International Journal of Engineering, Science and Mathematics
Vol. 6 Issue 8, December 2017 (Special Issue)
ISSN: 2320-0294 Impact Factor: 6.765
Journal Homepage: http://www.ijesm.co.in, Email: ijesmj@gmail.com Double-Blind
Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

ENHANCED HYBRID CRYPTOSYSTEM FOR SOURCE CODE

Sashi Bhusan Maharana
Dept. of Computer Science & Engineering
Centurion University of Technology & Management (CUTM), Paralakhemundi, Odisha, India

Dr. Jatindra Kumar Dash
Dept. of Computer Science & Engineering
National Institute of Science & Technology (NIST), Berhampur, Odisha, India

Dr. Sarojananda Mishra
Dept. of Computer Science & Engineering
Indira Gandhi Institute of Technology (IGIT), Sarang, Odisha, India

Sambeet Patro
Dept. of Computer Science & Engineering
Centurion University of Technology & Management (CUTM), Paralakhemundi, Odisha, India

Abstract

This paper aims to provide security to the source code file on a network with the use of cryptographic approach. This achieves advantage of individual cryptographic approach by building an enhanced hybrid cryptosystem. Source code will be legible only to authorized consumers.

Keywords:

Source code security;
Enhanced cryptosystem;
MPC.

Author correspondence:

Sashi Bhusan Maharana,
Dept. of Computer Science & Engineering
Centurion University of Technology & Management (CUTM), Paralakhemundi, Odisha, India
sbmaharana@gmail.com

1. Introduction

Transmission security is the capability to send source codes electronically from one computer to another so that only the intended recipient receives and read the source code, and the code text received is identical to the source code sent. Designing a 100 percent transmission secure network is still challenging – no matters how secure the design, someone could always break it. This does not mean one should sit back and wait for the interception to happen. Instead, efforts have been made to build system to discourage people from attempting to break in, and to make it costly for the hacker to decode.

Cryptography

Cryptography is the study of mathematical techniques related to aspects of information security, such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques.

2. Research Method

CORRELATED STUDIES

A. Private Key Vs. Public Key Cryptography

Symmetric key and public key encryption schemes have various advantages and disadvantages, some of them are common to both.

Advantages of symmetric key cryptography:

- Symmetric key ciphers can be designed to have high rate date throughput. Some hardware implementations achieve encrypt rates of hundreds of Megabytes per second, while software implementations may attain throughput rates in the megabytes per second range.
- Keys for symmetric key ciphers are relatively short.
- Symmetric key ciphers can be employed as primitives to construct various cryptographic mechanisms including pseudo random number generators, hash functions, and computationally efficient digital signature schemes, to name just a few.
- Symmetric key ciphers can be composed to produce stronger ciphers. Simple transformations, which are easy to analyze, but are weak on their own, can be used to construct strong product ciphers.

Disadvantages of symmetric key cryptography:

- In two party communications, the key must remain secret at both sides.
- In a large network, there are many key pairs to be managed.
- In a two party communications between entities A and B, sound cryptographic practice dictates that the key be changed frequently and perhaps for each communication session.
- Digital signature mechanisms arising from symmetric key encryption typically require large keys for the public verification function.

Advantages of public key cryptography:

- Only the private key must be kept secret (authenticity of public keys must, however, be guaranteed).
- Depending on the mode of usage, a private key/public key pair may remain unchanged for considerable period of time.
- Many public key schemes yield relatively efficient digital signature mechanisms. The key used to describe the public verification function is typically much smaller than symmetric key counterpart.
- In a large network, the number of keys necessary may be considerably smaller than in the symmetric key scenario.

Disadvantages of public key cryptography:

- Throughput rates for the most popular public key encryption methods are several orders of magnitude slower than the best known symmetric key schemes.
- Key sizes are typically much larger than those required for symmetric key encryption, and the size of the public key signatures is larger than that of tags providing data origin authentication from symmetric key techniques.
- No public key scheme has proved to be secure. The most effective public Key encryption schemes found to date have their security based on the presumed difficulty of a small set of number theoretical problems.

