Article can be accessed online at http: //www.publishingindia.com

SudoKrypt: A Novel Sudoku based Symmetric Encryption Scheme

Anjali Deshmukh[*], Alin Nagraj[**], Mansi A. Radke[***]

Abstract

In this paper a novel symmetric encryption scheme, SudoKrypt, with a block size of 128 bits has been presented using a Latin square Sudoku as the key. For SudoKrypt, the key used is Hexadoku and the key size is 1024 bits. The proposed algorithm gives good diffusion due to high degree of randomness because of multiple options for encrypting the same value. In addition, the proposed approach overcomes the drawback of DES of having weak keys after parity drop operation. AES is computation bound. Proposed algorithm requires less CPU cycles as it is not computation bound and hence is found to have comparable speed to that of AES in terms of encryption and decryption time. Also, the algorithm is very simple to understand, implement and use. The algorithm can typically be used for offline communication where pdf and word documents are encrypted and sent to the receiver via email or other media. The key exchange between transmitter and receiver can take place through a standard key transmission algorithm like RSA.

Keyword: Symmetric Key Encryption; Sudoku; Hexadoku

Introduction

A Sudoku is 9X9 logic based combinatorial puzzle. The puzzle setter provides a partially completed grid, which typically has a unique solution. A completely filled Sudoku satisfies the property that each of the nine 3X3 grids in the Sudoku contains each number from 1 to 9 exactly once. Also, each row and each column in the Sudoku contain each number from 1 to 9 once.

	7	5		9				6
	2	3		8			4	
8					3			1
5			7		2			
	4		8		6		2	
			9		1			3
9			4					7
	6			7		5	8	
7				1		3	9	

Fig. 1: Example of a Sudoku Puzzle

The algorithm was initially developed by us for character encryption using a 9X9 Sudoku as the key. From this basic idea, the actual algorithm which encrypts the plaintext byte-wise has evolved. In the latter one, a 16 X 16 Sudoku also known as Hexadoku has been used as the key. The algorithm has been further modified to work on blocks of 128 bits as per NIST (National Institute of Standards and Technology) specifications. [8]

In the case of 9X9 Sudoku as the key, the Sudoku Solver is a part of the algorithm and is available with both the parties involved in the communication. The key size, though bigger than AES (Advanced Encryption Standard) and DES (Data Encryption Standard), cannot be claimed as an advantage as the key has relationship among its elements. Transmission of the big key on the other hand appears to be an overhead; however a minimum of only 17 clues need to be transmitted during the key exchange

[*] Honeywell IIPL, Software Development, Pune, Maharashtra, India. Email: anjalideshmukh2501@gmail.com
[**] Carnegie Mellon University, Computer Science Dept., Pittsburgh, USA. Email: alinnagraj@gmail.com
[***] Visvesvaraya NIT, Computer Science Dept., Nagpur, Maharashtra, India. Email: mansiaradke@gmail.com

for a 9X9 Sudoku. Key size is of 200 bits for the scheme which uses a 9X9 Sudoku as the key.

It has been proved by Gary McGuire et al. [2] that a minimum of at the least 17 clues are necessary for a Sudoku to have a unique solution. With only up to 16 keys, multiple answers are there for the Sudoku puzzle. This property has been used to reduce the key size. Only 17 appropriate clues are transmitted during key exchange. All the clues are numbers between 1 and 9 and each number takes maximum of 4 bits for storage. For 17 clues we need 17X4 bits i.e. 68 bits. For each clue the position (row index and column index in the sub-block) in the Sudoku needs to be mentioned. As each sub-block in the Sudoku is a 3X3 structure, 2 bits are required to store the row number and 2 bits for the column number. Thus one byte is sufficient for storing the position and the value both. Also, the concept of a changeover bit pattern has been introduced which is an 8 bit pattern of all 11111111. For each sub block, there is one byte for indicating the position in the sub block and the value there. If multiple clues are there in one sub-block, they are indicated by one byte each. When we move from sub block 1 to 2, we indicate with the changeover bit pattern. Similarly when we change from sub block 2 to sub block 3, sub block 7 to sub block 8 and so on we use the changeover bit pattern. Here an important thing to note is this pattern will never appear for any value in the Sudoku as 1111 represents a value 15 which is not a valid value for a 9X9 Sudoku entry. The maximum number of changeover bytes would be 8 for a 9X9 Sudoku. Hence the key to be exchanged, by any well-established key exchange algorithm like RSA[7], consists of 25 bytes i.e. 200 bits for a 9X9 Sudoku.

