UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING

NETWORK SECURITY (EHN 410)

PRACTICAL 2 GUIDE

Contents

1 INTRODUCTION

The AES encryption algorithm was created by Joan Deamon and Vincent Rijmen as a substitute for DES and was accepted for use by the National Institute of Standard Technology (NIST) in 2001. It is based on the Rijndael key scheduler, which makes use of a substitution-permutation network to create the secret keys. The advantage of the AES encryption algorithm over DES is the fact that the key sizes can easily be varied, thus allowing scalability when increasing the algorithm's complexity and security. The subsequent literature provided in this document describes the general procedures required for AES 128-bit encryption.

1.1 BLOCK SIZES

AES 128-bit encryption makes use of 128-bit plaintext block sizes (also commonly referred to as the *state array*). Hence a plaintext block consisting of 128-bits are encrypted per iteration. The blocks are of size 4×4 with each block containing a single byte. Each plaintext block is created column-wise, meaning that the first column will have the first 4 bytes and the last column the last 4 bytes. Table 1 provides an example of how the blocks are populated.

byte1	byte5	byte9	byte13
byte2	byte6	byte10	byte14
byte3	byte7	byte11	byte15
byte4	byte8	byte12	byte16

Table 1: 128-bit block of AES

It is common when encrypting a large file that the amount of data is not a multiple of the block size (i.e. 128 bits). Hence padding is required to increase the length of the block to the correct size (only when using a block cipher). Many different types of padding can be used to increase a block's size to 128-bits. **For this practical zero padding will be used.**

1.2 GENERAL AES PROCEDURE

The AES algorithm has four main components used for encryption and decryption. Each is briefly listed below:

1. Key Expansions - Rijndael's key schedule (to expand the key to 176 bytes).
2. Initial round - Set up the round key (XOR with block with the key)
3. Mixing rounds - Multiple rounds of byte permutations and substitutions.
4. Final Round - A last round of mixing, the output of which is the ciphertext.

2 RIJNDAEL'S KEY SCHEDULER

The Rijndael's key scheduler is used to expand the secret key to 176 bytes for encryption/decryption. It consists of five main components:

1. A 32-bit word rotation.
2. The exponentiation of 2 of a user-specified value, also called the round constant (RCon).
3. The Substitution box (S-box) of Rijndael.
4. The core key scheduler.
5. The actual expansion of the key.

2.1 32-BIT WORD ROTATION

The rotation procedure takes as input 4 bytes (32-bits) and shifts it to the left by one byte. Additionally, the input's most significant byte "wraps around to become the least significant byte of the 32-bit word. The example below is provided for clarity:

Input: 3A 65 71 1B
Output: 65 71 1B 3A

2.2 THE S-BOX

Calculating the S-Box and inverse S-box for Rijndael is complicated, making use of affine transformations along with Rijndael's finite field along with multiplicative inverses. As a result, many implementations make use of look-up tables in order to speed up computation and reduce the complexity. For this practical, such a lookup table may be used to simplify the implementation. The following tables provide the S-Box and its inverse:

0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5
0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0
0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC
0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x04	0xC7	0x23	0xC3	0x18	0x96	0x05	0x9A
0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0
0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B
0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85
0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5
0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17
0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88
0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C
0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9
0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6
0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E
0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94
0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68
0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

