Application Report
SLAA547–August 2012

C Implementation of Cryptographic Algorithms

Jace H. Hall

ABSTRACT

This application report discusses the implementations of the AES, DES, TDES, and SHA-2 cryptographic algorithms written in the C programming language. These software cryptographic solutions were made for devices without hardware acceleration for these algorithms. This document does not go into common methods or practices using these algorithms; however, it does describe how to use the algorithms in program code as well as the nature of the algorithms themselves.

NOTE: This document may be subject to the export control policies of the local government.

Contents

List of Figures

List of Tables

1 Software Benchmarks

All code was tested and benchmarked on the MSP430 platform using IAR as the compiler tool. The optimization columns in the benchmark tables indicate the type of optimization used in IAR. Table 1 describes the settings used.

Table 1. Optimization Settings in IAR for Benchmark Testing

Optimized for:	Optimization Level	Aggressive Unrolling	Aggressive In-Lining
Size	High => Size	No	No
Speed	High => Speed	Yes	Yes

1.1 AES Benchmarks

Table 2. Benchmarks for AES Library Functions Encrypting One 16 Byte Block

		Optimization				Optimization	
AES (ENC/DES Function)		**Speed**	**Size**	**AES (ENC Only Function)**		**Speed**	**Size**
Memory (kB)	RAM (B)	34	34	Memory (kB)	RAM (B)	34	34
	Const	0.55	0.55		Const	0.29	0.29
	Code	1	0.83		Code	0.67	0.51
Clock Cycles (kilo-cycles)		79	12.3	Clock Cycles (kilo-cycles)		7.3	11.3

1.2 DES Benchmarks

Table 3. DES Code Size Benchmarks

	Optimization	
DES Code Size	**Speed**	**Size**
RAM (B)	288	288
Const (kB)	2.3	2.3
Code (kB)	3.3	2.17

Table 4. Performance of Several DES Modes

	Optimization	
DES Clock Cycle Count (kilo-cycles)	**Speed**	**Size**
DES (FULL) (One Data Block)	41	42.6
3DES (FULL) (One Data Block)	135.6	143.1
DES Key Scheduler (EN0 or DE1 modes)	34.7	36
DES Key Scheduler (ENDE mode)	69	72
DES Encode/Decode (One Data Block)	2.7	3.8
DES CBC Encode/Decode (2-block chain)	5.5	7.7
3DES CBC Encode/Decode (2-block chain)	139	149.7

1.3 SHA-2 Benchmarks

Table 5. Benchmarks for SHA-256 Library Function

SHA-256 (Data < 448 bits) [1]		Optimization	
		Speed	**Size**
Memory (kB)	RAM	0.328	0.328
	Const	0.264	0.328
	Code	3.72	1.87
Clock Cycles (kilo cycles)		34.1 (67)	44.3 (86.7)

[1] Values in () indicate a hashing of 448 bits < Data< 960 bits or 2 blocks of data.

2 Using Library Functions

The algorithms were implemented using C. The following sections show how an encryption or decryption can be calculated using the functions provided in this application report.

2.1 AES 128

2.1.1 Encrypting With AES 128

The following code example shows how an AES encryption can be performed.

```
#include "msp430xxxx.h"
#include "TI_aes.h"
//#include “TI_aes_encr_only.h” //Alternative method

int main( void )
{
unsigned char state[] = { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30, 0xd8, 0xcd, 0xb7,
                            0x80, 0x70, 0xb4, 0xc5, 0x5a};
unsigned char key[]   = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
aes_enc_dec(state, key, 0);  // ”0” indicates Encryption
//aes_encrypt(state, key); //Alternative Method of Encryption
return 0;
```

This short program defines two arrays of the type unsigned character. Each array is 16 bytes long. The first one contains the plaintext and the other one the key for the AES encryption.

After the function *aes_enc_dec( )* returns, the encryption result is available in the array state.

2.1.2 Decrypting With AES 128

Decryption can be done in a similar way to encryption. First, two arrays are defined. When a decryption needs to be performed, one array contains the key and the other one the cipher text.

After the function *aes_enc_dec( )* returns, the decryption result is available in the array state.

```
#include "msp430xxxx.h"
#include "TI_aes.h"

int main( void )
{
unsigned char state[] = { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
                            0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a};
unsigned char key[]   = {0x00, 0x01, 0x02, 0x03, 0x04, 05, 0x06, 0x07,
                            0x08, 0x0, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
aes_enc_dec(state, key, 1); // “1” indicates Decryption
return 0;
}
```

2.2 DES

2.2.1 Setting the Key Schedule for DES

The following code example shows how to set the key schedule for DES encryption or decryption rounds. This step must be performed before encryption or decryption can begin.

```
#include "msp430xxxx.h"
#include "TI_DES.h"

int main( void )
{  des_ctx       dc1; // Key schedule structure
   des_ctx       dc2; // Key schedule structure

  unsigned char key[8] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd, 0xfe};

Des_Key(&dc1, key, EN0 );  // Sets up key schedule for Encryption only
Des_Key(&dc1, key, DE1 );  // Sets up key schedule for Decryption only
Des_Key(&dc2, key, ENDE ); // Sets up key schedule for Encryption and Decryption

return 0;
}
```

2.2.2 Encrypting and Decryption With DES

The following code example shows a full encryption then decryption process on a single block of data. The key scheduler is set to populate both key schedules. The results of the operations are stored in the original data array.

```
#include "msp430xxxx.h"
#include "TI_DES.h"

int main( void )
{
   des_ctx       dc1; // Key schedule structure
   unsigned char *cp;
   unsigned char data[] = {0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0xd4, 0x30};
   unsigned char key[8] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xfe};
   cp = data;

   Des_Key(&dc1, key, ENDE); // Sets up key schedule for Encryption and
                                Decryption
   Des_Enc(&dc, cp, 1); //Encrypt Data, Result is stored back into Data
   Des_Dec(&dc, cp, 1); //Decrypt Data, Result is stored back into Data

return 0;
}
```

2.2.3 Encryption and Decryption With DES CBC Mode

The following code example shows a full encryption then decryption process on multiple blocks of data using Cipher-Block Chaining (CBC). The key scheduler is set to populate both key schedules. The results of the operations are stored in the original data array.

```
#include "msp430xxxx.h"
#include "TI_DES.h"

int main( void )
{
  des_ctx       dc1; // Key schedule structure
  unsigned char *cp;
  unsigned char data[] = { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
                           0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a};
  unsigned char key[8] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xfe};
  cp = data;
  Des_Key(&dc1, key, ENDE ); // Sets up key schedule for Encryption and
                               Decryption
  DES_Enc_CBC(&dc, cp, 2); //Encrypt Data, Result is stored back into Data
  DES_Dec_CBC(&dc, cp, 2); //Decrypt Data, Result is stored back into Data
return 0;
}
```

2.3 3DES

2.3.1 Encrypting and Decrypting With Triple DES

The following code example shows the encryption and decryption process using 3DES with and without CBC. The key scheduler is set to populate both key schedules. The results of the operations are stored in the original data array.

```
#include "msp430xxxx.h"
#include "TI_DES.h"

int main( void )
{
   des_ctx       dc1; // Key schedule structure
   unsigned char *cp;
   unsigned char data[]   = {0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30, 0xd8,
                             0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a};
   unsigned char key[8]   = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
   unsigned char key1[8]  = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xfe};
   unsigned char key2[8]  = {0x01,0x23,0x45,0x67,0x89,0xab,0xdc,0xfe};
   cp = data;

   ///First 8 bytes of Data will be Encrypted then Decrypted
   TripleDES_ENC( &dc, cp, 1, key, key1, key2);   // 3DES Encrypt
   TripleDES_DEC( &dc, cp, 1, key, key1, key2);   // 3DES Decrypt

   /// All 16 Bytes of Data will be Encrypted then Decrypted with CBC
   TripleDES_ENC_CBC( &dc, cp, 2, key, key1, key2); // 3DES Encrypt
   TripleDES_DEC_CBC( &dc, cp, 2, key, key1,  key2); // 3DES Decrypt

return 0;
}
```

2.4 SHA-2

2.4.1 Hashing With SHA-256

The following code example shows an example of a data hash using SHA-256.

```
#include "msp430xxxx.h"
#include "TI_SHA2.h"

uint32_t M[32]; //Message array to be hashed
uint64_t L = 0x0000000000000000;  //Bit Length of message to be hashed
uint32_t Ha[8]; // Hash Array to be used during calculation and to store result

int main( void )
{
   M[0] =0x41424344;  //Data
   M[1] =0x45464748;  //Data
   M[2] =0x494A4B4C;  //Data
   L = 0x0000000000000060 //Length == 96 bits or 0x60 bits

   SHA_256(M, L, Ha, 1); // “1” indicates SHA-256 mode

return 0;
}
```

Although this example does not show full initialization of the array M[ ], all relevant values have been populated with meaningful data. M[ ] must be initialized to sizes equal to a 512-bit block of data or hashing block. If the message to be hashed exceeds 448 bits within a hashing block, then an additional hashing block must be reserved. Table 6 explains minimum sizes of M[ ] according to message size.