The total number of possible 9X9 Sudokus is 6.67×10^{21} as determined by Bertram Felgenhauer et al. [1]. This is our key space for the character encryption algorithm. The total number of possible 16X16 valid Sudokus has not been yet found outto the best of our knowledge. However, the value for 9X9 possible Sudokus itself is very huge and that of 16X16 Hexadokus is definitely much more than this.Hence our key space for the byte encryption algorithm is quite huge undoubtedly.

We describe our initial idea in detail in Section 2, followed by our proposed algorithm in Section 3. We elaborate on our experimental setup and results in section 4. Section 5 concludes the paper.

Character Encryption Algorithm (Basic Idea and Concepts with a Toy Example)

The key Sudoku also referred to as Sudoku 1 will be used for first 9 alphabets i.e. a to i. Similarly second Sudoku will be used for the next nine alphabets i.e. from j to r and so on. The indices of the Sudoku are as given in figure 2.

A. Encryption

As it is a character oriented cipher and there are 26 characters in English, for the cipher to be able to encrypt all 26 characters, we need at the least 26 possible encryptions.

1	7	5	2	9	4	8	3	6
6	2	3	1	8	7	9	4	5
8	9	4	5	6	3	2	7	1
5	1	9	7	3	2	4	6	8
3	4	7	8	5	6	1	2	9
2	8	6	9	4	1	7	5	3
9	3	8	4	2	5	6	1	7
4	6	1	3	7	9	5	8	2
7	5	2	6	1	8	3	9	4

Key Sudoku

(cell value + 1) mod 9

2	8	6	3	1	5	9	4	7
7	3	4	2	9	8	1	5	6
9	1	5	6	7	4	3	8	2
6	2	1	8	4	3	5	7	9
4	5	8	9	6	7	2	3	1
3	9	7	1	5	2	8	6	4
1	4	9	5	3	6	7	2	8
5	7	2	4	8	1	6	9	3
8	6	3	7	2	9	4	1	5

Permuted Sudoku

Fig. 2. Operation to Generate New Sudoku from the Key Sudoku

One Sudoku could encrypt 9 characters and hence we permuted this Sudoku by performing the following operation on each cell.(Cell value + 1)mod 9 and generated the second Sudoku for the next 9 characters. Similarly, we permuted the second Sudoku by the operation (cell value + 1) mod 9 and generated the third Sudoku. Also, if we wanted to include special characters, punctuations

like comma, full stop or spaces for that matter, we could use an additional 4th Sudoku by performing the same operation on 3rd Sudoku to generate the fourth one and use it for encryption.

Now, for the first alphabet "a", there are nine possible encryption values. The encrypted values are a three tuple consisting of <Sudoku number, column number, cell value>.

Now, for the first alphabet "a", there are nine possible encryption values. The encrypted values are a three tuple consisting of <Sudoku number, column number, cell value>. The proposed encryption scheme is explained with the help of table 1.

As we can see in the above diagram, there are 9 possible encryption values for the alphabet "A" which are:

(1 1 1) – Indicating Sudoku 1, column 1 and cell value 1, (1 2 7), (1 3 5), (1 4 2), (1 5 9), (1 6 4), (1 7 8), (1 8 3), and (1 9 6).

The diagram highlights the possible encryption values for the alphabet "d". Out of these 9 values, one is chosen randomly by a pseudorandom number generator which outputs a random number between 1 and 9. The table 1 shows all possible encryption values for each alphabet from a to i.

Table 1. Encryption Scheme for the First 9 Alphabets

A	111	127	135	142	159	164	178	183	196
B	116	122	133	141	158	167	179	184	195
C	118	129	134	145	156	163	172	187	191
D	115	121	139	147	153	162	174	186	198
E	113	124	137	148	155	166	171	182	199
F	112	128	136	149	154	161	177	185	193
G	119	123	138	144	152	165	176	181	197
H	114	126	131	143	157	169	175	188	192
I	117	125	132	146=	151	168	173	189	194

As we can see, each entry in this table is unique. This never conflicts as tables generated by two Sudokus can never be the same. Note that this is a case insensitive scheme.

