TrueCrypt: Analysis and Implementation

Introduction

TrueCrypt is a software package used for the encryption and obscuring of local files on a system. It incorporates a number of encryption algorithms and features including the encryption and hiding of an operating system. The purpose of this software is to protect sensitive data on the host system from potential threats and misuses by others. The following report provides a brief review and explanation of the various algorithms used by TrueCrypt, specifically its encryption and hash algorithm. This review is followed by a summary of its encryption services and a brief analysis of how the software impacts the system.

Analysis

Encryption

The encryption used in TrueCrypt consists of three different block ciphers. The purpose of these ciphers is to provide confidentiality by 'diffusing' and 'confusing' the plaintext data in the ciphertext. Diffusing is accomplished by spreading the data out across the encrypted block so that what may have originally been bits belonging to a single word may now be several bits spread out across the entire block. Confusing can be regarded as replacing one word/character or bit/byte for another so that the output is confusing and doesn't make sense when looking at normally. TrueCrypt use AES, Twofish and Serpent as its block ciphers, each using a 128 bit key and 128 bit plaintext as input.

Encryption: AES

The Advanced Encryption Standard (or Rijndael) is the current cipher standard set by NIST (the National Institute of Standards and Technology). It uses a 128, 192 or 256 bit key and performs a number of rounds (number of rounds depends on the key length used) on a plaintext input of 128 bits, eventually producing an output ciphertext of 128 bits. With the exception of the first initialization round and the final round, each round is identical to one another and can be broken up into four specific operations.

Byte Substitution:

During the byte substitution step the plaintext is broken down into 16 bytes (8bits x 16 = 128bits) whose values are then used to determine a substitution. The substitute value for the byte is selected from a preexisting data table (see appendix A) using the leftmost 4 bits as the row index for the table and the remaining 4 bits as the column index. Once the value is determined from the table the original plaintext (or indexing) byte is replaced with the listed value.

Row Shifting:

The arrangement of the 128 plaintext bits (and the subsequent Byte Substituted bits) can be regarded as a 4x4 matrix with each element being a byte and the matrix itself being filled row-wise, where the first byte would go in row 1 column 1, the second byte would go in row 1 column 2 and so on. During the

Row Shift operation each row is cyclically shifted left a number of times equal to the rows specific index (i.e. the first/top row is row 0; the second is row 1 etc.).

Column Mixing:

The Column Mixing step can be regarded as a mapping from one set of values to another by means of a linear transformation. The operation itself is a matrix multiplication of a transformation matrix and the byte substituted and row shifted plaintext. The purpose of this step is to further diffuse the plaintext throughout the data block being encrypted. The transformation matrix is given in Appendix A.

Round Key Addition:

For each round a separate sub key is computed from the original primary key. During the Round Key Addition step the output of the previous step is bitwise XORd with the current round key. This corresponds to addition over GF(2).

The initialization and final rounds of Rijndael make use of the previous four steps except that they do not use every step. The initialization round only performs a Round Key Addition while the final round performs every step except Column Mixing before computing the final ciphertext.

Round Key Generation

For each round a separate key is determined for use in the cipher. This key is based on the original key. For a 128 bit key the source key is arranged in a 4x4 matrix with each element of the matrix being a byte of the key. This matrix then has 40 additional columns appended to it with the original four columns having the index (i) of 0, 1, 2 and 3. The new columns are then filled out according to the following algorithm:

For columns with index i not evenly divisible by 4:

$$C(i) = C(i-4) \oplus C(i-1)$$

Where *C* is a column vector with index i and entries $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$

$\oplus$ is the XOR operation

For columns with index i divisible by 4:

$$C(i) = C(i-4) \oplus B(i-1)$$

Where *B(i-1)* is the same as C(i-1) but with it's Byte entries *a*, *b*, *c* and *d* replaced using the same substitution matrix used in the Byte Substitution step, yielding *a'*, *b'*, *c'* and *d'* and the performing the following operation:

$$B(i-1) = \begin{bmatrix} a" \\ b" \\ c" \\ d" \end{bmatrix} = \begin{bmatrix} a^{'} \oplus r(i) \\ b' \\ c' \\ d' \end{bmatrix}$$

$$r(i) = \begin{cases} 00000010_2^{(1-4)/4} \quad , & i < 36 \\ 00000010_2^{(1-4)/4} \oplus 1\ 00011011_2, & i \geq 36 \end{cases}$$

Note that the exponent term in the r(i) function acts as a shift operator, shifting left for values of i greater than 4. Upon realizing this, it is easy to see that an overflow would occur for values of i greater than or equal to 36. For this reason the exclusive or operation is performed, reducing the value over $GF(2^8)$. Note that the value $1\ 00011011_2$corresponds to a generator polynomial of $GF(2^8)$, specifically $x^8 + x^4 + x^3 + x + 1$.

Encryption: Twofish

Much like Rijndael, Twofish is a block cipher that makes use of a key size of 128, 192 or 256 bits and a plaintext of 128 bits. Compared to Rijndael, Twofish is quite complex, but makes use of many similar functions. The basic process of Twofish is given as follows (depicted in figure 1):

1. The plaintext is broken up into four 32 bit words and each is XORd with a 32 bit expanded key (The first word is XORd with K_0, the second word with K_1 and so on).
2. The first word is broken up into 4 bytes, each of which is applied to a substitution box (or S-box, like the lookup table mentioned in AES). The second word is first rotated left by 8 bits and then is also applied to the same set of S-boxes.
3. From here both the first and second words are applied to an MDS matrix (Maximum Distance Separable) which serves to diffuse the newly substituted data of the 32 bit word amongst its 4 bytes.
4. After the MDS matrix multiplication the first word is applied to a pseudo-Hadamard Transform:
$$a^{'} = a + b \; mod \; 2^{32}$$
where *a* is the first word, *b* is the second word and *a'* is the new first word.
Using the 'new' first word as input, the second word is applied to the same transform, which can equivalently be represented as:
$$b^{'} = a + 2b \; mod \; 2^{32}$$
This operation serves to diffuse the two words amongst each other.
5. At this point the first two words are XORd with a round key each.
6. Following this step the third word is XORd with the output of the first word (the new first word or $W_0{'}$) and then rotated right by one bit producing what will be the new third word ($W_2{'}$). At the same time the fourth word is rotated left by one bit and then XORd with the output of the operations on the second word ($W_1{'}$) producing what will be treated as the new fourth word ($W_3{'}$).
7. The next round begins (starting at step 2) with the first and second words (W_0 and W_1) at the beginning of the previous round becoming this rounds third and fourth words while the new first and second words are the output from the previous round ($W_2{'}$ and $W_3^{'}$respectively).

8. Steps 2 through 7 are repeated for a total (including the first) of 16 rounds.
9. The first and second words are swapped with the third and fourth words, effectively undoing the seventh step of the final round.
10. The words are XORd with another set of round keys ($K_4 - K_7$) producing the ciphertext.

