Simplified AES for Low Memory Embedded Processors

GJCST Classification
E.3,C.3,C.1.M

Suhas J Manangi, Parul Chaurasia, Mahendra Pratap Singh

***Abstract*-Embedded Processors have 2 main constraints[1]: Memory and Processing Power. Encryption algorithm running on such processors should be specially customized to overcome these 2 constraints. AES[2] is a well-known standard algorithm for encryption but its memory and processing power requirements are very high. This paper proposes modifications to AES to optimize it to overcome above 2 constraints.**

Keywords-AES, Encryption, Decryption, Embedded Processors.

I. Introduction

Embedded systems are those which are complete devices often including hardware and mechanical parts. For Eg: Digital watches, MP3 players, mobile phones, videogame consoles, GPS receivers, Wireless Sensor Nodes, dedicated routers and network bridges etc. Embedded systems have mainly 2 constraints[1]: Low memory and Low Processor Capacity. In this paper a simplified version of Advanced Encryption Standard is designed for low memory embedded processors. This algorithm can be used in cases where data or network security is required involving embedded systems like Wireless Sensor Nodes, Mobile Phones etc. The proposed algorithm tries to reduce amount of memory needed by AES and also increasing amount of processing needed for encryption and decryption.AES[2] is Advanced Encryption Standards designed from larger collection of Rijndael algorithm. AES has 3 flavors AES-128, AES-192, AES-256 with block size 128 and number of rounds is 10, 12, and 14 respectively.

II. Proposed Algorithm (LMEP-S-AES)

LMEP-S-AES is a simplified version of AES, it has all the functions AES has but made suitable for embedded systems by optimizing on its memory and processor requirements.
Encryption part of LMEP-S-AES has 5 sub functions: Key Expansion, Adding Key Round, Substitution Function, Row Transformation function and Mix Column function. Decryption part of LMEP-S-AES has 5 sub functions: Key Expansion, Inverse Mix Column functions, Inverse Row Transformation function, Inverse Substitution function and Adding Key Round

1) *Key Expansion*

Key length here is not fixed but can vary between 1 Byte to 16 Bytes[3]. Initial key is expanded to 16 bytes expanded key. This same key is applied in each round during encryption and also during decryption. Key is basically XORed with data blocks during encryption as well as decryption. Initial key is expanded using substitution S Box mentioned in algorithm in Figure 1.

```
KeyExpansion ( ByteString InputKey, Int KeyLen )
{
   RoundKey = InputKey

   for ( i=0; i<16; i++ )
   {
      RoundKey[i] = SubByte ( RoundKey[i%KeyLen] )
   }

   return RoundKey
}
```

Figure 1: Key Expansion

2) *Encryption*

Block size is same as AES, 128 bits arranged in 4x4 bytes called State Box. Encryption algorithm is explained in Figure 2. Number of rounds here is reduced to three and transformation functions are applied in the order of adding round key, substation, row rotation and mix column functions.

```
Encryption ( ByteString Input, ByteString Output, ByteString InputKey, int ByteKeyLen )
{
   ByteString State[4][4] = Input
   ByteString RoundKey = KeyExpansion ( InputKey, ByteKeyLen )

   for ( i=0; i<3; i++)
   {
      State = AddRoundKey ( State, RoundKey )
      State = SubByte ( State )
      State = ShiftRows ( State )
      State = MixColumns ( State )
   }

   State = AddRoundKey ( State, RoundKey )

   Output = State
}
```

Figure 2: Encryption[2] [3]

a) *Add Round Key*

For each round key used is same, and is XORed with data in State Box. Complete algorithm is shown in Figure 3 below.

About- Department of Computer Engineering B.Tech in Computer Engineering suhasjm@gmail.com, stylishparul@gmail.com, mps_82@aol.inNational Institute of Technology Karnataka –

```
void AddRoundKey ( ByteString RoundKey )
{
   k=0;
   for ( i=0; i<4; i++ )
   {
      for ( j=0; j<4; j++ )
      {
         State[i][j] = State[i][j] XOR RoundKey[k++]
      }
   }

   return ;
}
```

Figure 3: Add Round Key[2] [3]

b) *Substitution Transformation*

S Box is modified into 4x4 Matrix from 16x16 Matrix in AES. Each byte in State Box is split into 2 Nibble. Each Nibble is replaced by another nibble from S-Box in same manner of AES. First 2 bits of nibble represents row number and next 2 bits represents column number. For Eg: if Nibble value is 1010 then present nibble is replaced with 2nd row, 2nd column nibble of S-Box. Substitution of bytes is shown in Figure 4, breaking of each Byte into 2 nibbles is shown in figure 5, substituting each nibble with nibble from S-Box is shown in Figure 6. S-Box is shown in Table 1.

