Making AES Stronger: AES with Key Dependent S-Box

Krishnamurthy G N, V Ramaswamy

Bapuji Institute of Engineering and Technology, Davangere-577004, Karnataka, India

Summary
With the fast evolution of digital data exchange, security information becomes much important in data storage and transmission. In this paper, we show a new property of Advanced Encryption Standard (AES)[1],[2],[3] using S-box and Inverse S-box. We also show how this property can be used to make the S-box key dependent[5],[6],[7],[9],[24] and hence make AES stronger. This has been done without changing the basic operations of AES. The importance lies in the fact that the S-box is made Key dependent without changing its values (ranging from 00 to FF) and without touching Inv-S-box. Detailed explanations of implementation are given.

Key words:
Cryptography, Encryption, Advanced Encryption Standard (AES), Key dependent S-box, Inverse S-box, Key expansion

1. Introduction

In October 2000, after a four year effort to replace the aging DES, NIST announced the selection of Rijndael[1],[2] as the proposed AES (NIST 2004). Draft of the Federal Information Processing Standard (FIPS) [3] for the AES was published in February 2001, Standardization of AES was approved after public review and comments, and published a final standard FIPS PUB-197 [3] in December 2001. Standardization was effective in May 2002 (NIST 2004).
Rijndael[1],[2] is a block cipher developed by Joan Daemen and Vincent Rijmen[1]. The algorithm is flexible in supporting any combination of data and key size of 128, 192, and 256 bits. However, AES merely allows a 128 bit data length that can be divided into four basic operation blocks. These blocks operate on array of bytes and organized as a 4×4 matrix that is called the state. For full encryption, the data is passed through Nr rounds (Nr = 10, 12, 14) [1], [2], [3]. These rounds are governed by the following transformations:

(i) SubByte transformation: Is a non linear byte Substitution, using a substation table (S-box), which is constructed by multiplicative inverse and affine transformation. It provides nonlinearity and confusion.
(ii) ShiftRows transformation: Is a simple byte transposition, the bytes in the last three rows of the state are cyclically shifted; the offset of the left shift varies from one to three bytes. It provides inter-column diffusion.
(iii) MixColumns transformation: Is equivalent to a matrix multiplication of columns of the states. Each column vector is multiplied by a fixed matrix. It should be noted that the bytes are treated as polynomials rather than numbers. It provides inter-byte diffusion.
(iv) AddRoundKey transformation: Is a simple XOR between the working state and the roundkey. This transformation is its own inverse. It adds confusion.

The encryption procedure consists of several steps as shown in Fig. 1. After an initial addroundkey, a round function is applied to the data block (consisting of SubBytes, Shiftrows, Mixcolumns and AddRoundKey transformation, respectively). It is performed iteratively (Nr times) depending on the key length. The decryption structure as shown in Fig. 2 has exactly the same sequence of transformations as the one in the encryption structure. The transformations Inv-SubBytes, Inv-ShiftRows, Inv-MixColumns, and AddRoundKey allow the form of the key schedules to be identical for encryption and decryption.
The AES algorithm [1], [2] is designed to use one of three key sizes (Nk). AES-128, AES-196 and AES-256 use 128 bit (16 bytes, 4 words), 196 bit (24 bytes, 6 words) and 256 bit (32 bytes, 8 words) key sizes respectively. In this paper we will only emphasize on AES-128. The AES -128 key expansion algorithm, takes as an input a four word (16 bytes) key, produces a linear array of forty four words (176 bytes) keys. This is sufficient to provide a four word round key for the initial AddRoundKey stage and each of the 10 rounds of cipher.
This paper introduces a new, key-dependent Advanced Encryption standard algorithm, AES-KDS, to ensure that no trapdoor is present in the cipher and to expand the key-space to slow down attacks.
The paper is organized as follows: Section 2 presents the proposed AES-KDS. Section 3 explains timing and security aspects. Section 4 shows the experimental results. Section 5 summarizes and concludes the paper. References are given in Section 6.

Manuscript received September 5, 2008.
Manuscript revised September 20, 2008.