B. The Multi-Phase Algorithm

This is a recently developed encryption algorithm which is (in our opinion) nearly uncrackable. The reason why will be pretty obvious when we take a look at the method itself.

ENHANCED HYBRID CRYPTO-SYSTEM

So, why are there so many different types of cryptographic schemes? Why can't we do everything we need with just one?

The answer is that each scheme is optimized for some specific objective(s) of cryptography. Hash functions, for example, are well suited for ensuring data integrity because any change made to the contents of a source code will result in the consumer calculating a different hash value than the one placed in the transmission by the provider. Since it is highly unlikely that two different source codes will yield the same hash value, data integrity is ensured to a high degree of confidence.

In this proposal the following algorithms have been used to develop a Hybrid cryptosystem application:

1. For secret key cryptography a modified version of a brand new "MULTI-PHASE" algorithm proposed by R. E. Frazier (the author of S.F.T. Inc) with 128-bit key is used. Modification has been done to the original algorithm for effectiveness.
2. For public key cryptography the most used algorithm RSA has been used.
3. For hashing some algorithm mentioned in could be used.

DESIGN OF ENHANCED HYBRID CRYPTO-SYSTEM

The "MULTI-PHASE" encryption algorithm [Here after known as MPC (Multi Phase Cipher)] is character oriented, operating on a block of 8 bits with an 128-bit key. This functional description of the algorithm is based on the paper "Data Encryption Techniques", using the different general layout, terminology, and pseudo code style.

The MPC uses the following primitive operations:

1. Two's-complement addition of words, denoted by "+".
2. Bitwise Exclusive-OR, denoted by "^".
   Description: It compares each bit of its first operand to the corresponding bit of its second operand. If one bit is zero and the other is 1, the corresponding result bit is set to 1.Otherwise; the corresponding result bit is set to zero.
3. Bitwise AND, denoted by "&".
   Description: It compares each bit of its first operand to the corresponding bit of its second operand. If both are 1, the corresponding result bit is set to 1. Otherwise; the corresponding result bit is set to zero.
4. Bitwise Right Shift Operator, denoted by ">>".
   Description: This is used to shift the bits of an expression to the right, maintaining the sign. The syntax is as follows:
   Result = expression 1 >> expression 2
   The >> operator shifts the bits of expression1 right by the number of bits specified in expression2.The sign bit of expression1 is used to fill the digits from left. Digits shifted off from right are discarded.
5. Bitwise Left Shift Operator, denoted by "<<".
   Description: This is used to shift the bits of an expression to the left. The syntax is as follows:
   Result = expression 1 << expression 2
   The << operator shifts the bits of expression1 left by the number of bits specified in expression2.
6. Bitwise OR Operator, denoted by "|".
   Description: The **|** operator looks at the binary representation of the values of two expressions and does a bitwise OR operation on them
7. Bitwise XOR Assignment Operator, denoted by "^=".
   Description: Used to perform a bitwise exclusive OR on an expression. The syntax is as follows:
   Result ^= Expression
   Using the ^= operator is exactly the same as specifying:
   Result = Result ^ Expression
8. '0x' before any number indicates hexadecimal representation.

These operations are directly and most efficiently supported by most processors.

The MPC Algorithm:

MPC consists of four components, a 'Seed Generation' algorithm, a 'Random number generation' algorithm an 'Encryption' algorithm and a 'Decryption' algorithm.

SEED GENERATOR:

The pseudocode for the proposed seed generator algorithm is provided by figure 4.1 below.

In put:*K* The session key (16 characters).

Output:*S* The seed for the random number generator (32-bit).

```
    // Step 1: Initialization of S-Box

1   unsigned longpoly←0xEDB88320        // 32-bit constant
2   for i←0  to 255
loop 1
3       c ←i
4        forj←8 to 1
5        loop 2
6           if (c & 1)
7               c ← (c>>1) ^ poly
8            else
9              c ← c>>1
10       end of loop 2
11      S-Box[i] ← c
end of loop 1

    // Step 2: Seed generation

12  S ←0xFFFFFFFF
13  forі← 0 to 15
14     S ← ((S>>8) &0xFFFFFFFF) ^ S-Box [ (S ^ K[i]) &0xFF ]
15  return(S ^ 0xFFFFFFFF)    // Seed for the Random No. Generator
```

Fig.1 Pseudo-code for seed generation algorithm

Input to the algorithm is a key (K) of minimum 16 characters. Output of the algorithm is a 32-bit seed(S) which will be the input for the random number generator. The algorithm has two steps – Initialization of S-Box and Seed generation. The S-Box is a array of size 256. This is initialized by performing the right shift (>>) and bitwise XOR operation (^) on a constant (poly) and a variable (c) as mentioned by the loop 1 [line 3-11] in the figure 4.1. The constant poly is a 32-bit constant (Hexadecimal represented). It may be chosen any 32-bit constant. For example purpose it has been taken as 0xEDB88320 (integer value 3988292384). By choosing poly as a 32-bit value, each location in the S-Box can be initialized to 32-bit value. This helps in the generation of a 32-bit seed in the next step.

In step 2, the seed is first initialized with 0xFFFFFFFF. It may also be initialized to any other 32-bit value. Then the pre-final seed is calculated using the initialized seed and the value in the S-Box as mentioned in the line 16 in the figure 4.1. The final 32-bit seed is generated by XORing the pre-final seed with the initialized seed as mentioned in the line 17 in figure 4.1.

RANDOM NUMBER GENERATOR:

The pseudocode for the proposed random number generation algorithm is provided by the figure 4.2.

Input to the algorithm is the 32-bit seed (S). Output of the algorithm is a two dimensional table of random numbers. The size of this two dimensional array is 256*256. Each random number is generated by using seven variables which are initialized using the 32-bit seed. The variable rand_x is calculated using the expression 'rand_x = rand_x * 69069 + 1' to have a periodicity greater than 256. For the implementation of the encryption algorithm, it is required that no random value should be repeated within a single row. By choosing the integer 69069 provides a greater periodicity. The random numbers are generated by shifting

the bits of the variables to left and right and taking the sum as mentioned in the loop (line 9-18) in the figure 2.

```
Input:S                    // The 32-bit seed generated by the seed generator

Output:RandTable[256] [256] // Two dimensional array of Random numbers

      // Step 1: Initialization of Variables

 1      unsigned integerrand_x, rand_y, rand_z, rand_w, rand_carry, rand_k, rand_m
2rand_x← seed | 1
3rand_y← seed |2
4rand_z← seed |4
5rand_w← seed |8
6 rand_carry←0

      // Step 2: Random Table Generation

 7      forі← 0 to 255
 8      for j ←  0 to 255
loop
 9          rand_x = rand_x * 69069 + 1     // * is the multiplication operator
 10         rand_y ^= rand_y<< 13
 11         rand_y ^= rand_y>> 17
 12         rand_y ^= rand_y<< 5
 13         rand_k← (rand_z>> 2) + (rand_w>> 3) + (rand_carry>> 2)
 14         rand_m←rand_w + rand_w + rand_z + rand_carry

 15         rand_z←rand_w;
 16         rand_w←rand_m;
17          rand_carry←rand_k>> 30;

 18         Rand Table [ i ] [  j ] = rand_x + rand_y + rand_w
end of the loop
```

Fig.2 Pseudo-code for random number generation algorithm

ENCRYPTION ALGORITHM:

After generating the two dimensional array (size 256 x 256) of random numbers the encryption algorithm is as follows.