Table 2: S-box used in Rijndaels key expansion

0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38
0xbf	0x40	0xa3	0x9e	0x81	0xf3	0xd7	0xfb
0x7c	0xe3	0x39	0x82	0x9b	0x2f	0xff	0x87
0x34	0x8e	0x43	0x44	0xc4	0xde	0xe9	0xcb
0x54	0x7b	0x94	0x32	0xa6	0xc2	0x23	0x3d
0xee	0x4c	0x95	0x0b	0x42	0xfa	0xc3	0x4e
0x08	0x2e	0xa1	0x66	0x28	0xd9	0x24	0xb2
0x76	0x5b	0xa2	0x49	0x6d	0x8b	0xd1	0x25
0x72	0xf8	0xf6	0x64	0x86	0x68	0x98	0x16
0xd4	0xa4	0x5c	0xcc	0x5d	0x65	0xb6	0x92
0x6c	0x70	0x48	0x50	0xfd	0xed	0xb9	0xda
0x5e	0x15	0x46	0x57	0xa7	0x8d	0x9d	0x84
0x90	0xd8	0xab	0x00	0x8c	0xbc	0xd3	0x0a
0xf7	0xe4	0x58	0x05	0xb8	0xb3	0x45	0x06
0xd0	0x2c	0x1e	0x8f	0xca	0x3f	0x0f	0x02
0xc1	0xaf	0xbd	0x03	0x01	0x13	0x8a	0x6b
0x3a	0x91	0x11	0x41	0x4f	0x67	0xdc	0xea
0x97	0xf2	0xcf	0xce	0xf0	0xb4	0xe6	0x73
0x96	0xac	0x74	0x22	0xe7	0xad	0x35	0x85
0xe2	0xf9	0x37	0xe8	0x1c	0x75	0xdf	0x6e
0x47	0xf1	0x1a	0x71	0x1d	0x29	0xc5	0x89
0x6f	0xb7	0x62	0x0e	0xaa	0x18	0xbe	0x1b
0xfc	0x56	0x3e	0x4b	0xc6	0xd2	0x79	0x20
0x9a	0xdb	0xc0	0xfe	0x78	0xcd	0x5a	0xf4
0x1f	0xdd	0xa8	0x33	0x88	0x07	0xc7	0x31
0xb1	0x12	0x10	0x59	0x27	0x80	0xec	0x5f
0x60	0x51	0x7f	0xa9	0x19	0xb5	0x4a	0x0d
0x2d	0xe5	0x7a	0x9f	0x93	0xc9	0x9c	0xef
0xa0	0xe0	0x3b	0x4d	0xae	0x2a	0xf5	0xb0
0xc8	0xeb	0xbb	0x3c	0x83	0x53	0x99	0x61
0x17	0x2b	0x04	0x7e	0xba	0x77	0xd6	0x26
0xe1	0x69	0x14	0x63	0x55	0x21	0x0c	0x7d

Table 3: Inverse S-box used in Rijndaels key expansion

2.3 THE EXPONENTIATION OF 2

The exponentiation of 2 is commonly referred to as the *RCon* operation. The easiest way to calculate the value is to know that the current value of *RCon* is always double the previous one **except** in the case where all the bits are zero and only the most significant bit is one (i.e. 0x80). In such a case, the new *RCon* value becomes 0x1B. The following equation can be use

to calculate the *RCon* value:

$$RCon(i) \quad = \quad 2 \times RCon(i-1) \tag{1}$$

where i is the iteration number.

Note: The RCon values can have a maximum of 8-bits and can therefore have maximum value of 0xFF.

2.4 CORE KEY SCHEDULER

The core key scheduler makes use of the previous operations to transform the bytes using the following procedure:

1. Rotate the input 32-bit word as described previously.
2. Use Rijndaels S-box on all four bytes.
3. On the leftmost byte XOR the byte with exponentiation of 2 (i.e. the Rcon operation).

2.5 KEY EXPANSIONS

So far, none of the bytes in the keys have been expanded to the required 176 bytes. In order to expand the key to its required length, the following procedure is used:

1. The first 16 bytes of the expanded key are simply the encryption key that the user entered.
2. Set the RCon value to 1.
3. While there are not 176 bytes in the expanded key, the following is executed:
   (a) Create a 4-byte temporary variable, *t*.
   (b) Assign the value of the previous four bytes in the expanded key (i.e. bytes 12-15) to *t*.
   (c) Send *t* to the core key scheduler along with the RCon value.
   (d) Increment RCon by 1.
   (e) XOR the output of the core key scheduler with a four-byte block 16 bytes before the expanded key (i.e bytes 0-3). The result becomes the next 4 bytes of the expanded key.
   (f) Execute the following three times to create the next twelve bytes of expanded key:
      i. Create a 4-byte temporary variable, *t*.
      ii. XOR the output of the core key scheduler with a four-byte block 16 bytes before the expanded key.
      iii. The result becomes the next 4 bytes of the expanded key.

