Abstract

In October 2000, Rijndael defeated all the other candidates and was announced as the winner of the contest of the new Advanced Encryption Standard (AES) by the National Institute for Standards and Technology (NIST). Since it was standardized, this encryption algorithm has become the most widely-used cryptosystem in the world due to its elegant, efficient and simple design with higher security features. On the other hand, it also becomes the focus of intensive research into efficient implementation techniques and associated techniques. In 2002, Barkan and Biham proposed the concept of dual ciphers of Rijndael and indicated several interesting applications of dual ciphers, and one of them was whether the dual ciphers would enjoy a better performance during encryption and decryption. This resulted in a range of variants, which so far have not been studied from a practical perspective. Since Rijndael is widely applied in many applications, such as smart card, network software and so forth, provided that there are variants retaining the similar security properties with higher performances, it could help dramatically improve the efficiency of all the applications using the algorithm.

In this thesis, we will follow the proposal of Barkan and Biham and investigate the variants of Rijndael with a focus on the performance. To be specific, we will review and implement Rijndael algorithms using several mainstream methods. After that, since there are 240 dual ciphers of Rijndael in total, before implementing the corresponding dual ciphers, we need to find the ones with relatively higher performance. Finally, after comparing and evaluating the original Rijndael ciphers with their corresponding dual ones with suitable benchmark methods, we draw conclusions with regards to the design of Rijndael.

- I implemented 5 mainstream Rijndael algorithms.
- I found certain dual ciphers with relatively higher performance and implemented them
- The research carried out and reported in this thesis consists of a comparison and evaluation of the performance of above 10 Rijndael implementation methods, which results in a conclusion that the dual ciphers retaining similar security properties could improve the performance of Rijndael algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Background

The Data Encryption Standard (DES), which was developed in 1970s, was the previously predominant algorithm for encryption of electronic data. However, with the development of computer hardware which accelerates attacks based on brute force search, it is no longer regarded as a secure encryption algorithm mainly because of the size of secret key. Thus, the National Institute for Standards and Technology (NIST) announced to a contest develop a new encryption standard, the Advanced Encryption Standard (AES), to replace the outdated and insecure DES.

In October 2000, after careful selection, Rijndael defeated all the other candidates and was announced as the winner and the contest of the new AES by the NIST. In the report of AES selection [9], it says,

"Rijndael appears to be consistently a very good performer in both hardware and software across a wide range of computing environments regardless of its use in feedback or non-feedback modes. Its key setup time is excellent, and its key agility is good. Rijndael's very low memory requirements make it very well suited for restricted space environments, in which it also demonstrates excellent performance. Rijndael's operations are among the easiest to defend against power and timing attacks. Additionally, it appears that some defense can be provided against such attacks without significantly impacting Rijndael's performance.

Finally, Rijndael's internal round structure appears to have good potential to benefit from instruction-level parallelism."

Since it was published, Rijndael has become the most widely-used cryptosystem in the world due to its elegant, efficient and simple design with high security features [9]. According to the cryptanalysis in [9], Rijndael is secure against all known attacks. Meanwhile, its elegant algebraic structure fits various platforms well. Therefore, the encryption algorithm of Rijndael is now not only adapted for civil use, such as smart cards, but also for military or government use. In June 2003, the U.S. Government announced that AES may be used to protect classified information.

1.2 Research Motivation

As a fairly new encryption algorithm, Rijndael, it becomes the focus of intensive research into efficient implementation techniques and attach techniques.In [1] that appeared at ASIACRYPT 2002, Barkan and Biham asked an interesting question: what if the constants, including the irreducible polynomial, the matrix in MixColumns operation and so on, applied in Rijndael are replaced by other parameters. They came to the conclusion that if the parameters were altered carefully according to some principles, the replacement would produce the new dual ciphers which would have the similar security properties. Meanwhile, they also put forward the hypothesis that such new dual ciphers might have some additional properties. One of them is that the dual ciphers might allow optimization of the cipher, which means in some case the dual ciphers might actually be faster to compute than the original one.

It is known to all that Rijndael plays increasingly significant roles in telecommunication, financial transactions and so on in our day-to-day life. Considering the majority of these applications are base on 8-bit platforms, e.g., bank cards, SIM cards, wireless sensors and so forth, and 32-bit or more platforms, say PC, Xbox360 or servers, it is quite necessary to investigate the performances of Rijndael on these platforms. Thus, if the performance of it is improved, all of its applications could benefit from it. A good example is that we may not need to suffer from the time costing confirmation of online transactions. What's more, in order to prevent the Differential Power Attack (DPA), we usually insert some dummy operations or do some masking operations which come at the cost of performance. In other words, sometimes for the sake of security, we have to suffer the inefficiency and dual ciphers with high performance could minimize its effects. Besides, the benefit of the dual ciphers could be combined with other optimizations, say AES-NI, which could improve the performance even more.

In addition, due to the fact that ordinary optimizations of Rijndael are neither efficient enough nor secure to satisfy present needs, bit-slicing, a more secure and efficient technique, will be applied in this project. As a matter of fact, bit-slicing is not a new encryption optimizing technique. Early in 1997 before Rijndael appeared, in [5], Biham put forward this fast method to accelerate the DES implementation in software. In [5], the author showed that by applying this technique the speed of encryption would be almost three times faster than the fastest implementation at that time. He also suggested that bit-slicing could be applied in any cipher. Due to these factors, just after Rijndael was selected as AES to take the place of DES, a large amount of research was conducted to apply bit-slicing to Rijndael. In [3, 12, 16, 24], how to use bit-slicing in Rijndael on different platforms are discussed.

This project will be focused on implementations of Rijndael and its dual ciphers on 8-bit and 32-bit platforms. At the same time, we will also study the performance of them with the bit-slicing technique from a practical perspective. The novelty of this project lies in implementing Rijndael variants which have not been studied from a practical stand-point. Moreover, the conclusion of it would be reused in the future block cipher designs so as to make them better. Now that Rijndael algorithm is nowadays widely applied in many applications, such as smart cards, network software and so forth, provided that there are Rijndael variants retaining the similar security properties with high encryption performances, it could help dramatically improve the efficiency of all the applications using Rijndael algorithm.

1.3 Objects and Aims

The overall aim of this project is to answer the question whether any variant of Rijndael provides an advantage in terms of non-security metrics, in particular performance. In order to achieve this aim, it has the following objectives:

- To survey state-of-the-art of Rijndael implementation.
- To implement the original Rijndael using variety of technologies, such as bit-slicing, according to current Rijndael algorithm.
- To develop a precise approach to benchmarking the implementation results above.
- To implement Rijndael variants with techniques used in the second bullet point.
- To evaluate and compare the comprehensive performances of different Rijndael variants via the methods mentioned above.
- To draw conclusions with regards to design of Rijndael.

1.4 Thesis Outline

The structure of the thesis is organized as follows,

Chapter 2 will give an overview of the background and context of the project and its relation to work already done in the area. We will introduce the Rijndael encryption algorithm, which is mainly composed of four steps, then we will show the concepts of Rindael dual ciphers mentioned in [1]. Finally, more emphasis will be focused on describing the state-of-the-art techniques used to implement Rijndael on different platforms.

Chapter 3 will exclusively describe the implementations of Rijndael and its variants. Following the methods mentioned in last chapter, we are going to inject the concept of dual ciphers into these methods during implementation. In terms of each technique on the platforms as well as the properties of dual ciphers, some performance analysis will be conducted roughly so that certain potential higher performance are put forward in theory. In this chapter, more attention would be paid to the bit-slicing technique since compared with other methods, it seems to be influenced much more by the dual cipher.

Chapter 4 will analyze the dual ciphers we found in Chapter 3 from both practical and theoretical aspects precisely so as to test and verify the analysis we made in terms of the efficiency.

The conclusion and further work will be described in Chapter 5.

Chapter 2

Technical Basis

In this chapter, we will first discuss the Rijndael algorithm which will be the core of the whole thesis. After that, the concept of dual ciphers will be introduced so that the discussion about Rijndael variants in the following chapters will be easy to understand. Finally, we will go through the state-of-the-art techniques applied to implement Rijndael, including the methods on 8-bit platforms and 32-bit platforms and bit-slicing technique.

Before introducing Rijndael algorithm, we would like to look at some concepts of block cipher first since it is a kind of block cipher. A block cipher is a deterministic algorithm operating on fixed-length groups of bits, which are called blocks. Usually block ciphers operate blocks of plain text with a block of secret key and produce blocks of cipher text accordingly. Below is the operation of a block cipher [29],

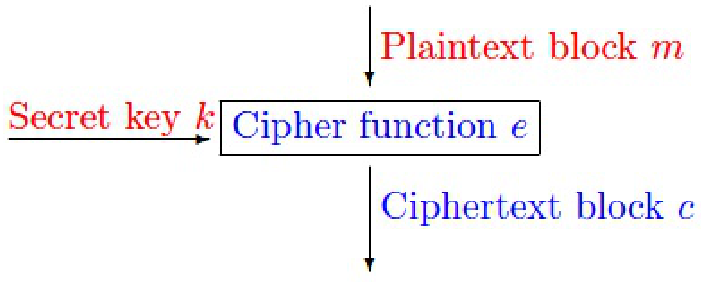


Figure 2.1: Block Cipher. (From [29])

The modern design of block cipher is based on the concept of an iterated product cipher which was suggested and analyzed by Claude Shannon [28]. The block cipher obtains the security by run a simple function multiple rounds by taking the output of last round as the input of this round and each round will use a new round key which is generated by key schedule. DES and all the final five candidates of AES apply the technique. Obviously, the more rounds are, the more security the cipher obtains.

2.1 Formal Description of Original Rijndael

As mentioned above, Rijndael was selected as the AES by the NIST. Hence, sometimes Rijndael is considered as the AES. Actually, Rijndael is a block cipher with variable key length and block length. Specifically, both key length and block length can be 128, 192 or 256 bits. On the other hand, the AES only supports key lengths of 128, 192 or 256 bits with the fixed block length which is 128 bits. In the following discussion, for simplicity sake, we only consider the Rijndael whose block length and key length are 128 bits with 10 rounds iteration.

2.1.1 Overview of Rijndael

There are four main transformations in the whole Rijndael algorithm. They are `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. All the transformations operate on an intermediate result which is called the state. The Rijndale algorithm (Algorithm 1) is as follows [9],

Algorithm 1 Rijndael Encryption Pseudo-code

```
Input: A 128-bit plain text P; An 11-element sequence of 128-bit round keys K;
Output: A 128-bit cipher text C;
 1: $S = P$;
 2: $S = AddRoundKey(S, K_0)$;
 3: for $i = 1$ to 9 do
 4:     $S = SubBytes(S)$;
 5:     $S = ShiftRows(S)$;
 6:     $S = MixColumns(S)$;
 7:     $S = AddRoundKey(S, K_i)$;
 8: end for
 9: $S = SubBytes(S)$;
10: $S = ShiftRows(S)$;
11: $C = AddRoundKey(S, K_{10})$;
12: return C;
```

The plain text and round keys can be regarded as a 4 by 4 element two-dimensional byte array (which is 128 bits in total), such as

$$\begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix}$$

2.1.2 The SubBytes Transformation

This transformation, actually, is combined with two parts, linear operation and non-linear operation. The linear operation is an affine transformation which is defined as follows,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{2.1}$$

Note that $b_0 ...\ b_7$ is the outcome of `SubBytes` transformation and $a_0 ...\ a_7$ is the intermediate value which is computed in the non-linear operation.

The non-linear operation is to compute the multiplicative inverse of an element whose length is 8 bits over GF(2^8) via the irreducible polynomial

$$P(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.2}$$

That is to say, given an element a, its multiplicative inverse element a^{-1} over GF(2^8) is defined as,

$$a * a^{-1} = 1 \bmod P(x) \tag{2.3}$$

Obviously, the calculating the multiplicative inverse over definite field is not as easy as that in real number field. There are a few options. They are either quite inefficient or relatively hard to conduct.

- Going through all the element in the field to get the multiplicative inverse.
- Using Extended Euclidean Algorithm (XGCD) to computer the multiplicative inverse.
- Computing a^{254} which is equal to a^{-1} according to Fermats little theorem.
- Decomposing GF(2^8) into smaller fields which is isomorphic to each other for computing the multiplicative inverse over smaller fields will be easier.

On the other hand, because of the complexity of calculating the multiplicative inverse, usually it is done off-line rather than online and all the outcomes stored in the table called S-Box. The transformation is as follows, where $b_{i,j}$ = S-Box($a_{i,j}$)

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \longrightarrow \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} \tag{2.4}$$

2.1.3 The ShiftRows and MixColumns Transformation

2.1.3.1 The ShiftRows Transformation

This transformation is quite simple. The first row is not shifted, the second row is shifted 1 byte from right to left, the third and fourth row are shifted 2 and 3 bytes respectively.

$$\begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} \longrightarrow \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix} \tag{2.5}$$

2.1.3.2 The MixColumns Transformation

In the `MixColumns` transformation, each column is multiplied by a fixed 4 by 4 matrix in GF(2^8) and the polynomial is

$$x^4 + 1 \tag{2.6}$$

The outcome is computed by the equation as follows,

$$\begin{pmatrix} b'_{0,j} \\ b'_{1,j} \\ b'_{2,j} \\ b'_{3,j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \odot \begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \tag{2.7}$$

Note that this transformation is only involved with multiplication by 02 and 03, which means only two operations (shift and XOR) are needed (multiplication by 02 or doubling in computer means a shift operation). The multiplication by 02 could be implemented with a shift operation and a conditional XOR operation, which depends on whether the most significant bit is one or not. As for the 03, when we get the result of multiplication by 02, we could just conduct the XOR operation.

2.1.4 The AddRoundKey Transformation and Key Schedule

2.1.4.1 The AddRoundKey Transformation

The operation in `AddRoundKey` transformation is fairly easy since it is only involved in XOR operation. What we need to do is to XOR the state values with the round keys byte by byte. The transformation is as follows,

$$\begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \oplus \begin{pmatrix} k_{0,4k} & k_{0,4k+1} & k_{0,4k+2} & k_{0,4k+3} \\ k_{1,4k} & k_{1,4k+1} & k_{1,4k+2} & k_{1,4k+3} \\ k_{2,4k} & k_{2,4k+1} & k_{2,4k+2} & k_{2,4k+3} \\ k_{3,4k} & k_{3,4k+1} & k_{3,4k+2} & k_{3,4k+3} \end{pmatrix} \tag{2.8}$$

where k is the round number which is from 0 to 10.

2.1.4.2 Key Schedule

Now the only thing left is to compute the 11-element sequence of 128-bit round keys from the original secret key. The key generation algorithm (Algorithm 2) is as follows [29], where K_0,...K_{11} is the 11 round keys and each of them is 128 bits. `RotBytes` is the function which rotates a word to the left by a single byte and `SubBytes` is one of the transformation in Rijndael. RC_i are a sequence of round constants.

Algorithm 2 Rijndael Key Generation Pseudo-code

```
Input: A 128-bit secret key K;
Output: An 11-element sequence of 128-bit round keys K;
1: $W_0 = K_0, W_1 = K_1, W_2 = K_2, W_3 = K_3$;
2: for $i = 1$ to 10 do
3:     $T = RotBytes(W_{4i-1})$;
4:     $T = SubBytes(T)$;
5:     $T = T \oplus RC_i$;
6:     $W_{4i} = W_{4i-4} \oplus T$;
7:     $W_{4i+1} = W_{4i-3} \oplus W_{4i}$;
8:     $W_{4i+2} = W_{4i-2} \oplus W_{4i+1}$;
9:     $W_{4i+3} = W_{4i-1} \oplus W_{4i+2}$;
10: end for
```

2.1.5 Decryption

The decryption algorithm is quite similar to the encryption algorithm. What we need to do is just to perform all the operations reversely. The decryption algorithm (Algorithm 3)is as follows, where `InverseSubBytes, InverseShiftRows` and `InverseMixColumns` are the inverse operations accordingly.

2.1.6 Further Discussion and Analysis

Rijndael, as an excellent symmetric key encryption algorithm, has many advantages compared with other block ciphers, say DES and the other four candidates of AES in the final round.