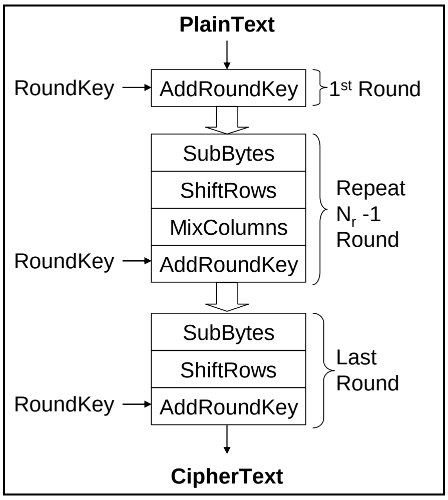


Fig. 1 AES algorithm- Encryption Structure

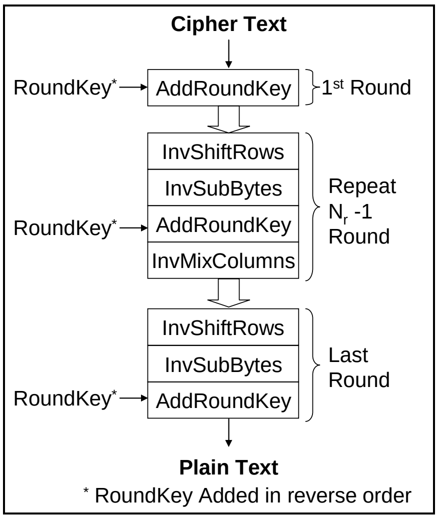


Fig. 2 AES algorithm- Decryption Structure

Many people have tried to modify AES algorithm to improve its performance [24]. More relevant to our work is a technique (Fahmy et al., 2005; Fahmy, Shaarawy, Hadad, Salama and Hassanain, SEITT 2005). In Fahmy[5] et al. (2005), an attempt is made to make AES key dependent[5],[6],[7],[9],[23] (KAES)[5]. In that the AES S-box is completely replaced by a new S-box. This eliminates completely Inverse S-box, which violates AES design and hence requires thorough analysis regarding its security, because AES S-box is tested thoroughly for linear, differential and algebraic attacks.

2. AES-KDS

AES-KDS is block cipher in which the block length and the key length are specified according to AES specification: three key length alternatives 128, 192, or 256 bits and block length of 128 bits. We assume a key length of 128 bits, which is likely to be the one most commonly implemented.

The encryption and decryption process AES-KDS resembles that of AES with the same number of rounds, data and key size. The round function resembles that of AES, but is composed of 5 stages rather than 4 stages. The extra stage named Rotate S-box is introduced at the beginning of the round function. The other four stages remain unchanged as it is in the AES and follow the Rotate S-box stage. However, the decryption process will have only 4 stages as in he AES. But the InvSubBytes operation is modified to nullify the effect of the Rotate_S-box operation of encryption. This is followed by a description of key expansion and generation of shift offset-matrix.

The input to the encryption and decryption algorithms is a single 128-bit block. This block is depicted as a square matrix of bytes. This block is copied into the state array, which is modified at each stage of encryption or decryption. After the final stage, state is copied to an output matrix. Similarly, 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words: each word is four bytes and the total key schedule is 44 words for the 128-bit key, a round key similar to a state. The process of encryption and decryption is as depicted in Fig. 3 and Fig. 4 respectively.

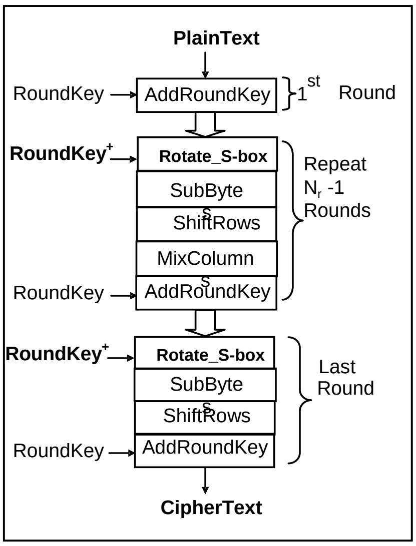


Fig. 3 AES-KDS algorithm- Encryption Structure

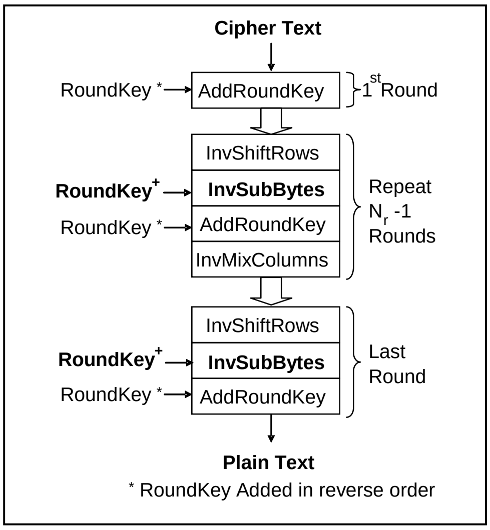


Fig. 4 AES-KDS algorithm- Decryption Structure

2.1 Rotate_S-box and SubBytes / InvSubBytes transformations

AES-KDS uses rotated AES S-box for its *SubBytes[8]* operation. To show how AES-KDS works, let us see how *Rotate_S-box* and *SubBytes / InvSubBytes* transformations work. A detailed study and analysis of AES S-box and Inverse S-sox reveals the following property. Consider AES S-box as shown in *Fig. 5*.

		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	3B	52	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

Fig. 5 AES S-box

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	**D4**	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Fig. 6 Inverse S-box

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	FF	F3	D2	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64
	1	5D	19	73	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE
	2	5E	0B	DB	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91
	3	95	E4	79	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65
	4	7A	AE	08	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B
	5	BD	8B	8A	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86
	6	C1	1D	9E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE
	7	55	28	DF	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0
	8	54	BB	16	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE
	9	D7	AB	76	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C
	A	A4	72	C0	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71
	B	D8	31	15	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB
	C	27	B2	75	09	83	2C	1A	1B	6E	5A	A0	3B	52	D6	B3	29
	D	E3	2F	84	53	**D1**	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A
	E	4C	58	CF	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50
	F	3C	9F	A8	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10

Fig. 7 S-box rotated left 125(or 7D in Hex) times

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	**51**	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Fig. 8 Inverse S-box demonstrating substitution for D1

In the `SubBytes` step, each byte in the state is replaced with its entry in the S-box;

b_{ij} = *S-box(a_{ij}).*

This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over **GF**(2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), and also any opposite fixed points.
Consider a byte, say D4 of the state. This will be replaced by 48(in Hex) as shown in *Fig. 5.*

48(Hex)=S-box(D4)

During Decryption the *InvSubBytes[8]* operation performs the inverse operation using Inverse S-box as shown in *Fig. 6*.

a_{ij} = Inv-S-box(b_{ij}).