Now, suppose the random function returns the value (147) for the letter 'd'. The binary coded decimal for (147) is as follows:0000 0001 01000111

We do a circular right shift by 4 bits on this bit string, to ensure that encrypted data gets further diffused. The bit string obtained after this operation is as follows:
0111 0000 0001 0100

Now, these 16 bits are to be laid row wise in 4 rows and 4 columns as follows:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

The next step is to pick these bits column by column and rearrange them as follows: 0000 1001 1000 1010. This completes the encryption step. The permutation box used above is

1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

B. Decryption

The decryption proceeds as follows: Suppose the value to be decrypted is: 0000 1001 1000 1010. We lay the bits column wise now and these 16 bits are arranged in 4 rows and 4 columns and we pick the data row wise to get

0 1 1 1 0 0 0 0 0 0 0 1 0 1 0 0.

The permutation box used here is the same as used in encryption. Then we perform a circular left shift by 4 bits as follows: 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 Suppose the cipher text to be decrypted is 0000 0001 0100 0111, then 0001 is the Sudoku number, 0100 i.e. 4 is the column number and 0111 i.e. 7 is the cell value. We know our key, hence we go to the appropriate Sudoku number 1, 2, 3, or 4 as required (1 is available and 2, 3 4, can be generated by the operation explained above in this section). In that Sudoku, we locate the cell value in that particular column by a hash function and the row number indicates which letter it is. For example, if the cipher text for an alphabet or letter is 0000 0001 0100 0111, then we need the first Sudoku i.e. the original key Sudoku. We go to the first Sudoku and we locate the value 7 in the 4th column by a hash function. The row number is 4 in this case and hence, the decrypted alphabet is 'd'. Row 1 indicates letter 'a', row 2 indicates letter 'b' and so on.

Table 2. Encryption of the Word "DAGGER" with Initial Algorithm using 9x9 Sudoku

Letter	*D*	*A*	*G*	*G*	*E*	*R*
Intermediate Cipher text	147	178	119	165	124	268
Final Cipher Text	9,138	129,19	128,11	9,26	8,18	129,48

Thus we can see that the scheme is completely reversible. It is simple yet hides the letter frequencies. For example, consider the word "dagger". This can be encrypted as shown in table 2. Though the alphabet 'g' is repeated, the encryption of both the 'g's is different. We need 2 bytes for each letter. Thus for every plaintext byte we generate cipher text of 2 bytes. This scheme is the base from where we started and got our final idea of byte-wise/block wise encryption using a 16X16 Sudoku/Hexadoku as the key. Note that our character encryption algorithm has its own flaws and is not the proposed scheme. It is only used to elaborate the concepts required and to explain the root of our idea.

C. Proposed Algorithm SudoKrypt

In a Hexadoku the 16 unique numbers are represented for convenience as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The completely filled Hexadoku has the same properties as a 9X9 completely filled Sudoku i.e. each number from 0 to F occurs only once in each sub block, each row and each column. The sender and the receiver agree on a key Hexadoku. We have used the term Sudoku for Hexadoku henceforth. We mean to refer to a Hexadoku only. Also, the original key Sudoku is the 0th Sudoku and first permuted Sudoku is the 1st Sudoku, 8th permuted Sudoku is the 8th Sudoku and so on. Also indices are from 0 to 15 for Sudoku henceforth.

In SudoKrypt, all the elements of the Sudoku i.e. 256 cell values need to be transmitted on the receiver side. To the best of our knowledge, there is no available research on how many appropriate minimum clues are necessary to generate a unique Hexadoku. Hence, in this case the Hexadoku solver is not needed on the transmitter and receiver end as the complete key is transmitted when the session starts.

