_chaff_positions(chaff_positions,CHAFF_BLOCKS,(file_size/BLOCK_SIZE));

    /* byte swapping required here on position indices if not big-endian machine
       then write them to a temporary file */
    if(is_big_endian() != 0) {
        for(i=0;i<CHAFF_BLOCKS;i++) {
            swapped_chaff_positions[i] = chaff_positions[i];
            SWAP(swapped_chaff_positions[i]);
        }
        fwrite(swapped_chaff_positions,sizeof(long int),CHAFF_BLOCKS,tmp_aont);
    } else {
        fwrite(chaff_positions,sizeof(long int),CHAFF_BLOCKS,tmp_aont);
    }

    fseek(tmp_aont,0,SEEK_SET);

    /* apply AONT to chaff position indices */
    transform(tmp_aont,tmp_rsa,NULL,0);
    fclose(tmp_aont);

    /* encrypt chaff position indices with RSA */
    fseek(tmp_rsa,0,SEEK_SET);
    rsa_encrypt(tmp_rsa,out,e,n);

    total_blocks = CHAFF_BLOCKS + (file_size/BLOCK_SIZE);

    for(j=0; j<total_blocks; j++) {

        /* if current packet is a chaff packet, generate one
           and write it to file. Otherwise write valid packet */
        if(j == chaff_positions[chaff_pointer]) {

            generate_chaff_packet(chaff_packet,BLOCK_SIZE);
```

```
            chaff_pointer += 1;

            fwrite(chaff_packet,sizeof(char),BLOCK_SIZE,out);

        } else {
            /* read next input block */
            memset(input_block,0,BLOCK_SIZE);
            fread(input_block,sizeof(char),BLOCK_SIZE,tmp);

            /* write valid block to output file */
            fwrite(input_block,sizeof(char),BLOCK_SIZE,out);

        }

        counter++;

    }
}

void winnow(FILE *in, FILE *out, mpz_t *d, mpz_t *n) {

    int               i, counter=0,chaff_pointer=0;
    int               chaff_position_size = ((BLOCK_SIZE*sizeof(long int))
                                                                   +BLOCK_SIZE);
    int               chaff_positions_length =
                          (((int)ceil((double)chaff_position_size / RSA_BLOCK_SIZE) *
                          ((int)ceil((double)mpz_sizeinbase(*n,16)/2)))+sizeof(int));
    unsigned long int chaff_positions[CHAFF_BLOCKS];
    unsigned long int chaff_positions_block[chaff_positions_length];
    unsigned char     input_block[BLOCK_SIZE], aont_file[chaff_position_size];
    long int          file_size;
    FILE              *tmp, *tmp_rsa_in, *tmp_rsa_out, *tmp_aont_in, *tmp_aont_out;

    /* create up temporary files */
    tmp = tmpfile();
    tmp_rsa_in = tmpfile();
    tmp_rsa_out = tmpfile();
    tmp_aont_in = tmpfile();
    tmp_aont_out = tmpfile();

    /* copy encrypted chaff positions and to a temp file */
    fread(chaff_positions_block,sizeof(char),chaff_positions_length,in);
    fwrite(chaff_positions_block,sizeof(char),chaff_positions_length,tmp_rsa_in);

    /* decrypt chaff positions */
    fseek(tmp_rsa_in,0,SEEK_SET);
```

```
    rsa_decrypt(tmp_rsa_in,tmp_rsa_out,d,n);

    /* apply inverse AONT to chaff positions */
    fseek(tmp_rsa_out,0,SEEK_SET);
    fread(aont_file,sizeof(char),chaff_position_size,tmp_rsa_out);
    fwrite(aont_file,sizeof(char),chaff_position_size,tmp_aont_in);
    fseek(tmp_aont_in,0,SEEK_SET);

    inverse_transform(tmp_aont_in,tmp_aont_out);
    fseek(tmp_aont_out,0,SEEK_SET);

    /* chaff positions byte swapping needed here, if not big endian */
    fread(chaff_positions,sizeof(unsigned long int),CHAFF_BLOCKS,tmp_aont_out);
    if(is_big_endian() != 0) {
        for(i=0;i<CHAFF_BLOCKS;i++)
            SWAP(chaff_positions[i]);
    }

    /* get size of input file */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,chaff_positions_length,SEEK_SET);

    /* read file in blocks, if block has a valid index, write to temp file
        otherwise discard the block */
    while(((counter*(BLOCK_SIZE))+1) <= (file_size-chaff_positions_length)) {

        memset(input_block,0,BLOCK_SIZE);
        fread(input_block,sizeof(char),BLOCK_SIZE,in);

        if(counter == chaff_positions[chaff_pointer]) {
            chaff_pointer += 1;
        } else {
            fwrite(input_block,sizeof(char),BLOCK_SIZE,tmp);
        }

        counter++;
    }

    /* apply inverse AONT to temp file */
    fseek(tmp,0,SEEK_SET);
    inverse_transform(tmp,out);