So the value 48(in Hex) will be replaced by the original value D4 as shown in the figure below.

D4=Inv-S-box(48)

Now suppose we rotate the S-box left by a value say 125(or 7D in Hex). The new S-box will be as shown in *Fig. 7*.

Now suppose if we consider the same input D4, the rotated S-box will give a value D1 as shown in the figure. During Decryption the *InvSubBytes* operation performs the inverse operation using Inverse S-box. So Inverse S-box will produce a result 51(in Hex) as shown in *Fig. 8*.

51(in Hex)=Inv-S-box(D1)

The original value was D4 but what we are getting is 51(Hex). This leads to a wrong decryption. After a thorough analysis we could find out a way to get back the original value without changing the Inverse S-box. We can get back the original value just by subtracting a value used to rotate the S-box from the result obtained out of *InvSubBytes* operation.

So, a_{ij} = (Inv-S-box(b_{ij})-Number of times S-box bytes rotated) mod 256(FF+1 in Hex)

i.e.,
(InvS(D1)-7D) mod 256(or FF+1 in Hex) = (51- 7D) mod (FF+1) = D4.

This property holds good for all possible 256 rotations. Hence this property can be used to make the S-box key dependent [5],[6],[7],[9],[25].

The Rijndael S-box was specifically designed to be resistant to linear and differential cryptanalysis. This was done by minimizing the correlation between linear transformations of input/output bits, and at the same time minimizing the difference propagation probability. In addition, to strengthen the S-box against algebraic attacks, the affine transformation was added. In the case of suspicion of a trapdoor being built into the cipher, the current S-box might be replaced by another one. The authors claim that the Rijndael cipher structure should provide enough resistance against differential and linear cryptanalysis, even if an S-box with "average" correlation / difference propagation properties is used. This is the reason for keeping AES S-box values unchanged while making it key dependent.

Now by making S-box key dependent[5],[6],[7],[9],[25] AES will be much stronger[8]. We will now show how the above property of S-box can be used to make it key dependent using either of the following three cases depending on the level of security requirement. For moderate level security requirement Case 1 can be employed. For high security requirements Case 2 can be adopted. For very high level security Case 3 and Case 4 can be used.

Case 1:
Here different round keys are generated using a key expansion algorithm which is similar to that of AES key expansion algorithm. The round keys thus generated will used for finding a value that is used to rotate the S-box. The same round keys are used for *AddRoundKey* stage as well. Suppose for a particular round j, if the round key value is

2D9578565E262AA56F5F904A0B955B27 (each byte represented by 2-Hex digits).

The last byte 27(Hex) is used to rotate the S-box. The resulting S-box is used during the Subbyte operation.

Case 2:
Here different round keys are generated using a key expansion algorithm which is similar to that of AES key expansion algorithm. The round keys thus generated will used for finding a value that is used to rotate the S-box. The same round keys are used for *AddRoundKey* stage as well. Suppose for a particular round j, if the round key value is

06ACB47D588A9ED837D50E923C4055B5 (each byte represented by 2-Hex digits).

Here XOR operation of all the bytes is taken.

15(Hex)=06^AC^B4^7D^58^8A^9E^D8^37^D5^0E^92^3C^40^55^B5 (^ symbol used for XOR)

The resulting byte value 15(Hex) is used to rotate the S-box. The resulting S-box is used during the *SubBytes* operation. The advantage here is the rotation value is now dependent on entire round key rather than only on the last byte. The disadvantage is that it consumes little extra time. The following pseudo code describes the encryption operation for this case.

```
void encrypt(unsigned char state[4][4],unsigned
char key[16],unsigned char s_box[16][16],unsigned
long int expanded_key[])
{
       int round;
       unsigned long int mask=0xff;
       add_round_key(0,state,expanded_key);

       for(round=1;round<=9;round++)
       {
       rotate=(expanded_key[round*4]^expanded_ke
y[round*4+1]^expanded_key[round*4+2]^expanded_key
[round*4+3])&mask;
              create_s_box(s_box,rotate);
 // function to rotate S-box to left by
```

```
// a value equal to rotate
                substitute_bytes(state,s_box);
                shift_row(state);
                mix_column(state);

        add_round_key(round*4,state,expanded_key)
;
        }
        Rotate=(expanded_key[40]^expanded_key[41]
^expanded_key[42]^expanded_key[43])&mask;
        create_s_box(s_box,rotate);
        substitute_bytes(state,s_box);
        shift_row(state);
        add_round_key(40,state,expanded_key);
}
```

Case 3:

Here two sets of round keys are generated using a key expansion algorithm which is similar to that of AES key expansion algorithm. One set of round keys thus generated will used for finding a value that is used to rotate the S-box. The second set of round keys are used for *AddRoundKey* stage. From the first set of round keys, suppose for a particular round j, if the round key value is

EE0AF824 B02CD281 DF7342CB D4E619EC (each byte represented by 2-Hex digits).