1) 256 entries at a time, use the random number sequence to generate arrays of "Cipher Translation Tables" as follows
   - Fill an array of integers with 256 random numbers from one of the row of the RandTable [256] [256].
   - Sort the numbers using a method (like pointer) that lets us know the original position of the corresponding number.
   - Using the original positions of the now-sorted integers, generate a table of randomly sorted numbers between 0 and 255.
2) Now, generate a specific number of 256-byte tables. Let the random number generator continue 'in sequence' for all of these tables, so that each table is different.
3) Now we have the translation tables. Using these tables, generate a two dimensional Reference Table (RefTable[256][256]) as follows:
   - Fill the first column of the RefTable with elements of the first Cipher Translation Table.
   - Similarly, fill the second column of the RefTable with elements of the second Cipher Translation Table and so on up to 256th row of the RefTable.

Now all the elements of a singe column of the RefTable are the values between 0 and 255 and not repeated. Any character can be represented using the values of a single column.

Now we have the RefTable of size 256 x 256, the basic cipher works like this:

The previous byte's encrypted value is the index of the 256 x 256 RefTable. Alternatively, for improved encryption we can use more than one byte, and either use a 'checksum' or a CRC algorithm to generate the index byte. We can then 'mod' it with the # (number) of Cipher Translation Tables if we use less than 256 x 256 bytes table. Assuming the table is a 256 x 256 array, it would look like this:

Crypto I = RefTable [Crypto (I-1)] [Count]

Where 'Crypto I' is the encrypted byte, and 'Crypto (I-1)' is the previous byte's encrypted value (or a function of several previous values). Count refers to the column value of the RefTable. Naturally, the first byte will need a 'seed', which must be known. This may increase the total cipher size by an additional 8 bits if we use 256 x 256 tables. Or, we can use the key we generated the random list with, perhaps taking the CRC of it.

In this project the encryption has been performed using the following rule to provide better security:

Crypto I = RefTable[**fn (I/P, Crypto (I-1)**][**Count**]

Where 'fn' is a recoverable function i.e. this should not be a trapdoor problem. For the ease of implementation Bitwise-XOR function has been chosen, which is easily recoverable and provide better security.

DECRYPTION ALGORITHM:

Decryption algorithm also needs to generate the same RefTable as in encryption algorithm. After generation of the same RefTable the decryption process is as follows:

Decrypted Value I = fn (RefTable [Crypto I] [Count], Crypto (I-1))

Both encryption and decryption algorithms can be better understood with the help of an example.

3. Results and Analysis

Results:

This example implement MPC algorithm for the characters that can be represented using the ASCII value between 0 and 7. Though it is not possible to take a 256 x 256 table here this example shows how MPC works for a 7 x 7 bytes translation table.

Let the 128-bit **key** is:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101

Let the **seed** (previous encrypted value for the first byte) is 5.

The Encryption Process is as follows:

Based on the key and random number generation algorithm the Random Table of size 7 x 7 will be generated as given in Table 1.

COUNT							
(0)	143	1859	1639	827	511	499	343
(1)	398	961	205	617	1877	1873	1821

(2)	1171	887	1291	399	1091	1895	59
(3)	793	69	897	1421	41	533	785
(4)	2039	1931	527	707	999	699	895
(5)	1421	41	533	785	2013	1593	229
(6)	955	127	1651	983	491	239	1059

Table 1: RandTable [7] [7]

The seven Cipher Translation Tables (CTTs) are generated as follows:

- Count = 0

Rand[7] = 143 1859 1639 827 511 499 343
SRand[7]= 143 343 499 511 827 1639 1859
CTT0[7] = 0 6 5 4 3 2 1

- Count = 1

Rand [7] = 389 961 205 617 1877 1873 1821
SRand [7]= 205 389 617 961 1821 1873 1877
CTT1 [7] = 2 0 3 1 6 5 4

- Count = 2

Rand [7] = 1171 887 1291 399 1091 1895 59
SRand [7]= 59 399 887 1091 1171 1291 1859
CTT2 [7] = 6 3 1 4 0 2 5

- Count = 3

Rand[7] = 793 69 897 1421 41 533 785
SRand[7]= 41 69 533 785 793 897 1421
CTT3[7] = 4 1 5 6 0 2 3

- Count = 4

Rand[7] = 2039 1931 527 707 999 699 895
SRand[7]= 527 699 707 895 999 1931 2039
CTT4[7] = 2 5 3 6 4 1 0

- Count = 5

Rand[7] = 1421 41 533 785 2013 1593 229
SRand[7]= 41 229 533 785 1421 1593 2013
CTT5[7] = 1 6 2 3 0 5 4

- Count = 6

Rand[7] = 955 127 1651 983 491 239 1059
SRand[7]= 127 239 491 955 983 1059 1651
CTT6[7] = 1 5 4 0 3 6 2

Then the RefTable is generated by the filling the 0^{th} column of it with the elements of CTT0[7], 1^{st} column of it with the elements of the CTT1[7],and so on. Then the RefTable of size 7 x 7 will be as shown in Table2 below.

COU=> ROW	(0)	(1)	(2)	(3)	(4)	(5)	(6)
(0)	0	2	6	4	**2***	1	1

(1)	6	0	3	1	5	**6***	5
(2)	5	3	**1***	5	3	2	4
(3)	4	1	4	**6***	6	3	0
(4)	3	6	0	0	4	0	**3***
(5)	2	**5***	2	2	1	5	6
(6)	**1***	4	5	3	0	4	2

Table 2: Reference Table (RefTable [7][7])

Now the basic cipher works like this:

PLAIN TEXT INPUT [N] = 3 4 7 2 6 3 2 (ASCII Values)

Here N = 7

- For Count = 0

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):

Input (3)=> 011

Pre. Encry. Value (5)=> 101

(Seed) ---

XOR => 110 (6)

Encrypted Value = RefTable [6] [0] = 1.

- For Count = 1

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):

Input (4)=> 100

Pre. Encry. Value (1)=> 001

XOR => 101 (5)

Encrypted Value = RefTable [5] [1] = 5

- For Count = 2

Encrypted Value=RefTable[fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):-

Input (7)=> 111

Pre. Encry. Value (5)=> 101

XOR => 010 (2)

Encrypted Value = RefTable [2] [2]= 1

- For Count = 3

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):-

Input (2)=> 010

Pre. Encry. Value (1)=> 001

XOR => 011 (3)

Encrypted Value = RefTable [3] [3]= 6

- For Count = 4

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):-

Input (6)=> 110

Pre. Encry. Value (6)=> 110

XOR => 000 (0)
Encrypted Value = RefTable [0] [4] = 2

- For Count = 5

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]

fn(I/P, Pre. Encry. Value):
Input (3)=> 011
Pre. Encry. Value (2)=> 010

XOR => 001 (1)
Encrypted Value = RefTab [1] [5] = 6

- For Count = 6

Encrypted Value=RefTable [fn(I/P, Pre. Encry. Value)] [Count]
fn(I/P, Pre. Encry. Value):
Input (2)=> 010
Pre. Encry. Value (6)=> 110

XOR => 100 (4)
Encrypted Value = RefTable[4][6]= 3

CIPHER TEXT OUTPUT[N] = 1 5 1 6 2 6 3
Where as the is INPUT[N] = 3 4 7 2 6 3 2

The decryption process is as follows:

CIPHER TEXT INPUT [N] = 1 5 1 6 2 6 3
Here N = 7

- For Count = 0

Decrypted Value =fn(RefTable[Encry.Input][Count], Pre.Encry. Value)
= fn(RefTable[1][0] , Seed) 6=>110
= fn(Row , Seed) 5=>101
= fn(6,5) ---
= 3 011(3)

- For Count = 1

Decrypted Value=fn(RefTable[Encry. Input][count], Pre. Encry. Value)
= fn(RefTable[5][1] , 1) 5=>101
= fn(Row , 1) 1=>001
= fn(5,1) ---
= 4 100(4)

- For Count = 2

Decrypted Value = fn(RefTable[Encry.Input][count], Pre.Encry. Value)
= fn(RefTable[1][2] , 5) 2=>010
= fn(Row , 5) 5=>101
= fn(2,5) ---
= 7 111(7)

- For Count = 3

Decrypted Value=fn(RefTable[Encry. Input][count], Pre. Encry. Value)
= fn(RefTable[6][3] , 1) 3=>011
= fn(Row , 1) 1=>001
= fn(3,1) ---
= 2 010(2)

- For Count = 4
  Decrypted Value=fn(RefTable[Encry. Input][count], Pre. Encry. Value)
  = fn(RefTable[2][4] , 6) 0=>000
  = fn(Row , 6) 6=>110
  = fn(0,6) ----------
  = 6 110(6)
- For Count = 5
  Decrypted Value=fn(RefTable[Encry. Input][count], Pre. Encry. Value)
  = fn(RefTable[6][5] , 2) 1=>001
  = fn(Row , 2) 2=>010
  = fn(1,2) ----------
  = 3 011(3)
- For Count = 6
  Decrypted Value=fn(RefTable[Encry. Input][count], Pre. Encry. Value)
  = fn(RefTable[3][6] , 6) 4=>100
  = fn(4,6) 6=>110
  = 2 ----------
  010(2)

PLAIN TEXT OUTPUT[N] = 3 4 7 2 6 3 2

In this way MPC algorithm is able to recover the plain text from the cipher text.

Analysis:
Plain Text:
011 100 111 010 110 011 010
Cipher Text:
001 101 001 110 010 110 011
Change in number of bits is 8.

The MPC algorithm was implemented to analyze it closely. There are several points of view using which the algorithm can be analyzed.

A. Avalanche Effect
A desired property of any encryption algorithm is that small change in either the plain text or the key should produce a significant change in the cipher text. In particular a change in one bit of plain text or one bit of key should produce a change in many bits of the cipher text. If the change were small, this might provide a way to reduce the size of plain text or the key space to be searched.

MPC exhibits a strong avalanche effect. The **tests 1** given below shows the results taken where <u>two plain texts differ by one bit (underlined) were used.</u>

Test 1:-
Plain Text 1: jatindra
0<u>1</u>101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
Plain Text 2: *atindra
0<u>0</u>101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
With the key: jatindrakumardas
01101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
01101011 01110101 01101101 01100001 01110010 01100100 01100001 01110011
Cipher Text 1:

Character	ASCII Value	Binary Representation
‘}’	125	01111101
‘…’	133	10000101
‘U ’	85	01010101
‘’	20	00010100

‘ ‘	10	00001010
‘ M ’	77	01001101
‘ \ ’	92	01011100
‘ ’ ’	146	10010010

i.e. the binary representation of cipher text 1 is:
01111101 10000101 01010101 00010100 00001010 01001101 01011100 10010010
Cipher Text 2:

Character	ASCII Value	Binary Representation
‘–‘	150	10010110
‘ _’	1	00000001
‘&’	38	00100110
‘ª’	170	10101010
‘‰’	137	10001001
‘ð’	240	11110000
‘,’	44	00101100
‘ñ’	241	11110001

i.e. the binary representation of Cipher text 2 is:
10010110 00000001 00100110 10101010 10001001 11110000 00101100 11110001
The Cipher text 1 and cipher text 2 are different in 35 bit positions with a single bit change in the plain text.

The **test 2** given below shows the results taken where a single plain text is tested with two different keys that differ by one bit (underlined) were used.