}

int main(int argc, char **argv)
```

```
{
    FILE *in = NULL, *out = NULL, *key=NULL;
    mpz_t e, d, n;

    /* if Windows or Mac initialise random functions */
    #ifdef WIN32
        srand(time(NULL));
    #endif
    #ifdef sun
        srandom(time(NULL));
    #endif

    /* check input arguments */
    if(argc == 5) {
        in = fopen(argv[2],”rb”);
        if(in == NULL) {
            printf(”Error opening file %s\n”, argv[2]);
            return 1;
        }

        out = fopen(argv[3],”wb”);
        if(out == NULL) {
            printf(”Error opening file %s\n”, argv[3]);
            return 1;
        }

        key = fopen(argv[4],”rb”);
        if(key == NULL) {
            printf(”Error opening key file %s\n”, argv[4]);
            return 1;
        }

        if((strcmp(argv[1],”−c”)) == 0) {
            read_key(key,&e,&n);
            addchaff(in, out, &e, &n);
            mpz_clear(e);
            mpz_clear(n);
        } else if((strcmp(argv[1],”−w”)) == 0) {
            read_key(key,&d,&n);
            winnow(in, out, &d, &n);
            mpz_clear(d);
            mpz_clear(n);
        } else {
            usage();
            return 1;
        }
```

```
    } else if(argc == 2 && (strcmp(argv[1],"-g") == 0)) {
        generate_key_pair(&e,&d,&n);
    } else {
        usage();
        return 1;
    }

    return 0;
}
```

C.9 cw_pk_oaep.c

This is the source code for the hybrid Chaffing and Winnowing OAEP scheme that was produced.

```
/*
 *  cw_pk_oaep.c
 *
 *
 *  Created by John Larkin.
 *  Chaffing and Winnowing implementation
 *  using the "oaep" pre-processing method
 *  and RSA to encrypt the chaff packet positions
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <gmp.h>
#include "rsa.h"
#include "oaep.h"
#include "cw_lib.h"

#define BLOCK_SIZE      128
#define CHAFF_BLOCKS    128
#define MIN_BLOCKS      128

void usage(void) {
    printf("Usage: cw [-c|-w] <input file> <output file> <keyfile>\n");
}


void addchaff(FILE *in, FILE *out, mpz_t *e, mpz_t *n) {

    unsigned char       input_block[BLOCK_SIZE];
    unsigned char       chaff_packet[BLOCK_SIZE];
    unsigned long int   chaff_positions[CHAFF_BLOCKS];
    unsigned long int   swapped_chaff_positions[CHAFF_BLOCKS];
    int                 i, j, counter=0, chaff_pointer=0;
    long int            file_size, total_blocks;
    FILE                *tmp, *tmp_aont,*tmp_rsa;

```

```
    /* create temp files */
    tmp = tmpfile();
    tmp_aont = tmpfile();
    tmp_rsa = tmpfile();

    /* apply AONT to input file */
    transform(in, tmp, MIN_BLOCKS);

    /* get size of transformed file */
    fseek(tmp,0,SEEK_END);
    file_size = ftell(tmp);
    fseek(tmp,0,SEEK_SET);

    /* create chaff packet indices */
    set_chaff_positions(chaff_positions,CHAFF_BLOCKS,(file_size/BLOCK_SIZE));

    /* byte swapping required here on position indices if not big-endian machine
       then write them to a temporary file */
    if(is_big_endian() != 0) {
       for(i=0;i<CHAFF_BLOCKS;i++) {
          swapped_chaff_positions[i] = chaff_positions[i];
          SWAP(swapped_chaff_positions[i]);
       }
       fwrite(swapped_chaff_positions,sizeof(long int),CHAFF_BLOCKS,tmp_aont);
    } else {
       fwrite(chaff_positions,sizeof(long int),CHAFF_BLOCKS,tmp_aont);
    }


    fseek(tmp_aont,0,SEEK_SET);

    /* apply AONT to chaff position indices */
    transform(tmp_aont,tmp_rsa,0);
    fclose(tmp_aont);

    /* encrypt chaff position indices with RSA */
    fseek(tmp_rsa,0,SEEK_SET);
    rsa_encrypt(tmp_rsa,out,e,n);

    total_blocks = CHAFF_BLOCKS + (file_size/BLOCK_SIZE);

    for(j=0; j<total_blocks; j++) {

       /* if current packet is a chaff packet, generate one
          and write it to file. Otherwise write valid packet */
       if(j == chaff_positions[chaff_pointer]) {
```

```

            generate_chaff_packet(chaff_packet,BLOCK_SIZE);
            chaff_pointer += 1;

            fwrite(chaff_packet,sizeof(char),BLOCK_SIZE,out);

        } else {
            /* read next input block */
            memset(input_block,0,BLOCK_SIZE);
            fread(input_block,sizeof(char),BLOCK_SIZE,tmp);