The last byte EC(Hex) is used to rotate the S-box. The resulting S-box is used during the *SubBytes* operation. The advantage here is that it increases key expansion time and the rotation value is now dependent on round key other than what is used in *AddRoundKey* stage. The disadvantage is that it consumes extra time for generating new round key.

Case 4:

Here two sets of round keys are generated using a key expansion algorithm which is similar to that of AES key expansion algorithm. One set of round keys thus generated will used for finding a value that is used to rotate the S-box. The second set of round keys are used for *AddRoundKey* stage. From the first set of round keys, suppose for a particular round j, if the round key value is

C556E6B8 9021DC53 DB002238 6EB86774 (each byte represented by 2-Hex digits).

Here XOR operation of all the bytes is taken.

F7(Hex)=C5^56^E6^B8^90^21^DC^53^DB^00^22^38^6E^ B8^67^74 (^ symbol used for XOR)

The resulting bye value F7(Hex) is used to rotate the S-box. The resulting S-box is used during the *SubBytes* operation. The advantage here is the rotation value is now dependent on round key other than what is used in *AddRoundKey* stage. And also rotational value is dependent on the entire new round key rather than only on the last byte. The disadvantage is that it consumes little time extra.

The remaining 3 stages namely, *ShiftRows*, *MixColumns* and *AddRoundKey* transformations will remain as they are in the AES algorithm.

2.2 AES-KDS key expansion

One of the following two types of key expansion[1],[2],[8] is used in the AES-KDS algorithm.

Type 1:

The AES-KDS key expansion algorithm, takes as an input a four word (16 Bytes) key. In this case, first XOR operation of all the bytes of the key is carried out and the resulting 8-bit (byte) value is used for shifting the S-Box. This shifted S-Box is used to generate 11 subkeys, each of 4 words in length, totally a linear array of forty four words (176 Bytes). This is sufficient to provide a four word round key for the initial *AddRoundKey* stage and each of the 10 rounds of cipher. These round keys are also used for finding a value for rotating the S-box. The following pseudo code describes the expansion.

```
        unsigned char key[16]=1234567890ABCDEF;
        unsigned char temp=0;
        FILE *ky1;
        unsigned int rotate;
        for(i=0;i<16;i++)
        temp=temp^key[i];
        rotate=temp;
        create_s_box(s_box,rotate);
        key_expansion(expanded_key,key,s_box);
// as in original AES
        for(i=0;i<44;i++)
        fprintf(ky1,"%lx ",expanded_key[i]);
```

This expanded key is for first two cases, Case 1 and Case 2 of encryption procedure described above based on the requirements of the user.

Type 2:

Modern cryptography demands lengthy key schedule [11] algorithm. So in order to increase the key expansion time and to increase the security of the cipher we can make use of two sets of sub keys, one set can be used to shift the S-box, one in each round and the second set of sub keys are used as a regular *AddRoundKey* as in the original algorithm to perform add round key. This requires the key expansion algorithm to be executed twice, which indirectly helps to increase the time for key expansion. Use of two sets of keys helps in increasing the security. This operation is repeated during the decryption process.

During *InvSubBytes* operation of decryption, the same value that is used to rotate the S-box during *SubBytes* of Encryption is subtracted from the resulting *InvSubBytes* operation in order to nullify the effect of rotation of S-box. The AES-KDS key expansion algorithm, takes as an input a four word (16 Bytes) key. In this case, first XOR operation of all the bytes of the key is carried out and the resulting 8-bit (byte) value is used for shifting the S-Box. This shifted S-Box is used to generate 11 sub keys, each of 4 words in length, totally a linear array of forty four words (176 Bytes). This forms first set of round keys named *expanded_key1*. The round keys thus generated will be used for finding a value that is used to rotate the S-box in each round. These round keys are also used for finding a value for rotating the S-box, which will be used in generating second set of round keys named *expanded_key2*. This is sufficient to provide a four word round key for the initial *AddRoundKey* stage and each of the 10 rounds of cipher. The following pseudo code describes the expansion.

```
        unsigned char key[16]=1234567890ABCDEF;
        unsigned char temp=0;
        FILE *ky1;
        unsigned int rotate;
        for(i=0;i<16;i++)
        temp=temp^key[i];
        rotate=temp;
        create_s_box(s_box,rotate);
        key_expansion(expanded_key1,key,s_box);
// as in original AES
        for(i=0;i<44;i++)
        fprintf(ky1,"%lx ",expanded_key1[i]);
// First set of round keys
        for(i=0;i<43;i++)
        {

        expanded_key1[i+1]=expanded_key1[i]^expan
ded_key1[i+1];
        }
        for(i=0;i<=3;i++)
                for(j=0;j<=3;j++)
                {

        temp=expanded_key1[44]&mask;
                        temp=temp>>shift1;
                        shift1=shift1+8;
                        mask=mask<<8;
                        shift=shift^temp;
                 }
        create_s_box(s_box,shift);
        key_expansion(expanded_key2,key,s_box);
        for(i=0;i<44;i++)
        fprintf(ky2,"%lx ",expanded_key2[i]);
// Second set of round keys.
```

3. Timing and Security aspects

AES-KDS requires little extra time for encryption and decryption. The added stage in encryption, the Rotate S-box operation does not contain any calculation like multiplication or division. Here the bytes are just rotated and hence consume very less time. Decryption process does not have any extra stage we compared to AES, but one subtraction operation is carried out during the *InvSubByes* operation. The extra time taken for this is also negligible. Some time is consumed during the key expansion and to compute a value that is used for rotating S-box. This is affordable at the gain of security.