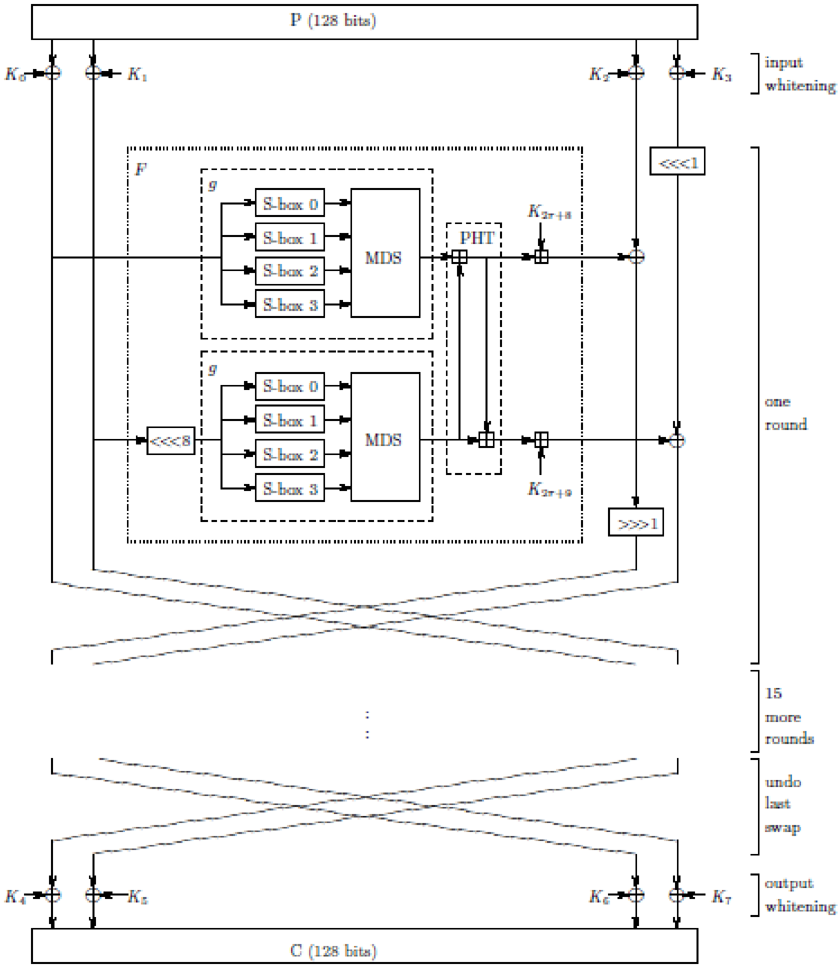


Figure 1) the Twofish Cipher Structure

An important note to make about Twofish however is that the S boxes used in the rounds (while the same 4 S-boxes for each round) are key dependant (dependant on the original source key, not the round keys). This adds an extra layer of security in that the S-boxes are now an unknown quantity to a would be attacker, making it much more difficult to crack a single round without knowledge of the key, since for two unique keys the primary substitution scheme will be different. The generation of the S-boxes (and similarly the round keys) is accomplished through a somewhat complicated process that is not fully explained here (due to its relatively low significance to TrueCrypts operation and the excessive length of the required explanation). Simply put however, the original key is used to generate three 2-word vectors (64 bits total), one of which is used specifically to generate the S-boxes. These three vectors are used in a series of conditional permutations (conditional as in the exact sequence of permutations, depending on the round number) and XORs, followed by a set of matrix multiplications and word operations (specifically rotations and modular additions) to produce the round keys and the four 8x8 S-boxes. For more information please see section 4.3 of [11].

Encryption: Serpent

Serpent is the third and final block cipher used by TrueCrypt to perform its data encryption. Just like Rijndael and Twofish, Serpent accepts keys size of 128, 192 or 256 bits and uses an input block of plaintext 128 bits long. The encryption process involves an initial permutation of the plaintext followed by 31 rounds of operations (described below), a modified 32nd round and then a final permutation (which is the inverse of the initial permutation).

Initial Permutation and Setup:

The initial permutation is simply a remapping of the 128 input bits into new positions. The new positions are specified in the reordered matrix (given in Appendix A) by the bit place or number that they originally occupied. Note that Serpent is specified using the little-endian standard, such that the least significant bit (bit 0) is first (or the leftmost bit). After this, the rest of the algorithm is arranged into four 32 bit words ($W_0, W_1, W_2 \text{ and } W_3$). These words are encrypted in parallel with each other but are not treated entirely independently, as shown below.

Rounds 0-30:

Each round starts with XORing the input with a 128 bit round key. The result is then broken up into groups of 4 bits which are each applied to an S-box (this operation is applied to each of the four bit groups in parallel). The S-box used depends on which round is currently being computed. There are a total of 8 different S-boxes, $S_0 - S_7$, where S_0 is used in rounds $R_0, R_8, R_{16}, \text{and } R_{24},\ S_1 \text{ in rounds}$ $R_1, R_9, R_{17}, \text{and } \mathrm{R}_{25}$ and so on. The S-Boxes are accomplished through logical operations but can be regarded as the mappings provided in Appendix A, where the input value of the 4 bits are used to select the column position which contains the new value to be used as the output. At this point the output is subjected to a series of logical operations defined as follows (in this order):

- $W_0$is rotated left by 13 bits
- $W_2$is rotated left by 3 bits
- $W_1$is replaced with the result of XORing $W_1, W_0 \text{ and } W_2$together
- $W_3$is replaced with the result of XORing W_3, W_2 and a version of W_2that has been shifted left by 3 bit positions.
- $W_1$is rotated left by 1 bit
- $W_3$is rotated left by 7 bits
- W_0 is replaced with the result of XORing $W_1, W_0 \text{ and } W_3$together
- W_2 is replaced with the result of XORing W_3, W_2 and a version of W_1that has been shifted left by 7 bit positions.
- W_0 is rotated left by 5 bits

- W_2 is rotated left by 22 bits

These operations are performed to diffuse the four words amongst each other as well as to maximize the effects of the S-Boxes and round key addition. Changing one bit in the input of an S-Box will cause two output bits to change, which in turn propagates through to the next round.

Round 31:

The final round is treated the same way as the first 31 rounds, except that the set of logical operations is not performed and instead the output of the S-Boxes is XORd against a final round key K_{32}.

Final Permutation:

The final permutation operation is simply the inverse of the initial permutation. The table is given in Appendix A and is indexed in the same manner as the Initial Permutation.

Key Generation:

Serpent uses thirty-three 128 bit round keys, one for each round and an additional key for the final round. The 33 round keys are derived from the original key which is 256 bits. Regardless of the input key size, Serpent always expands it to 256 bits by attaching a 1 to the most significant bit position and zero padding the rest up to the 256th bit (MSB) position. The 256 bit key is then broken up into eight 32 bit words and are indexed as $W_{-8}^k, W_{-7}^k, W_{-6}^k, W_{-5}^k, W_{-4}^k, W_{-3}^k, W_{-2}^k,$ and W_{-1}^k. These are then used to calculate $W_0^k - W_{131}$ as follows:

$$W_i^k = ROL[(W_{i-8}^k \oplus W_{i-5}^k \oplus W_{i-3}^k \oplus W_{i-1}^k \oplus \Phi \oplus p), 11]$$

Where ROL(x,y) specifies to rotate x left by y bits
Φ is the Hex word 9E3779B9
p is the Hex word 000001A5

And the round keys are formed according to:

$$K_i = S_{i+3 mod\, 8}(W_{4i}^k, W_{4i+1}^k, W_{4i+2}^k, W_{4i+3}^k)$$

Where S represents one of the S-Boxes used in the cipher
i is the round index

Mode of Operation: XTS

A mode of operation is a method for applying an encryption standard to a block of data that is equal to or greater than the standard block size used by the encryption cipher. The specific goal of using a mode of operation as opposed to a straight forward implementation of the cipher itself is to help protect against chosen plaintext attacks (CPA) and chosen ciphertext attacks (CCA). As an illustration of these threats let us assume that a system (computer) has its hard drive encrypted as a series of blocks, each encrypted independently of the other blocks by a straight forward implementation of the encryption cipher. A chosen plaintext attack could be performed on system by an attacker entering a set of plaintext blocks to be encrypted and stored on the device and then observing the resultant ciphertext on the drive. With enough such plaintext/ciphertext pairs the attacker would be able obtain the encryption key used by the cipher, and would then be able to access the rest of drive. Similarly, let us assume that the drive itself is not entirely full and consists of several similar blocks. By using the same key and encrypting each block independently of the others the attacker would be able to recognize such blocks simply by virtue of their similar appearance. The attacker could then analyze the system and make an educated guess as to the type of plaintext that was entered. Knowing the encryption algorithm used the attacker could then begin to extract a pattern from the ciphertext/plaintext relationship and begin to recognize it in other similar blocks, eventually obtaining enough ciphertext/plaintext pairs to extract a key from; such an attack is a chosen ciphertext attack. Using a mode of operation counters these attacks by relating the blocks to each other such that the encryption for each is not identical but rather a function of the key and its relation to the rest of the encrypted blocks. The XTS mode of operation accomplishes this in the following way (as described on the TrueCrypt Website):

$$C_i = E_{k1}\left(P_i \oplus \left(E_{k2}(n) \otimes a^i\right)\right) \oplus \left(E_{k2}(n) \otimes a^i\right)$$