Test 2:-
Plain Text 1: jatindra
01101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
Key 1: jatindrakumardas
01101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
01101011 01110101 01101101 01100001 01110010 01100100 01100001 01110011
Key 2: *atindrakumardas
00101010 01100001 01110100 01101001 01101110 01100100 01110010 01100001
01101011 01110101 01101101 01100001 01110010 01100100 01100001 01110011

The plain text (jatindra) converted to cipher text by the key ‘jatindrakumardas’ has already been obtained in the previous test. Now the cipher text 1 using the key 2 is:
Cipher Text 1:

Character	ASCII Value	Binary Representation
‘/’	47	00101111
‘ï’	239	11101111
‘Ü’	220	11011100
‘’’	180	10110100
‘<’	60	00111100
‘Þ’	222	11011110
‘ _’	2	00000010
‘Ø’	216	11011000

i.e. the binary representation of Cipher text 1 with key 2 is:
00101111 11101111 11011100 10110100 00111100 11011110 00000010 11011000

The Cipher text 1 with key 1 and Cipher text 1 using key 2 are different in 32 bit positions with a single bit change in the key.

B. Level of security

This is measured by the work factor. Work factor is defined as the minimum amount of work required to determine the secret key. If the work factor is't' years, for a sufficiently large't' the cryptographic scheme is, for all purposes, a secure system.

The security of an algorithm rests in the key. If you're using a cryptographically weak process to generate keys, then your whole system is weak. The cryptanalyzer need not cryptanalyze the encryption algorithm; he can cryptanalyze the key generation algorithm.

DES has a 56-bit key. Implemented properly, any 56-bit string can be the key; there are 256 (1016) possible keys. Norton Discreet for MS-DOS (versions 8.0 and earlier) only allows ASCII keys, forcing the high-order bit of each byte to be zero. The program also converts lowercase letters to uppercase (so the fifth bit of each byte is always the opposite of the sixth bit) and ignores the low-order bit of each byte, resulting in only 240 possible keys. These poor key generation procedures have made its DES ten thousand times easier to break than a proper implementation.

Table 3 gives the number of possible keys with various constraints on the input strings. Table 4.4 gives the time required for an exhaustive search through all of those keys, given a million attempts per second. Remember, there is very little time differential between an exhaustive search for 8-byte keys and an exhaustive search of 4-, 5-, 6-, 7-, and 8-byte keys.

All specialized brute-force hardware and parallel implementations will work here. Testing a million keys per second (either with one machine or with multiple machines in parallel), it is feasible to crack lowercase-letter and lowercase-letter-and-number keys up to 8 bytes long, alphanumeric-character keys up to 7 bytes long, printable character and ASCII-character keys up to 6 bytes long, and 8-bit-ASCII-character keys up to 5 bytes long.

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	460,000	$1.2*10^7$	$3.1*10^8$	$8.0*10^9$	$2.1*10^{11}$
Lowercase letters and digits (36):	1,700,000	$6.0*10^7$	$2.2*10^9$	$7.8*10^{10}$	$2.8*10^{12}$
Alphanumeric characters (62):	$1.5*10^7$	$9.2*10^8$	$5.7*10^{10}$	$3.5*10^{12}$	$2.2*10^{14}$
Printable characters (95):	$8.1*10^7$	$7.7*10^9$	$7.4*10^{11}$	$7.0*10^{13}$	$6.6*10^{15}$
ASCII characters (128):	$2.7*10^8$	$3.4*10^{10}$	$4.4*10^{12}$	$5.6*10^{14}$	$7.2*10^{16}$
8-bit ASCII characters (256):	$4.3*10^9$	$1.1*10^{12}$	$2.8*10^{14}$	$7.2*10^{16}$	$1.8*10^{19}$

Table 3: Number of Possible Keys of Various keyspaces

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	.5 sec	12 sec	5 min	2.2 hrs	2.4 day
Lowercase letters and digits (36):	1.7 sec	1 min	36 min	22 hrs	33 days
Alphanumeric characters (62):	15 sec	15 min	16 hrs	41 days	6.9 yrs
Printable characters (95):	1.4 min	2.1 hrs	8.5 days	2.2 yrs	210 yrs
ASCII characters (128):	4.5 min	9.5 hrs	51 days	18 yrs	2300 yrs
8-bit ASCII characters (256):	1.2 hrs	13 days	8.9 yrs	2300yrs	580,000yrs

Table 4: Exhaustive Search of Various Keyspaces (assume one million attempts per second)

The MPC algorithm uses printable characters (95) for the key generation and the key is 128 bits i.e. 16 Bytes long. Therefore the number of possible keys is $4.3*10^{31}$. This implies the work factor for this crypto system, based on the above assumption, is $1.36*10^{18}$ years. This is quite higher than the DES algorithm which implement 56 bit key, using any keyspace.

DES is a block cipher operated on a block of 64-bit. There is no feedback mechanism between the currently encrypting block and previously encrypted block. Let a same 64 bits block appears more than once. With same key that block will be encrypted to same cipher text. But though MPC is a stream cipher, it uses a feedback mechanism between the preceding characters. Therefore the same block appearing more than once will be encrypted to different cipher text. And this will be more difficult for the cryptanalyzer to cryptanalyze. The same above comparison can be made with the RSA.

C. Performance

This refers to the efficiency of the algorithm in a particular mode of operation. An encryption algorithm may be rated by the no of bits/sec at which it can encrypt. The MPC algorithm can encrypt at a rate of 3534 Kbytes/sec.

D. Ease of implementation

This refers to the difficulty of realizing the algorithm in a practical instantiation and might include the complexity of implementing the algorithm in either software or hardware environment.

DES and RSA include high computational work and tough to implement. These algorithms provide high level of security but do not provide better system performance.

The MPC algorithm is easy to implement and particularly suitable in an environment where computing power is limited. MPC provides a trade off between very high levels of security for the better system performance.

4. Conclusion

This paper presented an enhanced cryptosystem which could provide digital signature as well as encryption functions. Besides this, compression services can also be used. In the general protocol for developing an enhanced hybrid cryptosystem, after the source code has been encrypted at source, may be compressed using ZIP, for storage and transmission. At the consumer end, before decryption, the source code is decompressed.

An extended work to the cryptosystem may involve building a stronger algorithm for hash function; a better cryptosystem can be implemented. If a stronger hashing algorithm is used, the following cryptosystem based on the proposed general protocol can also be implemented.

For digital signature a hash value of the source code is created using one of the hashing algorithms. This source code digest can be encrypted using RSA with the provider's private key and, and included with the source code.

For source code encryption a better symmetric algorithm may also be used with one time session key set generated by the provider and consumer.

Future studies will involve developing an L-System based compiler to optimize performance by enhanced parsing. This compiler will seek for the encrypted source code file.

References

[1] William Stallings, **"Cryptography and Network Security"**, Pearson Education Asia

[2] Hal Tipton and Micki Krause, **"Handbook of Information SecurityManagement"**,CRC Press LLC

[3] Gil Held, **"Learn Encryption Techniques with BASIC and C++"**, Wordware Publishing, Inc

[4] R. E. Frazier, **"Data Encryption Techniques"**, Data Encryption Techniques.htm

[5] Bruce Schneier, **"Applied Cryptography, Second Edition"**, John Wiley & Sons, Inc.

[6] Alexander N. Pisarchik, **"Encryption and decryption of images with chaoticmap lattices"**, academia.edu, Chaos 16, 033118 (2006)

[7] Aayushi Jangid, **"CRYPTOGRAPHY"**, International Journal of Scientific &Engineering Research, Volume 1, Issue 2, November-2010, ISSN 2229-5518 IJSER © 2010, http://www.ijser.org