AES-KDS uses S-box whose entries ranging from 00 to FF as in the AES S-box. AES S-box was specifically designed to be resistant to linear and differential cryptanalysis [4], [12], [13] [14], [21]. It is secure against linear, differential and algebraic attacks[18]. AES-KDS does not even touch Inverse S-box. The aim of the algorithm was to make the S-box key dependent without changing design and by making minimum modifications to the implementation. In each round AES-KDS S-box can have 256 possible entries. Totally there are 10 rounds. So total number of possible S-boxes is given by,

$$256 \times 256 \times 256 \times 256 \times 256 \times 256 \times 256 \times 256 \times 256 \times 256 = 2^{80}$$

This gives the clear picture of the difficulty involved in the Cryptanalysis.

Moreover sub keys (round keys) are also generated after shifting the S-box. We can have 256 possible sub keys when only one set of keys are used. When two sets of keys are used, first set can have one of the 256 possible sub keys and the second set can have one of the 256 x 256 possible values.

4. Experimental Results

Key = ADF278565E262AD1F5DEC94A0BF25B27

SN	Plaintext	Ciphertext
1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	B2 43 B5 85 CA DD F4 4E F5 E6 6E D1 D7 08 B3 0B
2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01	37 14 10 49 D7 DE 2D 56 CC 74 66 6B CF 89 4C 95
3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 01	64 E2 C1 53 C4 79 78 DF FC 87 35 15 9F 4C 39 76
4	10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	2F 1D C4 1D DB 52 D6 9D A5 74 99 69 9B 16 31 9E

Table.1. Plaintext & Ciphertext samples

Key = ADF278565E262AD1F5DEC94A0BF25B28

SN	Plaintext	Ciphertext
1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	21 55 66 73 D8 BE 4F 9D 98 55 68 D0 06 DC E6 35
2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01	32 67 6F 89 15 6E 88 80 D0 82 07 9A 0E E2 35 07
3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 01	25 AE 71 C2 4F E0 7F A3 AD 23 35 84 31 47 2B 9E
4	10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	C9 89 71 71 A1 F0 9D 4A 80 A9 6D CF F3 EE 40 4C

Table.2. Plaintext & Ciphertext samples

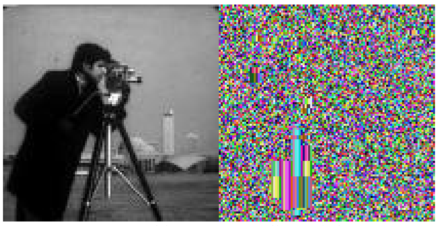


Fig. 9 Image and is corresponding Cipher-Image

The experimental results shown in Table.1 and Table.2 above are taken for Case 4 of encryption and posses very good avalanche characteristic. Encryption of a bitmap image Cman.bmp is also shown.

5.5 Security Analysis

5.5.1 Avalanche effect:

In cryptography, the **avalanche effect** refers to a desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions. The avalanche effect is evident if, when an input is changed slightly (for example, flipping a single bit) the output changes significantly (eg, half the output bits flip). In the case of quality block ciphers, such a small change in either the key or the plaintext should cause a drastic change in the ciphertext.

If a block cipher does not exhibit the avalanche effect to a significant degree, then it has poor randomization, and thus a cryptanalyst can make predictions about the input, being given only the output. This may be sufficient to partially or completely break the algorithm. It is thus not a desirable condition from the point of view of the designer of the cryptographic algorithm or device.

Constructing a cipher or hash to exhibit a substantial avalanche effect is one of the primary design objectives.We have taken 60000 samples each for the original algorithm and modified algorithm and noted down the Avalanche effect by changing the plain text by one bit. The results observed in security analysis are shown below. Tabulation of results observed by changing one bit of plaintext/key in the samples:

One bit change in Plaintext (60000 samples), using Key 1

Case	Number of samples	Number of times Original algorithm gives better Avalanche	Number of times Modified algorithm gives better Avalanche	Number of times the Original and Modified algorithms give same Avalanche
Case 1	60000	28630	28329	3041
Case 2	60000	28478	28558	2964
Case 3	60000	28298	28699	3003
Case 4	60000	28322	28541	3137

Table.3. Avalanche effect for 1 bit change in plaintext

One bit change in key, using Plaintext (60000 samples)

Case	Number of samples	Number of times Original algorithm gives better Avalanche	Number of times Modified algorithm gives better Avalanche	Number of times the Original and Modified algorithms give same Avalanche
Case 1	60000	28544	28475	2981
Case 2	60000	28717	27299	2984
Case 3	60000	28689	28325	2986
Case 4	60000	28557	28354	3089