First of all, we will compare Rijndael with DES. In terms of security, Rijndael has absolute advantages over DES. The key size and block size of DES is 56 bits and 64 bits respectively. However, Rijndael has at least 128-bit key size and block size, which means it will be more difficult to analyze Rijndael. In addition, as discussed above, Rijndael consists of three parts, non-linear transformation, linear transformation and key generation, where non-linear transformation prevent the linear analysis and differential analysis which are successfully applied to the DES; linear transformation diffuses the data distribution; key generation generate a new key every round. What's more, Rijndael works more efficiently than DES does. While implementing DES,

Algorithm 3 Rijndael Decryption Pseudo-code

Input: A 128-bit cipher text C; An 11-element sequence of 128-bit round keys K;
Output: A 128-bit plain text P;

```
1:  $S = C$;
2:  $AddRoundKey(S, K_{10})$;
3:  $InverseShiftRows(S)$;
4:  $InverseSubBytes(S)$;
5:  for $i = 9$ to 1 do
6:      $S = AddRoundKey(S, K_i)$;
7:      $S = InverseMixColumns(S)$;
8:      $S = InverseShiftRows(S)$;
9:      $S = InverseSubBytes(S)$;
10: end for
11: $P = AddRoundKey(S, K_0)$;
12: return $P$;
```

we need 16 rounds to iterate with 8 S-Boxes. On the other hand, only 10 rounds with 1 S-Box are needed in Rijndael. Besides, the structure of Rijndael enables it easily to be implemented on various platforms efficiently compared with DES. While implementing DES, even with various optimizations, it is still not very efficient in software in that the majority operations in DES are involved in permutations which are easy to rewrite in hardware but lead to a major performance bottleneck in software. However, Rijndael does not involve permutations.

Now let us move on to the other candidates of AES in the final round according to [18].

Table 2.1: The comparison of AES candidates. (From [18])

	key size	block size	rounds	main operations
Serpent	128, 192, 256 bits	128 bits	32	Substitution-permutation network
Twofish	128, 192, 256 bits	128 bits	16	Feistel network
RC6	128, 192, 256 bits	128 bits	20	Feistel network
MARS	128, 192, 256 bits	128 bits	32	Type-3 Feistel network
Rijndael	128, 192, 256 bits	128 bits	10, 12 or 14	Substitution-permutation network

1.Serpent

It has a block size of 128 bits and supports a key size of 128, 192 or 256 bits with 32 rounds. And Each round applies one of eight 4-bit to 4-bit S-Boxes 32 times in parallel [23]. So actually, Serpent is more securer than Rijndael. However, the evaluation criteria of AES are not just according to security aspect. Obviously, Serpent needs more S-Boxes each round and the number of rounds are more than 3 times of Rijndael. So it will works much slower than Rijndael does.

2.Twofish

Its block size and key size are the same as Serpent but with 16 rounds. But it is vulnerable to Key-dependent S-Boxes complicate analysis. Meanwhile, its design is too complicated.

3. RC6 and MARS

The disadvantages of RC6 is its lower security margin.Besides, while implemented on 8-bit platform, it needs pre-computing operation, which will be a big problem for many low-end smart cards. As for MARS, its problems are also obvious. Its design is too complex and it is not suitable for 8-bit platform, say smart cards, if without modification.

2.2 Variants of Rijndael

In [1], Barkan and Biham presented the concept of dual ciphers which are equivalent to the original in all aspects in terms of security properties. The definition of a dual cipher is as follows:

Definition 1 [1] *Two ciphers E and E′ are called Dual Ciphers, if they are isomorphic, i.e., if there exist invertible transformations f, g and h such that*

$$\forall P, K f(E_K(P)) = E'_{g(K)}(h(P)), \tag{2.9}$$

where P and K refer to the plain text and secret key respectively.

Note: it is obvious that if two ciphers are isomorphic or dual, the plain text, secret key and cipher text all have certain transformation.

In [1], they also define all the operations of Rijndael as follows:

Definition 2 [1]

-Operations in GF(2^8):

1. *Addition*
2. *XOR with a constant*
3. *Multiplication*
4. *Multiply by a constant*
5. *Raise to any power. This includes the inverse of x: x^{-1}*
6. *Any replacement of the order of elements*

   -Non-GF(2^8) operations:
7. *Linear transformations L(x) = Ax, for any boolean matrix A.*
8. *Any unary operation over elements in GF(2^8).*

2.2.1 Square Dual Ciphers

In terms of Definition 1, if we set transformation as f(x) = g(x) = h(x) = x^2 , then the equation become

$$(E_K(P))^2 = E^2_{K^2}(P^2), \tag{2.10}$$

So E^2 is the square dual cipher of E.Note that all the square computations are implemented in GF(2^8).

Meanwhile according to [1], E^2 is defined by modifying the constants of E. So except the operations which are involved in constants, all the operations remain the same. That is to say, only the second, the fourth, the seventh and the last operations above require to be modified (raised to square). As discussed above, it is obvious that the constants which are involved in these operations in Rijndael are just constant matrix in Equation (2.7) and the matrix and vector in Equation (2.1) and the round constant RC_i in key schedule. The transformation is square, so for polynomials, vectors and round constant RC_i, just square them, and for matrix A, they are converted to QAQ^{-1} , where Q and $Q^{-1}$are defined in [1] as follows,

$$Q = \begin{pmatrix} 0&0&0&0&0&0&1&1 \\ 0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&1&1&0 \\ 0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&1&1&1 \\ 0&1&0&0&0&1&0&0 \\ 0&0&0&0&1&0&1&1 \\ 1&0&0&0&1&0&1&0 \end{pmatrix}, Q^{-1} = \begin{pmatrix} 1&0&0&1&0&1&0&1 \\ 0&1&1&1&0&0&0&0 \\ 0&0&0&1&1&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&1&0&0&0&0&0&1 \\ 0&1&0&1&0&0&0&0 \\ 0&1&0&1&0&1&0&0 \\ 0&1&0&1&0&1&0&1 \end{pmatrix} \tag{2.11}$$

Specifically, the Equation (2.7) is converted to the polynomial

$$\begin{pmatrix} b'_{0'j} \\ b'_{1'j} \\ b'_{2'j} \\ b'_{3'j} \end{pmatrix} = \begin{pmatrix} 04&05&01&01 \\ 01&04&05&01 \\ 01&01&04&05 \\ 05&01&01&04 \end{pmatrix} \odot \begin{pmatrix} b_{0'j} \\ b_{1'j} \\ b_{2'j} \\ b_{3'j} \end{pmatrix} \tag{2.12}$$

Similarly, the affine transformation (2.2) can be regarded as Ax+b now becomes QAQ^{-1}x + b^2 , and we finally get the new transformation is,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 0&1&1&0&0&1&0&0 \\ 1&0&1&0&0&0&1&1 \\ 1&0&0&1&1&0&1&1 \\ 1&0&0&0&1&1&0&0 \\ 0&1&0&1&0&0&0&0 \\ 0&0&1&0&0&1&0&0 \\ 1&1&0&1&0&0&0&1 \\ 0&1&0&0&1&0&0&1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{2.13}$$

As for Round Constant RC_i which was (02^{i-1}), it now can be replaced by (03^{i-1}).

Thus, the only problem needs to be solved is how to compute matrix Q and Q^{-1}. Actually, Q=(x^0),(x^2),(x^4),(x^6),(x^8),(x^{10}),(x^{12}),(x^{14})mod the polynomial (2.2), and x is vector.

So above is all the transformation from E to E^2, and similarly we could produce E^4 from E^2. Furthermore, E^8,E^{16},E^{32},$E^{64}$and $E^{128}$are all dual ciphers, and totally there are 8 square dual ciphers.

2.2.2 Irreducible Polynomials

In the last subsection, most of the constants such as the vector and the matrix in (2.1), the constant matrix in Equation (2.7) and the round constant RC_i have been replaced. However, what about the others, say the irreducible polynomials which are used to do modular arithmetic? In [1], the authors claim "There are 30 irreducible polynomials of degree 8" and conclude "the choice of the irreducible polynomial of Rijndael is arbitrary".

The definition of irreducible polynomial dual cipher is quite similar to that in the last subsection. According to the Definition 1, if we set the transformation as f(x) = g(x) = h(x) = Rx, then the equation becomes

$$E_K(P)R = E^R_{KR}(PR), \tag{2.14}$$

where R is a binary matrix which transforms the origin polynomial to the new one.

Since there are 30 irreducible polynomials of degree 8 and each of them has 8 square dual ciphers, the total number of dual cipher will be 240. In [32], the authors gave a more general way to represent these 240 dual ciphers. Specifically, the irreducible polynomials are regarded as the two numbers with binary or hexadecimal format. For example, for the polynomial

$$P(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.15}$$

it can be perceived as {00011011} or {1B}. As for the square dual ciphers, they use the different generators to replace. (The generator should be a primitive element and for the original Rijndael, it is chosen as {03}.)Thus, all of the dual ciphers can be represented as the format {R(x), b}, where R(x) is the polynomial in hexadecimal format and b is the generator.

According to algorithm in [32], the mapping matrix of dual cipher is formed from generator b, such as T = [b^0, b^{25}, b^{50}, b^{75}, b^{100}, b^{125}, b^{150}, b^{175}]. Therefore, if the dual cipher is {R(x), b}, then each transformation will be as follows.

As mentioned above, the `AddRoundKey` and the `ShiftRows` transformations are not influenced by dual ciphers. So are their inverse operations. As for the `MixColumns` transformation, the four elements in the matrix will be replaced by [b^{25}, b^0, b^0, b] and the four elements in its inverse operation will be [b^{223}, b^{199},b^{228}, b^{104}]. Finally, the `SubByes` transformation will become T(const) + (T * A * T^{-1}) * a, where T is the mapping matrix mentioned above, A is the affine transformation in Equation (2.1) and the const is the constant (63).

Now we will take {{11d}, {02}} as an example. Both `AddRoundKey` and `ShiftRows` transformations keep the same operations as the original. However, after multiplied with mapping matrix, the Equation (2.1) becomes

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{2.16}$$

Similarly, the `MixColumns` transformation becomes,

$$\begin{pmatrix} b'_{0,j} \\ b'_{1,j} \\ b'_{2,j} \\ b'_{3,j} \end{pmatrix} = \begin{pmatrix} 03 & 02 & 01 & 01 \\ 01 & 03 & 02 & 01 \\ 01 & 01 & 03 & 02 \\ 02 & 01 & 01 & 03 \end{pmatrix} \odot \begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \tag{2.17}$$

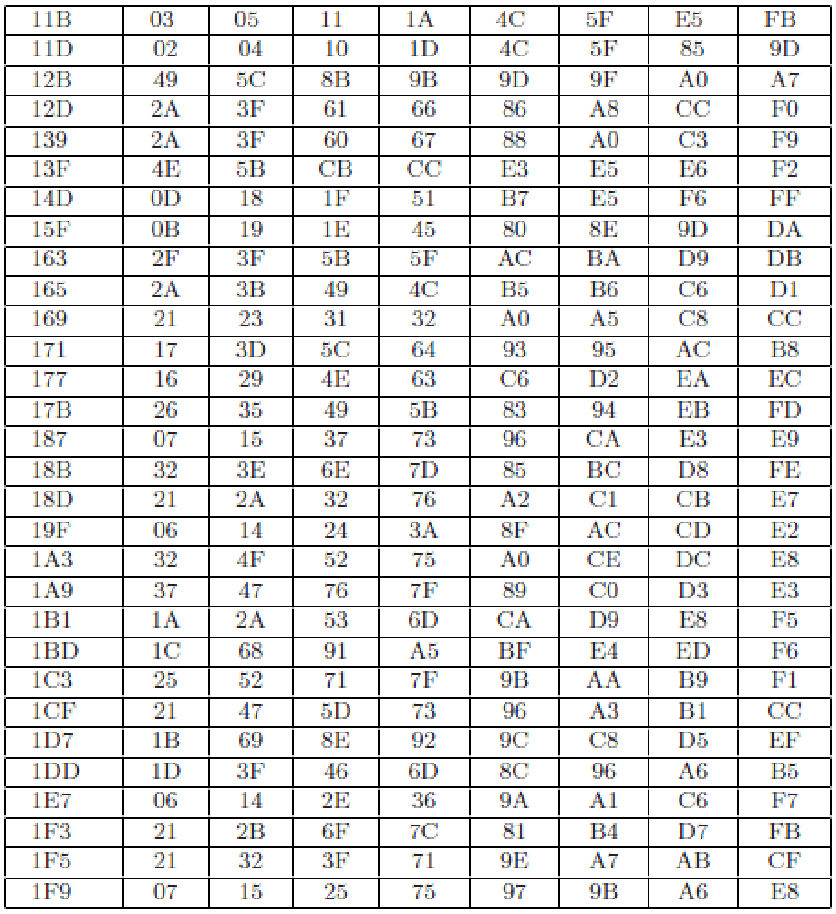
Now the only question left is how to get all the corresponding generators. The full description of the dual ciphers could be found in [2]. Below (Figure 2.2) is all the dual ciphers in the format such as {R(x), b} from [32]. The solution to find all the generator b has been given in [32].

2.3 Some Basic Concepts

Since most of the optimization techniques for Rijndael, to some extent, are usually relevant to or based on the computer architecture. Therefore, it is necessary to give a rough description of the computer architecture so that the following optimization implementations would not be so obscure to understand.

2.3.1 Cache Memory and Register

In modern computer architecture, the heart of a computer is the microprocessor, which is also commonly called the Central Processing Unit (CPU). CPU which is a single silicon chip consists of the Arithmetic Logic Unit(ALU) and the Control Unit(CU), while ALU is used to handle with arithmetic operation, CU is used to control the flow of data through the processor. Before conducting the arithmetic operation, we must fetch the data from storage as fast as possible so that the operations can be carried out quickly [30]. Since ALU is the unit which handle with arithmetic operation, if the data storage is as close as possible to the ALU, the operations would be carried out



11B	03	05	11	1A	4C	5F	E5	FB
11D	02	04	10	1D	4C	5F	85	9D
12B	49	5C	8B	9B	9D	9F	A0	A7
12D	2A	3F	61	66	86	A8	CC	F0
139	2A	3F	60	67	88	A0	C3	F9
13F	4E	5B	CB	CC	E3	E5	E6	F2
14D	0D	18	1F	51	B7	E5	F6	FF
15F	0B	19	1E	45	80	8E	9D	DA
163	2F	3F	5B	5F	AC	BA	D9	DB
165	2A	3B	49	4C	B5	B6	C6	D1
169	21	23	31	32	A0	A5	C8	CC
171	17	3D	5C	64	93	95	AC	B8
177	16	29	4E	63	C6	D2	EA	EC
17B	26	35	49	5B	83	94	EB	FD
187	07	15	37	73	96	CA	E3	E9
18B	32	3E	6E	7D	85	BC	D8	FE
18D	21	2A	32	76	A2	C1	CB	E7
19F	06	14	24	3A	8F	AC	CD	E2
1A3	32	4F	52	75	A0	CE	DC	E8
1A9	37	47	76	7F	89	C0	D3	E3
1B1	1A	2A	53	6D	CA	D9	E8	F5
1BD	1C	68	91	A5	BF	E4	ED	F6
1C3	25	52	71	7F	9B	AA	B9	F1
1CF	21	47	5D	73	96	A3	B1	CC
1D7	1B	69	8E	92	9C	C8	D5	EF
1DD	1D	3F	46	6D	8C	96	A6	B5
1E7	06	14	2E	36	9A	A1	C6	F7
1F3	21	2B	6F	7C	81	B4	D7	FB
1F5	21	32	3F	71	9E	A7	AB	CF
1F9	07	15	25	75	97	9B	A6	E8

Figure 2.2: All the Dual Cipher. (From [32])

instantaneously. Thus, the data storage called register is introduced. Usually it only has limited numbers, say four or eight and only a small set of data to be processed will be stored in it.

It is apparent that four or eight registers absolutely impossible to satisfy the throughput of PCs or servers. This is where the computer's main memory comes in. Generally a large set of data will be stored in main memory and only a small portion of them will be sent to registers and processed by ALU (as showed in Figure 2.3). However, the main memory, to some extent, is a completely separated component of computer system, which means it is much farther away from the ALU. In other words, if there were no register and the ALU would have to read data from main memory directly, the computers would run very slowly [30].