Where *C* is the ciphertext
P is the plaintext
E is the encryption algorithm
i is the block index in the data unit (the data unit is the 512 byte block of data being encrypted)
k is the key being used by the encryption cipher (two different keys are used)
n is the data unit within the sector being encrypted
a is the primitive element 2 or x (or ….00010 in binary)
$\oplus$ represents the XOR operation
$\otimes$ represent the multiplication of two polynomials modulo $x^{128} + x^7 + x^2 + x + 1$ over GF(2)

This operation can alternately be represented by the following diagram:

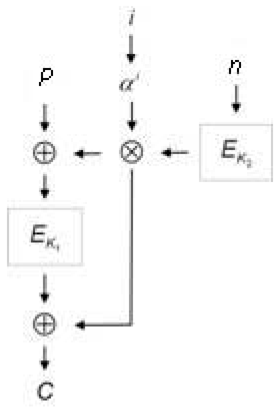


Figure 2) the XTS Operation Structure [9]

By incorporating the data unit (n) and block index (i) as a part of the encryption process XTS successfully differentiates the encryption used for one block from another, producing a unique ciphertext for identical plaintext in two separate blocks. Additionally, the ciphertext of one block does not depend on the ciphertext of another, thus any errors that occur will not propagate from one block to another.

Hash Algorithms

Hash algorithms are generally used to provide data integrity, such that the data being viewed is in fact the data that it is supposed to be and has not been tampered with. Hash algorithms accomplish this by using the data itself to calculate what is referred to as a message digest, such that the message digest uniquely represents the message from which it was derived. Hash algorithms are generally one way functions, in that a message digest cannot be used to determine what the original message was. Given this, a piece of data can be stored along with its message digest and encrypted to provide confidentiality. The data can later be viewed by someone with the appropriate key(s) but without the message digest they have no way of knowing if the data has been tampered with, even if the would be attacker was unable to read the data itself. With the message digest the user can first decrypt the data and then run the data (or message) through the hash algorithm once again to produce another message digest. The two digests can then be compared and if they match the data's integrity has been verified, or likewise disproven if they do not match. TrueCrypt makes use of has algorithms in a different way however. Due to the requirement that hash algorithms provide unique message digests for unique messages, they also serve as good pseudo random number generators. Considering this, along with the fact that even given the output of the hash function the source material remains obscured, hash algorithms are often utilized for key generation. TrueCrypt uses its hash functions in this way, producing keys (and the SALT – see Appendix B). TrueCrypt uses the following hash algorithms:

Hash Algorithm: SHA-512

The first hash algorithm to be considered is SHA-512, which is a variant of the SHA-2 algorithm, producing a large message digest (512 bits) from messages up to a size of $2^{128} - 1$bits. The algorithm first takes the input message and zero-pads (fills any empty bit places with 0s) it such that the message contains $896\ mod 1024$ bits (if the size is already congruent to this 1024 bits of zero-padding are applied anyways). A 128 bit value is then appended to the message indicating the size of the original message prior to zero padding. The message now consists of N 1024bit blocks of data, ($1 \leq N \leq 13$). These message blocks are then passed to the hashing algorithm which is performed sequentially for each 1024 bit block as shown in figure 3, where each output of the previous block operation is used by the next subsequent block operation.

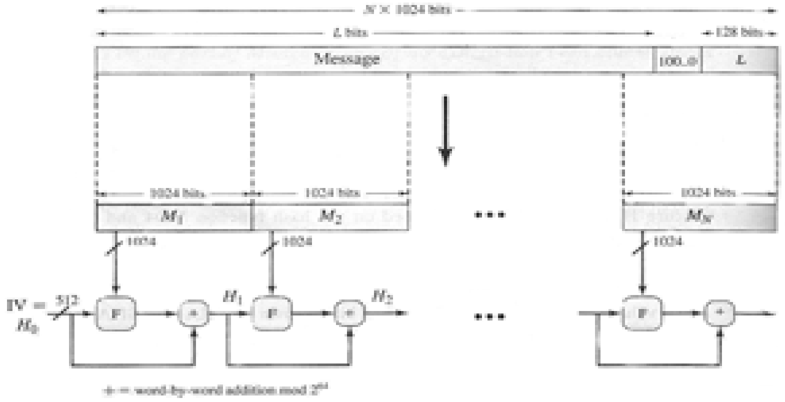


Figure 3) Block Level structure of SHA-512 [6]

In each round the message block is broken up into sixteen 64 bit words ($W_0 - W_{15}$). From these initial words $W_{16} - W_{79}$ are calculated using the following relationship [6]:

$$W_j = \sigma_0(W_{j-2}) + W_{j-7} + \sigma_1(W_{j-15}) + W_{j-16} \quad \text{for } j = 16, 17, 18.....79$$
$$\sigma_0(x) = ROR(x,1) \oplus ROR(x,8) \oplus SHR(x,7)$$
$$\sigma_1(x) = ROR(x,19) \oplus ROR(61,x) \oplus SHR(x,6)$$

Where $\oplus$ is the XOR operation
+ is addition modulo 64 (not simply the XOR operation but addition followed by the modulus operation)
ROR(x,y) is a right rotation of x by y bits
SHR(x,y) is a right shift of x by y bits

The message words are then used in the block function as shown in figure 4) which consists of 80 rounds. At the beginning of each block function the previous block functions 512 bit output is used as the input (for the first round an initialization vector is used). This input is broken up into eight 64 bit words called *a, b, c, d, e, f, g,* and *h*. During each round these words are updated using message word corresponding to the round number, a specific constant for the round K_j (which correspond to the fractions of the cubic roots to each of the first 80 prime numbers) and the following relationship [6]:

$$T_1 = h + [(e\ AND\ f) \oplus (NOT\ e\ AND\ g)] + [ROR(e,14) \oplus ROR(e,18) \oplus ROR(e,41)] + W_j + K_j$$
$$T_2 = [ROR(a,28) \oplus ROR(a,34) \oplus ROR(a,39)] + [(a\ AND\ b) \oplus (a\ AND\ c) \oplus (b\ AND\ c)]$$
$$a = T_1 + T_2$$
$$b = a$$
$$c = b$$
$$d = c$$
$$e = d + T_1$$
$$f = e$$
$$g = f$$
$$h = g$$

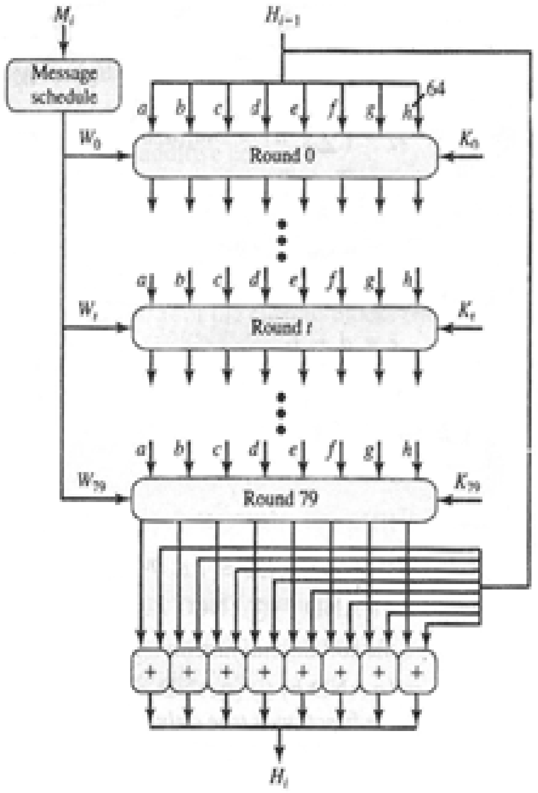


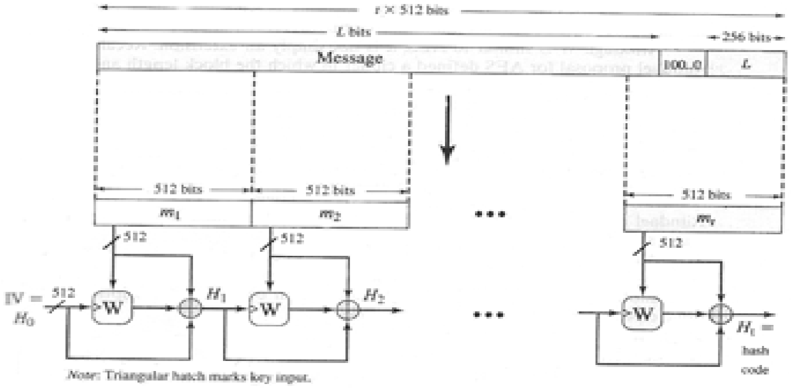
Figure 4) Round Level Structure of SHA-512 (operation performed on each block) [6]

Where AND is the logical 'and' operation
NOT is the logical inverse operation
j is the round index (0-79)

At the end of the 80th round (j=79) the output of the previous block function is added to the new version of *abcdefgh* to produce the output for that block function. Once all the blocks have been computed the final output is the message digest.

<u>Hash Algorithm: Whirlpool</u>

Whirlpool is a hash algorithm that makes use of a block cipher as its core function. The algorithm itself accepts up to $2^{256} - 1$ bits of input and pads the input up to a number of bits such that the result is congruent to $256 \ mod \ 512$. The padding is applied by first attaching a 1 and then whatever number of zeroes are required as well. Following this a value representing the size (stored as a 256 bit word) is attached to the input. From here the message is broken up into blocks of 512 bits to be used as the message inputs for the block cipher (shown in Figure 5) and is arranged as an 8x8 matrix of bytes. The first block round function begin by taking the message input, the output of the previous blocks round functions (H_{i-1} where $H_0$is an empty string, all zeroes) as an initial key (described below) and adding them together. The rest of the rounds are performing as follows:



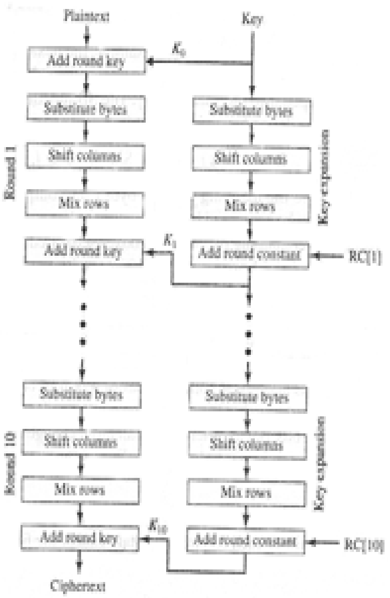


Figure 5) Whirlpool Block Level Structure [6]

Figure 6) Whirlpool Round Structure [6]

Byte Substitution

The byte substitution operates in the same manner as S-box, by using the leftmost four bits of a byte as a row index and the rightmost 4 bits as a column index for a lookup table from which the replacement value is selected. The hardware implementation of this is implemented somewhat differently but the result is the same and corresponds to the S-Box listed in Appendix A.

Column Shifting

The column shifting step is accomplished by rotating the bytes in each column of the 8x8 matrix downward a number of positions based on their column index. For instance the bytes in the first column of the matrix (with index 0) would not be rotated, or rather, are rotated 0 positions while the bytes in the eighth column (index 7) are rotated downwards by 7 spaces. This step serves the purpose of providing diffusion throughout the block, on a byte by byte level.

Row Mixing

Row mixing is accomplished by means of a matrix multiplication. The matrix operation is defined as: (in HEX)

$$RM = M_{CM}\begin{bmatrix} 01 & 01 & 04 & 01 & 08 & 05 & 02 & 09 \\ 09 & 01 & 01 & 04 & 01 & 08 & 05 & 02 \\ 02 & 09 & 01 & 01 & 04 & 01 & 08 & 05 \\ 05 & 02 & 09 & 01 & 01 & 04 & 01 & 08 \\ 08 & 05 & 02 & 09 & 01 & 01 & 04 & 01 \\ 01 & 08 & 05 & 02 & 09 & 01 & 01 & 04 \\ 04 & 01 & 08 & 05 & 02 & 09 & 01 & 01 \\ 01 & 04 & 01 & 08 & 05 & 02 & 09 & 01 \end{bmatrix}$$

Which is performed over $GF(2^8)$ using $x^8 + x^4 + x^3 + x^2 + 1$ (or 11D in Hex) to reduce the values during the multiplicative shifts. This step is used to diffuse the data amongst the rows so that diffusion is achieved in both dimensions and is not simply a reorganization of bytes.

Round Key Addition

At the end of each round the output is XORd with a round key which is generated from the source key for the current block operation, H_{i-1}. The round keys are generated by applying a parallel version of this block cipher, using H_{i-1} as the message input and a round specific constant for its key input (as shown in the right half of figure 6). Each round constant (an 8x8 byte matrix) for the key cipher is formed according to the following relationship:

For row 1:

$$RC_{0,j} = S(8r + j) \qquad \text{for } 0 \le r \le 9, 0 \le j \le 7$$

Where S(x) is the S-Box operation on input byte x
r is the round index (0-9)
j is the column index

For all other rows:

$$RC_{i,j} = 0$$

The output of each round of the key cipher is then used as the round key for the corresponding round in the block (or message) cipher.

After completing the final round the output is XORd with the output of the previous block operation (H_{i-1}) and the input message for the current block. This yields the output of the current block (H_i). This process is continued until each of the N blocks has been treated, resulting in $H_N$which is used as the message digest.