Table 6. Minimum Sizes of M[ ]

Message Size x (bits)	Minimum Size of Array M[ ]
× < 448	M[16]
448 ≤ × ≤ 512	M[32]
512 < × < 960	M[32]
960 ≤ × < 1024	M[48]

2.4.2 Hashing With SHA-224

The following code example shows a hashing of a message using SHA-224. Although an array of eight 32-bit words are used for the hashing process, only the first seven 32-bit words are used as the hash result.

```
#include "msp430x26x.h"
#include "TI_SHA2.h"

uint32_t M[32]; //Message array to be hashed
uint64_t L = 0x0000000000000000;  //Bit Length of message to be hashed
uint32_t Ha[8]; // Hash Array to be used during calculation and to store result

int main( void )
{
  M[0] =0x41424344;  //Data
  M[1] =0x45464748;  //Data
  M[2] =0x494A4B4C;  //Data
  L = 0x0000000000000060 //Length == 96 bits or 0x60 bits

SHA_256(M, L, Ha, 0); // “0”  indicates SHA-224 mode.

return 0;
}
```

3 Overview of Library Functions

The following sections describe all modes of operation and parameters for the Software Cryptography Library.

3.1 AES 128

Software implementation is of 128-bit AES encryption. This means the algorithm uses a 128-bit key in order to encrypt 128-bit blocks of data. The library was optimized for memory usage (Flash and RAM). There are two functions available from the library: *aes_enc_dec()* and *aes_encrypt()*. Both functions overwrite the data block given with its encrypted value.

Table 7. AES 128 Table of Contents

3.1.1 aes_enc_dec *(unsigned char *state, unsigned char *key, unsigned char dir);*

This function can encrypt or decrypt a message using AES. Use this function if both modes are needed. Data must be in hex form. Function does not convert ASCII text.

Inputs

- *Unsigned char* *state – Pointer to data block to be encrypted
- *Unsigned char* *key – Pointer to 128-bit key
- *Unsigned char* dir – Value that dictates Encryption (‘0’) or Decryption (‘1’)

3.1.2 aes_encrypt *(unsigned char *state, unsigned char *key);*

This function only performs AES encryption. Data must be in hex form. Function does not convert ASCII text. It is possible to decrypt messages while only using the encrypt function. This can be done by encrypting a plain text message with an AES decrypt action, then feeding that cipher text to the AES encryption function.

NOTE: A separate header and code file are made specifically for this function; this is intended for code size sensitive applications.

Inputs

- *Unsigned char* *state – Pointer to data block to be encrypted
- *Unsigned char* *key – Pointer to 128-bit key

3.2 DES and 3DES

Software implementation uses a 64-bit key in order to encipher 64-bit blocks. The DES takes in a 64-bit key, where every eighth bit is used for parity. Therefore, the effective key length is 56 bits. 3DES uses three 64-bit keys and, therefore, has an effective key length of 168-bits.

The DES library functions make use of key structure of type *des_ctx* defined in the helper file. This structure stores the key schedule for both encrypt and decrypt functions.

Table 8. DES and 3DES Table of Contents

Title **Page**

3.2.1 Des_Key *(des_ctx *(Key Structure), unsigned char *pucKey, short sMode);*

This function is the key scheduler for the DES. This step must be performed before calling the encrypt or decrypt function. Key must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure that will store the key schedule
- *unsigned char *pucKey* – Pointer to start of key array in need of scheduling
- *short sMode* -- Sets operation mode for the key scheduler
  - sMode = EN0 : Mode is set to schedule key for encryption
  - sMode = DE1: Mode is set to schedule key for decryption
  - sMode = ENDE: Mode is set to schedule for both encryption and decryption

3.2.2 Des_Enc *(des_ctx *(Key Structure),unsigned char *pucData, short sBlocks);*

This function performs a DES encryption process on data. Key schedules must be created before use. Data must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure containing scheduled keys
- *unsigned char *pucData* – Pointer to start of data array that will be enciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be enciphered

3.2.3 Des_Dec *(ddes_ctx *(Key Structure), unsigned char *pucData, short sBlocks);*

This function performs a DES decryption process on data. Key schedules must be created before use. Data must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure containing scheduled keys
- *unsigned char *pucData* – Pointer to start of data array that will be deciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be deciphered

3.2.4 DES_ENC_CBC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucIV);*

This function performs a DES encryption process with CBC mode. Key schedule must be created before use. Data must be in hex form. Function does not convert ASCII text. Updated IV vector is stored starting at location pucIV.

Inputs

- *des_ctx *Ks* -- Pointer to structure containing scheduled keys
- *unsigned char *pucData* – Pointer to start of data array that will be enciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be enciphered
- *unsigned char *pucIV* – Pointer to start of array of Initialization Vector (IV)

3.2.5 DES_DEC_CBC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucIV);*

This function performs a DES decryption process with CBC mode. Key schedule must be created before use. Data must be in hex form. Function does not convert ASCII text. Updated IV is stored starting at location pucIV.

Inputs

- *des_ctx *Ks* -- Pointer to structure containing scheduled keys.
- *unsigned char *pucData* – Pointer to start of data array that will be deciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be deciphered
- *unsigned char *pucIV* – Pointer to start of array of Initialization Vector (IV)

3.2.6 TripleDES_ENC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucKey1, unsigned char *pucKey2, unsigned char *pucKey3);*

This function performs a 3DES encryption process in the form: Enc $_{key3}$(Dec $_{key2}$(Enc $_{key1}$(Data))). Data and keys must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure that will store the key scheduler
- *unsigned char *pucData* – Pointer to start of data array that will be enciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be enciphered
- *unsigned char *pucKey1* – Pointer to the first key array location
- *unsigned char *pucKey2* – Pointer to the second key array location
- *unsigned char *pucKey3* – Pointer to the third key array location

3.2.7 TripleDES_DEC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucKey1, unsigned char *pucKey2, unsigned char *pucKey3);*

This function performs a 3DES encryption process in the form: Dec[key1](Enc[key2](Dec[key3](Data))). Data and keys must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure that will store the key scheduler.
- *unsigned char *pucData* – Pointer to start of data array that will be deciphered.
- *short sBlocks* – Value indicating how many 64-bit blocks need to be deciphered.
- *unsigned char *pucKey1* – Pointer to the first key location.
- *unsigned char *pucKey2* – Pointer to the second key location.
- *unsigned char *pucKey3* – Pointer to the third key location.

3.2.8 TripleDES_ENC_CBC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucKey1, unsigned char *pucKey2, unsigned char *pucKey3, unsigned char *pucIV);*

This function performs a 3DES encryption process in the form: Enc $_{key3}$(Dec $_{key2}$(Enc $_{key1}$(Data))) with CBC mode enabled. Data and keys must be in hex form. Function does not convert ASCII text. Updated IV is stored starting at location pucIV.

Inputs

- *des_ctx *Ks* -- Pointer to structure that will store the key scheduler
- *unsigned char *pucData* – Pointer to start of data array that will be enciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be enciphered
- *unsigned char *pucKey1* – Pointer to the first key array location
- *unsigned char *pucKey2* – Pointer to the second key array location
- *unsigned char *pucKey3* – Pointer to the third key array location
- *unsigned char *pucIV* – Pointer to start of array of Initialization Vector (IV)

3.2.9 TripleDES_DEC_CBC *(des_ctx *(Key Structure), unsigned char *pucData, short sBlocks, unsigned char *pucKey1, unsigned char *pucKey2, unsigned char *pucKey3, unsigned char *pucIV);*

This function performs a 3DES encryption process in the form Dec[key1](Enc[key2](Dec[key3](Data))) with CBC mode enabled. Data and keys must be in hex form. Function does not convert ASCII text.