            /* write valid block to output file */
            fwrite(input_block,sizeof(char),BLOCK_SIZE,out);

        }

        counter++;

    }

}

void winnow(FILE *in, FILE *out, mpz_t *d, mpz_t *n) {

    int                 i, counter=0,chaff_pointer=0;
    int                 chaff_position_size = ((BLOCK_SIZE*sizeof(long int))
                                                                    +BLOCK_SIZE);
    int                 chaff_positions_length =
                            (((int)ceil((double)chaff_position_size / RSA_BLOCK_SIZE) *
                            ((int)ceil((double)mpz_sizeinbase(*n,16)/2)))+sizeof(int));
    unsigned long int   chaff_positions[CHAFF_BLOCKS];
    unsigned long int   chaff_positions_block[chaff_positions_length];
    unsigned char       input_block[BLOCK_SIZE], aont_file[chaff_position_size];
    long int            file_size;
    FILE                *tmp, *tmp_rsa_in, *tmp_rsa_out, *tmp_aont_in, *tmp_aont_out;

    /* create up temporary files */
    tmp = tmpfile();
    tmp_rsa_in = tmpfile();
    tmp_rsa_out = tmpfile();
    tmp_aont_in = tmpfile();
    tmp_aont_out = tmpfile();

    /* copy encrypted chaff positions and to a temp file */
    fread(chaff_positions_block,sizeof(char),chaff_positions_length,in);
    fwrite(chaff_positions_block,sizeof(char),chaff_positions_length,tmp_rsa_in);
```

```

    /* decrypt chaff positions */
    fseek(tmp_rsa_in,0,SEEK_SET);
    rsa_decrypt(tmp_rsa_in,tmp_rsa_out,d,n);

    /* apply inverse AONT to chaff positions */
    fseek(tmp_rsa_out,0,SEEK_SET);
    fread(aont_file,sizeof(char),chaff_position_size,tmp_rsa_out);
    fwrite(aont_file,sizeof(char),chaff_position_size,tmp_aont_in);
    fseek(tmp_aont_in,0,SEEK_SET);

    inverse_transform(tmp_aont_in,tmp_aont_out);
    fseek(tmp_aont_out,0,SEEK_SET);

    /* chaff positions byte swapping needed here, if not big endian */
    fread(chaff_positions,sizeof(unsigned long int),CHAFF_BLOCKS,tmp_aont_out);
    if(is_big_endian() != 0) {
       for(i=0;i<CHAFF_BLOCKS;i++)
          SWAP(chaff_positions[i]);
    }

    /* get size of input file */
    fseek(in,0,SEEK_END);
    file_size = ftell(in);
    fseek(in,chaff_positions_length,SEEK_SET);

    /* read file in blocks, if block has a valid index, write to temp file
       otherwise discard the block */
    while(((counter*(BLOCK_SIZE))+1) <= (file_size-chaff_positions_length)) {

       memset(input_block,0,BLOCK_SIZE);
       fread(input_block,sizeof(char),BLOCK_SIZE,in);

       if(counter == chaff_positions[chaff_pointer]) {
          chaff_pointer += 1;
       } else {
          fwrite(input_block,sizeof(char),BLOCK_SIZE,tmp);
       }

       counter++;
    }

    /* apply inverse AONT to temp file */
    fseek(tmp,0,SEEK_SET);
    inverse_transform(tmp,out);

```

```
}

int main(int argc, char **argv)
{
    FILE *in = NULL, *out = NULL, *key=NULL;
    mpz_t e, d, n;

    /* if Windows or Mac initialise random functions */
    #ifdef WIN32
        srand(time(NULL));
    #endif
    #ifdef sun
        srandom(time(NULL));
    #endif

    /* check input arguments */
    if(argc == 5) {
        in = fopen(argv[2],”rb”);
        if(in == NULL) {
            printf(”Error opening file %s\n”, argv[2]);
            return 1;
        }

        out = fopen(argv[3],”wb”);
        if(out == NULL) {
            printf(”Error opening file %s\n”, argv[3]);
            return 1;
        }

        key = fopen(argv[4],”rb”);
        if(key == NULL) {
            printf(”Error opening key file %s\n”, argv[4]);
            return 1;
        }

        if((strcmp(argv[1],”−c”)) == 0) {
            read_key(key,&e,&n);
            addchaff(in, out, &e, &n);
            mpz_clear(e);
            mpz_clear(n);
        } else if((strcmp(argv[1],”−w”)) == 0) {
            read_key(key,&d,&n);
            winnow(in, out, &d, &n);
            mpz_clear(d);
            mpz_clear(n);
        } else {
```

```
            usage();
            return 1;
        }
    } else if(argc == 2 && (strcmp(argv[1],"-g") == 0)) {
        generate_key_pair(&e,&d,&n);
    } else {
        usage();
        return 1;
    }

    return 0;
}
```