Hash Algorithm: RIPEMD

RIPEMD-160 differs from the previous two hash algorithm in that its message digest is only 160 bits in length as opposed to 512 bits. The message input can be up to $2^{64} - 1$ bits long, and padding is performed so that the message is congruent with $448 \ mod \ 512$. The original length of the message is then appended in a 64 bit word. RIPEMD starts by dividing up its message into groups of 32 bit words. It processes these sixteen of these words in one of its block functions but will only be using ten at a time in two parallel paths (five words in each path). The overall structure of RIPEMD is similar to that of SHA-512 and Whirlpool in that its block function (specifically the two paths for the set of ten words) takes as input the message words and the output of the previous block function (H_{i-1}). The two paths operate on their set of 5 words independently of each other over 5 rounds (each round consisting of 16 steps, for a total of 80 steps). The exact functions performed in each round depends on the step number and which path is being considered, for ease of identification functions and values corresponding to the left path will be denoted with a subscript ‘l’ while those in the right path will have a subscript ‘r’. The functions are as follows:

For $0 \le j \le 15$ $\quad f_{l1}(x,y,z) = x \oplus y \oplus z$

$f_{r1}(x,y,z) = x \oplus \big(y\ OR\ (NOT\ z)\big)$

for $16 \le j \le 31$ $\quad f_{l2}(x,y,z) = (x\ AND\ y) OR\big((NOT\ x) AND\ y\big)$

$f_{r2}(x,y,z) = (x\ AND\ z) OR\big(y\ AND\ (NOT\ z)\big)$

for $32 \le j \le 47$ $\quad f_{l3}(x,y,z) = \ f_{r3}(x,y,z) = \big(x\ OR\ (NOT\ y)\big) \oplus z$

for $48 \le j \le 63$ $\quad f_{l2}(x,y,z) = (x\ AND\ z) OR\big(y\ AND\ (NOT\ z)\big)$

$f_{r2}(x,y,z) = (x\ AND\ y) OR\big((NOT\ x) AND\ y\big)$

for $64 \le j \le 79$ $\quad f_{r1}(x,y,z) = x \oplus \big(y\ OR\ (NOT\ z)\big)$

$f_{l1}(x,y,z) = x \oplus y \oplus z$

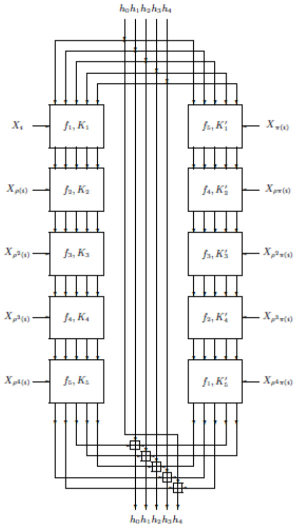


Figure 7) Structure of the two paths [3]

These functions are used in the each step in the following way:

$$T_l = ROL\Big(\big(A_l + f_{lj}\,(B_l, C_l, D_l) + M_i\big(r(j)\big) + K(j)\big), s_l(j)\Big) + E_l$$
$$A_l = E_l$$
$$E_l = D_l$$
$$D_l = ROL(C_l, 10)$$
$$C_l = B_l$$
$$B_l = T_l$$

$$T_r = ROL\Big(\big(A_r + f_{rj}\,(B_r, C_r, D_r) + M_i\big(r(j)\big) + K(j)\big), s_r(j)\Big) + E_r$$
$$A_r = E_r$$
$$E_r = D_r$$
$$D_r = ROL(C_r, 10)$$
$$C_r = B_r$$
$$B_r = T_r$$

Where j is the step index

I is the block index

A, B, C, D, E are the 32 bit words of the current message digest (initialized to the previous block function digest at the start of the current block function)

K* is a specific constant used for that round

M* is a 32 bit message word, indexed according to the path and round (discussed below)

s(x)* is a matrix of values representing a number of bits to rotate by. The index x is the index of the matrix being referenced

+ is addition modulo 2^{32}

*See Appendix A for information regarding these values.

At the end of the 5 rounds the message digest H_i is updated and then passed on to the next block function. The 160 bit digest H_i $(h_0, h_1, h_2, h_3, h_4)$ is updated using the digest from the previous round $h_{p0}, h_{p1}, h_{p2}, h_{p3}, h_{p4}$ (or the initial values for the first block– see Appendix A) as well as the A,B,C,D and E values from the left and right paths:

$$\begin{aligned} h_o &= h_{p1} + C_l + D_r \\ h_1 &= h_{p2} + D_l + E_r \\ h_2 &= h_{p3} + E_l + A_r \\ h_3 &= h_{p4} + A_l + B_r \\ h_4 &= h_{p0} + B_l + C_r \end{aligned}$$

Implementation

Process

TrueCrypt can be used to encrypt data in a number of different ways (providing different features) and using a number of different encryption techniques. As mentioned above AES, Twofish and Serpent are used as the encryption ciphers and are applied using the XTS mode of operation while the hash functions SHA-512, Whirlpool and RIPEMD-160 are used for creating key information. In addition to using one of the three available block ciphers TrueCrypt allows the user to chain the different ciphers together such that the data is encrypted sequentially by some or each of the different ciphers. The various cipher cascades available are Serpent-Twofish-AES, AES-Twofish-Serpent, Twofish-Serpent, AES-Twofish, Serpent-AES, where the first cipher in the cascade name is applied on top of or after the cipher name following it. TrueCrypt offers the following different encryption strategies:

TrueCrypt File

A TrueCrypt file, once completed, acts very similar to a file folder where anything contained within the folder is encrypted. The files themselves are made on the hard drive, in plain site (not hidden at all), and can be moved and relocated like any other file. To access the file it must be mounted as a drive using TrueCrypt, after which the user will be prompted for a password in order to gain access to the file. The TrueCrypt website provides a guide for setting up a TrueCrypt file and the volume creation wizard is very self explanatory (for all types of volumes) so as opposed to giving a implementation guide for each volume type a few notes are made on the finer points of the creation process instead. Partway through the file creation process the user will be prompted to choose the encryption method (a brief analysis of which concludes this report) and the hash algorithm to use for key generation. The choice of encryption algorithm should be based on the need for security versus speed. While each of the block ciphers used is still currently secure each offers various degrees of security and speed/system strain. After selection the cipher and hash algorithm to use the user will be brought to a menu indicating the file system type and be asked to make a series of random mouse movements for a time. This is because TrueCrypt uses data relating to mouse movements as a seed for the hash algorithms during key and salt generation, therefore the more random and the longer the duration of the mouse movements the better.

Encrypted/Hidden Volume

An encrypted volume is a physical drive or partition (it could be a flash drive, a separate hard drive, partition or other permanent storage device) that isn't the system drive, such that it does not have an operating system installed on it. During the setup of the TrueCrypt volume the partition (or drive) will be formatted, losing any data that was originally on the drive. Once the drive is formatted it has to be mounted using TrueCrypt in order to access it (note that this is done using the mount device option in the main TrueCrypt menu instead of the mount file option used with TrueCrypt files). Without mounting the device it will appear as an unformatted drive when it is accessed by the file system. Once a TrueCrypt volume has been created a second volume, or rather a hidden volume can be created within it. This second volume is created with its own password and can be accessed when the original volume is mounted. Depending on which password is entered either the original volume or the hidden volume will be accessed. The volume headers (composed of data required to access the volume/files on it) are located at the beginning of the original volume, in a space specifically left for them, with the original volume header appearing first and the second volume header (or none if there is no hidden volume) following it before the data of the original. In this way the presence of the hidden volume remains secret since the space for the header volume is always left with what appears to be random data, even in the case where there is no hidden volume.

Encrypted/Hidden OS

TrueCrypt can also encrypt the system drive/partition. In doing so, a boot loader is installed which is responsible for prompting the user at startup for a password prior to loading the operating system. By encrypting the operating system the user can achieve an extra level of confidentiality for their system drive but more importantly they can also install a secondary, hidden operating system. This is done by creating two partitions in the system drive, the inner and outer partitions. The primary operating system is installed on the first partition while the second partition contains a hidden volume with a clone of the operating system stored on it. For this reason the second partition is always larger than the first since it requires space for the hidden volume as well. Depending on the file system used this could force the second partition to be 2.1 times as larger as the first (for NTSF). When a hidden operating system is installed it is accessed similar to how a hidden volume is accessed. During the boot sequence, when TrueCrypt asks the user for a password the user can supply either the password for the normal operating system or for the hidden one. TrueCrypt will then try and decrypt the data stored in the header files for the two operating systems (or the data in the location of where the two header files would be) and loads the operating system with the corresponding password. During the encryption process of the system drive the user will be prompted to create a TrueCrypt Rescue Disk. This is to be used in case of instances where the TrueCrypt boot loader becomes damaged for some reason or if other similar corruption issues occur at launch. It is important to note that TrueCrypt will not proceed until it has verified that the user has a working rescue disk for this purpose.