Table.4. Avalanche effect for 1 bit change in key

5.5.1.1 Strict Avalanche Criterion (SAC)

It is a property of boolean functions of relevance in cryptography. A function is said to satisfy the strict avalanche criterion if, whenever a single input bit is complemented, each of the output bits should change with a probability of one half. The SAC builds on the concepts of completeness and avalanche and was introduced by Webster and Tavares in 1985. Following tables (Table. 5 and Table. 6) show the SAC by changing one bit of plaintext/key in the samples:

Case	Number of samples	Number of times Original algorithm gives better Avalanche	Number of times Modified algorithm gives better Avalanche	Number of times the Original and Modified algorithms give same Avalanche
Case 1	60000	27144	27163	5693
Case 2	60000	27262	28118	5620
Case 3	60000	27074	27249	5677
Case 4	60000	27278	26952	5820

Table.5. Strict Avalanche Criteria for 1 bit change in plaintext

Case	Number of samples	Number of times Original algorithm gives better Avalanche	Number of times Modified algorithm gives better Avalanche	Number of times the Original and Modified algorithms give same Avalanche
Case 1	60000	27239	27119	5642
Case 2	60000	27243	27122	5635
Case 3	60000	26975	27214	5811
Case 4	60000	27200	27113	5687

Table.6. Strict Avalanche Criteria for 1 bit change in key

The above results show that the modification to AES will not violate the security and is not vulnerable in any way. So the modified algorithm introduces confusion to the greater extent without violating diffusion.

5.5.2 Security Analysis using Digital Images

In this section, to evaluate the efficiency of modified AES (AES-KDS) cipher for application to digital images and to compare with that of AES, some experiments results are given to prove the efficiency. AES-KDS and AES ciphers are applied to several digital images. Before encryption/decryption, we must extract the image header for the image to be encrypted / decrypted. So, we must study the file format for image to determine all parts of the file header and to determine the beginning of the data stream to be encrypted. Then, the AES-KDS and AES ciphers are applied to the image. We have used bitmap grey scale(0-255) images, Lena and Cman as the original images (plainimages).

5.5.2.1 Encryption Quality Analysis of AES-KDS Block Cipher

All previous studies on image encryption were based on the visual inspection to judge the effectiveness of the encryption technique used in hiding features. Visual inspection is insufficient in evaluating the amount of information hidden. So, we need to have a mathematical measure to evaluate the degree of encryption quantity, which we will call the encryption quality. The main goal here is to use a mathematical model for the measurement of the amount of encryption quantity of AESKDS and to compare it with that of AES. In all experiments, we use the grey-scale two images-- Lena, and Cman, each of size grey-scale (0-255) as the original images (plainimages).

Measurement of Encryption Quality

With the application of encryption to an image a change takes place in pixels values as compared to those values before encryption. Such change may be irregular. This means that the higher the change in pixels values, the more effective will be the image encryption and hence the encryption quality. So the encryption quality may be expressed in terms of the total changes in pixels values between the original image and the encrypted one. A measure for encryption quality may be expressed as the deviation between the original and encrypted image. The quality of image encryption may be determined as follows: Let F, F' denote the original image (plainimage) and the encrypted image (cipherimage) respectively, each of size M*N pixels with L grey levels. $F(x, y)$,$F'(x, y) \in \{0,.., L-1\}$ are the grey levels of the images F , F' at position (x, y) , $0 \leq x \leq M-1$, $0 \leq y \leq N-1$. We will define $H_L(F)$ as the number of occurrence for each grey level L in the original image (plainimage), and $H_L(F')$ as the number of occurrence for each grey level L in the encrypted image (cipherimage). The encryption quality represents the average number of changes to each grey level L and it can be expressed mathematically as

$$\text{Encryption Quality} = \frac{\sum_{L=0}^{255} \left| H_L(F') - H_L(F) \right|}{256}$$

Following table (Table. 7) shows results of Encryption quality of AES and its comparison with AES-KDS. From the results we can conclude that the modification to AES will not affect the Encryption Quality of he cipher in any way.

Key K1 = ADF278565E262AD1F5DEC94A0BF25B27
Key K2 = ADF278565E262AD1F5DEC94A0BF25B28

Encryption Quality (E.Q) of AES and AES-KDS						
Image	Key	Algorithm type				
		AES	AES-KDS			
			Case 1	Case 2	Case 3	Case 4
Cman	K1	128.109375	128.773438	128.441406	128.738281	126.714844
Cman	K2	126.390625	127.882812	129.136719	128.878906	126.933594
Lena	K1	55.515625	56.835938	56.539062	57.703125	56.710938
Lena	K2	56.945312	55.406250	56.226562	56.296875	56.007812

Table.7. Encryption quality of AES and AES-KDS

5.5.2.2 Key sensitivity test

Assume that a 128-bit ciphering key is used. A typical key sensitivity test has been performed, according to the following steps:

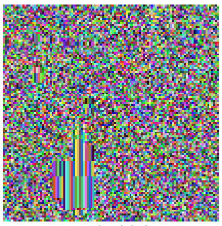
First, an image is encrypted by using the test key K1="ADF278565E262AD1F5DEC94A0BF25B27".

Then, the least significant bit of the key is changed, so that the original key becomes, say K2="ADF278565E262AD1F5DEC94A0BF25B28" in this example, which is used to encrypt the same image.