Inputs

- *des_ctx *Ks* -- Pointer to structure that will store the key scheduler
- *unsigned char *pucData* – Pointer to start of data array that will be deciphered
- *short sBlocks* – Value indicating how many 64-bit blocks need to be deciphered
- *unsigned char *pucKey1* – Pointer to the first key location
- *unsigned char *pucKey2* – Pointer to the second key location
- *unsigned char *pucKey3* – Pointer to the second key location
- *unsigned char *pucIV* – Pointer to start of array of Initialization Vector (IV)

3.3 SHA-256 and SHA-224

The software implementation uses a 256-bit hash to hash, a hashing block of 512 bits as described in the document *FIBS PUB 180-3*. Data to be hashed must be in hex form. Function does not convert ASCII text. Message array must be a multiple of a hashing block with array elements being 32 bits in length. Function is written in C99 notation for portability reasons.

Table 9. SHA-256 and SHA-224 Table of Contents

3.3.1 SHA_256 *(uint32_t *Message, uint64_t Mbit_Length, uint32_t *Hash, short sMode);*

Inputs

- *uint32_t *Message* – Pointer to array of 32-bit longs to be hashed. Size of array must be a multiple of a hashing block (512 bits or sixteen 32-bit longs).
- *uint64_t Mbit_length* -- 64-bit value containing the precise number of bits to be hashed within the Message array.

NOTE: If Mbit_Length %(mod) 512 >= 448 bits, then an additional hashing block is needed. You must allocate the additional 512 bits.

- *uint32_t *Hash* – Pointer to array of eight 32-bit longs. The final hash value is stored here.
- *short sMode* – Determines if the algorithm run is SHA-224 or SHA-256.
  - Mode is equal to “False”, SHA-224 is used. Final Hash == Hash[0-6].
  - Mode is equal to “True”, SHA-256 is used. Final Hash == Hash[0-7].

4 Cryptographic Standard Definitions

4.1 AES

The Advanced Encryption Standard (AES) was announced by the National Institute of Standards and Technology (NIST) in November 2001. It is the successor of Data Encryption Standard (DES), which cannot be considered as safe any longer, because of its short key with a length of only 56 bits.

To determine which algorithm would follow DES, NIST called for different algorithm proposals in a sort of competition. The best of all suggestions would become the new AES. In the final round of this competition the algorithm Rijndael, named after its Belgian inventors Joan Daemen and Vincent Rijmen, won because of its security, ease of implementation, and low memory requirements.

There are three different versions of AES. All of them have a block length of 128 bits, whereas, the key length is allowed to be 128, 192, or 256 bits. In this application report, only a key length of 128 bits is discussed.

4.1.1 Basic Concept of Algorithm

The AES algorithm consists of ten rounds of encryption, as can be seen in Figure 1. First the 128-bit key is expanded into eleven so-called round keys, each of them 128 bits in size. Each round includes a transformation using the corresponding cipher key to ensure the security of the encryption.

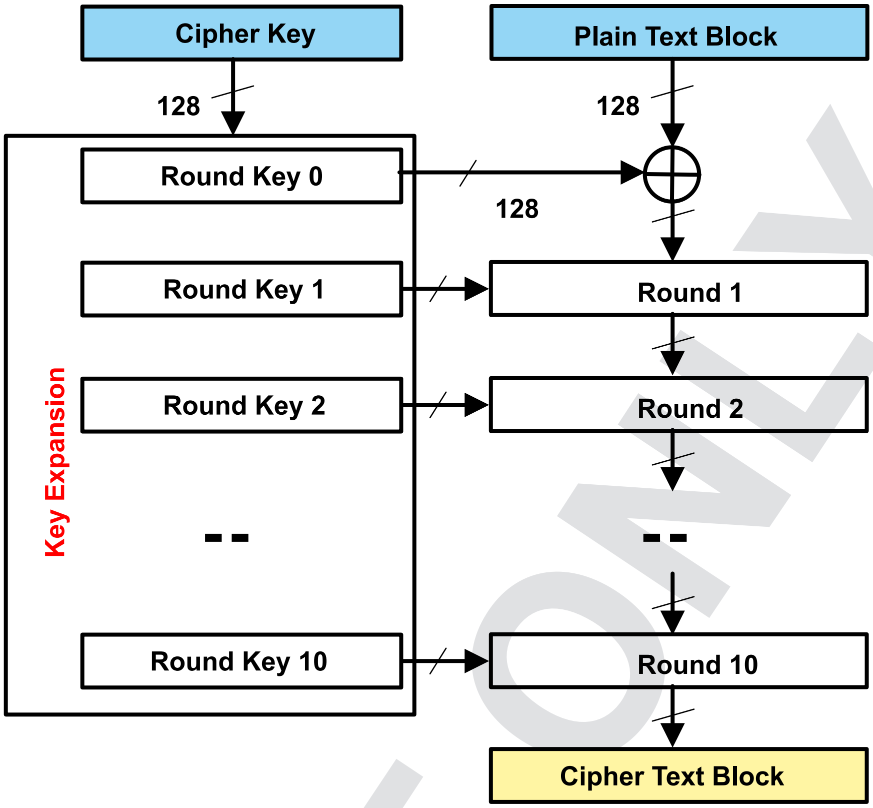


Figure 1. AES Algorithm Structure

After an initial round, during which the first round key is XORed to the plain text (Add roundkey operation), nine equally structured rounds follow. Each round consists of the following operations:

- Substitute bytes
- Shift rows
- Mix columns
- Add round key

The tenth round is similar to rounds one to nine, but the Mix columns step is omitted. In the following sections, these four operations are explained.

4.1.2 Structure of Key and Input Data

Both the key and the input data (also referred to as the state) are structured in a 4x4 matrix of bytes. Figure 2 shows how the 128-bit key and input data are distributed into the byte matrices.

The State

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

The Key

k_0	k_4	k_8	k_{12}
k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}

Figure 2. Structure of the Key and the State

4.1.3 Substitute Bytes (Subbytes Operation)

The Subbytes operation is a nonlinear substitution. This is a major reason for the security of the AES. There are different ways of interpreting the Subbytes operation. In this application report, it is sufficient to consider the Subbytes step as a lookup in a table. With the help of this lookup table, the 16 bytes of the state (the input data) are substituted by the corresponding values found in the table (see Figure 3).

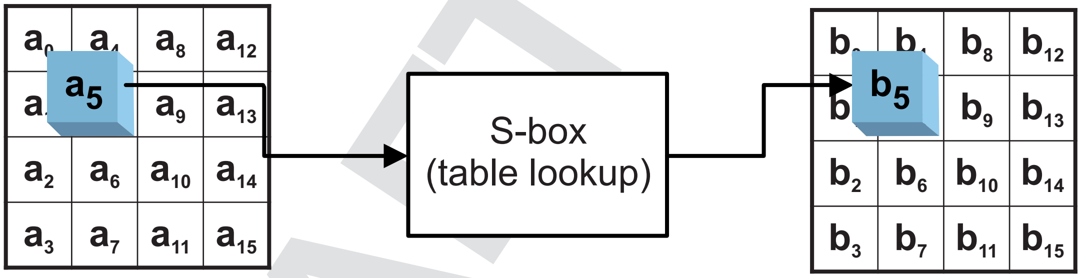


Figure 3. Subbytes Operation

4.1.4 Shift Rows (Shiftrows Operation)

As implied by its name, the Shiftrows operation processes different rows. A simple rotate with a different rotate width is performed. The second row of the 4x4 byte input data (the state) is shifted one byte position to the left in the matrix, the third row is shifted two byte positions to the left, and the fourth row is shifted three byte positions to the left. The first row is not changed.

Figure 4 illustrates the working of Shiftrows.