Plausible Deniability

Plausible deniability is a general term for hiding the fact that the user has hidden data, volume or operating systems even if it is apparent that the user has encrypted data on their system. This is accomplished by use of hidden volume and/or a hidden operating system since while it is obvious that TrueCrypt is installed on your system, and that you have an encrypted drive, volume or files, there is no clear indication to prove that the user has a secondary hidden volume or operating system. This is important for instances where the user may be coerced into revealing their password for an encrypted device. In an event like this the user only has to reveal the password for the original or dummy drive. In order to be able to maintain this level of security it is important for the system to be free of clues that would suggest the presence of a hidden operating system or volume. Doing so is referred to as maintaining plausible deniability. Towards this ends the TrueCrypt website offers a set of suggestions and practices that would help to remove the risk of leaving or creating such clues. Some of these practices include using the dummy operating system as you primary OS so that an active use record is maintained (a low level of use for that system could be discovered and would be suspicious) and placing dummy files that may appear sensitive (but are not, or at least not as sensitive) on the original drive that also contains a hidden volume. During the setup of a hidden operating system or volume TrueCrypt also takes steps to help ensure plausible deniability such as overwriting the original (non-hidden) operating system after the creating of the hidden OS in order to eliminate any logs of the TrueCrypt functions used in its creation. One final note is that while random data (equivalently: encrypted data) may appear suspicious and serve as a sign that encrypted data is present it can also be easily excused since there are many reasons and ways in which random data can exist on a system (i.e. the drive has just been recently formatted).

Multiple Users

In instances where a system is used by multiple users, particularly when multiple users do not share the same level of security access or require that their own data remain available only to themselves or a subset of the user group, it is important to maintain a level of access restriction. TrueCrypt will only allow a drive to be mounted by someone with the necessary password, and as such this password should only be given to those who are allowed to access the drive. However, once a drive has been mounted it remains accessible to any of the systems users while it remains mounted. Switching the user does not dismount a drive. It is also important to note that while any user can mount a drive they have access to only the user who mounted a drive (or the system administrator) can dismount the drive. For this reason it is important to maintain a clear and regimented set of procedures (involving the dismounting of any drives) when switching between users in order to maintain the desired level of confidentiality.

Evaluation

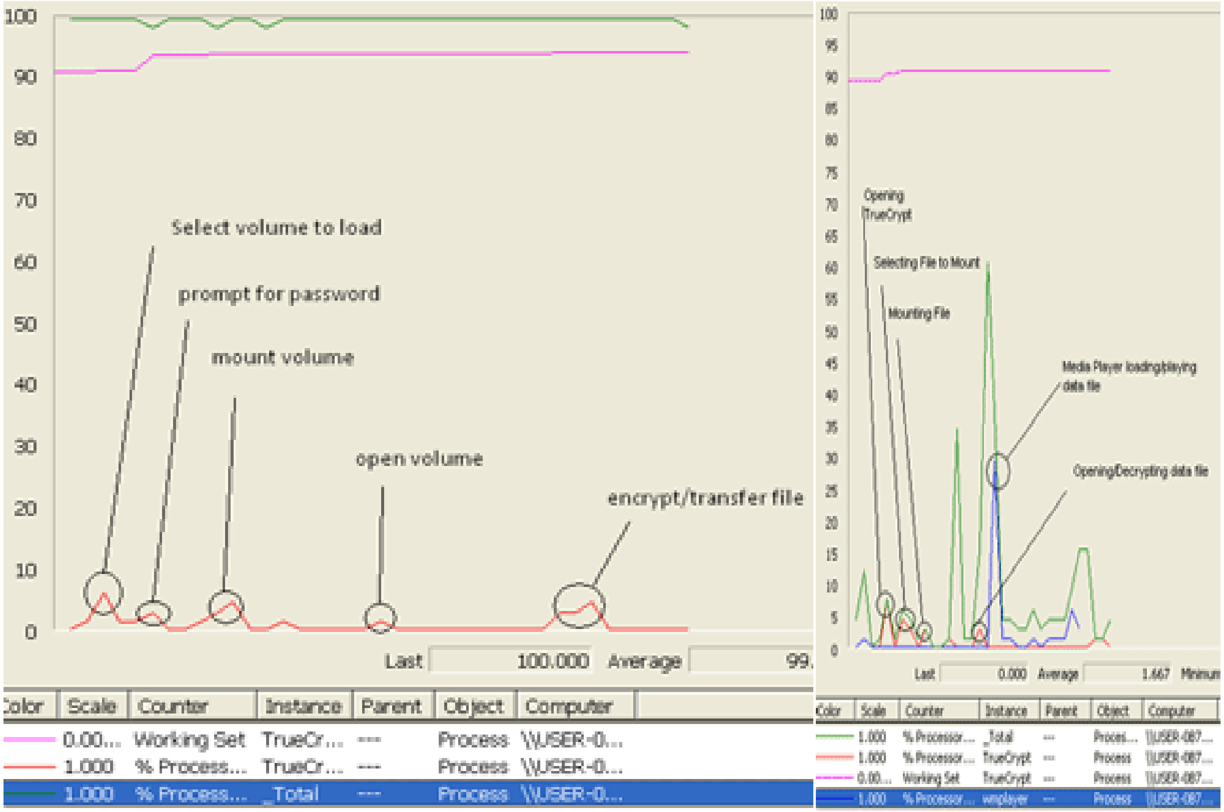
The following presents a brief performance evaluation of TrueCrypt under its various encryption schemes. The evaluation was performed on the following system:

- Intel Pentium 4 CPU 2.00 GHz
- 256 MB of RAM
- Microsoft Windows XP Service Pack 3

The performance test was done using the same file for each encryption scheme. The file itself is an audio file 8.56 MB in size and has a bit rate of 320 kbps. It is important to note that TrueCrypt encrypts/decrypts its data 'on the fly' such that all computation is done in RAM and no data is saved on the drive except for the final encrypted version (or decrypted version if the file is being moved). In the case of an application reading a file from a TrueCrypt file or volume, TrueCrypt will decrypt the entire file (in the case of a small file) for the application at once and will not need to be called again until it is time to save the file. In the case of a large file (such as a video file like a movie) TrueCrypt will decrypt the file in sections for the application, providing the decrypted data as needed. As a result the rate of calls to TrueCrypt will depend on the bitrate of such a file. For the file used for the performance evaluation this call was only needed once and as such only the first few seconds (particularly the time required to run the performance tool, mount the file, open the file location and begin playing the file) were recorded since only they possessed relevant data on TrueCrypts performance. The data was gathered using Microsoft's built in performance monitoring utility (perfmon.msc):

	Encrypt RAM (bytes)		Processor				Decrypt RAM (bytes)		Processor			
	Min	Max	AVG % Processor	Min	Max	Time (s)	Min	Max	AVG % Processor	Min	Max	Time (s)
AES	9113600	9113600	0.568	0	6.25	11	8962048	9129984	0.827	0	7.813	34
Twofish	9121792	9383322	0.962	0	6.25	39	9388032	9416704	0.092	0	1.563	17
Serpent	6352896	7475200	0.725	0	4.688	28	7495680	7467008	0.25	0	1.563	25
Serpent-Twofish-AES	7499776	7540736	0.756	0	10.938	31	7491584	7520256	0.276	0	1.563	34
AES-Twofish-Serpent	7528448	8765440	1.078	0	12.5	29	8716288	8749056	0.732	0	10.938	32
Twofish-Serpent	8769536	8781824	1.227	0	10.938	19	8744960	8744960	0.195	0	1.563	24
AES-Twofish	8740864	8769536	1.116	0	10.938	21	8716288	8744960	0.313	0	3.125	20
Serpent-AES	8744960	8781824	1.897	0	14.063	18	8716288	8744960	0.213	0	1.563	22