3 AES ROUNDS

As mentioned previously, the AES algorithm consists of a multiple of rounds. Each round has a number of sub-procedures which need to be implemented. The actual implementation details for each of these procedures can be found in the Additional Resources section. Fig. 1 provides the general procedure for the AES encryption algorithm. Note that "bXX" refers to the byte number in the expanded key.

3.1 INITIAL ROUND

In the initial round each byte of the state is combined with the expanded round key using a bitwise XOR.

3.2 THE MIXING ROUNDS

The mixing rounds contains the following procedures:

1. Subbytes - non-linear substitution where each block is replaced according to a lookup table
2. ShiftRows - transposition step where the last three rows of the state are shifted cyclicly
3. MixColumns - mixing operation, combining the four bytes in each column
4. AddRoundKey

These rounds will go through 10 cycles of repetition before moving on to the final round.

3.3 THE FINAL ROUND

The final round makes use of the same procedures as with mixing rounds, with the exception of the MixColumns procedure. Thus the final round only consist of the following steps:

1. Subbytes
2. ShiftRows
3. AddRoundKey

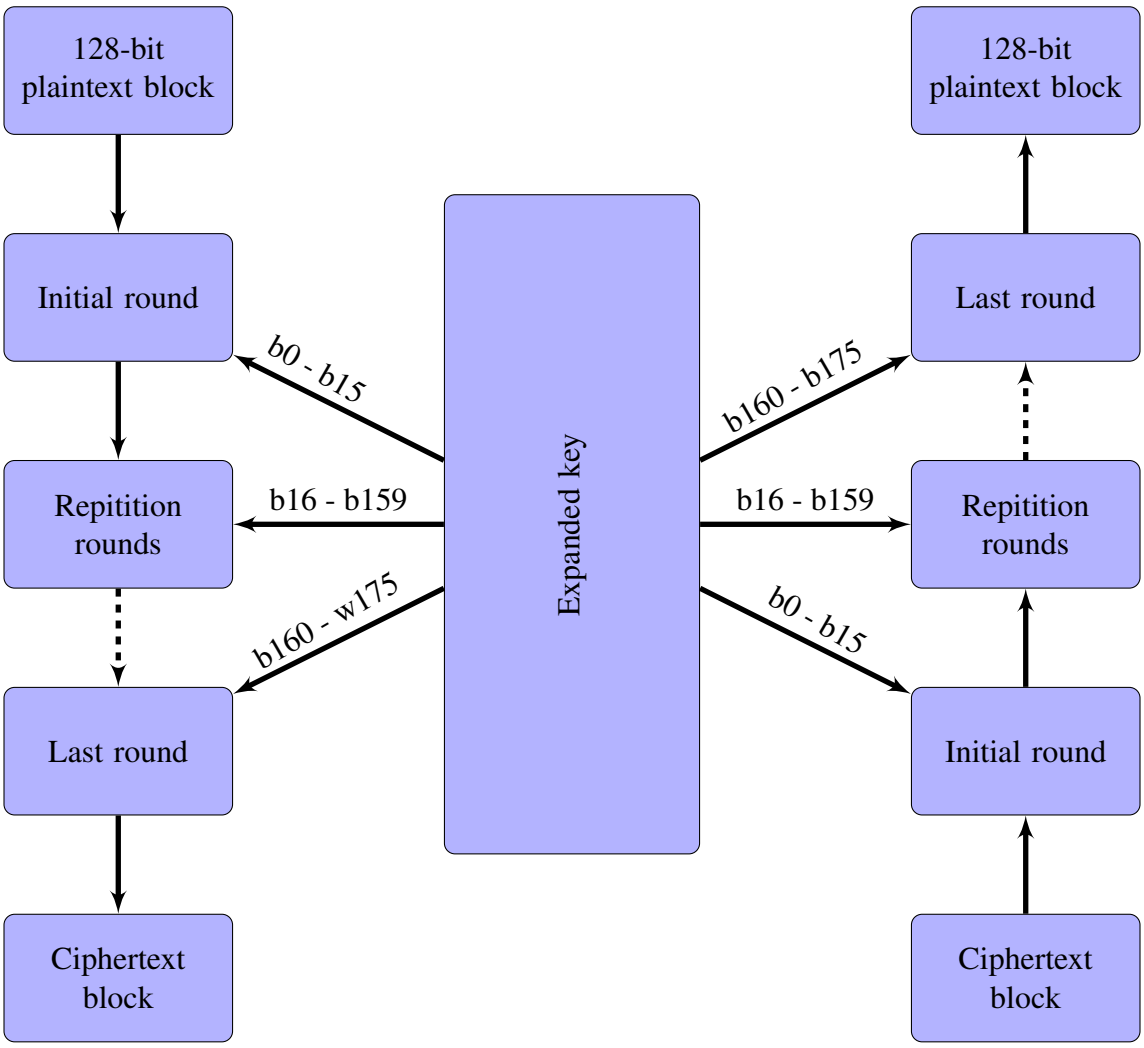

Figure 1: The AES encryption/decryption algorithm

4 TEST VECTORS

The following section provides some test vectors that can be used to confirm that each method works independently.

4.1 MIXING THE COLUMNS

The following test vector of **hexadecimal numbers** can be used to test if mixing the columns was implemented successfully:

Input:

74	20	61	73
68	69	20	74
69	73	74	2e
73	20	65	2e

Output:

4a	a8	b3	7a
6c	47	d8	c7
5b	cf	6	29
7b	3a	3d	93

4.2 SHIFTING THE ROWS

The following test vector of **hexadecimal numbers** can be used to test if shifting the rows was implemented successfully:

Input:

74	20	61	73
68	69	20	74
69	73	74	2e
73	20	65	2e