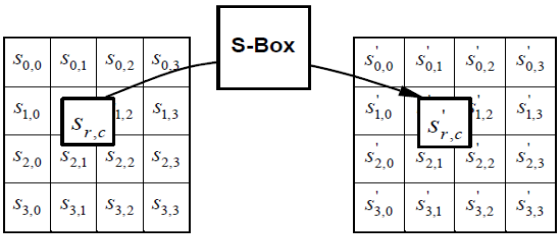

Figure 4: Byte Substitution[2] [3]

```
Byte SubByte ( Byte B )
{
   Nibble N1, N2
   N1, N2 = B
   N1 = SubNibble ( N1 )
   N2 = SubNibble ( N2 )
   B = N1, N2
   return B;
}
```

Figure 5: Substitution Byte Function

```
Nibble SubNibble ( Nibble N )
{
   Base4 r,c
   r,c = N
   N = S-Box[r][c]
   return N
}
```

Figure 6: Substitution Nibble Function

	0	1	2	3
0	21	20	23	10
1	03	32	00	01
2	12	33	30	02
3	11	22	13	31

Table 1: S-Box

c) *Row Transformation*

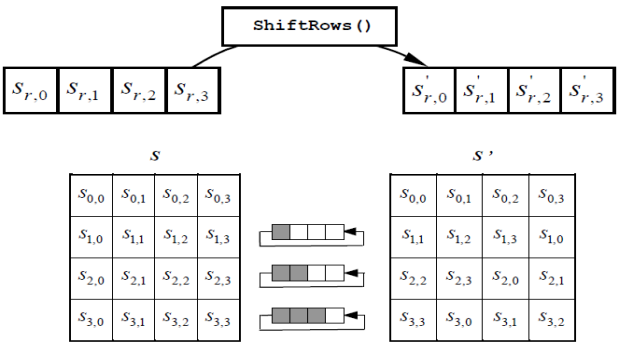

Figure 7: Shift Rows[2] [3]

d) *Column Transformation*

Mix Columns function is exactly same as in AES algorithm. This is represented as matric multiplication as shown below.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 8: Matrix Multiplication[2] [3]

Above matrix multiplication is expanded as below. This includes scalar multiplication thus ensuring final value not exceeding 8 bits length.

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Figure 9: Scalar Multiplication[2] [3]

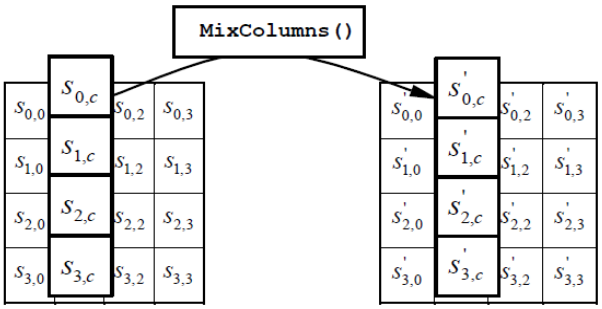

Figure 10: Column Transformation[2] [3]

3) *Decryption*

```
Decryption ( ByteString Input, ByteString Output, ByteString InputKey, int ByteKeyLen )
{
  ByteString State[4][4] = Input
  ByteString RoundKey = KeyExpansion ( InputKey, ByteKeyLen )

  State = AddRoundKey ( State, RoundKey )
  for ( i=0; i<3; i++)
  {
     State = InvMixColumns ( State )
     State = InvShiftRows ( State )
     State = InvSubByte ( State )
     State = AddRoundKey ( State, RoundKey )
  }

  Output = State
}
```

Figure 11: Decryption Function[2] [3]

a) *Add Round Key*

```
void AddRoundKey ( ByteString RoundKey )
{
   k=0;
   for ( i=0; i<4; i++ )
   {
      for ( j=0; j<4; j++ )
      {
         State[i][j] = State[i][j] XOR RoundKey[k++]
      }
   }

   return ;
}
```

Figure 12: Add Round Key Function[2] [3]

b) *Inverse Substitution Transformation*

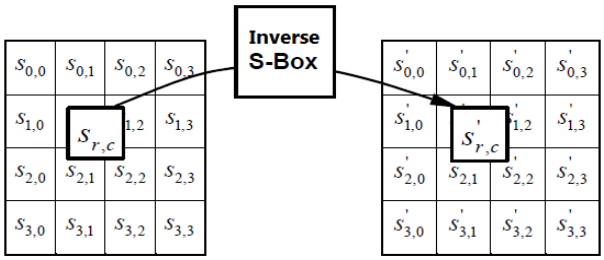

Figure 13: Inverse Byte Substitution[2] [3]

```
Byte InvSubByte ( Byte B )
{
   Nibble N1, N2
   N1, N2 = B
   N1 = InvSubNibble ( N1 )
   N2 = InvSubNibble ( N2 )
   B = N1, N2
   return B;
}
```