As can be seen the table is broken up into two sections, one for the encryption process and one for the decryption. Each section presents the same information regard the minimum and maximum amount of RAM used by TrueCrypt during the duration of the test as well as the percentage of the processor used (min max and average values), the test time is listed as a means to normalize the average values with each other so that they are not taken out of context. The tests themselves can be broken up into phases as shown in figure 8. It is worth mentioning that both the decryption and encryption test begin with mounting the volume. This is done so that the encrypted/decrypted file is not already in the RAM at the beginning of the test, providing more reliable results. From the data table it is clear that TrueCrypt uses relatively static amount of RAM while the processor time varies depending on the encryption cipher.



Last 100.000 Average 99.

olor	Scale	Counter	Instance	Parent	Object	Computer
🟪	0.00...	Working Set	TrueCr...	---	Process	\\USER-0...
🟥	1.000	% Process...	TrueCr...	---	Process	\\USER-0...
🟩	1.000	% Process...	_Total	---	Process	\\USER-0...

Last 0.000 Average 1.467 Minimum

Color	Scale	Counter	Instance	Parent	Object	Computer
🟩	1.000	% Processor...	_Total	---	Proces...	\\USER-087...
🟥	1.000	% Processor...	TrueCrypt	---	Process	\\USER-087...
🟪	0.00...	Working Set	TrueCrypt	---	Process	\\USER-087...
🟦	1.000	% Processor...	wmplayer	---	Process	\\USER-087...

Figure 8) the steps and corresponding processor usage involved in the data recording phase for file encryption and decryption

The inclusion of the three different blocks ciphers allows the user to balance their choice versus security requirements. While each cipher is considered safe and has yet to be cracked, each has a different safety factor, which is represented by the number of rounds in the algorithm divided by the number of rounds that have been jeopardized. AES has the lowest safety factor of the three with 1.42 (10/7) followed by Twofish with 16/6 and then Serpent with 32/9. Upon regarding the table however it appears that of the three standard choices (non cascaded) Serpent appears to be the best in both terms of security and system resources used, while AES shows moderate performance for encryption and somewhat poor performance for decryption while Twofish behaves in the opposite manner. This being said however, all three algorithms perform well and the case can easily be made for a more rigorous performance evaluation for a definitive result.

Conclusion

Based on the review and analysis the following points can be made about TrueCrypt

- Effective, versatile program with regards to the encryption mechanisms it utilizes and incorporates

- It can be useful for mobile security (through use of the movable TrueCrypt file) or for in-depth local security (using an encrypted volume or hidden OS), for single or multi-user systems while also being able to provide a degree of plausible deniability assuming that appropriate user precautions are taken.
- Can be used on a variety of platforms (FAT, NTSF) and operating systems (Windows Linux and Mac OSX)
- Efficient in terms of the system resources it uses.

References

[1] "TrueCrypt Free Open Source On-The-Fly Encryption", http://www.truecrypt.org/docs/

[2] Bruce Schneier, "Bruce Schneier: Twofish", http://www.schneier.com/twofish.html

[3] Antoon Bosselaers, "The RIPEMD Page", 25 August 2004, http://homes.esat.kuleuven.be/~bosselae/ripemd160.html,

[4] "Serpent Homepage", http://www.cl.cam.ac.uk/~rja14/serpent.html

[5] Paulo S. L. M. Barreto, "Whirlpool Homepage", 25 November 2008 http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html

[6] William Stalling, Cryptography and Network Security Principles and Practices 4th Edition, Pearson Education 2006

[7] W. Trappe, L. C. Washington, Introduction to Cryptography with Coding Theory 2nd Edition, Pearson education 2006

[8] Phillip Rogaway, Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC, 24 September 2004, http://www.cs.ucdavis.edu/~rogaway/papers/offsets.pdf

[9] Luther Martin, Voltage Superconductor: Understanding AES-XTS, http://superconductor.voltage.com/2009/07/understanding-aesxts-part-1.html

[10] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, page 16

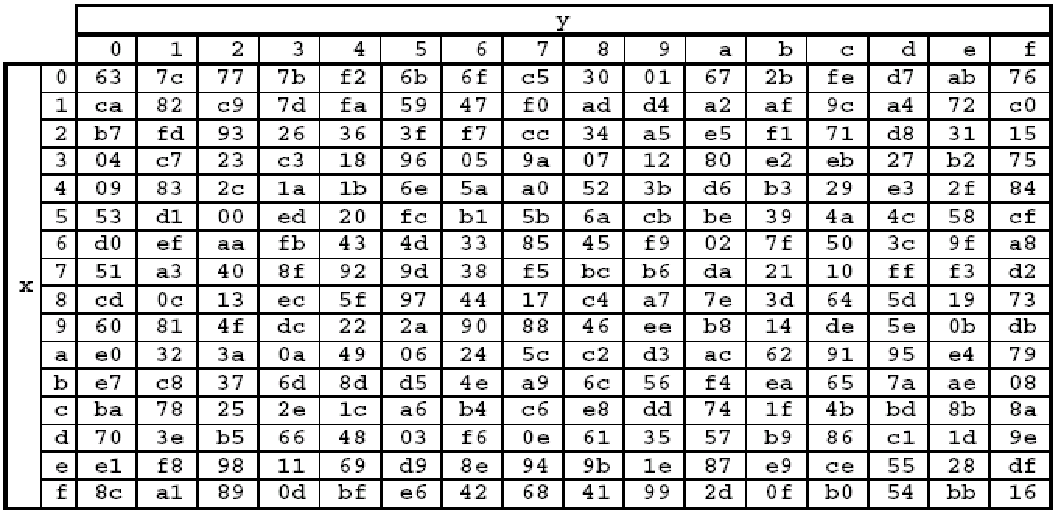
[11] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128 bit block Cipher", NIST AES Submission, June 15th 1997

[12] Morris Dworkin, CSRC Cryptography Toolkit, December 4 2001, http://csrc.nist.gov/archive/aes/index.html

Appendix A

Rijndael

Byte Substitution Matrix:



		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 7. S-box: substitution values for the byte xy (in hexadecimal format).