Although the introduction of registers and main memory at least partly improves the performance of a computer, the speed gap between the main memory and registers is still a bottleneck, which costs a significant amount of time and will kill most of the

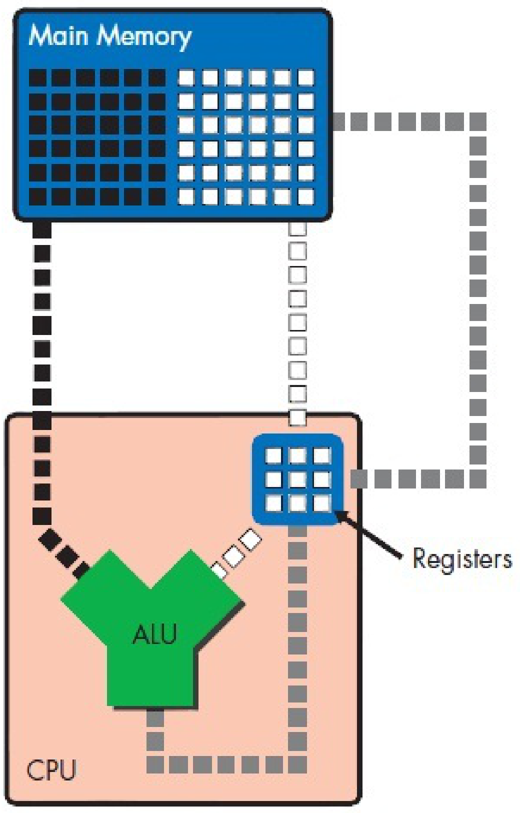


Figure 2.3: Registers and Main Memory. (From [30])

performance. Thus, the cache memory is introduced to fill the speed gap. A cache is a memory area which is placed between main memory and processor [22], but it is much faster and has smaller memory space compared with main memory (as showed in Figure 2.4). So usually it could be regarded as a buffer. That is to say, some of data which may be accessed frequently would be stored in cache. As a result, once these data are required, CPU would access the cache to get them directly instead of accessing the main memory, which will significantly improve the processing efficiency.

Note if the required data are in the cache, this is called cache-hit and processor could get the data quickly. But if the required data are not in the cache, this is called cached-missed and CPU will have to get the data from the main memory. So caches behave in a data-dependent manner and this result in execution time to vary. Thus, some attackers will make use of this kind of properties of cache to attack encryption devices, which called cache attack.

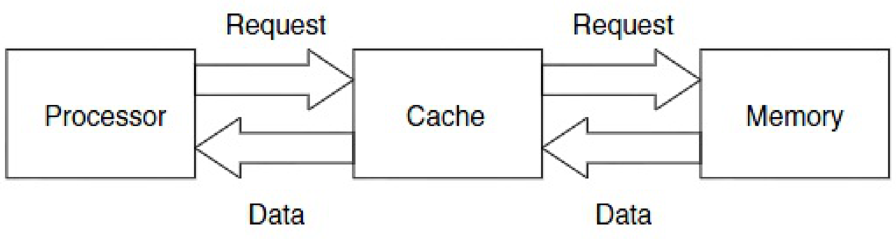


Figure 2.4: Cache Operation. (From [22])

2.3.2 Scalar Processor and Vector Processor

Scalar processor is the simplest computer processor. It processes each datum with each instruction, therefore it is also classified as a Single Instruction stream, Single Data Stream (SISD) processor. One the other hand, vector processor, also called Single Instruction stream, Multiple Data Stream (SIMD) processor [30], could handle a multiple streams of data with single instruction. The figure below (Figure 2.5) is the SISD and SIMD,

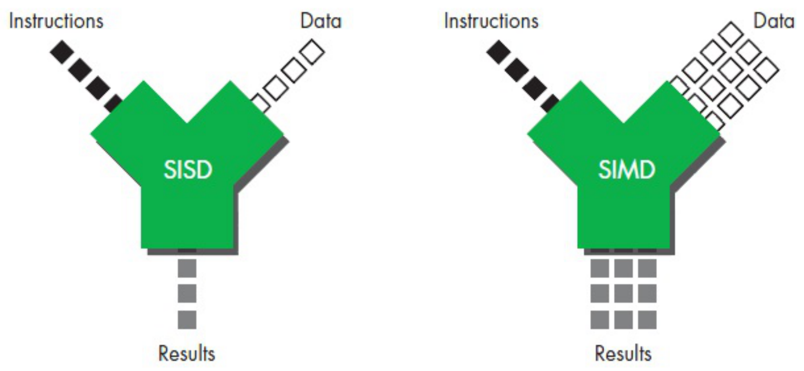


Figure 2.5: SISD and SIMD. (From [30])

Vectors, actually, are nothing more than a group of data or scalar and all of them are the same type. For a simplistic example, if a scalar processor processes a variable in the program, then a vector processor does an array of this kind of variable in parallel. Since vector processors usually handle with a group of data once a time, the vector operations are different from those in scalar processors. Here are some basic operations in vector processors [30],

- intra-element arithmetic
- intra-element non-arithmetic
- inter-element arithmetic

- inter-element non-arithmetic

The intra-element arithmetic and intra-element non-arithmetic are the vector operations that happen between vectors and vectors (as showed in Figure 2.6) and the only difference between them is that the first one dealing with arithmetic operations say addition, multiplication etc, while the latter is involved in logical operations, such as AND, NOT and XOR. As for inter-element arithmetic and inter-element non-

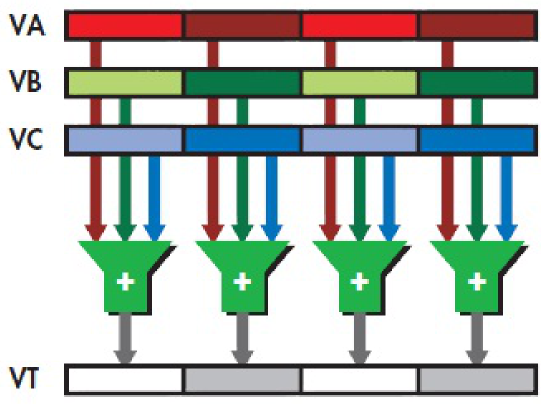


Figure 2.6: Intra-element Arithmetic and Intra-element Non-arithmetic Operation. (From [30])

arithmetic, they are operations which happen between the elements in a single vector (as displayed in figure 2.7 and 2.8) and the inter-element non-arithmetic one are operations like vector permute, which rearrange the order of the elements in an individual vector.

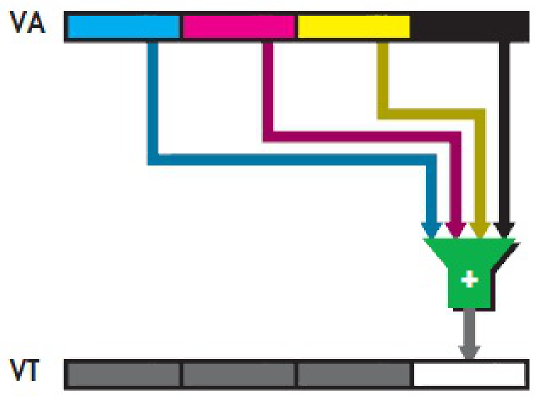


Figure 2.7: Inter-element Arithmetic Operation. (From [30])

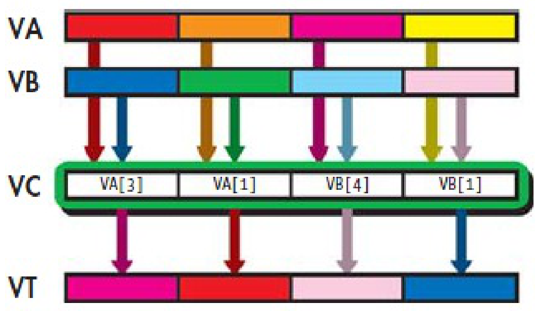


Figure 2.8: Inter-element Non-arithmetic Operation. (From [30])

2.4 Mainstream Techniques for Implementing Rijndael

As discussed above, due to the elegant design structure, Rijndael can be implemented efficiently in terms of not only software optimization but also hardware optimization [3, 4, 5, 9, 12, 15, 16, 24, 26, 27, 31] in different platforms. Therefore, whole spectrum cryptographic devices, from high-end machines, such as dedicated cryptographic servers, to some widely applied smart cards, such as SIM cards, take it as encryption standard [20]. In this section, some mainstream software implementation techniques of Rijndael will be discussed.

2.4.1 Optimization for Standard Processors

2.4.1.1 Implementation on 8-Bit Platforms

From the description of Rijndael algorithm given above, it is obvious that this algorithm is well suited 8-bit processors such as smart cards inherently, for all the operations of Rijndael algorithm are conducted on individual byte. Thus, on 8-bit platforms, no extra space in the register is wasted while processing 1 byte with per instruction. Since all the operations of Rijndael involved in the algorithm on 8-bit platform are very simple and the structure of the algorithm is straightforward, here we are not going to talk more details about the implementation on 8-bit platform.

2.4.1.2 Implementation on 32-Bit Platforms

Because of the fact that all the operations of Rijndael algorithm are conducted on individual byte, it is not particularly efficient on unconstrained platforms. Thus, while designing Rijndael, Daemen and Rijmen meanwhile put forward another implementation which will work much more efficiently on 32-bit or greater platforms, which are the main platforms in today's PC or cryptographic servers or workstations.

The idea of this method is quite simple. We could combine the `SubBytes`, `ShiftRows` and `MixColumns` into one step. In other words, we merge the the intermediate results of them into four tables, each of which is composed of 256 elements and each element is 32 bits. Since each element in these tables is 32 bits which are equal to the size of the register, every time 32 bits data could be processed with per instruction simultaneously. Specifically, , the equations of `SubBytes, ShiftRows and MixCloumns` become [9],

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \odot \begin{pmatrix} S-box(a_{0,j+c_0}) \\ S-box(a_{1,j+c_1}) \\ S-box(a_{2,j+c_2}) \\ S-box(a_{3,j+c_3}) \end{pmatrix} \tag{2.18}$$

where S-Box is the result of affine transformation (2.1) which is stored in a 256 bytes lookup table accordingly and c_i denotes the bytes shifted from right to left. Furthermore, this equation could be regarded as this,

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} Sbox(a_0, j + c_0) \oplus \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} Sbox(a_{1,j+c_1}) \oplus \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} Sbox(a_{2,j+c_2}) \oplus \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} Sbox(a_{3,j+c_3}) \tag{2.19}$$

And then it could be translated into,

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} = T_0(a_{0,j+c_0}) \oplus T_1(a_{1,j+c_1}) \oplus T_2(a_{2,j+c_2}) \oplus T_3(a_{3,j+c_3}) \tag{2.20}$$

Thus, 4 new lookup tables (each of $T_i(a_{i,j+c_i})$ presenting as a new lookup table) are introduced to take the place of S-Box. The advantage of this method is that we could avoid the affine transformation in `SubBytes,` rotations in `ShiftRows` and multiplication in GF(2^8) in `MixColumns` since all of them are replaced by lookup table. Now the only question is how to obtain the four lookup tables. This will not be a problem since they are composed of the elements in S-Box which is computed in advance as well.

Although this method saves quite a few operation steps during the implementation, we cannot ignore the fact it requires much more memory. Each table requires 1 KB and there are 5 tables in totally. So 5 KB memory is needed while the implementation, which is unacceptable for some embedded platforms. What's more, it is more likely to suffer from cache attack. Thus, in [4], Bertoni et al. gave an approach to implement Rijndael with high-performance and low-footprint. The core idea of this approach is to compute the 4 byte data in parallel. In order to process the 32 bits data once, we need to transpose the whole state matrix. Of course, some transformations of Rijndael would need to be modified accordingly.

`SubBytes` In this transformation, no modification is required since the operation is independent of the positions of bytes.

`ShiftRows` In this transformation, the only modification is that we no longer shift rows of state matrix; instead, the columns are shifted because of the transposition.

`AddRoundKey` As for this transformation, the operation is invariant because it is only involved in bitwise XOR operations which is also independent of the positions of bytes.

`MixColumns` After the transposition, the whole operations in this transformation will be completely reconstructed and the new transformation is considerably sped-up.

Before the optimization, we could compute each of column according the Equation (2.7). After further transformation, it could become,

$$\begin{pmatrix} b'_{0,j} \\ b'_{1,j} \\ b'_{2,j} \\ b'_{3,j} \end{pmatrix} = 02 \begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \oplus 03 \begin{pmatrix} b_{3,j} \\ b_{0,j} \\ b_{1,j} \\ b_{2,j} \end{pmatrix} \oplus \begin{pmatrix} b_{2,j} \\ b_{3,j} \\ b_{0,j} \\ b_{1,j} \end{pmatrix} \oplus \begin{pmatrix} b_{1,j} \\ b_{2,j} \\ b_{3,j} \\ b_{0,j} \end{pmatrix} \tag{2.21}$$

The equation above could also be regarded as the following,

$$y_0 = 02x_0 \oplus 03x_1 \oplus x_2 \oplus x_3 \tag{2.22}$$

where both x and y are 32-bits words. After the state matrix is transposed, the 4 bytes word (x_i) could be processed in parallel. Here are the rest of the equations,

$$y_1 = x_0 \oplus 02x_1 \oplus 03x_2 \oplus x_3 \tag{2.23}$$

$$y_2 = x_0 \oplus x_1 \oplus 02x_2 \oplus 03x_3 \tag{2.24}$$

$$y_3 = 03x_0 \oplus x_1 \oplus x_2 \oplus 02x_3 \tag{2.25}$$

These four equations above could be calculated by three steps [4].

$$\begin{aligned} y_0 &= x_1 \oplus x_2 \oplus x_3 \\ y_1 &= x_0 \oplus x_2 \oplus x_3 \\ y_2 &= x_0 \oplus x_1 \oplus x_3 \\ y_3 &= x_0 \oplus x_1 \oplus x_2 \\ y_i &= 02x_i; \end{aligned}$$

$$y_i = y_i + x_i + x_{((i+1) \bmod 4)}; \tag{2.26}$$

In this way, fewer XOR gates and doubling operations are involved, which means the time performance is improved.

2.4.2 Implementation with Bit-Slicing Technique

The idea of bit-slicing technique was first introduced in [5] by Biham in 1997 to speed up the DES implementation in software. As displayed in [5], using bit-slicing technique enables the implementation approximately five times faster. To be specific, suppose we have a CPU with a 64-bit register width, then in bit-slicing implementation, the 64 bits in the register would act as 64 one-bit processors handling different data encryption. In other words, we simulate the way of vector processors (SIMD processors) dealing with data so that the algorithm could be implemented in parallel. Therefore, intuitively this technique could improve the performance N times, provided that we have a CPU with a N-bit register width. Here is a figure of bit-slicing technique, As showed in

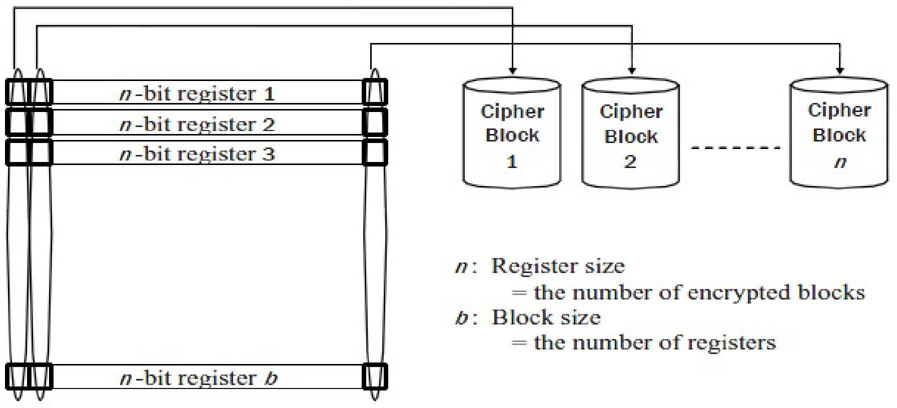


Figure 2.9: The basic concept of bit-slicing. (From [15])

Figure 2.9, all the first bits in each cipher block are operated in the register 1 with the same instruction simultaneously. And all the second bits are operated in the register 2, and so on. Obviously, the more bits a register contains, the more cipher blocks could be handled. That is to say, a bigger register could work more efficiently. Besides, from what we have discussed above, it is apparent that bit-slicing technique could be implemented regardless the encryption algorithms. So not just DES, it is also applied to Rijndael or other block cipher.

Another advantage of bit-slicing technique is to avoid looking up tables. As discussed above, most of the implementations of `SubBytes` transformation depends on table lookups before bit-slicing is introduced since compared with calculating the complicated permutation and substitution of each element, it is fairly efficient. However, as a matter of fact, table lookup is not a good technique while dealing with transformation in S-Box. First of all, it requires both additional space to store the table and extra time to look up it. And we know that the space of registers is limited, so usually the tables would be stored in cache or even main memory, which means more time is wasted in data transferring. Even worse, it is vulnerable to cache-timing attacks. Looking up table would always require different extra time depending on the elements looked up in the S-Box. Specifically, the attackers usually could guess whether some particular data are in the cache or not (cache hit or cache miss) depending on these time information and finally recover the secret keys. However, bit-slicing is independent of table lookup, which means such attacks will never happen. Hence, bit-slicing could not only improve the efficiency of implementation but also be immune to the attack based on cache-timing analysis.

On the other hand, in spite of the fact that bit-slicing technique has considerable advantages over ordinary techniques, we could not implement it directly since this implementation uses a non-standard representation. Therefore, the conversion from standard domain to bit-sliced domain is inevitable if both of input and output data are required in standard format.

Assuming that we have a 64-bit CPU and 64 block ciphers (each of them contains 128 bits) needs to be handled. Thus, if we implement them without bit-slicing technique, they would be treated like Table 2.2,

Table 2.2: The bit stored in 64-bit CPU standard format. (From [24])

b0,64	..	b0,4	b0,3	b0,2	b0,1	b0,0
b0,127	..	b0,68	b0,67	b0,66	b0,65	b0,64
b1,64	..	b1,4	b1,3	b1,2	b1,1	b1,0
b1,127	..	b1,68	b1,67	b1,66	b1,65	b1,64
b2,64	..	b2,4	b2,3	b2,2	b2,1	b2,0
b2,127	..	b2,68	b2,67	b2,66	b2,65	b2,64
b3,64	..	b3,4	b3,3	b3,2	b3,1	b3,0
b3,127	..	b3,68	b3,67	b3,66	b3,65	b3,64
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
b63,64	..	b63,4	b63,3	b63,2	b63,1	b63,0
b63,127	..	b63,68	b63,67	b63,66	b63,65	b63,64

where $b_{x,y}$ mean the yth bit in the xth block cipher.

In order to convert these data into bit-slicing domain, we need to rearrange the whole table as following Table 2.3, the first bit of each block cipher needs to be extracted and put into the first line and the second bit of each block cipher needs to be put into the second line, and so on.

To some extent, this conversion is kind of similar to the transposition of matrix. And it has been discussed in [10, 15, 24]. On the other hand, while implementing bit-slicing in closed environment, it is not necessary to convert the input and output data.

2.5 Summary

In this chapter, we have talked about each steps of Rijndael encryption, `SubBytes`, `ShiftRows`, `MixColumns`, `AddRoundKey` and round key generation. And according

Table 2.3: The bit stored in 64-bit CPU bit-sliced format. (From [24])

b63,0	..	b4,0	b3,0	b2,0	b1,0	b0,0
b63,1	..	b4,1	b3,1	b2,1	b1,1	b0,1
b63,2	..	b4,2	b3,2	b2,2	b1,2	b0,2
b63,3	..	b4,3	b3,3	b2,3	b1,3	b0,3
b63,4	..	b4,4	b3,4	b2,4	b1,4	b0,4
b63,5	..	b4,5	b3,5	b2,5	b1,5	b0,5
b63,6	..	b4,6	b3,6	b2,6	b1,6	b0,6
b63,7	..	b4,7	b3,7	b2,7	b1,7	b0,7
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
..	..	..	..	..	..	..
b63,126	..	b4,126	b3,126	b2,126	b1,126	b0,126
b63,127	..	b4,127	b3,127	b2,127	b1,127	b0,127

to the description, we compare Rijndael with other block cipher, say DES and the other candidates in the final round of AES and analysis their advantages and disadvantages according to their security properties, algorithm complexity, feasibility on various platforms and so on. After that, we introduce the concept of dual ciphers which are put forward by Barkan and Biham and illustrate some examples of Rijndael variants. If just considering the square dual ciphers and irreducible polynomials, there are totally 240 dual ciphers in Rijndael and all of them have the same security properties as the original one. Then, some state-of-the-art Rijndael implementations which will be used in the next chapter are talked so that the work we have done will not be so hard to understand.

Chapter 3

Implementation of Rijndael and its Variants

In this chapter, we are going to discuss the Rijndael and its variants from a more practical stand-point. Following the Rijndael implementation techniques illustrated in last chapter, we will introduce and use the concept of dual ciphers in terms of implementation. For each method, according to its property of implementation, certain dual ciphers with optimal design will be implemented. What's more, in this chapter, more attention will be paid to the bit-slicing technique since it is more likely to be influenced by the dual ciphers compared with other implementation techniques.

3.1 Implementation Details of Rijndael Variants

Before discussing the different kinds of implementations, we will review and discuss the dual ciphers of Rijndael. As described in last chapter, dual ciphers will only have influence on those operations that are involved in constants. That is to say, only `SubBytes`, `MixColumns` and the round constants in the key schedule will need to be investigated while we implement variants of Rijndael.

In last chapter, we introduce the relation between Rijndael and its dual cipher. It can be described as following figure 3.1, where P, C and K are plain text, cipher text and secret key respectively. T is a mapping matrix could be represented as [b^0, b^{25}, b^{50}, b^{75}, b^{100}, b^{125}, b^{150}, b^{175}] and all the generator values b have been listed in [32].

The `SubBytes` transformation now becomes

$$f(y) = (T * A * T^{-1})y + T(C) \tag{3.1}$$

where T is the mapping matrix mentioned above and A is the original affine transformation matrix and the C is the constant which is (01100011) in binary format (in Equation 2.1) in the original Rijndael. As for the constant matrix in `MixColumns`, it

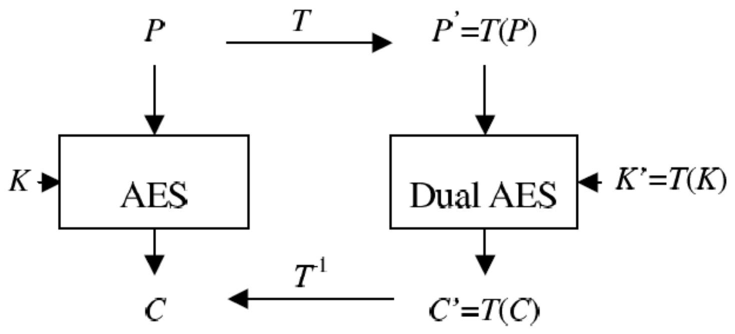


Figure 3.1: The Relation Between Rijndael and Its Dual Cipher. (From [32])

now is regarded as following matrix,

$$\begin{pmatrix} b^{25} & b & b^0 & b^0 \\ b^0 & b^{25} & b & b^0 \\ b^0 & b^0 & b^{25} & b \\ b & b^0 & b^0 & b^{25} \end{pmatrix}$$

and b is the generator above.

In this thesis we are not going to discuss the round constants used in the key schedule operation for two reasons. First of all, the round constants will not influence the performances of Rijndael on byte level implementation (non-bit-slicing technique) since each time it will always cost a XOR gate no matter what the round constants are. Besides, even for the bit-slicing technique, they still will not affect the efficiency too much compared with other two transformations (`SubBytes` and `MixColumns`). Secondly, while we measuring the performance of each method, the cost of key schedule will not be taken into account since the round keys are only dependent on the secret key and usually we will not change the secret key for each block encryption. Therefore, there are only two transformations which need to be modified we will discuss in the following sections, `SubBytes` and `MixColumns`.

3.2 On the 8-bit Platform

3.2.1 8-bit with S-Box

In this method, we will only discuss the `MixColumns` transformation since the `SubBytes` operation is done by table lookup, which means no matter what dual ciphers we choose, the efficiency of `SubBytes` will not improve or degrade.

In the original Rijndael, the `MixColumns` transformation works as,

$$\begin{pmatrix} a'_{0,j} \\ a'_{1,j} \\ a'_{2,j} \\ a'_{3,j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \odot \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} \tag{3.2}$$

The multiplication by 02 is usually denoted as xtime and is implemented by a shift operation and a conditional XOR operation. We need to check whether its most significant bit is 1 or not. If it is one, then we need to do a left shift operation followed by a XOR operation to deal with the overflow in . Otherwise, we just do the left shift operation. Here is an example, assume that we want to calculate the 02 * x, where x is {95} in hexadecimal format or {10010101} in binary format. Now that the most significant bit is one, we first shift 1 bit to left and get {00101010} in binary format. Then since all the operations are are implemented in GF(2^8), we do the XOR operation with {1B} or {00011011} which stands for the polynomial

$$P(x) = x^8 + x^4 + x^3 + x + 1 \tag{3.3}$$

Finally, we get the outcome {00110001} or {31} in hexadecimal format.

As for the multiplication by 03, we simply do a XOR operation with the outcome of multiplication by 02. Take {95} as an example again, its results is {10100100} or {A4} in hexadecimal format. Similarly, we could write all elements of GF(2^8) as a sum of powers of 02. For example, 09 * x could be written as 08 * $x \oplus x$, where 08 * x can be viewed as 02 * 02 * 02 * x. Intuitively, the bigger the constant is, the longer time it will cost since more XOR operations and shift operations are involved.

Considering the `MixColumns` transformation in dual ciphers now becomes

$$\begin{pmatrix} a'_{0,j} \\ a'_{1,j} \\ a'_{2,j} \\ a'_{3,j} \end{pmatrix} = \begin{pmatrix} b^{25} & b & b^0 & b^0 \\ b^0 & b^{25} & b & b^0 \\ b^0 & b^0 & b^{25} & b \\ b & b^0 & b^0 & b^{25} \end{pmatrix} \odot \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} \tag{3.4}$$

and it still works in GF(2^8), the multiplication could also be implemented by a repeated use of XOR a sequence of shift and XOR operations.

Since the performance of this method depends on the constants in the matrix, we have gone through all the 240 dual ciphers in [2] and found that there is no dual ciphers with fewer operations. In other words, all constants are equal to or great than 02 and 03. There is one dual cipher having the same performance as the original one, and it is {{11d}, {02}} in general representation. Its constant matrix is as follows,

$$\begin{pmatrix} 03 & 02 & 01 & 01 \\ 01 & 03 & 02 & 01 \\ 01 & 01 & 03 & 02 \\ 02 & 01 & 01 & 03 \end{pmatrix}$$

3.2.2 8-bit without S-Box

In this subsection, the `MixColumns` operation and its cost are the same as last subsection. Therefore, we are going to investigate how to implement the `SubBytes` without table lookup and its impact on the performance of dual ciphers. The implementation of `SubByte`, actually, could be regarded as two parts as in the equation (2.1). The affine transformation is operated over GF(2), which is not hard to deal with. However, as discussed in last chapter, there is no quite ideal method to compute the the multiplicative inverse. Here, we calculate a^{254} to handle the inverse operation since it is equal to a^{-1} over GF(2^8) according to Fermat's little theorem.

As for the exponentiation calculation, it is not as hard as the inverse calculation. We do it with the left-to-right binary exponentiation algorithm in [17]: Since the

Algorithm 4 Left-to-right binary exponentiation

```
Input: g over field G; a positive integer e = $e_t$... $e_1 e_0$;
Output: $g^e$;
1: $A = 1$;
2: for $i = t$ to 0 do
3:     $A = A * A$;
4:     if $e_i = 1$ then
5:         $A = A * g$;
6:     end if
7: end for
8: return $A$;
```

multiplication is easy to compute in GF(2^8)(it could be implemented by a repeated use of a sequence of shift and XOR operations), the exponentiation calculation is easy as well. Besides, as we can see that the cost of exponentiation is independent of the irreducible polynomial in the encryption algorithm.

Now let us look at the affine transformation

$$y = A \odot x \oplus c$$

we searched though the 240 dual ciphers, and found that there is no dual cipher with value c is 0, which means this XOR gate is always inevitable. As for the constant matrix A, usually we need to do the multiplication bit by bit and then add them up. Here we calculate it in byte level instead of bit level. In the following equation, all the elements in matrix and vector are either 0 or 1. According to this equation, if only a_0 and a_4 are 1, then the first column $x_{0,0}x_{0,1}x_{0,2}x_{0,3}x_{0,4}x_{0,5}x_{0,6}x_{0,7}$ and the fifth column

$x_{4,0}x_{4,1}x_{4,2}x_{4,3}x_{4,4}x_{4,5}x_{4,6}x_{4,7}$ will be added up.

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} x_{0,0} & x_{1,0} & x_{2,0} & x_{3,0} & x_{4,0} & x_{5,0} & x_{6,0} & x_{7,0} \\ x_{0,1} & x_{1,1} & x_{2,1} & x_{3,1} & x_{4,1} & x_{5,1} & x_{6,1} & x_{7,1} \\ x_{0,2} & x_{1,2} & x_{2,2} & x_{3,2} & x_{4,2} & x_{5,2} & x_{6,2} & x_{7,2} \\ x_{0,3} & x_{1,3} & x_{2,3} & x_{3,3} & x_{4,3} & x_{5,3} & x_{6,3} & x_{7,3} \\ x_{0,4} & x_{1,4} & x_{2,4} & x_{3,4} & x_{4,4} & x_{5,4} & x_{6,4} & x_{7,4} \\ x_{0,5} & x_{1,5} & x_{2,5} & x_{3,5} & x_{4,5} & x_{5,5} & x_{6,5} & x_{7,5} \\ x_{0,6} & x_{1,6} & x_{2,6} & x_{3,6} & x_{4,6} & x_{5,6} & x_{6,6} & x_{7,6} \\ x_{0,7} & x_{1,7} & x_{2,7} & x_{3,7} & x_{4,7} & x_{5,7} & x_{6,7} & x_{7,7} \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}$$

$$= \begin{pmatrix} x_{0,0}a_0 \oplus x_{1,0}a_1 \oplus x_{2,0}a_2 \oplus x_{3,0}a_3 \oplus x_{4,0}a_4 \oplus x_{5,0}a_5 \oplus x_{6,0}a_6 \oplus x_{7,0}a_7 \\ x_{0,1}a_0 \oplus x_{1,1}a_1 \oplus x_{2,1}a_2 \oplus x_{3,1}a_3 \oplus x_{4,1}a_4 \oplus x_{5,1}a_5 \oplus x_{6,1}a_6 \oplus x_{7,1}a_7 \\ x_{0,2}a_0 \oplus x_{1,2}a_1 \oplus x_{2,2}a_2 \oplus x_{3,2}a_3 \oplus x_{4,2}a_4 \oplus x_{5,2}a_5 \oplus x_{6,2}a_6 \oplus x_{7,2}a_7 \\ x_{0,3}a_0 \oplus x_{1,3}a_1 \oplus x_{2,3}a_2 \oplus x_{3,3}a_3 \oplus x_{4,3}a_4 \oplus x_{5,3}a_5 \oplus x_{6,3}a_6 \oplus x_{7,3}a_7 \\ x_{0,4}a_0 \oplus x_{1,4}a_1 \oplus x_{2,4}a_2 \oplus x_{3,4}a_3 \oplus x_{4,4}a_4 \oplus x_{5,4}a_5 \oplus x_{6,4}a_6 \oplus x_{7,4}a_7 \\ x_{0,5}a_0 \oplus x_{1,5}a_1 \oplus x_{2,5}a_2 \oplus x_{3,5}a_3 \oplus x_{4,5}a_4 \oplus x_{5,5}a_5 \oplus x_{6,5}a_6 \oplus x_{7,5}a_7 \\ x_{0,6}a_0 \oplus x_{1,6}a_1 \oplus x_{2,6}a_2 \oplus x_{3,6}a_3 \oplus x_{4,6}a_4 \oplus x_{5,6}a_5 \oplus x_{6,6}a_6 \oplus x_{7,6}a_7 \\ x_{0,7}a_0 \oplus x_{1,7}a_1 \oplus x_{2,7}a_2 \oplus x_{3,7}a_3 \oplus x_{4,7}a_4 \oplus x_{5,7}a_5 \oplus x_{6,7}a_6 \oplus x_{7,7}a_7 \end{pmatrix}$$

Therefore, the number of XOR gates is dependent on the variable $a_0a_1a_2a_3a_4a_5a_6a_7$ and the constant matrix. If there is 3 columns in the constant matrix is all 0, then at most 3 XOR gates could be reduced. However, there is no such matrix whose whole column is all 0 in all 240 dual ciphers. In other words, neither affine transformation nor multiplication inverse transformation will help improve the efficiency in terms of dual ciphers if we use this method to implement them.

3.3 On the 32-bit Platform

3.3.1 32-bit with T-table

As discussed in last chapter, this method combines different steps of the round transformation, `SubBytes, ShiftRows` and `MixColumns,` in a single set of lookup tables. On the other hand, the two operations will be modified in dual ciphers are `SubBytes` and `MixColumns.`That's to say, no matter how the constants are modified in these two transformations, there will be no performance impact since both of them finally will be implemented by table lookup whose cost actually only depends on the plain text and secret key.

3.3.2 32-bit without T-table

The implementation on 32-bit platform described above is accelerated by pre-computing part of round transformations and storing the results into the table. However, the requirement of the memory is an issue to be considered. Considering an embedded processor without much memory but with a 32-bit data-path may not be suitable for this approach, it makes sense to implement Rijndael with a mix of possible high performance and low-footprint methods.

Thus, Bertoni et.al put forward an approach without T-table on 32-bit platform in [4]. The principle of this technique is to improve the performance by packed operations so as to overcome the bottleneck of `MixColumns` transformation. To be specific, we make use of the bit length of the platform and could regard the four-byte elements as a unit and compute xtime of the packed vector in parallel. Compared with the standard implementation on 32-bit platform, it could save much more instructions. In the standard implementation of `MixColumns`, for each column, it will cost a single doubling (xtime), four XOR gates and 3 rotations. As for this method, no rotation is required. Therefore, for each block, we could save 12 rotation instructions since there are four columns for one each block.

Now that the `MixColumns` transformation of this technique is just to compute the the four-byte elements in parallel, the cost of it still depends on the constants in the matrix. And as discussed in previous section, there is no dual cipher with fewer operations to complete the `MixColumns` transformation. Hence, there is still no dual ciphers with higher performance compared with the original one. The one with the same efficiency is the dual cipher {{11d}, {02}} in general representation explained in last chapter.

3.4 Bit-slicing

The idea of bit-slicing technique appeared in 1997, Biham put forward this fast method to accelerate the DES implementation. In [5], he also mentioned that it could be applied in any cipher. Thus, just after Rijndael was announced as the winner of AES, a larger number of results [3, 12, 13, 15, 16, 20, 24] tried to apply bit-slicing to Rijndael. Most of them implement it in assembly language using SSE instructions. In this thesis, we implement Rijndael in C language so as to compare its performance with other approaches.

As analyzed before, bit-sliced ciphers use a non-standard format to represent data. Sometimes, its input and output need to be converted. In [10, 24], the authors gave an algorithm to deal with the conversion between normal domain and bit-slicing domain. However, it seems that this algorithm does not work while transposing the matrix. Hence, we assume that all the implementation is completed in a closed environment so that the data can be kept in bit-sliced representation and matrix transposition could be omitted.

3.4.1 The Original Rijndael

Bit-slicing technique is to implement the encryption algorithm on bit level rather than byte level. Therefore, certain transformations in Rijndael would be totally different from that in normal ones. It is fairly necessary to give a specific description of these transformations. Before that, it makes sense to introduce the platform we implement on since the data representation depends on the bit length of the processor. The platform is Snowy (Intel(R) Xeon(R) CPU X5460 @3.16GHz) whose instruction set is 64-bit long. So we could view the input data as that in Table 2.3. The first row stores the first bit of all the 64 blocks, the second row stores the second bit of all 64 blocks and so on.

3.4.1.1 The Bit-sliced SubBytes Transformation

The `SubBytes` transformation in bit-sliced domain also could be perceived as two parts, the linear affine transformation and the non-linear multiplicative inverse transformation. As a matter of fact, there is no difference between normal domain and bit-sliced domain in terms of the affine transformation in `SubBytes` transformation. As showed in previous chapter, the Equation (2.1) is calculated over bit level. That is to say, we could adapt this affine transformation in bit-sliced domain without any modification.

The Non-linear Multiplicative Inverse Transformation in Bit-sliced Domain Now the only problem needs to be solved is the non-linear multiplicative inverse transformation. It is true that the method we mentioned in Section 3.2 could be applied to calculate the inverse since it can be operated "bit" by "bit". However, we must notice that the cost of this method would be unacceptable. As for the rest of methods listed in Chapter 1, we found neither XGCD nor exhaustive search would be the good approach. Therefore, the only way to compute the multiplicative inverse is composite field.

The idea of composite field to compute the multiplicative inverse is first introduced by Rijmen in [25]. After that, some researchers followed this idea and came up with a few optimizing approach [6, 7, 8, 19, 31]. Here we use the method proposed by Oswald et.al in [31] to implement the multiplicative inverse. Directly computing the inverse of a seventh-degree polynomial is not easy, but the calculation of the inverse of a fourth-degree or less polynomial is relatively easy, as pointed out by Rijmen in [25]. So the idea is to decompose the bigger field GF(2^8) into the smaller but isomorphic field GF(2^4) so that the degree of modular polynomial could be decreased which could help calculate the inverse a little easier.

In this case, the elements in GF(2^8) need to be represented as those in GF(2^4). According to [31], the element a in GF(2^8) could be regarded as a linear polynomial with coefficients in GF(2^4),

$$a = a_h * x + a_l \tag{3.5}$$

where a is in GF(2^8), a_h and a_l are the elements with four bits in GF(2^4).

Similarly, this new Polynomial (3.5) requires an irreducible modular polynomial as well so that the results of its calculation will always be a two-term polynomial. Therefore, the irreducible modular polynomial is given in [31] as follows,

$$P(x) = x^2 + x + e \tag{3.6}$$

where all the coefficients are in hexadecimal format. On the other hand, the coefficients in Polynomial (3.5) are in GF(2^4) which also require the irreducible modular polynomial to do the calculation. It is given in [31] as well,

$$Q(x) = x^4 + x + 1, \tag{3.7}$$

Now that the two irreducible modular polynomials and the representation polynomial are available, we could conduct the calculations in GF(2^4) and its results are isomorphic to that in GF(2^8). From [31], we find the principle of basic calculations in in GF(2^4), such as multiplication, square and multiplication inverse, are similar to byte computation.

1. The multiplication is given by

$$q(x) = a(x) \otimes b(x) \bmod Q(x)$$

where a(x), b(x) and q(x) are in GF(2^4).

$$\begin{aligned} q_0 &= a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3, \\ q_1 &= a_1b_0 \oplus (a_0 \oplus a_3)b_1 \oplus (a_2 \oplus a_3)b_2 \oplus (a_1 \oplus a_2)b_3, \\ q_2 &= a_2b_0 \oplus a_1b_1 \oplus (a_0 \oplus a_3)b_2 \oplus (a_2 \oplus a_3)b_3, \\ q_3 &= a_3b_0 \oplus a_2b_1 \oplus a_ab_2 \oplus (a_0 \oplus a_3)b_3 \end{aligned} \tag{3.8}$$

2. The square is given by

$$q(x) = a(x)^2 mod Q(x)$$

where a(x) and q(x) are in GF(2^4).

$$\begin{aligned} q_0 &= a_0 \oplus a_2, \\ q_1 &= a_2, \\ q_2 &= a_1 \oplus a_3, \\ q_3 &= a_3 \end{aligned} \tag{3.9}$$

3. The inverse operation is given by,

$$q(x) = a(x)^{-1} \bmod Q(x)$$

where a(x) and q(x) are in GF(2^4).

$$\begin{aligned} q_0 &= a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3 \oplus a_0 \oplus a_0a_2 \oplus a_1a_2 \oplus a_0a_1a_2, \\ q_1 &= a_0a_1 \oplus a_0a_2 \oplus a_1a_2 \oplus a_3 \oplus a_1a_3 \oplus a_0a_1a_3, \\ q_2 &= a_0a_1 \oplus a_2 \oplus a_0a_2 \oplus a_3 \oplus a_0a_3 \oplus a_0a_2a_3, \\ q_3 &= a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3 \oplus a_0a_3 \oplus a_1a_3 \oplus a_2a_3 \end{aligned} \tag{3.10}$$

Note $a_i b_j$ stands for an AND operation between bit a_i and bit b_j.

Now that we have known how to conduct the basic calculation in GF(2^4), we could compute the multiplication inverse of the Polynomial (3.5). Similar to definition of inverse in GF(2^8), it is defined as follows,

$$(a_h * x + a_l) \otimes (a'_h * x + a'_l) = 1 \bmod P(x)$$

where a_h, a_l, a'_h and a'_l are in GF(2^4). Hence, the inverse can be derived [31],

$$(a_h * x + a_l)^{-1} = a'_h * x + a'_l = (a_h \otimes d) * x + (a_h \oplus a_l) \otimes d \tag{3.11}$$

where d is as follows,

$$d = ((a_h^2 \otimes e) \oplus (a_h \oplus a_l) \oplus a_l^2)^{-1} \tag{3.12}$$

where {e} is the coefficient in Polynomial (3.6).

Up to now, we have discussed the multiplication inverse calculation in GF(2^4). However, we still cannot get the multiplication inverse in GF(2^8). According to [31], there need some transition between GF(2^4) and GF(2^8). Before computing the inverse in GF(2^4), we must transform GF(2^8) to GF(2^4) since all the calculations are conducted in GF(2^4) now. The mapping function from GF(2^8) to GF(2^4) is as follows,

$$\begin{pmatrix} a_{l0} \\ a_{l1} \\ a_{l2} \\ a_{l3} \\ a_{h0} \\ a_{h1} \\ a_{h2} \\ a_{h3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \tag{3.13}$$

where $a_{l0}...a_{l3}$ and $a_{h0}...a_{h3}$ are in GF(2^4), $a_0...a_7$ is in GF(2^8).

After completing the inverse calculation in GF(2^4), we need the inverse mapping function so as to go back to GF(2^8) since all the rest of Rijndael transformations are operated in GF(2^8). The mapping matrix, as a matter of fact, is the inverse matrix of matrix in Equation (3.13). It is showed as following Equation (3.14)

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \odot \begin{pmatrix} a_{l0} \\ a_{l1} \\ a_{l2} \\ a_{l3} \\ a_{h0} \\ a_{h1} \\ a_{h2} \\ a_{h3} \end{pmatrix} \tag{3.14}$$

where $a_{l0}...a_{l3}$ and $a_{h0}...a_{h3}$ are in GF(2^4), $a_0...a_7$ is in GF(2^8).

To sum up, the whole operations in bit-sliced `SubByte` transformation in the encryption algorithm could be described with four steps as showed in Figure 3.2 below from [31],

- mapping the element in GF(2^8) to elements in GF(2^4).
- calculate the multiplication inverse in GF(2^4).
- mapping the elements in GF(2^4) back to element GF(2^8).
- conduct the affine transformation with the inverse value obtained.

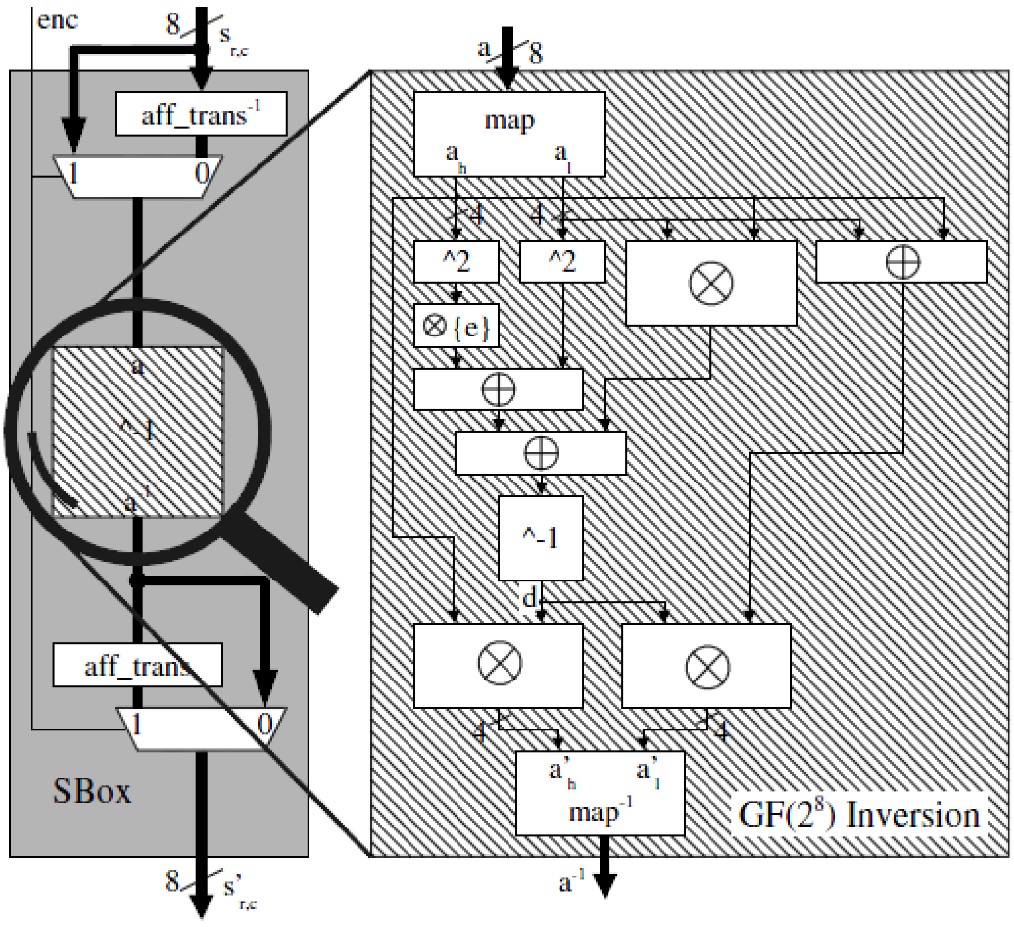


Figure 3.2: Operations in Bit-sliced SubBytes Transformation. (From [31])

3.4.1.2 The Bit-sliced ShiftRows Transformation

When in bit-sliced domain, the `ShiftRows` transformation just needs some slight modifications compared with the `SubBytes` transformation. Specifically, now we just need to shift the "word" instead of byte. The word in state are as following Table 3.1,

Table 3.1: The words in state before `ShiftRows` transformation.

b0..b7	b32..b39	b64..b71	b96..b103
b8..b15	b40..b47	b72..b79	b104..b111
b16..b23	b48..b55	b80..b87	b112..b119
b24..b31	b56..b63	b88..b95	b120..b127

Thus, according to the principle of the `ShiftRows` transformation, it will look like as following Table 3.2 after the operation,

Table 3.2: The words in state after `ShiftRows` transformation.

b0..b7	b32..b39	b64..b71	b96..b103
b40..b47	b72..b79	b104..b111	b8..b15
b80..b87	b112..b119	b16..b23	b48..b55
b120..b127	b24..b31	b56..b63	b88..b95

Note: The "word" here stands for a sequence of bits whose length are 64-bit.

3.4.1.3 The Bit-sliced MixColumns Transformation

The principle of the multiplication with constants in the `MixColumns` transformation in bit-sliced domain is the same as that in normal implementation. However, while implementing this transformation on bit level, we need more operations. In [13], the author gave the results of the multiplication by 02 and 03 as follow Table 3.3,

Table 3.3: Multiplication by 02 and 03 on bit level. (From [13])

bit	x	02 * x	03 * x
0	x0	x7	x0 + x7
1	x1	x0 + x7	x0 + x1 + x7
2	x2	x1	x1 + x2
3	x3	x2 + x7	x2 + x3 + x7
4	x4	x3 + x7	x3 + x4 + x7
5	x5	x4	x4 + x5
6	x6	x5	x5 + x6
7	x7	x6	x6 + x7

Note: this table is not entirely the same as that in [13] since we have corrected some mistakes in that table in the paper.

The principle of this multiplication is the same. The multiplication by 02 is involved in shift and XOR operations. In normal implementation, supposed that we multiply $b_7b_6b_5b_4b_3b_2b_1b_0$ with 02, if b_7, the most significant bit is 1, then the shift operation would be followed by a XOR operation. Otherwise, no XOR operation follows. Then we could notice that the results in above table are always applied to this principle. If b_7 is 0, then we only need to conduct the shift operation and the results is $b_6b_5b_4b_3b_2b_1b_00$ which is equal to $b_6b_5b_4(b_3 + b_7)(b_2 + b_7)b_1(b_0 + b_7)b_7$. On the other hand, if b_7 is 1, then we need to do a XOR operation with $0x1b$ after the shift operation and the result is $b_6b_5b_4(b_3 + 1)(b_2 + 1)b_1(b_0 + 1)1$ which is still equal to $b_6b_5b_4(b_3 + b_7)(b_2 + b_7)b_1(b_0 + b_7)b_7$.

Furthermore, according to the results in Table 3.1, we could calculate each byte in the Equation (3.15) in bit level like the Equations (3.16).

$$\begin{pmatrix} b'_{0,0} & b'_{0,1} & b'_{0,2} & b'_{0,3} \\ b'_{1,0} & b'_{1,1} & b'_{1,2} & b'_{1,3} \\ b'_{2,0} & b'_{2,1} & b'_{2,2} & b'_{2,3} \\ b'_{3,0} & b'_{3,1} & b'_{3,2} & b'_{3,3} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \odot \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} \tag{3.15}$$

$$\begin{aligned}
b'_{i,j}[0] &= b_{(i) \bmod 4,j}[7] \oplus b_{(i+1) \bmod 4,j}[0] \oplus b_{(i+1) \bmod 4,j}[7] \oplus b_{(i+2) mod 4,j}[0] \oplus b_{(i+3) \bmod 4,j}[0] \\
b'_{i,j}[1] &= b_{(i) \bmod 4,j}[0] \oplus b_{(i) \bmod 4,j}[7] \oplus b_{(i+1) \bmod 4,j}[0] \oplus b_{(i+1) \bmod 4,j}[1] \oplus b_{(i+1) \bmod 4,j}[7] \\
&\quad \oplus b_{(i+2) \bmod 4,j}[1] \oplus b_{(i+3) \bmod 4,j}[1] \\
b'_{i,j}[2] &= b_{(i) \bmod 4,j}[1] \oplus b_{(i+1) \bmod 4,j}[1] \oplus b_{(i+1) \bmod 4,j}[2] \oplus b_{(i+2) \bmod 4,j}[2] \oplus b_{(i+3) \bmod 4,j}[2] \\
b'_{i,j}[3] &= b_{(i) \bmod 4,j}[2] \oplus b_{(i) \bmod 4,j}[7] \oplus b_{(i+1) \bmod 4,j}[2] \oplus b_{(i+1) \bmod 4,j}[3] \oplus b_{(i+1) \bmod 4,j}[7] \\
&\quad \oplus b_{(i+2) \bmod 4,j}[3] \oplus b_{(i+3) \bmod 4,j}[3] \\
b'_{i,j}[4] &= b_{(i) \bmod 4,j}[3] \oplus b_{(i) \bmod 4,j}[7] \oplus b_{(i+1) \bmod 4,j}[3] \oplus b_{(i+1) \bmod 4,j}[4] \oplus b_{(i+1) \bmod 4,j}[7] \\
&\quad \oplus b_{(i+2) \bmod 4,j}[4] \oplus b_{(i+3) \bmod 4,j}[4] \\
b'_{i,j}[5] &= b_{(i) \bmod 4,j}[4] \oplus b_{(i+1) \bmod 4,j}[4] \oplus b_{(i+1) \bmod 4,j}[5] \oplus b_{(i+2) \bmod 4,j}[5] \oplus b_{(i+3) \bmod 4,j}[5] \\
b'_{i,j}[6] &= b_{(i) \bmod 4,j}[5] \oplus b_{(i+1) \bmod 4,j}[5] \oplus b_{(i+1) \bmod 4,j}[6] \oplus b_{(i+2) \bmod 4,j}[6] \oplus b_{(i+3) \bmod 4,j}[6] \\
b'_{i,j}[7] &= b_{(i) \bmod 4,j}[6] \oplus b_{(i+1) \bmod 4,j}[6] \oplus b_{(i+1) \bmod 4,j}[7] \oplus b_{(i+2) \bmod 4,j}[7] \oplus b_{(i+3) \bmod 4,j}[7]
\end{aligned} \tag{3.16}$$

where $b_{i,j}[x]$ stands for the xth bit in byte $b_{i,j}$.

3.4.1.4 The Bit-sliced AddRoundKey Transformation

This transformation is almost the same as that in the normal ones. It is just simply implemented by XOR the bit-sliced round keys with the bit-sliced state ″word″ by ″word″. Here ″word″ stands for a sequence of bits whose length are 64-bit.

3.4.2 The Dual Cipher

Up to now, we have analyzed the all the original Rijndael transformation in bit level. Due to the fact that in Rijndael dual ciphers, only `SubBytes` and `MixColumns` transformations will be modified, in this subsection, just these two transformations will be discussed.

Before discussing the bit-sliced dual cipher, we would like to introduce the concept of the Hamming weight so that further analysis would be easy to understand. (In [1], the authors claimed that the Hamming weight had implications on implementation efficiency.) According to the definition of it, the Hamming weight of a string is the number of symbols that are different from the zero-symbol of the alphabet used. In this thesis, the Hamming weight stands for the number of 1 in the 0-1 matrix and vector. Now let us look at the following equation,

$$\begin{aligned}
\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} &= \begin{pmatrix}
x_{0,0} & x_{1,0} & x_{2,0} & x_{3,0} & x_{4,0} & x_{5,0} & x_{6,0} & x_{7,0} \\
x_{0,1} & x_{1,1} & x_{2,1} & x_{3,1} & x_{4,1} & x_{5,1} & x_{6,1} & x_{7,1} \\
x_{0,2} & x_{1,2} & x_{2,2} & x_{3,2} & x_{4,2} & x_{5,2} & x_{6,2} & x_{7,2} \\
x_{0,3} & x_{1,3} & x_{2,3} & x_{3,3} & x_{4,3} & x_{5,3} & x_{6,3} & x_{7,3} \\
x_{0,4} & x_{1,4} & x_{2,4} & x_{3,4} & x_{4,4} & x_{5,4} & x_{6,4} & x_{7,4} \\
x_{0,5} & x_{1,5} & x_{2,5} & x_{3,5} & x_{4,5} & x_{5,5} & x_{6,5} & x_{7,5} \\
x_{0,6} & x_{1,6} & x_{2,6} & x_{3,6} & x_{4,6} & x_{5,6} & x_{6,6} & x_{7,6} \\
x_{0,7} & x_{1,7} & x_{2,7} & x_{3,7} & x_{4,7} & x_{5,7} & x_{6,7} & x_{7,7}
\end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} \\
&= \begin{pmatrix}
x_{0,0}a_0 \oplus x_{1,0}a_1 \oplus x_{2,0}a_2 \oplus x_{3,0}a_3 \oplus x_{4,0}a_4 \oplus x_{5,0}a_5 \oplus x_{6,0}a_6 \oplus x_{7,0}a_7 \oplus c_0 \\
x_{0,1}a_0 \oplus x_{1,1}a_1 \oplus x_{2,1}a_2 \oplus x_{3,1}a_3 \oplus x_{4,1}a_4 \oplus x_{5,1}a_5 \oplus x_{6,1}a_6 \oplus x_{7,1}a_7 \oplus c_1 \\
x_{0,2}a_0 \oplus x_{1,2}a_1 \oplus x_{2,2}a_2 \oplus x_{3,2}a_3 \oplus x_{4,2}a_4 \oplus x_{5,2}a_5 \oplus x_{6,2}a_6 \oplus x_{7,2}a_7 \oplus c_2 \\
x_{0,3}a_0 \oplus x_{1,3}a_1 \oplus x_{2,3}a_2 \oplus x_{3,3}a_3 \oplus x_{4,3}a_4 \oplus x_{5,3}a_5 \oplus x_{6,3}a_6 \oplus x_{7,3}a_7 \oplus c_3 \\
x_{0,4}a_0 \oplus x_{1,4}a_1 \oplus x_{2,4}a_2 \oplus x_{3,4}a_3 \oplus x_{4,4}a_4 \oplus x_{5,4}a_5 \oplus x_{6,4}a_6 \oplus x_{7,4}a_7 \oplus c_4 \\
x_{0,5}a_0 \oplus x_{1,5}a_1 \oplus x_{2,5}a_2 \oplus x_{3,5}a_3 \oplus x_{4,5}a_4 \oplus x_{5,5}a_5 \oplus x_{6,5}a_6 \oplus x_{7,5}a_7 \oplus c_5 \\
x_{0,6}a_0 \oplus x_{1,6}a_1 \oplus x_{2,6}a_2 \oplus x_{3,6}a_3 \oplus x_{4,6}a_4 \oplus x_{5,6}a_5 \oplus x_{6,6}a_6 \oplus x_{7,6}a_7 \oplus c_6 \\
x_{0,7}a_0 \oplus x_{1,7}a_1 \oplus x_{2,7}a_2 \oplus x_{3,7}a_3 \oplus x_{4,7}a_4 \oplus x_{5,7}a_5 \oplus x_{6,7}a_6 \oplus x_{7,7}a_7 \oplus c_7
\end{pmatrix}
\end{aligned} \tag{3.17}$$

this is a basic matrix affine transformation and all the elements in the matrix and vector are either 0 or 1.

We notice that the fewer the number of 1 in the multiplier matrix and vector is, the fewer XOR gates are needed to compute the final result (matrix). In other words, the Hamming weight of the multiplier matrix and the vector determines how long it will cost to get the final result. Therefore, intuitively if the matrix and vector in the affine transformation in Equation (2.1) have lower Hamming weight, then the `SubByte` transformation in bit-sliced format will work more efficiently. We searched through [2] and found that the dual cipher {{11d}, {02}} has the lowest Hamming weight which is 20. (The Hamming weight of the matrix is 17 and the Hamming weight of vector is 3. Below is its affine transformation.)

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0 \\ 1&1&0&0&1&0&0&0 \\ 0&1&1&0&0&1&0&0 \\ 0&0&1&1&0&0&1&0 \\ 0&0&0&1&1&0&0&1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Note, in dual cipher, the affine transformation is usually involved in three steps such as Equation (3.1). First, we need to multiply the affine matrix A with the mapping matrix T. After that, the result needs to be multiplied with the inverse matrix T^{-1}. Finally, we need to do the XOR operation with the vector. Here we regarded the first two steps (T * A * T^{-1}) as the affine transformation.

3.4.2.1 The MixColumns Transformation

In previous section, we have showed that the multiplication by a constant in GF(2^8) could be represented by a repeated XOR and shift operation and described how to conduct the multiplications by a constant in bit-sliced domain. For example, if the irreducible polynomial is $x^8 + x^4 + x^3 + x + 1$ and the constant is $a_7a_6a_5a_4a_3a_2a_1a_0$, after multiplying with 02, the result will be $b_6b_5b_4(b_3 + b_7)(b_2 + b_7)b_1(b_0 + b_7)b_7$.

We notice that there are four coefficients in the irreducible polynomial are non-zero, and in order to get the result $b_6b_5b_4(b_3 + b_7)(b_2 + b_7)b_1(b_0 + b_7)b_7$, totally we need to conduct four XOR operations. That's to say, when the number of shift operation is fixed, the more the non-zero coefficients in the irreducible polynomial are, the more instructions are required. Besides, when the number of non-zero coefficients in the irreducible polynomial is fixed, the bigger the constant we multiply is (this means more XOR operations are required), the more time it will cost. Therefore, to find the dual cipher with most efficient performance in `MixColumns` transformation is to find the one with the fewest non-zero coefficients in the irreducible polynomial and smallest constants in the matrix (3.4).

After searching the irreducible polynomial whose non-zero coefficients is the fewest, we found that {11b}, {1b1}, {171}, {11d}, {1a9}, {11b}, {12d}, {169}, {139}, {1a3}, {18b}, {165}, {11b}, {14d}, {163}, {187} and {1c3} were all polynomials with four non-zero coefficients excluding the most significant bit. As for the smallest constants in the matrix (3.4), we found 02, 03 is the smallest constant pair. Their corresponding dual ciphers could be represented as {{11d}, {02}} and {{11b}, {03}}, the latter is the original Rijndael. Luckily, both of them are ones with the fewest non-zero coefficients in the irreducible polynomial.

However, one thing we cannot ignore is that under some circumstance, multiplied with a bigger constant will result in fewer XOR gates for some of XOR gates are eliminated. Therefore, what we found is just relatively more efficient ciphers in terms of `MixColumns` transformation.

3.4.2.2 The SubBytes Transformation

Up to now, it seems that we could come up with a conclusion about a ”design for performance” approach since now we know how to conduct each transformation in bit-sliced domain and found the dual cipher with relatively better performance in affine transformation in `SubBytes` and in `MixColumns` transformation.

However, there are still two issues we have not discussed: why we use Polynomials (3.7) and (3.6) as the modular polynomial and how we get the mapping matrix and its inverse matrix in Equations (3.13) and (3.14).

To the first issue, actually, the modular polynomials are not limited to (3.7) and (3.6) and any irreducible polynomials with fourth-degree and its corresponding second-degree polynomials could take the place of Polynomial (3.7) and (3.6) respectively [14]. In GF(2^4), we know that there are merely three polynomials with fourth-degree satisfying the standard (irreducible polynomial). They are $y^4 + y + 1$, $y^4 + y^3 + 1$ and $y^4 + y^3 + y^2 + y + 1$. So for Polynomial (3.7), there will be three options. Once the polynomials with fourth-degree are decided, we could choose the coefficient in the polynomial P(z) (3.6). For the first polynomial, the coefficient could be w^7, w^{14}, w^{13} and w^{11}. For the second one and third one, it could be w, w^2, w^4 and w^8 [14]. So, in total we have 12 options to choose the polynomials while conducting the inverse calculation.

1. Q(y) = $y^4 + y + 1$ and its corresponding polynomial with second-degree,

$$P(z) = z^2 + z + 9 \tag{3.18}$$

$$P(z) = z^2 + z + b \tag{3.19}$$

$$P(z) = z^2 + z + d \tag{3.20}$$

$$P(z) = z^2 + z + e \tag{3.21}$$

2. Q(y) = $y^4 + y^3 + 1$ and its corresponding polynomial with second-degree,

$$P(z) = z^2 + z + 2 \tag{3.22}$$

$$P(z) = z^2 + z + 4 \tag{3.23}$$

$$P(z) = z^2 + z + 9 \tag{3.24}$$

$$P(z) = z^2 + z + e \tag{3.25}$$

3. Q(y) = $y^4 + y^3 + y^2 + y + 1$ and its corresponding polynomial with second-degree,

$$P(z) = z^2 + z + 3 \tag{3.26}$$

$$P(z) = z^2 + z + 5 \tag{3.27}$$

$$P(z) = z^2 + z + 9 \tag{3.28}$$

$$P(z) = z^2 + z + e \tag{3.29}$$

As for the second issue, it is quite complicated. Here we will just give a very brief description of the algorithm about how to find the mapping matrix and much more details about it has been analyzed in [14, 21, 26]. The algorithm could be demonstrate in subsequence,

- Compute a root a such that P(a) = 0.
- Find t such that result of R(a^t)(mod Q(y), P(z)) is zero, where R(x) is the irreducible polynomial with eighth-degree, such as Polynomial (2.2). Once we got the value t, the its conjugate roots could be a^{2t}, a^{4t}, a^{8t}, a^{16t}, a^{32t}, a^{64t}, a^{128t}.
- Set each the roots, such as a^t, we found as b, then the mapping matrix could be represented as M = [b^0, b^1, b^2, b^3, b^4, b^5, b^6, b^7]. The inverse mapping matrix M′ satisfying M * M′ = 1.

Here is an example, assume R(x) = $x^8 + x^4 + x^3 + x + 1$, Q(y) = $y^4 + y + 1$ and P(z) = $z^2 + z + \lambda$. In the first step, we find the root (10) in hexadecimal format. In the second step, t is equal to 5, thus all the roots would be a^5, a^{10}, a^{20}, a^{40}, a^{80}, a^{160}, a^{65} and a^{130}. We take a^5 as an instance in the third step. Since the root is a^5, then the mapping matrix could be viewed as M = [1, a^5, a^{10}, a^{15}, a^{20}, a^{25}, a^{30}, a^{35}] and we calculate the value of each element. Consequently, the mapping matrix is [1, 59, 2c, 23, 4d, 95, 47, d0] in hexadecimal format. (All the elements need to be perceived as a 0-1 matrix while conducting the mapping transformation.)

According to the above analysis, there are three different Q(y) to choose and for each Q(y), there are four options for us to choose P(z). Totally, while computing the multiplication inverse in GF(2^8), we have 12 polynomials to choose from Equation (3.18) to Equation (3.29). What's more, for each group of Q(y), P(z) and R(x), we could generate 8 mapping matrix due to the conjugate roots. Therefore, totally there are 3 * 4 * 8 =96 mapping matrix which could be chosen while computing the multiplication inverse. If we take 8 square dual ciphers and 30 irreducible polynomials into account, there will be 2880 * 8 * 30 = 23040 different `SubBytes` transformations in all. In other words, we are still far way from the conclusion about a "design for performance" approach.

Before conducting the further analysis about the dual cipher with better performance, we would like to discuss and sum up the implication of each variable on the performance so that it would help us find the dual cipher easier. First of all, according to analysis in last chapter, it seems that the square dual cipher would only affect the mapping matrix T and its corresponding inverse matrix T^{-1} in Equation (3.1). Similarly, the different irreducible polynomials R(x) will generate different the mapping matrices as well. What's more, in terms of above analysis, they will also affect the mapping matrix for transformation from GF(2^8) to GF(2^4) and its corresponding inverse matrix. As for the polynomials Q(y) and P(z), the choice of Q(y) will influence the choice of P(z). Besides, the mapping matrix for transformation from GF(2^8) to GF(2^4) will be also influenced by both of them. Finally, considering the Equations (3.11) and (3.12), we believe the polynomial P(z) will have influence on the inverse operation.(The value e could vary depending on the choice of P(z).) In the subsequence Table 3.4, we summarize above analysis.

Table 3.4: The Influence of Each Variable on SubBytes Transformation.

	mapping matrix 1	mapping matrix 2	inverse operation
square dual cipher	Yes	No	No
R(x)	Yes	Yes	No
Q(y)	No	Yes	Yes
P(z)	No	Yes	Yes

where mapping matrix 1 denotes the matrix in affine transformation and mapping matrix 2 denotes the matrix in inverse transformation.

Now that we have already 240 dual ciphers, we now need to choose a pair of Q(y) and P(z) from the 12 options. Remembering the Equation (3.11) and Equation (3.12), the coefficient of P(z) is a parameter of the inverse transformation in GF(2^4). From Equation (3.12), we know that the multiplication with a_h^2 varies depending on the value of exponent of w.

Here we take Q(y) = $y^4 + y + 1$ as an example. The parameter of the inverse transformation could 9, b, d and e and all of them are in hexadecimal format. The operations of multiplication with these four number are as follows,

1. $r = a * 9 \bmod$ Q(y),

$$\begin{aligned} r_0 &= a_0 \oplus a_1, \\ r_1 &= a_2, \\ r_2 &= a_3, \\ r_3 &= a_0 \end{aligned} \tag{3.30}$$

2. $r = a * b \bmod \mathrm{Q}(y)$,

$$\begin{aligned} r_0 &= a_0 \oplus a_1 \oplus a_2, \\ r_1 &= a_3, \\ r_2 &= a_0, \\ r_3 &= a_0 \oplus a_2 \end{aligned} \tag{3.31}$$

3. $r = a * d \bmod \mathrm{Q}(y)$,

$$\begin{aligned} r_0 &= a_0 \oplus a_3, \\ r_1 &= a_0 \oplus a_1 \oplus a_3, \\ r_2 &= a_1 \oplus a_2, \\ r_3 &= a_2 \oplus a_3 \end{aligned} \tag{3.32}$$

4. $r = a * e \bmod \mathrm{Q}(y)$,

$$\begin{aligned} r_0 &= a_1 \oplus a_2 \oplus a_3, \\ r_1 &= a_0 \oplus a_1, \\ r_2 &= a_0 \oplus a_1 \oplus a_3, \\ r_3 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \end{aligned} \tag{3.33}$$

As showed in above equations, we could notice that if we choose $z^2 + z + 9$ as P(z) while conducting the inverse operation only one XOR gate is involved, which is the fewest XOR gate needed compared with $z^2 + z + b$, $z^2 + z + d$ and $z^2 + z + e$ which require 3, 5 and 8 respectively. Although there are some optimizations to reduce the number of XOR gates, the final winner is still $z^2 + z + 9$. Similarly, we checked the number of XOR gates in other two polynomials and found there is no pair of polynomials with fewer XOR gate. Therefore, we choose $y^4 + y + 1$ and $z^2 + z + 9$ as the potential better dual ciper.

Now we could come up with a locally optimal solution. According to the discussion above, among the 240 dual ciphers, {{11d}, {02}} is the relatively better one in terms of the affine transformation in `SubBytes` step and `MixColumns` step. (The affine transformation here denotes (T * A * T^{-1})). In addition, we choose $y^4 + y + 1$ as Q(y) and due to the number of XOR gates in multiplication, $z^2 + z + 9$ is selected as P(z). Following is comparison of the mapping matrix between original Rijndael and that of the dual cipher we choose.

{{11d}, {02}}, $y^4 + y + 1$ and P(z) = $z^2 + z + 9$ mapping matrix in inverse operation,

$$M = \begin{pmatrix} 1&1&0&1&0&0&0&0 \\ 0&1&0&0&0&1&0&0 \\ 0&1&0&1&1&0&0&1 \\ 0&1&1&0&0&1&0&0 \\ 0&0&0&0&1&0&0&1 \\ 0&1&0&1&1&0&0&0 \\ 0&0&1&0&0&1&1&0 \\ 1&0&0&0&0&1&0&0 \end{pmatrix}, M^{-1} = \begin{pmatrix} 1&0&1&0&1&0&0&0 \\ 0&1&0&0&0&0&0&1 \\ 0&1&0&1&0&0&0&0 \\ 0&1&1&0&1&0&0&1 \\ 0&0&1&0&1&1&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&1&0&1&0&0&1&1 \\ 0&0&1&0&0&1&0&0 \end{pmatrix} \tag{3.34}$$

{{11b}, {03}}, $y^4 + y + 1$ and $P(z) = z^2 + z + e$ mapping matrix in inverse operation,

$$M = \begin{pmatrix} 1&0&0&0&1&1&1&0\\ 0&1&1&0&0&0&0&0\\ 0&1&0&0&0&0&0&1\\ 0&0&1&0&1&0&0&0\\ 0&0&0&0&1&1&1&0\\ 0&1&0&0&1&0&1&1\\ 0&0&1&1&0&1&0&1\\ 0&0&0&0&0&1&0&1 \end{pmatrix}, M^{-1} = \begin{pmatrix} 1&0&0&0&1&0&0&0\\ 0&0&0&0&1&1&0&1\\ 0&1&0&0&1&1&0&1\\ 0&1&0&0&1&1&1&0\\ 0&1&0&1&1&1&0&1\\ 0&0&1&0&1&1&0&0\\ 0&1&1&1&1&0&0&1\\ 0&0&1&0&1&1&0&1 \end{pmatrix} \tag{3.35}$$

Following the comparison of affine transformation between original Rijndael and that of the dual cipher we choose.

{{11d}, {02}}, $y^4 + y + 1$ and $P(z) = z^2 + z + 9$ affine transformation in `SubBytes` transformation,

$$\begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 1&0&0&1&0&0&0&0\\ 1&1&0&0&1&0&0&0\\ 0&1&1&0&0&1&0&0\\ 0&0&1&1&0&0&1&0\\ 0&0&0&1&1&0&0&1 \end{pmatrix} \odot \begin{pmatrix} a_0\\ a_1\\ a_2\\ a_3\\ a_4\\ a_5\\ a_6\\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 0\\ 0\\ 1\\ 0\\ 0\\ 1\\ 1\\ 0 \end{pmatrix} \tag{3.36}$$

{{11b}, {03}}, $y^4 + y + 1$ and $P(z) = z^2 + z + e$ affine transformation in `SubBytes` transformation,

$$\begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} = \begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \odot \begin{pmatrix} a_0\\ a_1\\ a_2\\ a_3\\ a_4\\ a_5\\ a_6\\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 0 \end{pmatrix} \tag{3.37}$$

As showed above, we notice that the affine transformation, the inverse operations and the inverse mappings are all better than those in the original Rijndael in terms of the Hamming weight and numbers of XOR gate, which means in theory, this dual cipher should be a potential candidate with higher performance.

3.5 Summary

In this chapter, we have discussed and analyzed the implementation original Rijndael and its variant on different platforms and different techniques. On the 8-bit platform, if we apply S-Box to implement the `SubBytes` transformation, then the only part will be modified in dual cipher is the `MixColumns` transformation and we found there was no dual cipher with better performance compared with the original one. If we perform the the `SubBytes` transformation on 8-bit platform with Fermats little theorem, we still did not find a better dual cipher. On the 32-bit platform, when we implement Rijndael with the T-table lookup, no matter what dual cipher we apply, performance will always keep the same. As for the implementation without T-table on the 32-bit platform, the result seems to be the same as that on 8-bit platform with S-Box since all we need to modify is `MixColumns` transformation. Consequently, there is no dual ciphers with better performance compared with original Rijndael with normal technique.

In addition, while implementing Rijndael variants with bit-slicing technique, we found that there are totally 23040 options for us to choose. In this thesis, in terms of the analysis, we choose the dual cipher which is local optimum and in theory it seems to be more efficient than the original Rijndael.

Chapter 4

Implementation Results Analysis and Evaluation

In the last chapter, we analyzed the properties of each implementation technique. According to the analysis, the implications of dual ciphers on the implementation performance are roughly discussed in theory only. Therefore, it is now necessary to investigate whether the analysis we made holds in practice. In this chapter, we will consider the cost of all the Rijndael transformations of each implementation technique, and evaluate the results of them so as to answer the question we proposed in the Chapter 1.

Before conducing the analysis and evaluation, it is necessary to mention some details about the methods. First of all, in order to consistent with the analysis in Chapter 3, we will not take the cost of key schedule into account. Secondly, we will use the number of XOR gates to measure the time complexity since most optimizations are involved in reducing the number of XOR gates. Thirdly, as showed in Figure 3.1, to implement the dual ciphers, usually we need to convert the plain text and the secret key first before encryption algorithm conducted and after the cipher text is produced, it needs to be converted again. However, it is obvious that these operations are fairly time-consuming and will affect the performance of dual ciphers, thereby now we do not convert plain text, secret key and cipher text before and after encryption; instead, we view the dual ciphers as a totally new cipher. Finally, all the implementations are conducted and tested on Snowy (Intel(R) Xeon(R) CPU X5460 @3.16GHz) so as to make the results comparable to each other.

4.1 The Cost of Rijndael and Its Variants on 8-bit Platform

4.1.1 Implementation with S-Box

In terms of this method, neither the `SubByes` nor `ShiftRows` transformations is involved in XOR gates. For the `AddRoundKey` transformation, its operation is independent of dual cipher and always requires 16 XOR to conduct each time.

As analyzed above, the `MixColumns` will affect the implementation performance. In our implementation, the doubling or xtime calculation will always take 1 XOR gate. Considering the constant matrix in the Equation 3.2, for each byte, intuitively we will need 6 XOR gates, where one gate for doubling operation, two for tripling and three for adding them up. Here we optimize this operation and reduce one XOR gate for each byte. Usually we will conduct this transformation as Equation 4.1,

$$b' = 02 * b_0 \oplus 02 * b_1 \oplus b_1 \oplus b_2 \oplus b_3 \tag{4.1}$$

but now we operate it as Equation 4.2.

$$b' = 02 * (b_0 \oplus b_1) \oplus b_1 \oplus b_2 \oplus b_3 \tag{4.2}$$

Therefore, totally in this transformation, 80 XOR gates are required.

On the other hand, we notice that the elements 02 and 03 in the constant matrix among all the dual ciphers are the smallest one. And we implement the dual cipher {{11d}, {02}} whose coefficients are the smallest for comparison. Now in terms of the number of XOR gates, both of them require 96 in total. We tested them on Snowy and found that they are 609.8125 and 611.5625 CPU cycles per byte respectively. The results are almost in accord with the analysis we made.

4.1.2 Implementation without S-Box

This method is exactly the same as the one we analyzed above except for the `SubBytes` steps. As discussed in last chapter, this transformation could be viewed as two parts, linear affine transformation and non-linear multiplicative inverse. For the first part, the number of XOR gates in the multiplication is dependent on the result of inverse and the constant matrix. Since there is no matrix whose whole column is 0 and the result of inverse is variable, we choose the average value 4 as the number of XOR gates. For the addition, we just need one XOR gate to conduct. On the other hand, in order to make the program feasible to dual cipher, some modifications are made so that extra transformation is required before the XOR operation is conducted and totally 9 XOR gates are used for each byte. The non-linear part of the `SubBytes` steps is the exponential operation. And averagely it requires 132 gates for each byte. (There are 11 multiplications and each one need 12 XOR gates average.) Therefore, just in terms of this transformation, it requires 2256 XOR gates averagely, which is very inefficient. For the rest of operations, they are completely the same as above and we still implement the dual cipher {{11d}, {02}}. The implementation results on Snowy show 41227.3125 CPU cycles required to encrypt each byte for the original Rijndael and 41205.375 cycles for the dual cipher.

4.2 The Cost of Rijndael and Its Variants on 32-bit Platform

4.2.1 Implementation with T-table

As for this approach, it is mainly dependent upon the table lookup. The only operations we need to conduct is the `AddRoundKey` and part of `MixColumns` transformations. In the `AddRoundKey` transformation, we did one XOR operation for four bytes since they are packed as a unit now. For the `MixColumns` step, the multiplication with constant could be completed by table lookup and we just need to add up the intermediate results, which will cost 3 XOR for four bytes. Therefore, for each round, it requires 16 XOR gates totally. Here we still implement the dual cipher {{11d}, {02}} for comparison. After implementing them, we discover that this approach is quite efficient and outcomes are 43.0625 and 43.5625 CPU cycles per byte.

4.2.2 Implementation without T-table

As explained before, before encrypting the data, the state matrix usually needs transposition. However, here for simplicity, we will regard that the state matrix we are going to encrypt has already been transposed and will not consider the cost of this operation. The `SubBytes` transformation in this method is implemented by table lookup and only `MixColumns` transformation requires to be modified revised deeply. Similarly, we still packed the four bytes as a unit and conduct this transformation as the first approach. So for each unit, it will cost 5 XOR gates. In other words, in `MixColumns` transformation, only 20 XOR gates are needed. In addition, in the `AddRoundKey` transformation, four XOR gates are required. Since the bottleneck of this approach is `MixColumns` as well in terms of the dual ciphers, we still choose {{11d}, {02}} to implement. The results are 368.125 and 365.125 CPU cycles per byte.

4.3 The Cost of Rijndael and Its Variants with Bit-slicing

Unlike the methods discussed above, owning to the special property of bit-slicing, the total cost of XOR gates will be always the same on different platforms as long as the mapping between GF(2^8) and GF(2^4) maintains the same. So the longer the register of a CPU, the more data would be encrypted, thus the more efficient it will be. Therefore, after obtaining the processor cycle clock, we will always divide it by the length of register. Now let us look at the cost of each part of transformation.

4.3.1 The Cost of SubBytes in Bit-slicing

In this step, as usual we are going to divide it into two parts. In the linear affine transformation of the original Rijndael, we notice the Equation (2.1) and there will be 44 XOR gates totally for each byte, which is equal to the Hamming weight of the matrix and vector. In the non-linear multiplication inverse calculation, it is composed of two parts, mapping between GF(2^8)and GF(2^4) and inverse calculation in GF(2^4).

For example, if $Q(y) = y^4 + y + 1$ and $P(z) = z^2 + z + e$, then according to Matrices (3.35), the Hamming weight will be 53 in all. In other words, 53 XOR gates are needed for mapping between between GF(2^8)and GF(2^4). As for the inverse calculation, we notice that it consists of 2 square calculations, 3 additions, 3 multiplications, 1 constant multiplication and 1 inverse operation on GF(2^4). The addition is just to conduct XOR operation word by word and requires 4 XOR gates. For the rest of them, they have been listed in Equations (3.8), (3.9), (3.10) and (3.33). So for each byte, there will be 101 XOR gates required in total.

In our implementation, for the original Rijndael, we choose $Q(y) = y^4 + y + 1$ and $P(z) = z^2 + z + 9$ so as to compare the dual cipher. Its corresponding mapping matrices are as follows,

$$M = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}, M^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad (4.3)$$

The Hamming weight of these two matrices is 57, which is slightly larger than previous one, 53. Its inverse operations are the same as described above except that the the Equation (3.33) is replaced by Equation (3.30). Therefore, for each byte, there will be 94 XOR gates required in total.

As for the dual cipher, reviewing the Equation (3.36), for dual cipher {{11d}, {02}}, the linear affine transformation will cost 20 XOR gates. Furthermore, the matrices (3.34) showed that the mapping between GF(2^8)and GF(2^4) will take 43 XOR gates in all. The inverse calculation on GF(2^4) is still 94 XOR gates since all the operations are the same.

Furthermore, we notice that the affine transformation and mapping transformation from GF(2^4)to GF(2^8) in the `SubBytes` transformation could be merged into one operation without extra cost if we multiply these two matrices in advance, which could reduce one multiplication between matrices and save a few XOR gates. Thus, after the multiplication the affine transformation in both the original Rijndael and its dual cipher {{11d}, {02}} become,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{4.4}$$

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \odot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{4.5}$$

Now the Hamming weight of the original Rijndael is reduced from 73 to 35 in terms of these two operations. And in dual cipher {{11d}, {02}}, the number of XOR gates is decreased by 16 , from 41 to 25. Accordingly, we implemented them on Snowy and obtained the testing results that 251.94 cycles per byte for the original Rijndael and 208 cycles for dual cipher {{11d}, {02}}.

4.3.2 The Cost of MixColumn in Bit-slicing

According to the Table 3.3, we know that, in `MixColumn` transformation, for each byte, it will cost three XOR gates to conduct the doubling operation and another eight XOR gates to perform the tripling one. In addition, in order to compute one byte in this step, there will be a doubling operation, a tripling one and three addition (XOR). Thereby, totally it will require 38 XOR gates in this transformation for each byte. The operation of dual cipher {{11d}, {02}} is the same as that of the original one.