Figure 14: Inverse Byte Substitution Function

```
Nibble InvSubNibble ( Nibble N )
{
   Base4 r,c
   r,c = N
   N = InvS-Box[r][c]
   return N
}
```

Figure 15: Inverse Nibble Substitution Function

	0	1	2	3
0	12	13	23	10
1	03	30	20	32
2	01	00	31	02
3	22	33	11	21

Table 2: Inverse S-Box

c) *Inverse Row Transformation*

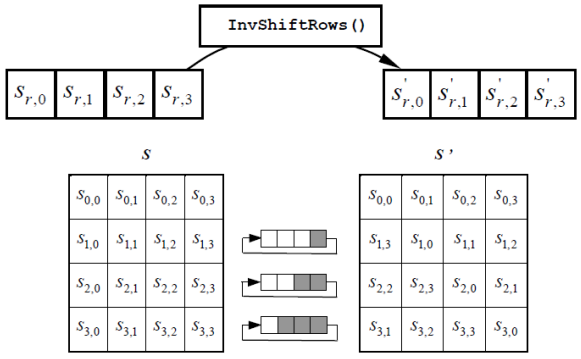

Figure 16: Inverse Row Transformation[2] [3]

d) *Inverse Column Transformation*

Inverse Mix Columns function is exactly same as in AES algorithm. This is represented as matric multiplication as shown below.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 17: Matrix Multiplication[2] [3]

Above matrix multiplication is expanded as below. This includes scalar multiplication thus ensuring final value not exceeding 8 bits length.

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

Figure 18: Scalar Multiplication[2] [3]

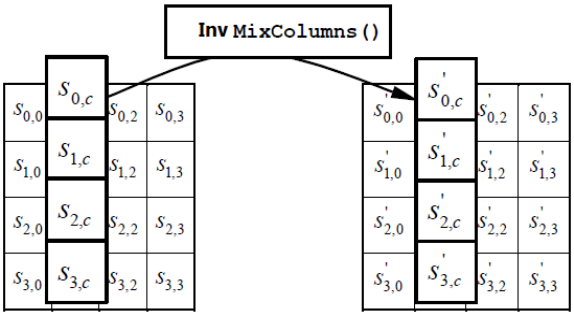

Figure 19: Inverse Column Transformation[2] [3]

III. Comparison of AES with proposed algorithm (LMEP-S-AES)

	AES	LMEP-S-AES
Key Length	128, 192, 256 Bits	Variable Length (Max 128 bits)
Number of Rounds	10, 12, 14	3
Round Key	Different for each round	Same for all rounds
Key Expansion	Complexcombination of SubByte() and Scalar Multiplication	Simplified
S-Box size	16x16	4x4
S-Box Memory	256 Bytes	8 Bytes
Inverse S-Box Memory	256 Bytes	8 Bytes

Table 3: Comparison between AES and LMEP-S-AES

IV. Analysis of LMEP-S-AES

1) *Memory Optimization*

Embedded processors need applications to be running in low memory constraints.

1. The proposed AES optimizes memory requirements by reducing S-Box and Inverse S-Box. Together S-Box needs 16 Bytes in comparison to 512 Bytes of actual AES algorithm.
2. Same key is used for al rounds, thus reducing memory consumption of expanded key like in AES algorithm.

2) *Processing Optimization*

Embedded processors are generally low end processors with processing power constraints.

I. The proposed algorithm optimizes processing power requirements by reducing number of rounds to 3.
II. Key Expansion algorithm is simplified.

III. Additional computations needed in choosing round key for each round is reduced, since same key is used for all rounds.

V. Conclusion

Embedded systems like Mobile phones, GPS receivers, Wireless Sensor Nodes etc handle sensitive data, hence requires data security mechanisms. AES algorithm which is a standard algorithm for data encryption is unsuitable for such scenarios where memory and processing power constraints are very high.The proposed Simplified AES for Low Memory Embedded Processors" algorithm is optimized to overcome these 2 constraints.

VI. References

1) Embedded Software", Edward A. Lee, Advances in Computers, Academic Press London 2002
2) Advanced Standard Encryption" Federal information Processing Standards Publication 197
3) Merging of RC5 with AES - Incorporating more Flexibility and Security in AES" ICIT 2009, ISBN: 978-0-07-068014-2