Column Mixing:

$$C_{cm} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} C_{rs\,00} & C_{rs\,01} & C_{rs\,02} & C_{rs\,03} \\ C_{rs\,10} & C_{rs\,11} & C_{rs\,12} & C_{rs\,13} \\ C_{rs\,20} & C_{rs\,21} & C_{rs\,22} & C_{rs\,23} \\ C_{rs\,30} & C_{rs\,31} & C_{rs\,32} & C_{rs\,33} \end{bmatrix}$$

Twofish

MDS Matrix (in HEX):

$$MDS = \begin{bmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{bmatrix}$$

Serpent

Initial Permutation:

$$\begin{bmatrix} 0 & 32 & 64 & 96 & 1 & 33 & 65 & 97 & 2 & 34 & 66 & 98 & 3 & 35 & 67 & 99 \\ 4 & 36 & 68 & 100 & 5 & 37 & 69 & 101 & 6 & 38 & 70 & 102 & 7 & 39 & 71 & 103 \\ 8 & 40 & 72 & 104 & 9 & 41 & 73 & 105 & 10 & 42 & 74 & 106 & 11 & 43 & 75 & 107 \\ 12 & 44 & 76 & 108 & 13 & 45 & 77 & 109 & 14 & 46 & 78 & 110 & 15 & 47 & 79 & 111 \\ 16 & 48 & 80 & 112 & 17 & 49 & 81 & 113 & 18 & 50 & 82 & 114 & 19 & 51 & 83 & 115 \\ 20 & 52 & 84 & 116 & 21 & 53 & 85 & 117 & 22 & 54 & 86 & 118 & 23 & 55 & 87 & 119 \\ 24 & 56 & 88 & 120 & 25 & 57 & 89 & 121 & 26 & 58 & 90 & 122 & 27 & 59 & 91 & 123 \\ 28 & 60 & 92 & 124 & 29 & 61 & 93 & 125 & 30 & 62 & 94 & 126 & 31 & 63 & 95 & 127 \end{bmatrix}$$

S-Boxes:

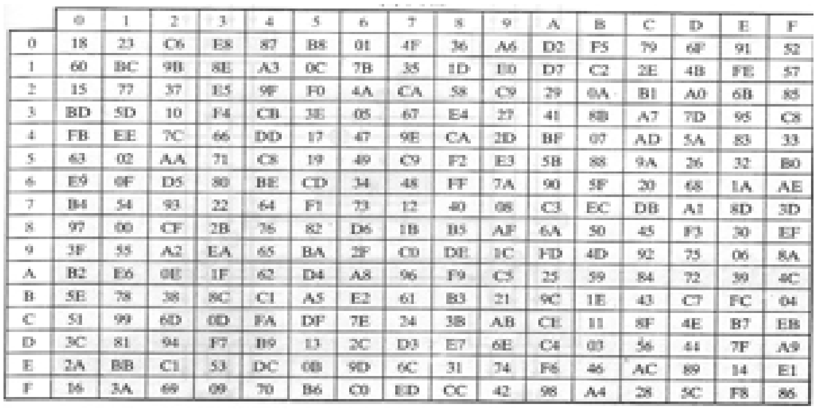
$S_0 =$ [3 8 15 1 10 6 5 11 14 13 4 2 7 0 9 12]
$S_1 =$ [15 12 2 7 9 0 5 10 1 11 14 8 6 13 3 4]
$S_2 =$ [8 6 7 9 3 12 10 15 13 1 14 4 0 11 5 2]
$S_3 =$ [0 15 11 8 12 9 6 3 13 1 2 4 10 7 5 14]
$S_4 =$ [1 15 8 3 12 0 11 6 2 5 4 10 9 14 7 13]
$S_5 =$ [15 5 2 11 4 10 9 12 0 3 14 8 13 6 7 1]
$S_6 =$ [7 2 12 5 8 4 6 11 14 9 1 15 13 3 10 0]
$S_7 =$ [1 13 15 0 14 8 2 11 7 4 12 10 9 3 5 6]

Final Permutation:

$$\begin{bmatrix} 0 & 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 & 56 & 60 \\ 64 & 68 & 72 & 76 & 80 & 84 & 88 & 92 & 96 & 100 & 104 & 108 & 112 & 116 & 120 & 124 \\ 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 & 33 & 37 & 41 & 45 & 49 & 53 & 57 & 61 \\ 65 & 69 & 73 & 77 & 81 & 85 & 89 & 93 & 97 & 101 & 105 & 109 & 113 & 117 & 121 & 125 \\ 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 & 34 & 38 & 42 & 46 & 50 & 54 & 58 & 62 \\ 66 & 70 & 74 & 78 & 82 & 86 & 90 & 94 & 98 & 102 & 106 & 110 & 114 & 118 & 122 & 126 \\ 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 & 35 & 39 & 43 & 47 & 51 & 55 & 59 & 63 \\ 67 & 71 & 75 & 79 & 83 & 87 & 91 & 95 & 99 & 103 & 107 & 111 & 115 & 119 & 123 & 127 \end{bmatrix}$$

Whirlpool

S-Box (In HEX):



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1	60	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2	15	77	37	E5	9F	F0	4A	CA	58	C9	29	0A	B1	A0	6B	85
3	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	C8
4	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5	63	02	AA	71	C8	19	49	C9	F2	E3	5B	88	9A	26	32	B0
6	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B	5E	78	38	8C	C1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
E	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F	16	3A	69	09	70	B6	C0	ED	CC	42	98	A4	28	5C	F8	86

RIPEMD-160 [3]

Initial H_0 Values:

$$h_0 = \texttt{67452301}_\texttt{x};\ h_1 = \texttt{EFCDAB89}_\texttt{x};\ h_2 = \texttt{98BADCFE}_\texttt{x};$$
$$h_3 = \texttt{10325476}_\texttt{x};\ h_4 = \texttt{C3D2E1F0}_\texttt{x};$$

Constant Values$K$:

$K_l(j) =$		$K_r(j) =$	
$\texttt{00000000}_\texttt{x}$	$(0 \le j \le 15)$	$\texttt{50A28BE6}_\texttt{x}$	$(0 \le j \le 15)$
$\texttt{5A827999}_\texttt{x}$	$(16 \le j \le 31)$	$\texttt{5C4DD124}_\texttt{x}$	$(16 \le j \le 31)$
$\texttt{6ED9EBA1}_\texttt{x}$	$(32 \le j \le 47)$	$\texttt{6D703EF3}_\texttt{x}$	$(32 \le j \le 47)$
$\texttt{8F1BBCDC}_\texttt{x}$	$(48 \le j \le 63)$	$\texttt{7A6D76E9}_\texttt{x}$	$(48 \le j \le 63)$
$\texttt{A953FD4E}_\texttt{x}$	$(64 \le j \le 79)$	$\texttt{00000000}_\texttt{x}$	$(64 \le j \le 79)$

32 bit Message Word $M(j)$

$M_l(j) =$

$$\begin{array}{ll}
j & (0 \leq j \leq 15) \\
7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8 & (16 \leq j \leq 31) \\
3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12 & (32 \leq j \leq 47) \\
1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2 & (48 \leq j \leq 63) \\
4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13 & (64 \leq j \leq 79)
\end{array}$$

$M_r(j) =$

$$\begin{array}{ll}
5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12 & (0 \leq j \leq 15) \\
6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2 & (16 \leq j \leq 31) \\
15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13 & (32 \leq j \leq 47) \\
8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14 & (48 \leq j \leq 63) \\
12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11 & (64 \leq j \leq 79)
\end{array}$$

Rotation bits $S(j)$

$S_l(j) =$

$$\begin{array}{ll}
11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8 & (0 \leq j \leq 15) \\
7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12 & (16 \leq j \leq 31) \\
11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5 & (32 \leq j \leq 47) \\
11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12 & (48 \leq j \leq 63) \\
9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6 & (64 \leq j \leq 79)
\end{array}$$

$S_r(j) =$

$$\begin{array}{ll}
8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6 & (0 \leq j \leq 15) \\
9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11 & (16 \leq j \leq 31) \\
9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5 & (32 \leq j \leq 47) \\
15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8 & (48 \leq j \leq 63) \\
8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11 & (64 \leq j \leq 79)
\end{array}$$

Appendix B

SALT:

A SALT is a series of random bits that is used to obscure a source key, intending to make dictionary attack much more difficult if not infeasible. For instance, assume a user has a password that is a word in a particular language. A dictionary attack will make use of this by trying the different likely words as a key for decrypting an encoded message. However, if a SALT is used, the bitstring representing the users word will be augmented by the SALT bitstring (according to the particular method being used), thus making the key used for encryption different (or rather a modified version) of that which would have normally corresponded directly to the key from the 'language word'.