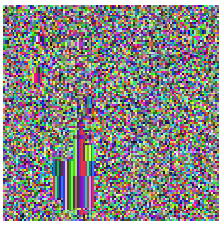
Finally, the above two ciphered images, encrypted by the two slightly different keys, are compared.

Case 1:

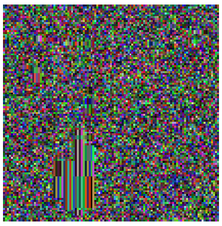
1. Plainimage Cman.bmp



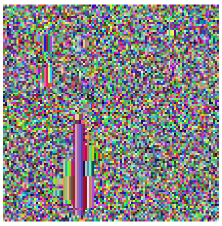
2.Encrypted with key K1



3. Encrypted with key K2



4. Difference of images in 2 & 3



5. Encrypted with key K1, but decrypted using K2

K1= ADF278565E262AD1F5DEC94A0BF25B27

K2= ADF278565E262AD1F5DEC94A0BF25B28

The above figures show plainimage(Cman.bmp) (1) and the encrypted images (2,3) using keys K1 and K2 respectively. The fourth image is the difference of the images encrypted. This cleary shows that even when one bit of key is changed, it has its influence on all the pixels. Morever when we tried to decrypt the image encrypted with K1 using the key K2, we got the image as shown in 5. This proves the sensitivity of the key of the modified algorithm. Similar results are obtained for Case 2, Case 3 and Case 4.

5.5.2.3 Correlation of two adjacent pixels

To test the correlation between two vertically adjacent pixels, two horizontally adjacent pixels, and two diagonally adjacent pixels in plainimage/cipherimage, respectively, the procedure is as follows: First, randomly select 1000 pairs of two adjacent pixels from an image. Then, calculate their correlation coefficient using the following two formulas:

$$\mathrm{cov}(x,y) = E(x - E(x))(y - E(y)),$$

$$r_{xy} = \frac{\mathrm{cov}(x,y)}{\sqrt{D(x)}\sqrt{D(y)}},$$

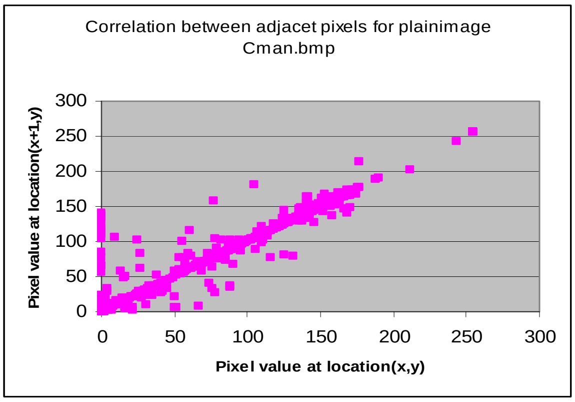
where x and y are grey-scale values of two adjacent pixels in the image. In numerical computations, the following discrete formulas were used:

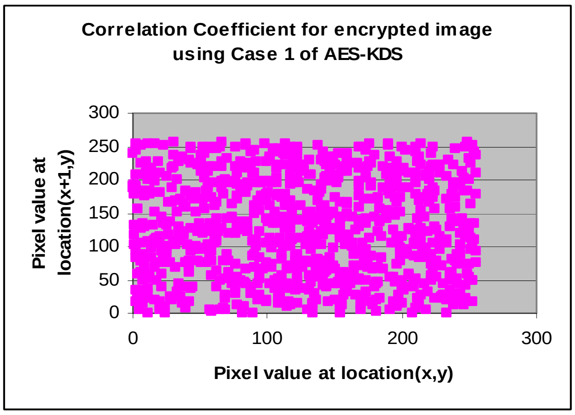
$$E(x) = \tfrac{1}{N}\sum_{i=1}^{N} x_i$$

$$D(x) = \tfrac{1}{N}\sum_{i=1}^{N} (x_i - E(x))^2,$$

$$\mathrm{cov}(x,y) = \tfrac{1}{N}\sum_{i=1}^{N} (x_i - E(x))(y_i - E(y)),$$

Figures below show the correlation distribution of two horizontally adjacent pixels in the plainimage/cipherimage for AES and Case 1 of AES-KDS block ciphers. The correlation coefficients are plain and cipher images are far apart. Similar results are obtained for AES, other Cases of AES-KDS.





Summary of results for both he ciphers are shown in table (Table. 8) below.

Image	Plainimage	AES	AES-KDS			
			Case 1	Case 2	Case 3	Case 4
Cman	0.452019	0.048484	0.032304	0.040144	0.045481	0.044151

Table.8. Correlation Coefficient for AES and AES-KDS

6. Conclusion

In this paper a new improved version of AES has been proposed. AES-KDS doesn't contradict the security of the AES algorithm. We tried to keep all the mathematical criteria for AES without change. We have improved the security of AES by making its S-box to be key dependent and by changing the key expansion procedure.

References

[1] J. Daemen and V. Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard." Springer-Verlag, 2002

[2] J. Daemen, V. Rijmen, "The block cipher Rijindael", Proceedings of the Third International Conference on smart card Research and Applications, CARDIS'98, Lecture Notes in computer Science, vol.1820, Springer, Berlin, 2000, pp.277_284.

[3] Federal Information Processing Standards Publications (FIPS 197), "Advanced Encryption Standard (AES) ", 26 Nov. 2001.