Output:

74	20	61	73
69	20	74	68
74	2e	69	73
2e	73	20	65

4.3 SUBBYTE STEPS

Using the following message in the Subbyte step results in:

Input:

"This is a test.."

Input(Block):

T		a	s
h	i		t
i	s	t	.
s		e	.

Output (Hex):

20	b7	ef	8f
45	f9	b7	92
f9	8f	92	31
8f	b7	4d	31

4.4 KEY EXPANSIONS

The following test vectors can be used to ensure that the key expansions work as expected:

Input vector (hex):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Output(hex):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
62	63	63	63	62	63	63	63	62	63	63	63	62	63	63	63
9b	98	98	c9	f9	fb	fb	aa	9b	98	98	c9	f9	fb	fb	aa
90	97	34	50	69	6c	cf	fa	f2	f4	57	33	b	f	ac	99
ee	6	da	7b	87	6a	15	81	75	9e	42	b2	7e	91	ee	2b
7f	2e	2b	88	f8	44	3e	9	8d	da	7c	bb	f3	4b	92	90
ec	61	4b	85	14	25	75	8c	99	ff	9	37	6a	b4	9b	a7
21	75	17	87	35	50	62	b	ac	af	6b	3c	c6	1b	f0	9b
e	f9	3	33	3b	a9	61	38	97	6	a	4	51	1d	fa	9f
b1	d4	d8	e2	8a	7d	b9	da	1d	7b	b3	de	4c	66	49	41
b4	ef	5b	cb	3e	92	e2	11	23	e9	51	cf	6f	8f	18	8e

4.5 FURTHER TESTS

To test the complete encryption system, the OpenSSL library can be used. Note that it does not make use of zero padding. Instead OpenSSL makes use of *PKCS#7* for padding (thus you will need to specify that no padding should be added). The following command can be used to test your complete encryption algorithm:

```
   echo -n 'Attack at dawn!!'  | openssl enc -aes-128-ecb
-K 61626364656667686 96a6b6c6d6e6f70 -nosalt -nopad | hexdump -C
```

where the "-K" delimiter is the cipher key used (in hex).

Note: All encrypted files are stored using ASCII values (e.g. " *Sdytwf/"; fw+<* Īăÿěæ* ")

5 ADDITIONAL RESOURCES

1. The Design of Rijndael- The Advanced Encryption Algorithm
2. Selent, Douglas. "Advanced encryption standard." Rivier Academic Journal 6.2 (2010): 1-14.
3. http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html
4. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf