International Journal of Technical Research and Applications e-ISSN: 2320-8163,
www.ijtra.com Volume 6, Issue 2 (MARCH-APRIL 2018), PP. 76-81

SYMMETRIC KEY ENCRYPTION USING A SIMPLE PSEUDO-RANDOM GENERATOR TO PROVIDE MORE SECURE COMMUNICATION

Brijgopal Bharadwaj[1], Shreyas Shukla[2], L. Shalini[3],
Department Of Computer Science Engineering
[1,2]Student, B. Tech., VIT, Vellore
[3]Assistant Professor (Senior), VIT, Vellore

***Abstract*— Symmetric key encryption is a cryptographic algorithm in which same keys are used for both, encryption and decryption of the information. Sometimes it is also referred to as the secret key cryptosystem and is the fastest mode of encryption. But, the usage of the same key for conversion and extraction of the data from the ciphertext is a major drawback of this algorithm, as this makes it vulnerable to a number of possible attacks, like the known-plaintext attacks and brute-force analysis. To enhance the security of this method, we are proposing a stream cipher that changes the key for every message or bit of information that is shared between the two parties involved in the communication. This ensures that a different ciphertext is produced even for the same message, every time it is encrypted, thus making the procedure immune to these possibilities.**

To achieve this, in our work, we have proposed a method of producing pseudo-random numbers generated from a seed value known beforehand to both the users. Using this pseudo-random generator, we can change the key every-time a message is shared between the sender and the receiver. The generator also depends upon the index of the message being communicated, making it even more difficult to break the cipher, if the knowledge about the sequence of the communication is unknown.

***Index Terms*— Symmetric Encryption, Pseudo-Random Generator, Stream Cipher, Forward Secrecy.**

I. INTRODUCTION

The human beings are often referred to by the experts as 'social animals'. One of the key reasons for this is our need to communicate with each other. But, often there comes a need to establish a localized communication with certain people around us, without the explicit disclosure of the thoughts being shared. This need was perceived quite early by our ancestors. And so, the contemporary brighter minds came up with some extraordinarily innovative methods to cater this need. These methods or algorithms gave birth to the art of selective communication that we know today as 'Cryptography'.

The basic need of any modern cryptographic algorithm is to obtain or generate a sufficiently random parameter which cannot be easily determined without the knowledge related to a certain private piece of information. The randomness of this parameter can be used to increase the computational complexity of breaking the cipher to a great extent. This is a vital and an integral part of any modern-day communication system or protocol, for the sake of increased security of the interaction between the two parties involved in communication. The need becomes even more apparent when the growth in the computational power available to the eavesdropper is considered. More is the degree of randomness of this parameter, harder it is for the attacker to generate the plaintext from the given ciphertext via a Brute-Force attack on the cipher.

The traditional or the conventional symmetric key encryption is one of the fastest modes of encryption, that can be deployed with a great computational ease and thus hold an upper hand in implementation, with respect to its other counterparts. But, symmetric encryption encounters certain fundamental drawbacks due to its design paradigms. To implement it efficiently in real time problems it is very important to address these drawbacks in the manner deemed necessary.

In this paper we have proposed a symmetric key cryptosystem that is designed to address and rectify a number of drawbacks of the traditional methods, in order to provide complete forward secrecy and confidentiality to the two parties involved in the communication. It should be noted that our algorithm does not provide any means to check the data integrity, implement authentication and ensure non-repudiation. Here, we also assume that the two parties are in a mutual agreement over a numerical parameter,that is used to generate different keys on both ends, for the sake of enhanced security.

II. ALGORITHM

In our algorithm, we have assumed that both the communicating members have the knowledge of a common shared secret number, referred to here as the 'seed'. The security of the entire following algorithm heavily depends on the confidentiality of this seed and is assumed to be sufficiently secure here. This seed may have been agreed upon via physical exchange of information or via asymmetric key encryption. Also, for the purpose of demonstration, we have implemented our algorithm using the XOR cipher to explain its working, though if it seems relevant, any other symmetric key algorithm may be used instead, with suitable modifications. A significant number of algorithms were attempted and out of them, the presented one was chosen. It is neither the quickest nor the shortest but is believed to be the best bargain between security, the simplicity of usage, the absence of a specialized table, and sensible execution. Once this is done we are in the position to initiate the following algorithm:

At the very beginning, the user will be provided with an option to choose between encryption of a message and its transmission or decryption of the message from a received transmitted file. Here we are importing/exporting the transmitted data from/in a binary text file which may be shared via any means even on an insecure channel.

Upon the selection of the option to encrypt, the user will be asked to enter the message and this message will be stored in an array. After that, a call to the function 'generate_key' will be made which prepares the key for the given instance of communication. It does so by making repeated calls to the function 'rand_key', which generates a large pseudo-random number, using the seed value, and the index of the communication, 'com_no'. It also changes the seed value upon each exit, to maintain the desired randomness of the 'result', the final random number. Then, we divide the digits of 'result' into the pairs of two and derive the corresponding ASCII characters for each pair thus obtained. This process is repeated as long as the size of the key is smaller than the maximum length specified by the users in the alias name 'MAX'. Once this is done, the message is XOR-ed with the key, in a cyclic fashion, and this encrypted text thus obtained is exported to the binary text file named 'transmission.txt'.

Similarly, if the option of decryption is selected, a similar call to the function generate_key is made as explained above, and results in the same key as generated on the transmission end, provided the indexing of the communication is maintained. Once the key is obtained, the received encrypted message is again XOR-ed with the key in a cyclic fashion, to obtain the original plaintext. This happens so, because of the unique property of the XOR function:

$$(A \oplus B) \oplus B = A$$

Corresponding pseudo-code is:

define Path as "C:/Users/Brijgopal Bharadwaj/Desktop/transmission.txt"
define as MAX 51
define a datatype of type unsigned long long int named big
declare keylen (to store value of length of key)
declare com_no (to count the number of communication) of type int
initialize com_no to 0.
Declare a,b and seed of type big and key[MAX] , msg[400] of type char

```
void SYMMETRIC()
{
    take the input of seed from the user and store it in the variable seed
    while(1)
    {

        Print "Enter your choice : 1. Encryption 2. Decryption 3. Exit
        CH=getch();
        If choice is 1 call Encode_main()
        Else if choice is 2 call Decode_main()
        Else display invalid choice.
    }
}

void Encode_main()
{
    Declare i of type int
    Declare ch of type char
    Open the file at path "PATH" in Writing mode.
    Increase the value of com_no by 1.
    Get the message from the user and store it in the array msg[].
    Call generate_key()
    Display the encoded message
    for(i=0;msg[i] not equal to NULL; i++)
    {
        ch=(msg[i] )XOR (key[i%(keylen/sizeof(char))]);
        display ch
        call the function char_to_bin(ch);
        write the bin array in the file
    }
    Close the binary file
}

void Decode_main()
{
    Declare ctr of type int and initialize it to 0.
    Open the file in read mode.
    Declare ch and str[8] of type char.
    Increase the value of com_no by 1.
    Display "Received Message"
    Call the function generate_key();
    while(fgets(str,9,fp) not equal to NULL)
    {
        ch=bin_to_dec(str);
        display ch
        ch=(ch ) XOR (key[ctr%(keylen/sizeof(char))])
        msg[ctr]=ch;
        increase the value of ctr by 1
    }
    Display the "Decoded message"
    Close the binary file.
}
```

```
void generate_key()
{
    Declare i of type int and initialize it to 0
    Initialize a to 0
    while(i is less than MAX-1)
    {
        if( not a)
        {
            b=rand_key(b)
            set a =b
        }
        key[i++]=a%100
        set a = a/100
    }
    key[i++]=NULL
    set keylen equal to i
    for(i=0;i<keylen;i++)
        display key[i]
}

big rand_key(big P)
{
    Declare next and result of type big
    seed equal to seed%1015;
    next equal to(seed) XOR (com_no)
    result equal to P

    next equal to next<<7
    next=next%15;
    result= (result)XOR(next)

    result = result + com_no
    result=result%1015;
    set seed = next;
    return result;
}
```

III. RESULTS AND DISCUSSION

In the following discussion, we analyze the results obtained from a basic ‘C' implementation of the algorithm described above. The XOR cipher used here works on the binary ASCII values of the characters present in the message or the key. It is possible here that the generated character, which belongs to the key or the result, may not be a printable entity due to the nature of the assigned character. Thus, instead of just printing the characters of the key or the ciphertext directly, we also provide their corresponding decimal ASCII value. For the characters that can’t be printed, we have used standard 3 alphabet codes for representation.

At the very beginning, the ‘seed’ value was randomly set to be 2321332. For the first instance of communication, the value of ‘com_no’ is 1. On the sender’s end, when the function-call was made for the ‘Encode_main’, the following message was entered for encryption:

“*We are located at (22.5N,65.8E). Come quickly*!”

Following this, a call was made to the function ‘generate_key’. It in-turn made repetitive calls to the function ‘rand_key’, each time ‘a’ became zero. Each call returned a large random number, which was then decomposed into two-digit numbers. Each of these two-digit numbers were then used to obtain the corresponding ASCII characters. This was done until the length of the compiled key was shorter than the specified length in ‘MAX’. The obtained results were:

Characters	**EM**	**ACK**	**CR**	**a**	**STX**	**LF**	**b**	**US**	**G**	**M**	**ETX**	**?**	**HT**
ASCII	25	06	13	97	02	10	98	31	71	77	03	63	09

Characters	**NUL**	**W**	**BEL**	**ETX**	**FS**	**V**	**!**	**GS**	**F**	**NAK**	**FF**	**T**	**>**
ASCII	00	87	07	03	28	86	33	29	70	21	12	84	62

Characters	**DC1**	**.**	**&**	**SP**	**FS**	**.**	**ETX**	**7**	**CAN**	**S**	**‘**	**‘**	**STX**
ASCII	17	46	38	32	28	46	03	55	24	83	39	39	02

Characters	**SP**	**R**	**&**	**6**	**SP**	**SOH**	**CR**	**ESC**	**?**	**BS**	**CR**	**NUL**	
ASCII	32	82	38	54	32	01	13	27	63	08	13	00	

The obtained key was then used for encryption, by performing a character-wise xor operation between the key and the input message in a cyclic fashion, until all the characters of the message were encoded. the generated ciphertext was as follows:

Characters	**N**	**c**	**-**	**NUL**	**p**	**O**	**B**	**s**	**(**	**.**	**b**	**K**	**l**
ASCII	78	99	45	00	112	111	66	115	40	46	98	75	108

Characters	**d**	**w**	**f**	**w**	**<**	**~**	**DC3**	**/**	**h**	**SP**	**B**	**x**	**BS**
ASCII	100	119	102	119	60	126	19	47	104	32	66	120	08

Characters	**$**	**NUL**	**RS**	**e**	**5**	**NUL**	**#**	**t**	**w**	**>**	**B**	**BEL**	**f**
ASCII	36	00	30	101	53	00	35	116	119	62	66	07	115

Characters	**U**	**;**	**E**	**]**	**L**	**X**	**,**						
ASCII	85	59	69	93	76	120	44						

This ciphertext was then stored in a file named *‘transmission.txt’*, in the binary format. The contents, as stored in the file were:

1001110	*0101000*	*1110111*	*1111000*	*0100011*	*0111011*
1100011	*0101110*	*0111100*	*0001000*	*1110100*	*1000101*
0101101	*1100010*	*1111110*	*0100100*	*1110111*	*1011101*
0000000	*1001011*	*0010011*	*0000000*	*0111110*	*1001100*
1110000	*1101100*	*0101111*	*0011110*	*1000010*	*1111000*
1101111	*1100100*	*1101000*	*1100101*	*0000111*	*0101100*
1000010	*1110111*	*0100000*	*0110101*	*1110011*	
1110011	*1100110*	*1000010*	*0000000*	*1010101*	

This file was then sent to the receiver's end, where its contents were extracted and were used along with the obtained key, same as that on the sender's end, for the XOR operation, in a cyclic fashion. This resulted in a successful decryption of the message and the following plaintext was obtained:

“We are located at (22.5N,65.8E). Come quickly!”

When the same message was again encrypted, this time with the value of the variable *‘com_no’* as ‘2’, the key generated was:

Characters	**_**	**ETB**	**/**	**-**	**%**	**SOH**	**M**	**SP**	**L**	**@**	**]**	**ACK**	**G**
ASCII	95	23	47	45	37	01	77	32	76	64	93	06	71

Characters	**SO**	**EM**	**CR**	**DLE**	**STX**	**EM**	**6**	**F**	**CAN**	**9**	**ACK**	**‘**	**+**
ASCII	14	25	13	16	02	25	54	70	24	57	06	39	43

Characters	**“**	**F**	**%**	**STX**	**CR**	**SYN**	**SI**	**P**	**FS**	**BEL**	**SP**	**NAK**	**7**
ASCII	34	70	37	02	13	22	15	80	28	07	32	21	55

Characters	**US**	**STX**	**HT**	**4**	**CAN**	**W**	**CR**	**BEL**	**_**	**3**	**CR**	**NUL**	
ASCII	31	02	09	52	24	87	13	07	95	51	13	00	

The corresponding ciphertext generated after encryption was:

Characters	**BS**	**r**	**SI**	**L**	**W**	**D**	**m**	**L**	**#**	**#**	**<**	**r**	**“**
ASCII	08	114	15	76	87	100	109	76	35	35	60	114	34

Characters	**j**	**9**	**l**	**d**	**“**	**1**	**EOT**	**t**	**6**	**FF**	**H**	**VT**	**GS**
ASCII	106	57	108	100	34	49	04	116	54	12	72	11	29

Characters	**ETB**	**h**	**GS**	**G**	**$**	**8**	**/**	**DC3**	**s**	**j**	**E**	**5**	**F**
ASCII	23	104	29	71	36	56	47	19	115	106	69	53	70

Characters	**j**	**k**	**j**	**_**	**t**	**.**	**,**						
ASCII	106	107	106	95	116	46	44						

When this was sent to the receiver’s end, it was again decrypted successfully to obtain the original plaintext. This exercise was performed to ensure that even for the same message, the generated ciphertext is different, and is sufficiently random.

The time complexity of the function ‘rand_key’ is O(1). It is so because of the absence of any iterative or recursive nature in the function. Whereas, the time complexity of the function ‘generate_key’ is O(n), where ‘n’ is the length of the key. The algorithm presented in our work has the characteristics of a stream cipher. It offers forward secrecy to the users involved in the communication, due to the mathematical inability of the attacker to compute the variable ‘next’ for any given iteration of the function ‘rand_key’ and ‘generate_key’, due to the lack of required necessary information.

The proposed cipher is designed to rectify a number of shortcomings of the original symmetric key system, in order to make it more secure, with a small trade-off on runtime. The following discussion tries to observe the security of the cipher against some of the possible attacks that can be performed on it, to gain some insight about the plaintext, or the keys.

In a ciphertext-only attack, the attacker has access to various encoded messages. He has no clue what the plaintext information or the secret key might be. The attack is considered successful if any amount of information regarding the underlying plaintext can be extracted, from the given ciphertext, or in some cases the key itself. This is the pinnacle of an attacker’s ambition, and so it should be ensured that the ciphertext does not divulge any significant data, when subjected to various cryptanalysis techniques. The algorithm proposed here has been subjected to the same, and to the best of our knowledge, is secured against such techniques.

In a known-plaintext attack, the eavesdropper/attacker has access to the ciphertext and its relating plaintext. He then tries to figure out the secret key or build up an algorithm which would enable him to decode any further messages. This gives the attacker considerably greater chances of breaking the cipher, than just by performing a ciphertext-only attack. This can be potentially used against our algorithm if the length of the known part of the sent message is greater than that of our key at that instance of communication. It is so, because then our algorithm will make a cyclic repetition of the key, and this will result in the revelation of the complete key sequence to the attacker. This will allow him to decipher all the data encrypted in that instance. But, on a brighter side he still won’t be able to predict the preceding and the succeeding keys without the knowledge of sufficiently long plaintext-ciphertext pairs in the following interaction, i.e. providing forward secrecy. This revelation for a given instance can also be ruled out if it is ensured that the size of the message being sent is kept shorter than that of the obtained key.

Our algorithm does not provide any security against a Man-In-The-Middle attack. In order to establish each other’s identity, any certification algorithm may be engaged before the deployment of our algorithm.

IV. CONCLUSION

The basic version of private key cryptosystem faces numerous security threats, due to its superficial security measures. In this paper we have proposed an algorithm which can be used to improve symmetric key cryptography where, we change the key every-time a message is shared between the sender and the receiver, in such a way that the previous and the succeeding keys do not share any direct correlation with each other, thus ensuring forward secrecy. It also ensures that the key being generated is also a function of the indexing of the communication sequence so as to increase the complexity of breaking the given cipher.

This security improvement is beneficial because the Symmetric key encryption finds application in many fields, including some other cryptographic algorithms which have the private key cryptosystem as one of their sub-steps. This development will ensure a better standard of security, to provide a comparatively safer and trustworthy mode of communication.

The underlying idea can be improved further-on if the need is so, to counter-act the ever-growing ease of computation. The algorithm can also be used in conjunction with other protocols/ciphers to incorporate advanced security measures like identity verification, chaotic maps for enhanced random behaviour, etc.

REFERENCES

[1] Smart, N. (December 30th, 2004). Cryptography: An Introduction, 3rd Edition, McGraw-Hill College.

[2] Kahate, A. (January 1st, 2008). Cryptography and Network Security, McGraw-Hill Education (India).

[3] Singh, S. (September 1999). The Code Book, Fourth Estate (United Kingdom).

[4] Wheeler, D., Needham R. (1994). TEA, a Tiny Encryption Algorithm, Cambridge University, England (United Kingdom).

[5] Singh, P., Shende P. (December 2014). Symmetric Key Cryptography: Current Trends, Chhatrapati Shivaji Institute of Technology (India)

[6] Rejani, R., Krishnan, D. (2015). Study of Symmetric key Cryptography Algorithms, Manonmanium Sundarnar University(India)

[7] Stallings W. (2006), Cryptography and Network Security: Principles and Practices, 4th Ed., Pearson Education

[8] Iqbal, S., Singh, S., Jaiswal, A. (May 2015). Symmetric Key Cryptography: Technological Developments in the Field, International Journal of Computer Applications (0975 – 8887) Volume 117 – No. 15

[9] Pandey, K., Rangari, V., Sinha, S. (July 2013). An Enhanced Symmetric Key Cryptography Algorithm to Improve Data Security, AISECT, Bhopal(India).

[10] Nath, A., Ghosh, S., Mallik, M. (July 2010). Symmetric key cryptography using random key generator, Proceedings of International conference on SAM2010 held at Las Vegas,Vol-2, and P-239-244.

APPENDIX

ASCII table: Winter, Dik T. (2010) [2003]. *US and International standards: ASCII*.