2	1	6	3	A	5	C	E	F	4	7	D	9	8	B	0
5	E	C	D	8	B	0	3	2	9	A	1	F	4	6	7
A	8	F	B	D	7	9	4	E	0	6	5	C	2	1	3
9	0	7	4	2	F	6	1	3	C	8	B	E	A	D	5
3	6	2	7	E	D	B	F	A	1	C	4	8	5	0	9
8	F	4	C	5	A	2	9	B	D	0	3	1	E	7	6
B	A	1	E	0	8	7	6	9	5	F	2	3	C	4	D
D	9	0	5	1	3	4	C	8	6	E	7	B	F	2	A
E	3	9	1	7	4	F	5	D	B	2	8	6	0	A	C
F	D	5	6	C	9	8	2	0	7	3	A	4	1	E	B
C	B	8	0	3	1	E	A	5	F	4	6	7	D	9	2
7	4	A	2	6	0	D	B	C	E	1	9	5	3	8	F
1	7	D	9	B	C	3	8	4	A	5	0	2	6	F	E
4	2	B	F	9	6	5	0	1	3	D	E	A	7	C	8
6	C	3	8	4	E	A	D	7	2	B	F	0	9	5	1
0	5	E	A	F	2	1	7	6	8	9	C	D	B	3	4

Fig. 4. Hexadoku Puzzle Completely Solved

D. Encryption in Proposed Approach

8 bits represent 256 numbers. One Sudoku is sufficient to encrypt 16 numbers. So, to encrypt 256 possible values of plaintext byte, we need 16 Sudokus. These are generated from the key Sudoku by the operation (cell value + 1) mod 16 on each cell, (cell value + 2) mod 16 on each cell and so on until (cell value + 15)mod 16 for the 15th permuted Sudoku. For each such number, we have 16 possible encryptions and we choose one randomly. For our block cipher which works on 128 bits of data. The encryption scheme is shown diagrammatically in figure 6. The Feistel function is depicted in figure 5. We have 16 blocks of 8 bits each. We process each of these 16 blocks as shown in Algorithm 3.

For example suppose the byte B of our 128 bit block to be encrypted is 00001100. We perform the following operations on it. The variables used are B, q, r, m, n, val, num, v, L, R, row index, column index, num byte, rowcolumn byte

1. Divide the number represented by B by 16. We mark the quotient as say q and the remainder as say r. We XOR the four corners of the original key Sudoku and obtain a number say 'm'. We then XOR the four numbers at the centre of the Sudoku and call the result as say n. We check the value in the row m and

column n of the original key Sudoku. Let the value be say v

```
Function encrypt(plaintext):
  i=xor four corner values from key_Sudoku;
  i=xor four centre values from key_Sudoku;
  v=key_sudoku(i,j);
  val=permute sudoku(v) and xor 4 corner values

   Sudoku_step(plaintext,val);
   Intermediate=Permute_step(intermdiate);
   final=Feistel_step(intermediate);
   Return final;

 Function Sudoku_step(plaintext):
   For k = 1 to 8 taking two bytes of plaintext at a
   time
     byteval=byte1;
     Quotient=byteval/16;
     Remainder=byteval%16;
     Intermediate(k)=quotient XOR val
     Shift_left4(intermediate);
     Permute_Sudoku(quotient);
     intermediate(k+8)=Randomly select i,j pair
     having value 'remainder' in the cell
     Byteval=byte1;
     Quotient=byteval/16;
     Remainder=byteval%16;
     Intermediate(k) = quotient XOR val
     Permute_Sudoku(quotient);
     intermediate(k+9)=Randomly select i,j pair
     having value 'remainder' in the cell
     return intermediate;

 function permute_step(intermediate):
   circular_right_shift4(intermediate);
   for i=1 to 8:
     append ith bit of each byte to intermediate;
   return intermediate;

 function feistel_step(intermediate):
   for i= 1 to 12:
     left=first 8 bits of 16bitword(i);
     right=last 8 bits of 16bitword(i);
     for j= 1 to 16:
       temp=left XOR (i,j) value at seq(j);
       swap(temp,right);
       swap(temp,right);
       append temp,right to final;
     return final;

 seq={2,7,1,11,16,13,4,6,8,12,9,3,10,15,5,14}
```

Algorithm 3: Encryption in Proposed Scheme

2. We go to the vth permuted Sudoku and XOR the four corners of this Sudoku to obtain a result value val. We XOR val with the quotient q obtained in step 1 to get the result say num. This forms the last 4 bits of the first byte of the intermediate cipher text for L.
3. In this case the represented number by the 8 bit pattern is 12. The quotient is 0 after dividing by 16 and remainder is 12. So, we use the 0th or the key Sudoku as the Sudoku number and obtain the last nibble of the first byte of the encrypted L. The remainder r obtained in step 1 is the cell value we search for in each row and get 16 values of (i,j).
4. Of these 16 (i,j) pairs we choose one randomly by a pseudorandom generator which selects a number randomly between 0 and 15.
5. The encryption of each byte is given by a 3 tuple as <num, row index, column index>. Since, the row index and column index each need only 4 bits for storage; we take the row number, left shift it by 4 bits and OR it with the column index to get a byte. Hence our encryption becomes <num byte, rowcolumn byte>. Note here that for each byte we have the first four bits of the num byte as 0000 since Sudoku number needs only 4 bits for its representation as they range from 0 to 15.
6. These steps from 1 to 5 are followed for each of the 16 bytes of data. So in total we get 32 bytes. However we pack these 32 bytes as follows to obtain 24 bytes.

Byte 1: (Sudoku no for 1st byte, Sudoku no for 2nd byte)
Byte 2: (Sudoku no for 3rd byte, Sudoku no for 4th byte)
……. and so on
Byte 8: (Sudoku no for 15th byte, Sudoku no for 16th byte)
Byte 9: (value of i for first byte, value of j for first byte)
Byte 10: (value of i for 2nd byte, value of j for 2nd byte)
……and so on
Byte 24: (value of i for 32nd byte, value of j for 32nd byte)

7. Now we perform a circular right shift by 4 bits. Whatever bit pattern we get we arrange it in 16X12 matrix by laying the numbers row wise and pick the bits column wise.
8. We have 24 bytes with us. We perform the further operations on each of the 12 blocks of 16 bits as mentioned in the further steps.
9. These 16 bits are passed through a fiestel like function. (This is not exactly a fiestel function but is highly inspired by the same. We refer to it as the fiestel function hence forth in our discussion). We consider v, and each row has a value of row and column (i,j) where v is present. We have 16 such (i,

j) pairs - one in each row. We take the pairs in the sequence given by the following permutation box.

2	7	1	11	16	13	4	6	8	12	9	3	10	15	5	14

So, we take (i2, j2) as the first pair to XOR it with our left byte L from the 16 bits data being considered. Then we exchange the left half L and the right half R of the encrypted data. We perform XOR on the left half of the data with the key (i7, j7) and get the result. We again exchange the two halves L and R and apply XOR operation on the L part of the encrypted data with key (i1, j1) and so on.

10. The final output after 16 rounds of XOR in the fiestel function is the encrypted data of 16 bits. All the 12 groups of 16 bits are passed independently through the fiestel function. These operations on each of the 12 groups of 16 bits can potentially be done in parallel as there is no data dependency. However, the overhead of spawning parallel threads is more than the actual computation time is what we observed from our experiments using Open MP library.
11. The output of the fiestel function is combined to form a block of 192 bits of data. Thus the encryption increases the size of the data by 1.5 times.

E. Decryption in Proposed Approach

The decryption is performed on 192 bits of ciphertext at a time as shown in Algorithm 4.

```
function decrypt(ciphertext):
  I=xor four corner values from key_Sudoku;
  J=xor four centre values from key_Sudoku;
  V=key_sudoku(i,j);
  Val=permute_sudoku(v) and xor 4 corner values;
  intermediate=reverse_feistel_step(ciphertext);
  intermediate=reverse_permute(intermediate);
  final=reverse_sudoku_step(intermediate,val);
  return final;

function reverse_fiestel_step():
  for i= 1 to 12:
      left=first 8 bits of 16bitword(i);
      right=last 8 bits of 16bitword(i);
      for j= 1 to 16:
        temp=left XOR (i,j) value at seq(j);
        swap(temp,right);
        swap(temp,right);
        append temp,right to final;
  return final;

seq={14 ,5,15,10,3,9,12,8,6,4,13,16,11,1,7,2}

function inverse_permute(intermediate):
  for i= 1 to 24:
      append ith bit of each 24bit segment to
      intermediate;
circular_left_shift4(intermediate);
  return intermediate;

function inverse_sudoku_step(intermediate,val):
  for k= 1 to 8:
    quotient= first 4bits of intermedita(k) XOR
    val;
    permute_sudoku(quotient);
    (i,j)= (first 4 bits of intermediate(k+8), last
    4 bits of intermediate(k+8));
    Remainder= value at (i,j) in the permuted
    Sudoku;
    Plaintext(i)=quotient*16+remainder;
    quotient= last 4bits of intermedita(k) XOR val;
    permute_sudoku(quotient);
    (i,j)= (first 4 bits of intermediate(k+9), last
    4 bits of intermediate(k+9));
    Remainder= value at (i,j) in the permuted
    Sudoku;
    Plaintext(i+1)=quotient*16+remainder;
  Return plaintext;
```