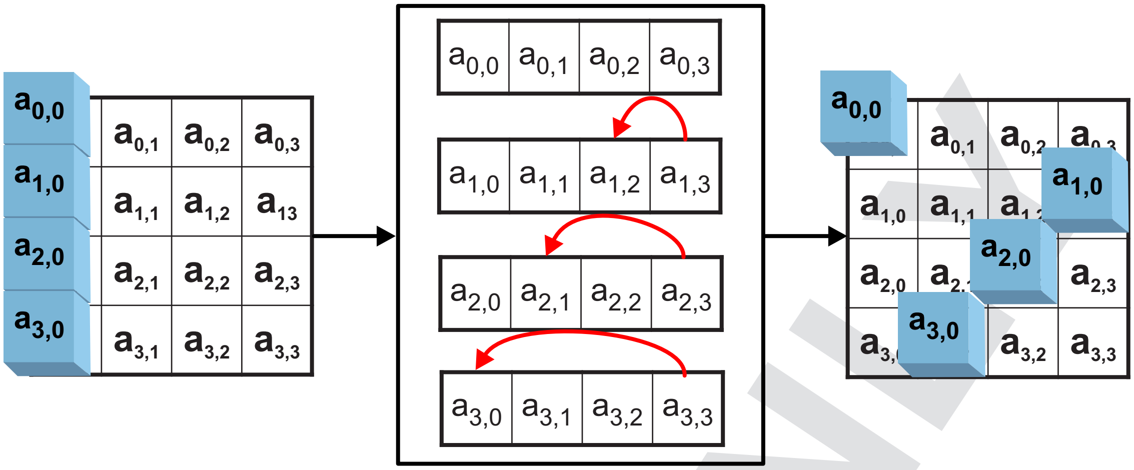


Figure 4. Shiftrows Operation

4.1.5 Mix Columns (Mixcolumns Operation)

Probably the most complex operation from a software implementation perspective is the Mixcolumns step. The working method of Mixcolumns can be seen in Figure 5.

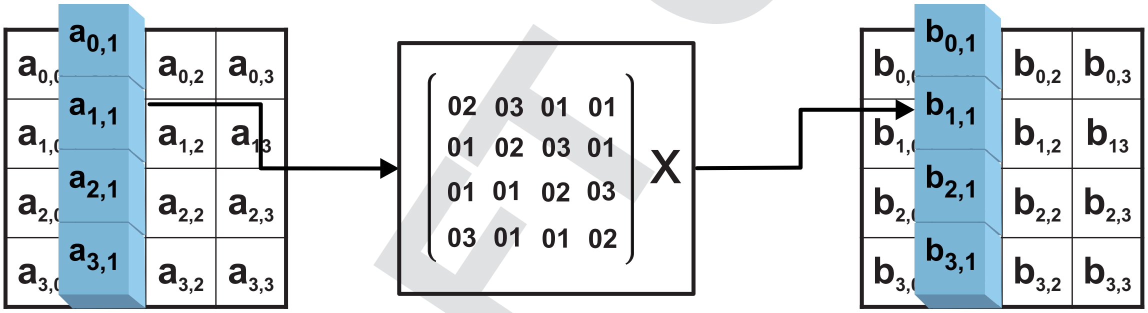


Figure 5. Mixcolumns Operation

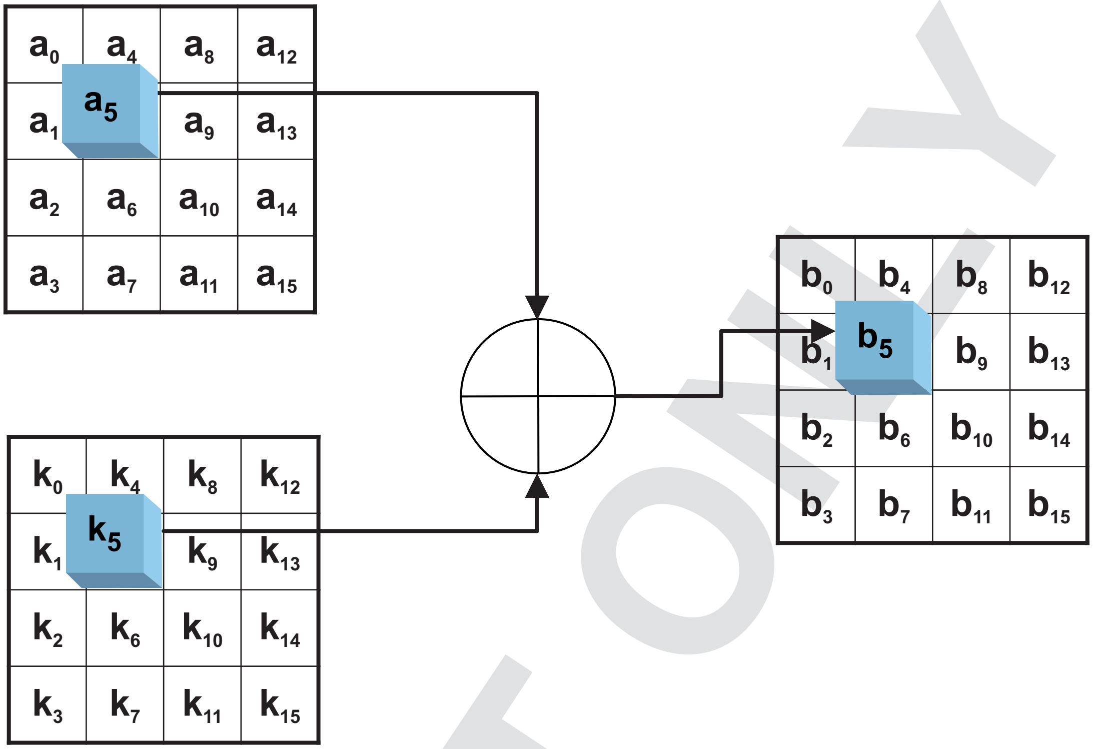
Opposed to the Shiftrows operation, which works on rows in the 4x4 state matrix, the Mixcolumns operation processes columns.

In principle, only a matrix multiplication needs to be executed. To make this operation reversible, the usual addition and multiplication are not used. In AES, Galois field operations are used. This document does not go into the mathematical details, it is only important to know that in a Galois field, an addition corresponds to an XOR and a multiplication to a more complex equivalent.

The fact that there are many instances of 01 in the multiplication matrix of the Mixcolumns operation makes this step easily computable.

4.1.6 Add Round Key (Addroundkey Operation)

The Addroundkey operation is simple. The corresponding bytes of the input data and the expanded key are XORed (see Figure 6).



a_0 a_4 a_8 a_{12}
a_1 a_5 a_9 a_{13}
a_2 a_6 a_{10} a_{14}
a_3 a_7 a_{11} a_{15}
k_0 k_4 k_8 k_{12}
k_1 k_5 k_9 k_{13}
k_2 k_6 k_{10} k_{14}
k_3 k_7 k_{11} k_{15}
b_0 b_4 b_8 b_{12}
b_1 b_5 b_9 b_{13}
b_2 b_6 b_{10} b_{14}
b_3 b_7 b_{11} b_{15}

Figure 6. Addroundkey Operation

4.1.7 Key Expansion (Keyexpansion Operation)

As previously mentioned, Keyexpansion refers to the process in which the 128 bits of the original key are expanded into eleven 128-bit round keys.

To compute round key (n+1) from round key (n) these steps are performed:

1. Compute the new first column of the next round key as shown in Figure 7:

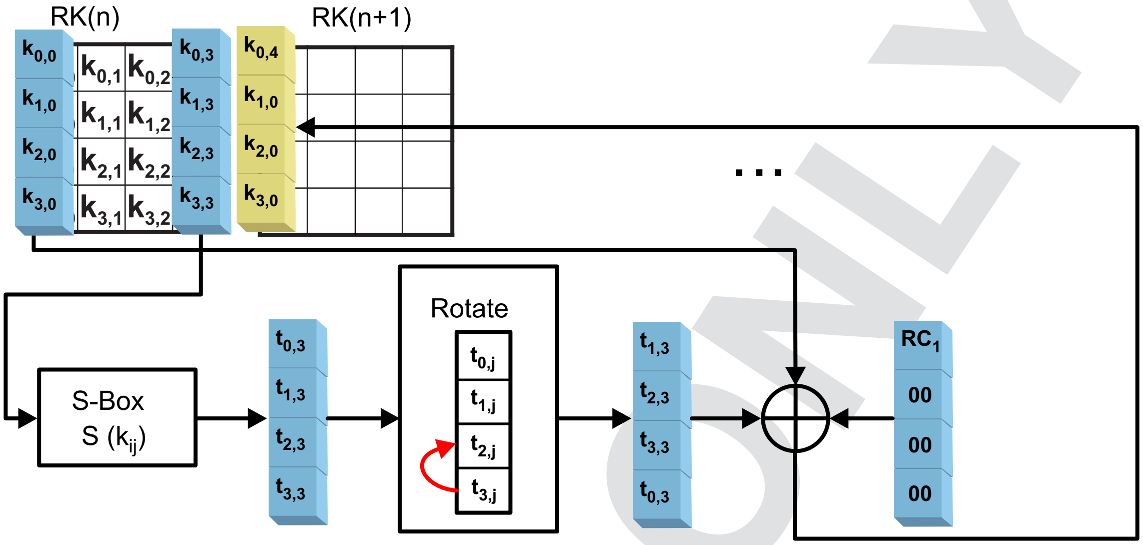

RK(n)
RK(n+1)
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$ $k_{0,4}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$ $k_{1,0}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$ $k_{2,0}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$ $k_{3,0}$
. . .
S-Box
S (k_{ij})
$t_{0,3}$ $t_{1,3}$ $t_{2,3}$ $t_{3,3}$
Rotate
$t_{0,j}$ $t_{1,j}$ $t_{2,j}$ $t_{3,j}$
$t_{1,3}$ $t_{2,3}$ $t_{3,3}$ $t_{0,3}$
RC_1
00
00
00

Figure 7. Expanding First Column of Next Round Key

First all the bytes of the old fourth column have to be substituted using the Subbytes operation. These four bytes are shifted vertically by one byte position and then XORed to the old first column. The result of these operations is the new first column.

2. Calculate columns 2 to 4 of the new round key as shown:

   (a) [new second column] = [new first column] XOR [old second column]

   (b) [new third column] = [new second column] XOR [old third column]

   (c) [new fourth column] = [new third column] XOR [old fourth column]

   Figure 8 illustrates the calculation of columns 2-4 of the new round key.

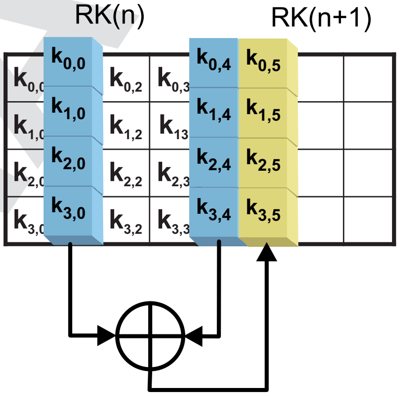

RK(n)
RK(n+1)
$k_{0,0}$ $k_{0,0}$ $k_{0,2}$ $k_{0,3}$ $k_{0,4}$ $k_{0,5}$
$k_{1,0}$ $k_{1,0}$ $k_{1,2}$ k_{13} $k_{1,4}$ $k_{1,5}$
$k_{2,0}$ $k_{2,0}$ $k_{2,2}$ $k_{2,3}$ $k_{2,4}$ $k_{2,5}$
$k_{3,0}$ $k_{3,0}$ $k_{3,2}$ $k_{3,3}$ $k_{3,4}$ $k_{3,5}$

Figure 8. Expanding Other Columns of Next Round Key

4.2 DES and 3DES

The Data Encryption Standard (DES) was developed in the 1970s by IBM and adopted as a standard by NIST by 1976. The DES algorithm itself has since then been declared insecure by NIST; however, it is believed to be reasonably secure in the form of Triple DES.

The DES algorithm consists of 16 rounds of data manipulation preceded by an initial permutation and followed by the inverse of the initial permutation. Figure 9 has a visual description of the algorithm structure. After the initial permutation, the data block is split in half into left and right blocks. The right block is sent through a function block with a round key and then is used as the left block for the next round. The left block is XOR'd with the result of the function block, the result of which is used as the right block in the next round. This is continued until the last round where the left and right blocks do not switch sides. At this point, the data is put through the inverse of the initial permutation resulting in the wanted cipher text.

4.2.1 DES Algorithm Structure

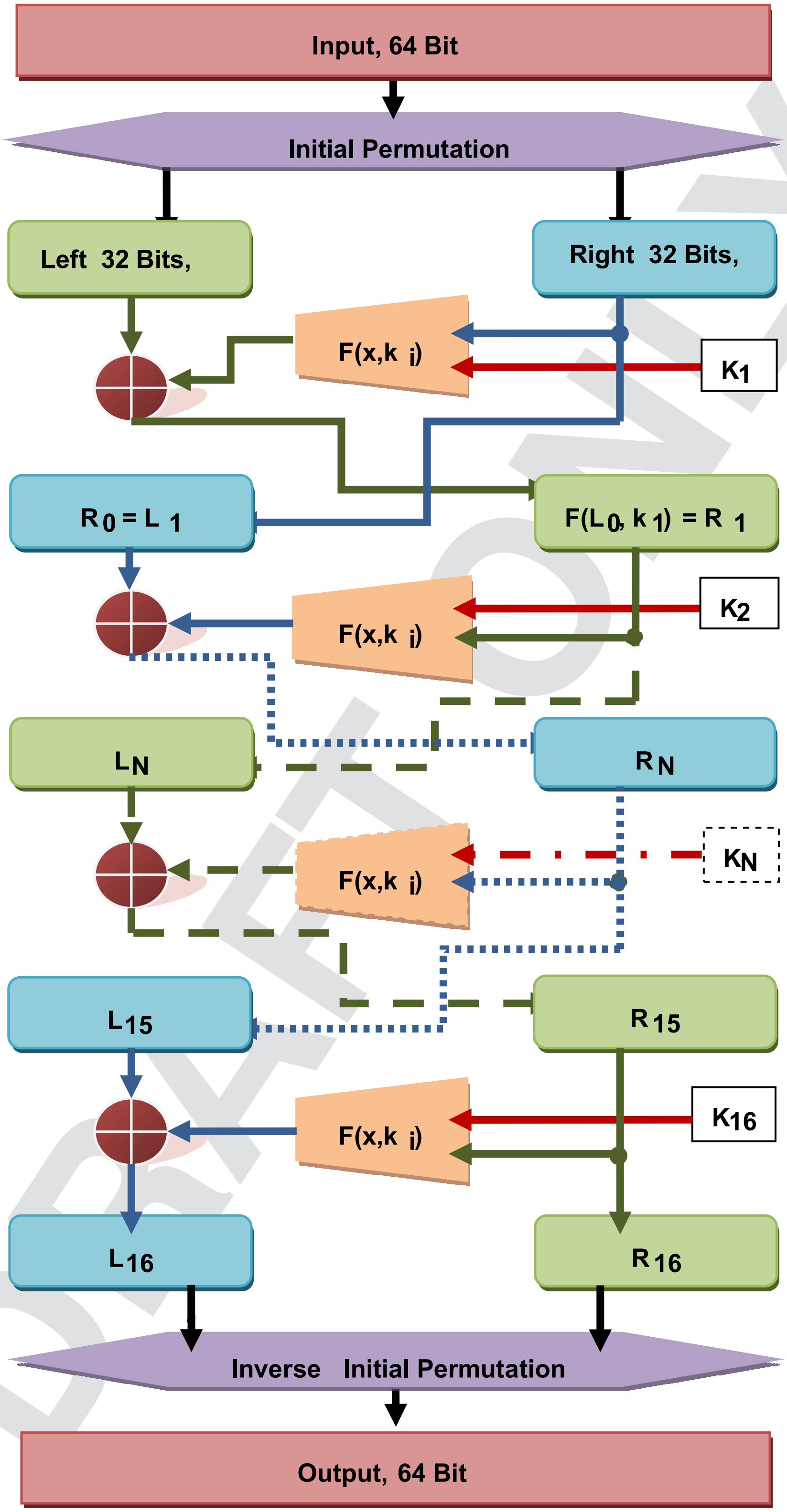


Figure 9. DES Algorithm Structure

4.2.2 The Function Block

The function block begins by expanding a 32-bit half block to 48 bits as shown in Figure 10.

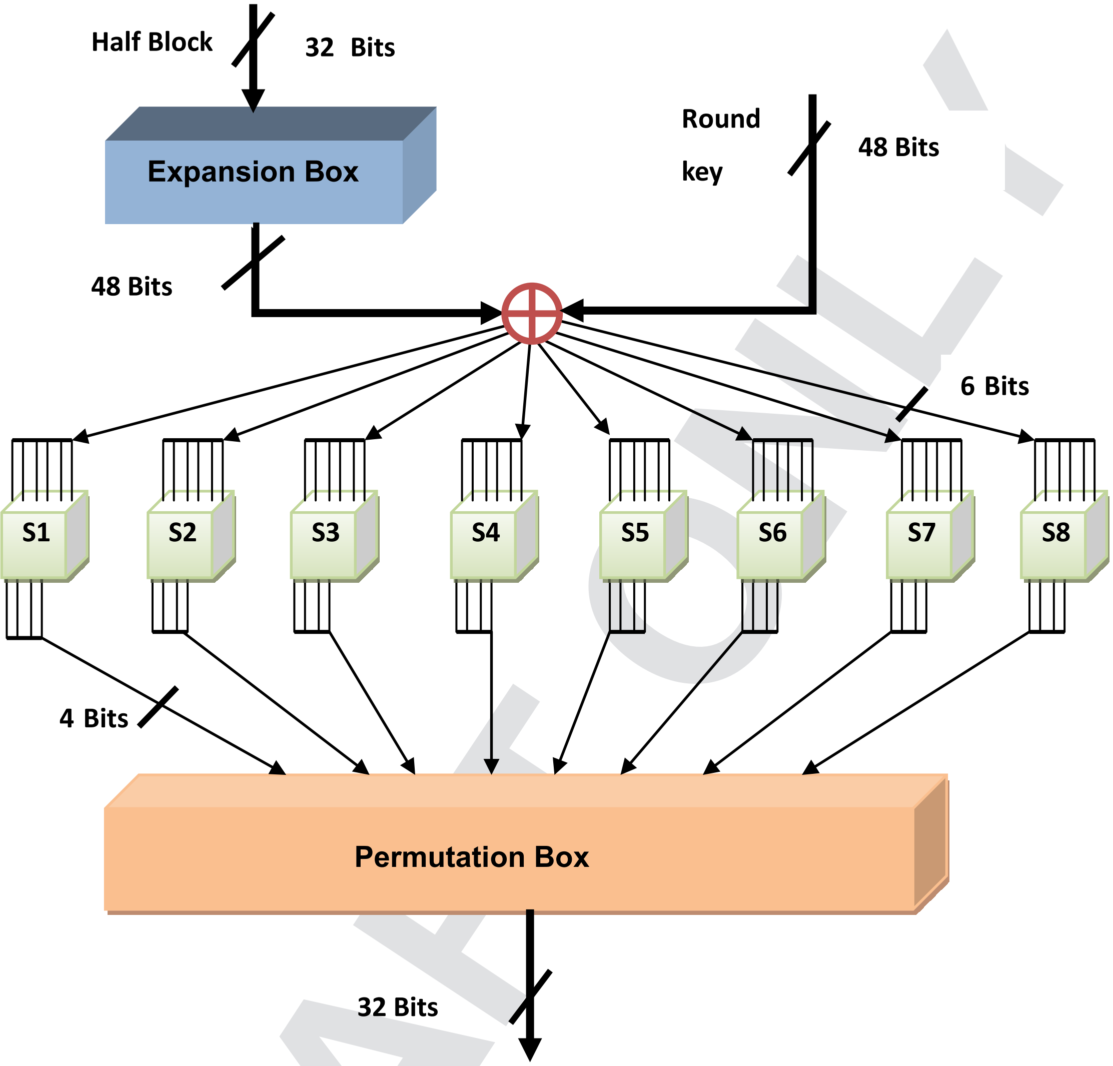


Figure 10. DES Function Block

The expanded block is then XOR'd with the round key. The resultant is the split into 6-bit increments and passed through eight S-boxes, with the six MSb going through S1 and the six LSb through S8. The S-boxes give 4-bit results which are concatenated (S1+S2+S3+S4+S5+S6+S7+S8) and sent through a 32-bit permutation box.

4.2.3 Key Schedule

The key schedule for all sixteen rounds of the DES algorithm must be calculated before encryption or decryption can occur. The key schedule process in this library is the most CPU intensive component of the algorithm. System speed can be increased by limiting the number of keys to be scheduled. Figure 11 describes how the key schedule is calculated. First, the 64-bit key is sent through a permutation box that reduces the bit count to 56. The result is split evenly and left rotated by 1-2 bits depending on the round. The rotate results are fed into a second permutation box that gives the round key used in the DES Function block.

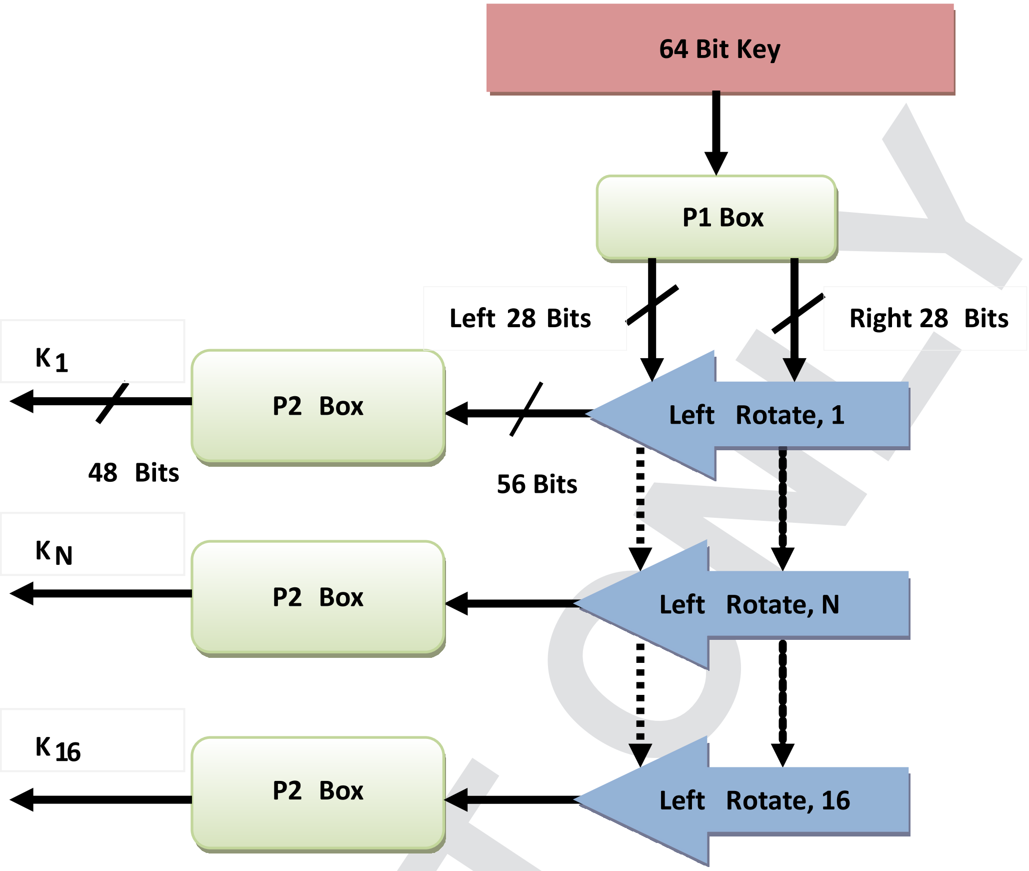


Figure 11. Key Schedule Function Diagram

4.2.4 Triple DES

Triple DES is a more secure form DES that implements three keys with a series of encodes and decodes. Figure 12 illustrates Triple DES Encoding and Decoding. In Triple DES, plain text is run through three alternating rounds of DES encoding and decoding with each round using a different key.

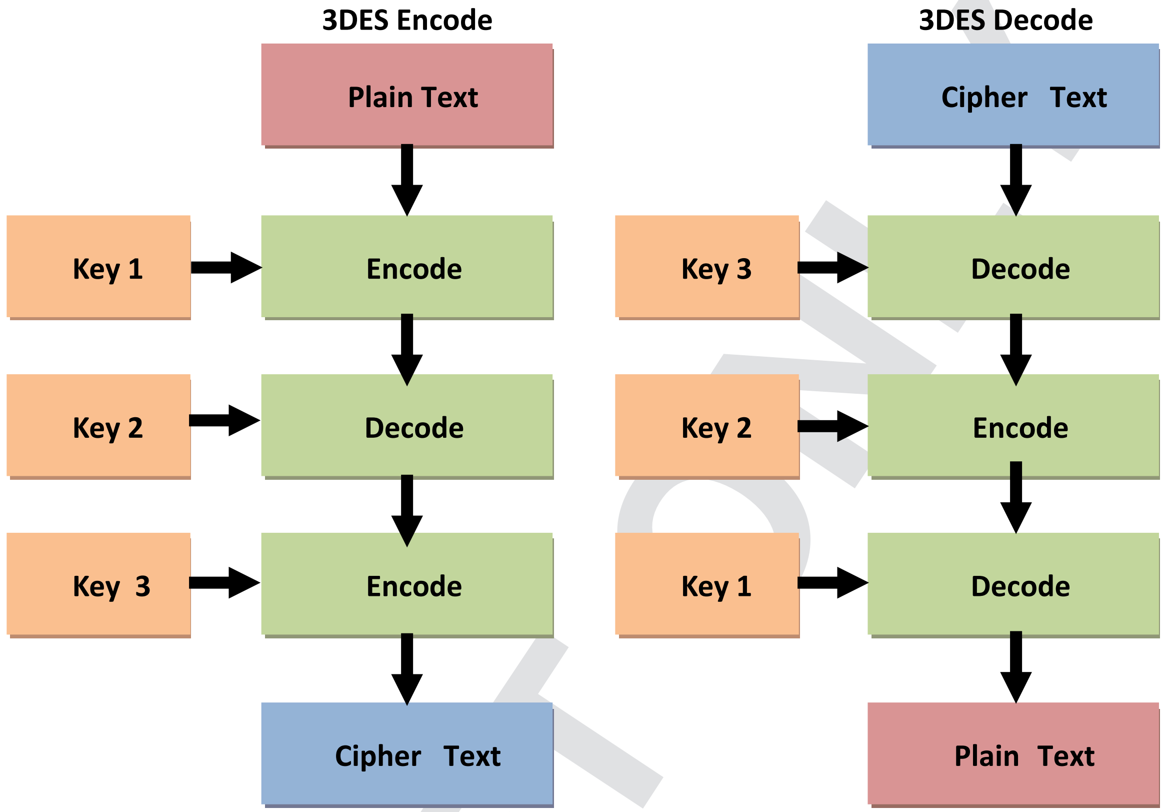


Figure 12. 3DES Encoding and Decoding Algorithms

4.2.5 Cipher Block Chaining (CBC) Mode

CBC is a common method to cipher multiple blocks of data. The mode introduces pseudo-randomness between cipher blocks in order to obscure data patterns between plaintext blocks. Figure 13 describes DES CBC modes for encryption and decryption.

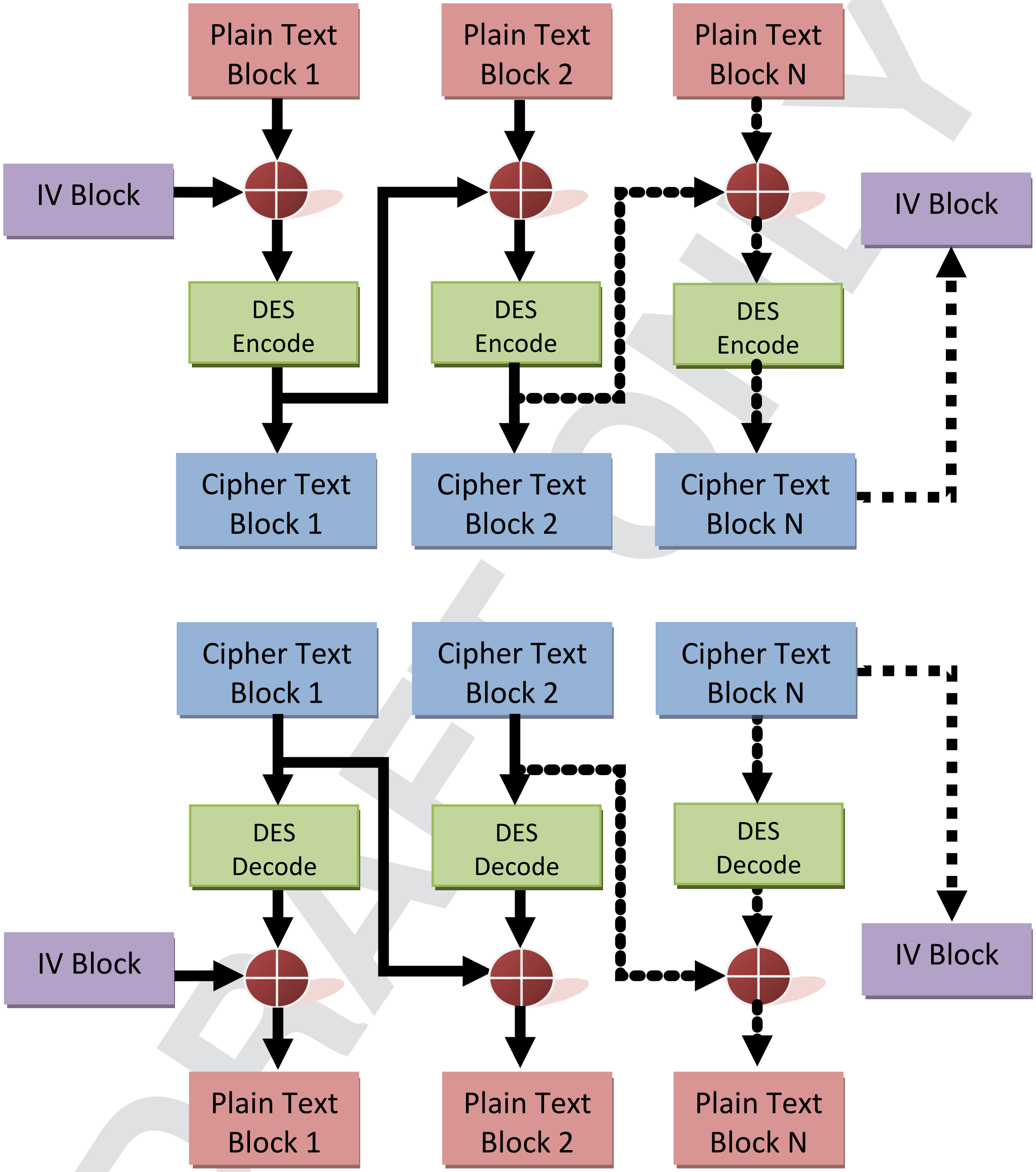


Figure 13. DES Encode and Decode in CBC Mode

Encoding in CBC modes begins with an XOR of the IV block and the first Plain text box. The result is encrypted to give the first block of Cipher text. This cipher text is then XOR’d with the next block of plaint text, which is then encoded. This process repeats until all data blocks are enciphered. The IV block is then updated to equal the last enciphered block.

Decoding in CBC happens in a similar way. In decoding, however, the XOR step happens after the decoding process. The first cipher text block is decoded then XOR'd with IV block to get the plain text. Continuing blocks are XOR'd with the previous cipher block after decoding, and the last cipher block is taken as the updated IV.

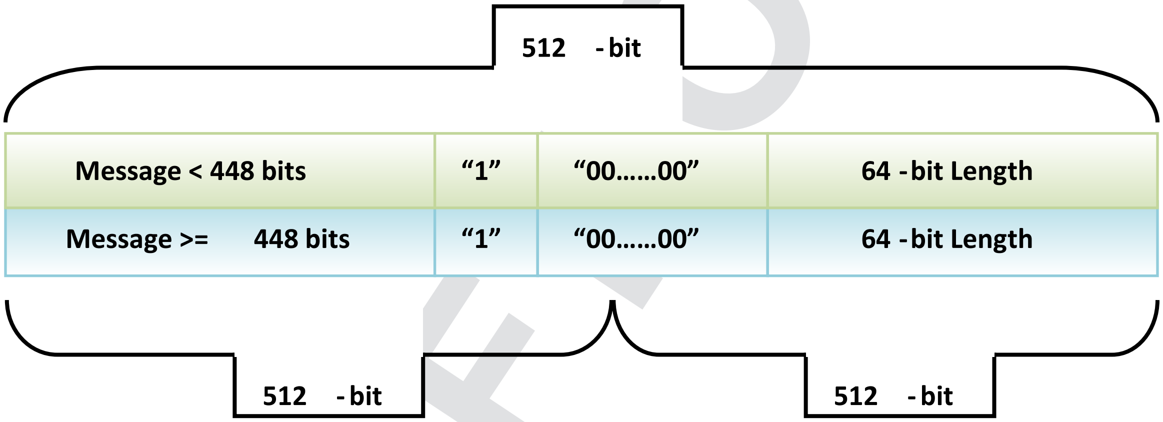
Triple DES with CBC works in the same way as DES with CBC. In Figure 13, replace the DES Encode module with 3DES Encode and the DES Decode module with 3DES Decode in order to have a visualization of the mode.

4.3 SHA-256 and SHA-224

Secure Hash Standard (SHA) 2 is a set of hashing algorithms developed by NIST in order to replace SHA-1. SHA-2 is a family of algorithms with message digests of 224, 256, 384 and 512 bits. The 224 and 384 variants are subsets of the 256 and 512, respectively. This library only implements SHA-256 and SHA-224.

4.3.1 Message Padding and Parsing

In order for a hash to be computed, the message must be padded to a multiple of a 512-bit hashing block. The last 64-bits of the last block is reserved for the bit count of the message. Figure 14 shows how padding is implemented. At the end of the message to be hashed a single "1" bit is appended followed by zeros. The zeroes continue until Message + Message Length + "1" + "00…00" = 512 bits.



Message < 448 bits	"1"	"00……00"	64 -bit Length
Message >= 448 bits	"1"	"00……00"	64 -bit Length

Figure 14. Example of Message Padding

4.3.2 SHA-256 Algorithm

The algorithm starts with an initialization vector of eight 32-bit words. These values are loaded into temp variables labeled A – H. A set of equations govern how these variables are combined and manipulated. The algorithm also calls for an array of hash constants (K_t), a message schedule (W_t), and the functions Ch, Ma, ∑0, and ∑1. The equations and functions are given in Section 4.3.3. Figure 15 gives a visualization of the hashing loop. This loop is repeated 64 times until the end of the message schedule. One message schedule covers only one hashing block of the full message. Once the loop is completed, the resulting temp variables are XOR'd with the initialization variables to form the current message digest H0-7. If other message blocks are to be processed, the temp values are loaded with the current message digest. At the end of the loop, the current results are XOR'd with the previous message digest. A full explanation of the algorithm can be found in FIPS PUB 180-3.

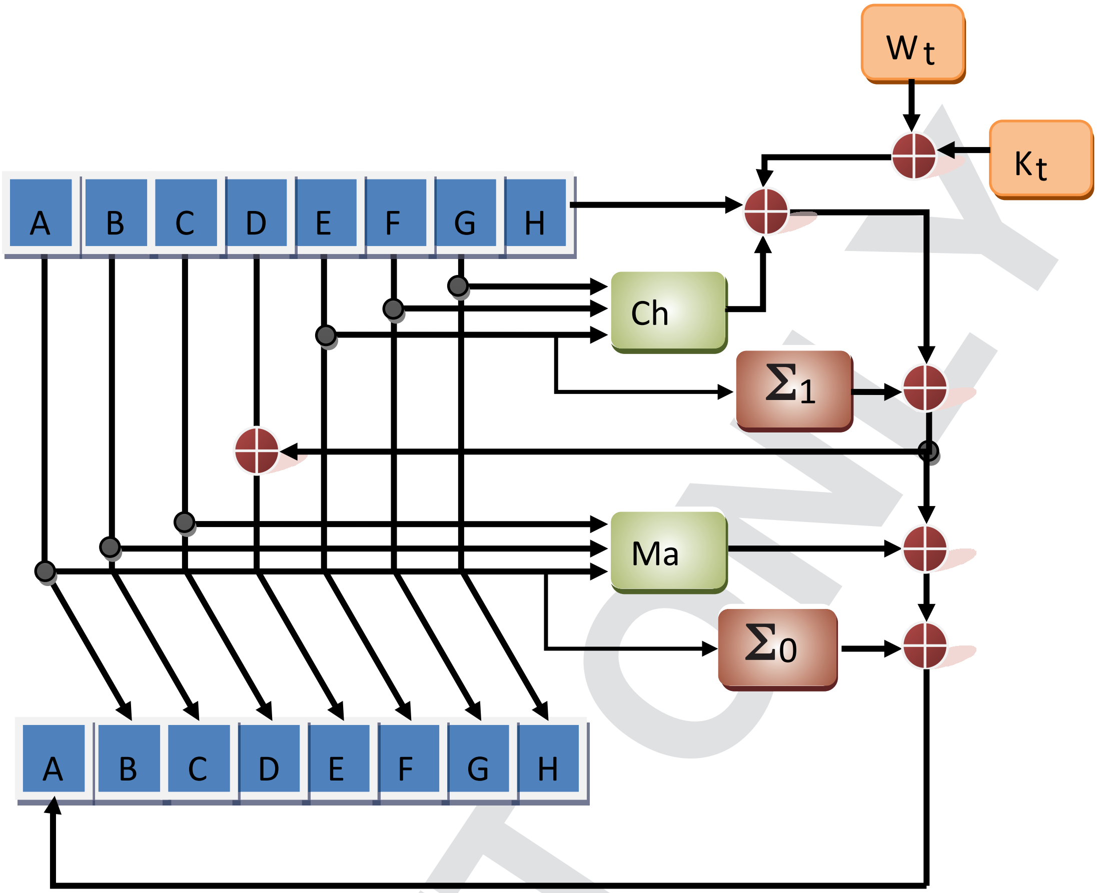


Figure 15. Visualization of the Hasing Loop of SHA-256

4.3.3 Equations Found in SHA-256 Algorithm

Symbols in Equations:

⊕ = Bitwise XOR
& = Bitwise AND
A' = Bitwise Compliment of A
>> = Shift Right
>>> = Rotate Right (1)

Functions:

ch(x,y,z) = (x & y) ⊕ (x' & z)

Ma (x, y, z) = (x & y) ⊕ (x & z) ⊕ (y & z)

σ_0(x)

$$\mathrm{ch}(x, y, z) = (x \,\&\, y) \oplus (x' \,\&\, z)$$

$$\mathrm{Ma}(x, y, z) = (x \,\&\, y) \oplus (x \,\&\, z) \oplus (y \,\&\, z)$$

$$\sigma_0(x) = (x \ggg 7) \oplus (A \ggg 18) \oplus (x \gg 3)$$

$$\sigma_1(x) = (x \ggg 17) \oplus (A \ggg 19) \oplus (x \gg 10)$$

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$

$$\Sigma_1(E) = (E \ggg 6) \oplus (A \ggg 11) \oplus (A \ggg 25)$$

$$\mathrm{W_t} = \boldsymbol{W_t} = \begin{cases} \boldsymbol{M_t^{(i)}}, & 0 \le t \le 15 \\ \boldsymbol{\sigma}_0(\boldsymbol{W_{t-2}}) \oplus \boldsymbol{W_{t-7}} \oplus \boldsymbol{\sigma}_1(\boldsymbol{W_{t-15}}) \oplus \boldsymbol{W_{t-16}}, & 16 \le t \le 15 \end{cases} \tag{2}$$

Loop Equations:

$$\boldsymbol{T_1} = \boldsymbol{h} \oplus \boldsymbol{K_t} \oplus \boldsymbol{W_t} \oplus \Sigma_1(\boldsymbol{E}) \oplus \boldsymbol{Ch}(\boldsymbol{e}, f, \boldsymbol{g})$$

$$\boldsymbol{T_2} = \boldsymbol{Ma}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) \oplus \Sigma_0(\boldsymbol{A})$$

$$\mathrm{h} = \mathrm{g}$$

$$\mathrm{g} = f$$

$$f = \mathrm{e}$$

$$\mathrm{e} = \mathrm{d} \oplus \mathrm{T_1}$$

$$\mathrm{d} = \mathrm{c}$$

$$\mathrm{c} = \mathrm{b}$$

$$\mathrm{b} = \mathrm{a}$$

$$\mathrm{a} = \mathrm{T_1} \oplus \mathrm{T_2} \tag{3}$$

4.3.4 SHA-224

SHA-224 is a subset of SHA-256 with a message digest of 224-bits. The algorithm is the same with the exception of different Hash initialization values. Also, only the first seven 32-bit (224-bits) words of the final message digest are used.

5 References

- Announcing the Advanced Encryption Standard (FIPS PUB 197)
- Data Encryption Standard (DES) (FIPS PUB 46-3)
- Security Hash Standard (SHS) (FIPS PUB 180-3)
- *AES128 – A C Implementation for Encryption and Decryption* (SLAA397)
- DES Modes of Operation (FIPS PUB 81)
- Schneier, Bruce; Applied Cryptography; John Wiley & Sons; 1996 (http://www.scheier.com/book-applied.html)