[4] H. Gilbert, M. Minier. A collision attack on 7 rounds of Rijndael. In The third Advanced Encryption Standard Candidate Conference, pages 230–241, NIST, April 2000. See http://www.nist.gov/aes.

[5] A. Fahmy, M. Shaarawy, K. El-Hadad, G. Salama and K. Hassanain, "A Proposal For A Key-Dependent AES" SETIT 2005, TUNISIA

[6] B. Schneier, Applied Cryptography: Protocols, Algorithms and Source Code in C. John Wiley and Sons, 1996.

[7] B. Schneier, "Description o f a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)", Fast Software Encryption, Cambridge Security Workshop roceedings (December 1993), Springer-Verlag, 1994, pp. 191-204.

[8] William Stallings, Cryptography and Network Security, Third Edition, Pearson Education, 2003.

[9] Kishnamurthy G.N, V.Ramaswamy, and Leela.G.H "Performance Enhancement of Blowfish algorithm by modifying its function" Proceedings of International Conference in CISSE 2006 university of Bridgeport, Bridgeport, CT, USA, pp 244-249.

[10] Adams, C. The CAST-128 Encryption Algorithm. RFC 2144, May 1997.

[11] Anne Canteaut(Editor) "Ongoing Research Areas in Symmetric Cryptography" Francois-Xavier Standaert, Gilles Piret, Jean-Jacques Quisquater, Cryptanalysis of Block Ciphers: A Survey, available at http://www.dice.ucl.ac.be/crypto/

[12] Howard M. Heys, A Tutorial on Linear and Differential Cryptanalysis, St. John's, NF, Canada.

[13] Pascal JUNOD, Statistical Cryptanalysis of Block Ciphers, Ph.D. Thesis, EPFL, 2005.

[14] Orr Dunkelman, Techniques for Cryptanalysis of Block Ciphers, Ph.D. Thesis, Israel Institute of Technology, 2006.

[15] Gilles-Fracois Piret, Block Ciphers: Security Proofs, Cryptanalysis, Design, and Fault Attacks, Ph.D. Thesis, UCL Crypto Group, 2005.

[16] Elad Pinhas Barkan, Cryptanalysis of Ciphers and Protocols, Ph.D. Thesis, Israel Institute of Technology, 2006.

[17] HM Heys and SE Tavares "On the security of the CAST encryption algorithm Canadian Conference on Electrical and Computer Engineering pp 332-335, Sept 1994, Halifax, Canada.

[18] Deepak Kumar Dalai, "On Some Necessary Conditions of Boolean Functions to Resist Algebraic Attacks", Ph.D Thesis, Applied Statistics Unit, Indian Statistical Institute, Kolkata, India, August, 2006.

[19] Krishnamurthy G N, V Ramaswamy, Leela G H, Ashalatha M E, "Blow-CAST-Fish, a New 64-bit Block Cipher", IJCSNS, ISSN : 1738-7906, Vol. 8, No.4, pp 282-290, April -2008, Korea.

[20] Krishnamurthy G N, V Ramaswamy, Leela G H, Ashalatha M E, "Performance enhancement of Blowfish and CAST-128 algorithms and Security analysis of modified Blowfish algorithm using Avalanche effect", Journal ISSN : 1738-7906Vol Number: Vol.8, No.3, pp 244-250, March -2008, Korea.

[21] R.C.W. Phan, "Impossible differential cryptanalysis of 7-round Advanced Encryption Standard (AES)", Information processing letters 91(2004) 33-38.

[22] Kishnamurthy G.N, V.Ramaswamy, and Leela.G.H "Performance Enhancement of CAST-128 algorithm by modifying its function" Proceedings of International Conference in CISSE 2006 university of Bridgeport, Bridgeport, CT, USA.

[23] Bruce Schneier, Doug Whiting (2000-04-07). "A Performance Comparison of the Five AES Finalists" (PDF/PostScript). Retrieved on 2006-08-13.

[24] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson (1998-06-15). "The Twofish Encryption Algorithm" (PDF/PostScript). Retrieved on 2007-03-04.

[25] R.L. Rivest, M.J.B. Robshaw, R.Sidney, and Y.L. Yin. The RC6 Block Cipher. v1.1, August 1998.

Krishnamurthy G N obtained his B.E. degree in Electronics & Communication Engineering from Kuvempu University in 1996 and M.Tech. degree in Computer Science & Engineering from Visveswaraya technological University, India in 2000. He is presently pursuing his Ph.D. from Visveswaraya Technological University, India under the guidance of Dr. V ramaswamy. He has published papers in national and international conferences, journals in the area of Cryptography. After working as a lecturer (from 1997) he has been promoted to Assistant Professor (from 2005), in the Department of Information Science & Engineering, Bapuji Institute of Engineering & Technology, Davangere, affiliated to Visveswaraya Technological University, Belgaum, India. His area of interest includes Design and analysis of Block ciphers; He is a life member of ISTE, India.

Dr. V Ramaswamy obtained his Ph.D. degree from Madras University, in 1982. He is working as Professor and Head in the Department of Information Science and Engineering. He has more the 25 years of teaching experience including his four years of service in Malaysia. He is guiding many research scholars and has published many papers in national and international conference and in many international journals. He has visited many universities in USA and Malaysia.