Algorithm 4: Decryption in Proposed Approach

We explain the same step wise as follows:

1. While decrypting, we consider 192 bits i.e. 24 bytes at a time.
2. We apply the fiestel function on each byte with the keys (i,j) now in the reverse sequence.

14	5	15	10	3	9	12	8	6	4	13	16	11	1	7	2

The sequence is given by the permutation box above.

3. We then arrange the data column wise in a 16 X 12 matrix and pick the data row wise.
4. We then perform circular left shift by 4 bits.
5. Now we have 24 bytes and we know their arrangement.
   Byte 1:(Sudoku no 1st byte, Sudoku no 2nd byte)
   Byte 2:(Sudoku no 3rd byte, Sudoku no 4th byte)
   ………. and so on
   Byte 8:(Sudoku no 15th byte, Sudoku no 16th byte)
   Byte 9:(value of i first byte, value of j first byte)
   Byte 10:(value of i 2nd byte, value of j 2^{nd}byte)
   ……and so on
   Byte 24:(value of i 32nd byte, value of j 32nd byte)
6. We obtain first (i,j) pair value from the 9th byte and its corresponding Sudoku number from the first 4 bits of the first byte. Similarly second (i,j) pair value

we obtain from the 10th byte and corresponding Sudoku number from the last 4 bits of the first byte and so on.

7. We calculate that byte value as 16* Sudoku number + value at (i,j) in that Sudoku.
8. We perform these steps for all the 16 bytes we need and our 128 bit block of plain text is ready.

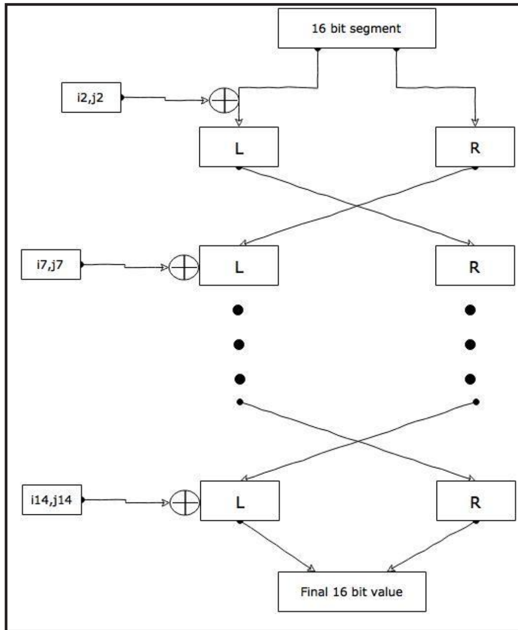

Fig. 5: The Feistel Function

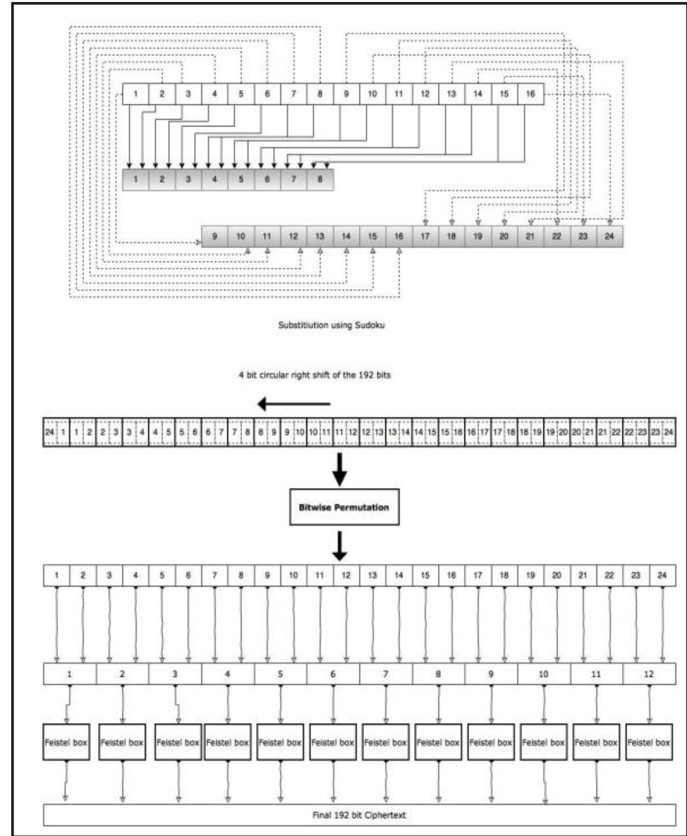

Fig. 6: Steps in the Encryption Process

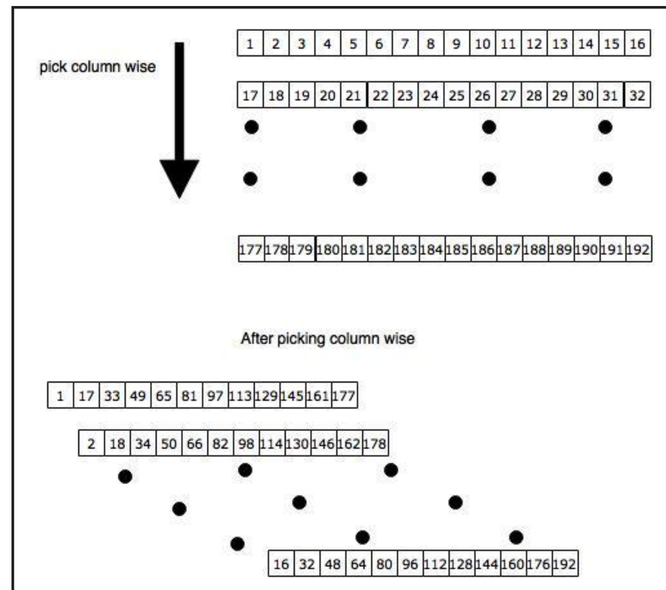

Fig. 7: Permutation Function

Experimental Results

We have compared our algorithm with the current standard for symmetric encryption i.e. the AES algorithm. One advantage of our algorithm is that it does not need any kind of substitution boxes which are used by AES and DES and their variants. We derive all our round keys from the key Sudoku. We also have overcome the drawback of DES of weak keys as we do not generate newer keys for the Feistel function by performing operations on the key bits like AES and DES and their variants; however our keys of the Feistel function do depend on the key Sudoku. Our algorithm is extremely simple, easy to implement and use and has the desirable properties of a symmetric cipher. The comparison of SudoKrypt (proposed algorithm) with AES in terms of encryption and decryption time is mentioned in figure 8. We have randomly generated a key which is then agreed by both sender and receiver and is transmitted using RSA key exchange algorithm. The machine on which we performed the experiments has a configuration of 2.4 Ghz, intel core i7 processor (i7-5500U) 8 GB RAM. The implementation has been done in C language

The table 3 shows the time required for encryption and decryption for each of the variants of AES, triple DES, Blowfish and SudoKrypt algorithm. The comparison of SudoKrypt with 128 bit key size AES is shown in the graph format in figure 8.

Table 3. Comparison of SudoKryptwith AES and its Variants, Triple DES and Blowfish

Algorithm	*Encryption Time (in seconds) Average of 10 randomly selected 128 bit data*	*Decryption Time (in seconds) Average of 10 different randomly selected data blocks of 128 bits*	*Block Size Pro-cessed (bits)*	*Speed in MB/sec*
SudoKrypt	0.015927	0.006032	128	1.737 X 10^-3
AES (128 bit)	0.014225	0.008277	128	1.356 X 10^-3
AES (192 bit)	0.010294	0.008369	128	1.635X 10^-3
AES (256 bit)	0.015193	0.007679	128	1.334 x 10 ^-3
Triple-DES	0.019716	0.007246	(2 X of 64 = 128)	1.132 X 10^-3

The X axis shows the size of files in KB. The Y axis shows the encryption time required in seconds. As we can see the proposed algorithm performs better on small file sizes and the speed drops as the file size increases.

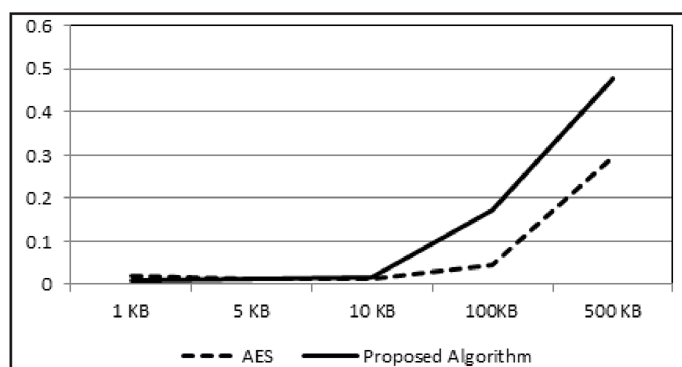

Fig. 8. Comparison of SudoKrypt with 128 bit AES on Different File Sizes (ECB mode)

Conclusion

We have successfully designed and implemented a symmetric block cipher SudoKrypt adhering to the NIST standards [8] and evaluated it against the established benchmarks. Our algorithm is comparable to AES in terms of encryption and decryption time as shown in figure 8. The transmission time dominates the encryption and decryption time and in our case the transmission time would be more as data size increases by 1.5 times. Our key size is 1024 bits which is big for a symmetric algorithm. In SudoKrypt for each byte we have 16 possible encryptions. For every such choice made, the next byte has 16 possible encryptions and so on. So in total we have 2^{128} possible encryptions of the same block. In addition, the possible (i,j) combinations is 16X16 and we use only 16 of those and this further adds to the number of possibilities. The properties of confusion and diffusion are fulfilled by our algorithm.

References

1. Bertram, F., & Jarvis, F. (2006). Mathematics of Sudoku I. *Mathematical Spectrum*, 39(1), 15-22.
2. McGuire, G., Tugemann, B., & Civario, G. (2014). There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem via hitting set enumeration. *Experimental Mathematics*, 23(2), 190-217.
3. Daemen, J., & Rijmen, V. (1999). AES proposal. *Rijndael Block Cipher*.
4. Forouzan, B. A., & Mukhopadhyay, D. (2011). *Cryptography and Network Security* (SIE). McGraw-Hill Education.
5. Stallings, W. (2006). *Cryptography and Network Security*, 4/E. Pearson Education India.
6. Daemen, J., & Rijmen, V. (2013). *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.
7. Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key crypto-systems. *Communications of the ACM*, 21(2), 120-126.
8. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4
9. https://github.com/B-Con/crypto-algorithms
10. https://github.com/kinghajj/hexadoku/blob/master/src/hexadoku/RandomBoard.java

Appendix A

The appendix A gives the pseudo code for encryption and decryption algorithms used in the toy example.

```
Function char_encryption(alpha):

   Intermediate=Sudoku_step(alpha);

   Final= permute(intermediate);

   Return final;
```

Algorithm 1: Character Encryption

```
Function Sudoku_step(alpha):
   Sudoku indices i,j
   If(alpha<'j'):
         J=Randomly select any one of the 9 values
         ati=(alpha-'a')%9
   Else if alpha<'s' :
         Generate Sudoku(1);
         J=Randomly select any one of the 9 values at
         i=(alpha-'a')%9;
   Else :
         Generate Sudoku(2);
         J=Randomly select any one of the 9 values at
         i=(alpha-'a')%9;

Function permute(inter):
   Right_circular_shift4(inter);
   For i =1 to 4:
         Append ith bit of each nibble to final;
   Return final;
```

Algorithm 2: Character Decryption

```
Function char_decrypt(cipher):

   Intermediate=inverse_permute(cipher);
   Plaintext=inverse_sudoku_step(intermediate);
   Return plaintext;

Function inverse_permute():
   For i =1 to 4:
         Append ith bit of each nibble to
   intermediate;
   Circular_left_shift4(intermediate);
   Return intermediate;

Function inverse_sudoku_step(inter):
   Num=Sudoku number from the first byte;
   Permute_sudoku(num);
   Character=alphabet at value 'a'+ i*num;
   Return character;
```