4.3.3 The Cost of ShiftRows and AddRoundKey Transformation

As analyzed in Chapter 2, these two transformation will not be affected by dual ciphers. The `ShiftRows` transformation will not be involved in XOR operation as usual. Now that the operations are performed on bit level, we will conduct the `AddRoundKey` transformation word by word and totally 128 XOR gates are needed (the length of each block are 128-bit.)

4.4 Summary and Evaluation

To sum up, in this chapter, we reviewed all the Rijndael implementation and deeply analyzed the their performance in both theoretical and practical aspect. In theory, we count the number of gates in each steps of different implementation approach. The Table 4.1 below summarize all the analysis results,

Table 4.1: The Number of XOR Gates in Each Transformation for Each Round.

Techniques and Platform	SubBytes	ShiftRows	MixColumns	AddRoundKey
8-bit with s-box	-	-	80	16
8-bit without s-box	2256	-	80	16
32-bit with t-table	-	-	12	4
32-bit without t-table	-	-	20	4
bit-slicing	2512	-	608	128

As for the dual cipher, the only difference lies in the cost of `SubBytes` transformation and it is 2240.

Before illustrating the results of our implementation, we would like to mention some consequence implemented in other papers. The performance varies from processor to processor, so even for the same technique, it still has different performances.

Table 4.2: The Bit-slicing Implementation Results in Other Papers.

Author	Performance
Konighofer [13]	347, 317, 303 cycles/block
Rebeiro et.al [24]	446, 280, 135 cycles/block
Matsui [15]	250, 418 cycles/block
Matsui et.al [16]	418, 491 cycles/block
Bernstein et.al [3]	167 cycles/block
Schwabe et.al [12]	150, 121, 112 cycles/block

The implementation results on Snowy are as following Table 4.3, and the code(rdtsc.h) used to test the performance of each technique is from http://www.mcs.anl.gov/ kazutomo/rdtsc.h

Table 4.3: The implementation Results on Snowy of Each Technique.

Techniques and Platform	Original Cipher	Dual Cipher
8-bit with s-box	609.8125 cycles/byte	611.5625 cycles/byte
8-bit without s-box	41227.3125 cycles/byte	41205.375 cycles/byte
32-bit with t-table	43.0625 cycles/byte	43.5625 cycles/byte
32-bit without t-table	368.125 cycles/byte	365.125 cycles/byte
bit-slicing	251.94 cycles/byte	208 cycles/byte

From Table 4.1, we could notice that bit-slicing will cost the most XOR gates in all the transformations. And this is quite easy to understand since it is operated on bit level. However, all these transformations are operated on 64 blocks in parallel (the length of register in CPU is 64 bit) and the all of these need to be divided 64. Thus, all of these would be very efficient. Furthermore, we notice that the processor (Intel(R) Xeon(R) X5460) supports Streaming SIMD Extensions (SSE), which means there are 8 registers whose length are 128-bit in the processor. That is to say, we could pack the 128 bits data rather than 64 bits data as a word and process them in parallel, and the performance would be improved further. Note the performance would not be doubled since there are 8 128-bit registers, which results in poor latency and throughput. In addition, 8-bit without s-box is fairly inefficient in terms of performance since it needs to conduct exponential operations. As for 32-bit with t-table, it need the fewest XOR gates, only 16 for each round, which should be the most efficient. And the results in Table 4.3 show the analysis is correct. However, what we cannot ignore is its efficiency is at the cost of storage. Almost of all the transformations are conducted by table lookup and all the tables covers 5 KB. This is applied to bit-slicing as well. Although it does not require table lookup, since it needs to process data in parallel, a great amount of data is input once, which also requires lots of space while computers processing the data.

In Table 4.3, we see that the difference between the original Rijndael and its dual cipher in the first four rows is quite small. The only exception perhaps is 8-bit without s-box, which may be affected by other programs while running. Therefore, if we implement the encryption on byte level or more, the performance cannot be benefit from the dual ciphers. Furthermore, we notice that the number of XOR gates in 8-bit with s-box quadruples that in 32-bit without t-table, however, the ratio of clock cycles between them is less than half. This may result from extra shift, AND, OR operations in 32-bit without t-table. For the last row, we see that the difference between the original Rijndael and its dual cipher is quite large, which is approximate 40 clock cycles per byte, which shows the performance could be improved by dual ciphers when we implementing them on bit level. Note we have tested all the programs for several times to get the results, however since the performance is always dependent on CPU loading and there always are other programs running on Snowy, we cannot make sure these results are very precise. All of them are the best performance we have obtained.

If we compare the implementation results in Table 4.2 with our results in Table 4.3, we will find that ours are relatively worse. On the other hand, it is pointless to compare these performance for three reasons. First of all, despite the same technique (bit-slicing), the way we process and store the data in the register is different, which results in the different data throughput. What's more, the platforms are different; the processors used to test are totally different and ours are even affected by other running programs. Last but not the least, the programming languages are different. Most of the methods mentioned in Table 4.2 are coded via assembly language. (Even for those who are coded in C language, they would be converted into assembly language before performance measurement.) And usually assembly language works more efficiently than C language does. So, it would be fair to test the dual ciphers with their methods which are coded in assembly language on the platforms suggested by them. Although limited by time and experiment condition, we are unable to perform these experiments, we are still able to say that the results in Table 4.2 would be benefit from dual ciphers due to the bit level operation.

Chapter 5

Conclusions and Further Work

In this thesis, we follow the proposals suggested in [1] that the dual cipher might be an optimization of the speed of the cipher for Rijndael and attempt to discuss whether this suggestion is true or not. According to this idea, we investigate the Rijndael and its variants deeply in both theoretical and practical way. Since the difference between the original cipher and dual ones is just the constants, we paid more attention on the transformations related to constants, such as `SubBytes` and `MixColumns` steps.

In theory, according to our analysis, on one hand, we found that with normal implementation methods (implemented on the byte level), there is no better dual ciphers in terms of the performance compared with the original one since the less Hamming weight could not benefit these implementations on the byte level and there is no smaller constants in `MixColumns` steps. On the other hand, while implementing Rijndael with bit-slicing technique, we found out that due to properties of this approach, the dual ciphers benefit from the advantage of the less Hamming weight, and the total numbers of XOR gates required in the implementation is fewer compared with the original one, which means fewer instructions are used.

In practice, we implement the original Rijndael with various techniques and test each of them performance (their processor clock cycles). After that, the dual ciphers with potential higher performance are implemented accordingly. Comparing the performance of them, we discovered that it is almost the same as what we had analyzed. The dual cipher implemented with the bit-slicing technique consumes fewer process cycle clocks compared with the original one. As for the normal implementation, we found that difference between the original cipher and the dual ones is quite small. Therefore, we believe that if the Rijndael variants are implemented in bit level, the efficiency of the algorithm could be benefit from certain dual ciphers.

On the other hand, there are still many issues to be investigated in the future. In this thesis, we just come up with a dual cipher with local optimization in terms of the bit-slicing technique due to the time limitation. Thereby, it would be necessary to find the dual cipher with the best performance among the 23040 ciphers. What's more, in this thesis, we applied the methods proposed by Oswald et.al to implement the `SubBytes` transformation, which is involved in decomposing GF(2^8) into GF(2^4).

However, in [7, 8], Canright put forward the method to decompose GF(2^8) into GF(2^2) and there will be 432 ways to implement the `SubBytes` transformation just in terms of the original Rijndael. Therefore, considering all the 240 dual ciphers, there would be 103680 ciphers to be investigated in total.

Another interesting research direction would be what if the Rijndael variants are implemented via the vectorization technique mentioned in [11]. Intuitively, the performance would be improved as well since it is also necessary to decompose GF(2^8) into GF(2^4) while conducting the calculation in GF(2^8), which means some bit level operations will be involved. Hence, it is quite necessary to implement the dual ciphers to check whether this is true.

References

[1] Elad Barkan and Eli Biham. In How Many Ways Can You Write Rijndael? In *ASIACRYPT*, pages 160–175, 2002. 2, 3, 11, 12, 13, 37, 53

[2] Elad Barkan and Eli Biham. The Book of Rijndaels. *IACR Cryptology ePrint Archive*, 2002:158, 2002. 14, 27, 37

[3] Daniel J. Bernstein and Peter Schwabe. New AES Software Speed Records. In *Proceedings of the 9th International Conference on Cryptology in India: Progress in Cryptology*, INDOCRYPT '08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag. 2, 19, 30, 50

[4] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 159–171, London, UK, UK, 2003. Springer-Verlag. 19, 20, 21, 30

[5] Eli Biham. A Fast New DES Implementation in Software. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, FSE '97, pages 260–272, London, UK, UK, 1997. Springer-Verlag. 2, 19, 21, 30

[6] Joan Boyar and René Peralta. A depth-16 circuit for the AES S-box. *IACR Cryptology ePrint Archive*, 2011:332, 2011. 31

[7] David Canright. A Very Compact S-Box for AES. In *CHES*, pages 441–455, 2005. 31, 54

[8] David Canright and Dag Arne Osvik. A More Compact AES. In *Selected Areas in Cryptography*, pages 157–169, 2009. 31, 54

[9] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002. 1, 6, 19, 20

[10] Gunnar Gaubatz and Berk Sunar. Leveraging the Multiprocessing Capabilities of Modern Network Processors for Cryptographic Acceleration. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, NCA '05, pages 235–238, Washington, DC, USA, 2005. IEEE Computer Society. 23, 30

[11] Mike Hamburg. Accelerating aes with vector permute instructions. In *CHES*, pages 18–32, 2009. 54

[12] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 1–17, Berlin, Heidelberg, 2009. Springer-Verlag. 2, 19, 30, 50

[13] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, CT-RSA'08, pages 187–202, Berlin, Heidelberg, 2008. Springer-Verlag. vi, 30, 35, 50

[14] Jyun-Wei Lyu. Design and Implementation of Composite-Dual Cipher Based on AES. Master's thesis, National Cheng Kung University, 2006. 39, 40

[15] Mitsuru Matsui. How far can we go on the x64 processors? In *Proceedings of the 13th international conference on Fast Software Encryption*, FSE'06, pages 341–358, Berlin, Heidelberg, 2006. Springer-Verlag. v, 19, 22, 23, 30, 50

[16] Mitsuru Matsui and Junko Nakajima. On the Power of Bitslice Implementation on Intel Core2 Processor. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 121–134, Berlin, Heidelberg, 2007. Springer-Verlag. 2, 19, 30, 50

[17] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. 28

[18] James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, and Edward Roback. Status Report On The First Round Of The Development Of The Advanced Encryption Standard. In *Journal of Research of the National Institute of Standards and Technology 104. URL: http://nvl.nist.gov/pub/ nistpubs/jres/104/5/cnt104-5.htm. Citations in this document: 3*, pages 435–459, 1999. vi, 10

[19] Naval Postgraduate School Monterey CA Dept of Mathematics and D. Canright. *A Very Compact Rijndael S-Box*. Storming Media, 2004. 31

[20] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software AES encryption. In *Proceedings of the 17th international conference on Fast software encryption*, FSE'10, pages 75–93, Berlin, Heidelberg, 2010. Springer-Verlag. 19, 30

[21] Christof Paar and Verein Deutscher Ingenieure. *Efficient VLSI Architecture for Bit Parallel Computation in Galios [Galois] Fields*. Fortschritt-Berichte VDI. VDI-Verlag, 1994. 40

[22] Dan Page. *A Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer, 2009. v, 16, 17

[23] Bart Preneel, Antoon Bosselaers, B. Preneel, A. Bosselaers, Vincent Rijmen, Jacques Stern, Sean Murphy, B. Van Rompay, Louis Granboulan, Eli Biham, Orr Dunkelman, V. Furman, Fran?ois Koeune, J. Stern, Jean-Jacques Quisquater, S. Murphy, Markus Dichtl, Pascale Serf, E. Biham, O. Dunkelman, V. Furman F. Koeune, Gilles Piret, J j. Quisquater, Lars Knudsen, and H. Raddum. Comments by the NESSIE Project on the AES Finalists, 2000. 10

[24] Chester Rebeiro, David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In *Proceedings of the 5th international conference on Cryptology and Network Security*, CANS'06, pages 203–212, Berlin, Heidelberg, 2006. Springer-Verlag. vi, 2, 19, 23, 24, 30, 50

[25] Vincent Rijmen. Efficient Implementation of the Rijndael S-box. 31

[26] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '01, pages 171–184, London, UK, UK, 2001. Springer-Verlag. 19, 40

[27] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, pages 239–254, London, UK, UK, 2001. Springer-Verlag. 19

[28] Claude Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal, Vol 28, pp. 656C715*, Oktober 1949. 5

[29] Nigel Smart. *Cryptography: an introduction*. Mcgraw-hill education. McGraw-Hill, 2003. v, 5, 9

[30] Jon Stokes. *Inside the machine: an illustrated introduction to microprocessors and computer architecture*. No Starch Press Series. No Starch Press, 2007. v, 14, 15, 16, 17, 18, 19

[31] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An ASIC Implementation of the AES SBoxes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, CT-RSA '02, pages 67–78, London, UK, UK, 2002. Springer-Verlag. v, 19, 31, 32, 33, 34

[32] Shee-Yau Wu, Shih-Chuan Lu, and Chi-Sung Laih. Design of AES Based on Dual Cipher and Composite Field. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 25–38. Springer, 2004. v, 13, 14, 15, 25, 26

Appendix

Now that except for bit-slicing technique, all the implementations are quite straightforward, here we just list some main transformation functions in bit-slicing implementation.

```
The following source code is the basic calculations in GF(2^4).
void multiplication(uint64_t* a, uint64_t* b, uint64_t* q)
{
        q[0] = (a[0] & b[0]) ^ (a[3] & b[1]) ^ (a[2] & b[2]) ^ (a[1] & b[3]);
        q[1] = (a[1] & b[0]) ^ ((a[0] ^ a[3]) & b[1]) ^ ((a[2] ^ a[3]) & b[2]) ^ ((a[1] ^ a[2]) & b[3]);
        q[2] = (a[2] & b[0]) ^ (a[1] & b[1]) ^ ((a[0] ^ a[3]) & b[2]) ^ ((a[2] ^ a[3]) & b[3]);
        q[3] = (a[3] & b[0]) ^ (a[2] & b[1]) ^ (a[1] & b[2]) ^ ((a[0] ^ a[3]) & b[3]);
}

void square(uint64_t* a, uint64_t* q)
{
        q[0] = a[0] ^ a[2];
        q[1] = a[2];
        q[2] = a[1] ^ a[3];
        q[3] = a[3];
}

void inverse(uint64_t* a, uint64_t* q)
{

        q[0] = a[0] ^ a[1] ^ a[2] ^ (a[0] & a[2]) ^ (a[1] & a[2]) ^ (a[0] & a[1] & a[2]) ^ a[3] ^ (a[1] & a[2] & a[3]);
        q[1] = (a[0] & a[1]) ^ (a[0] & a[2]) ^ (a[1] & a[2]) ^ a[3] ^ (a[1] & a[3]) ^ (a[0] & a[1] & a[3]);
        q[2] = (a[0] & a[1]) ^ a[2] ^ (a[0] & a[2]) ^ a[3] ^ (a[0] & a[3]) ^ (a[0] & a[2] & a[3]);
        q[3] = a[1] ^ a[2] ^ a[3] ^ (a[0] & a[3]) ^ (a[1] & a[3]) ^ (a[2] & a[3]) ^ (a[1] & a[2] & a[3]);
}

void addition(uint64_t* a, uint64_t* b, uint64_t* q)
{
        q[0] = a[0] ^ b[0];
        q[1] = a[1] ^ b[1];
        q[2] = a[2] ^ b[2];
        q[3] = a[3] ^ b[3];
}

void constantMultiplication(uint64_t* a, uint64_t* q)
{
        q[0] = a[0] ^ a[1];
        q[1] = a[2];
        q[2] = a[3];
        q[3] = a[0];
}


This function illustrates how the multiplicative inverse is conducted based on the basic calculation above.
void G256_inv(uint64_t* bit8)
{
        uint64_t ah[4];
        uint64_t al[4];

        ah[3] = bit8[7];
        ah[2] = bit8[6];
        ah[1] = bit8[5];
        ah[0] = bit8[4];

        al[3] = bit8[3];
        al[2] = bit8[2];
        al[1] = bit8[1];
        al[0] = bit8[0];
        uint64_t d[4], ahSquare[4], t0[4], t1[4], alSquare[4], t2[4], t3[4], t4[4], t5[4], t6[4];

        //t0 = A * ah2
        square(ah, ahSquare);
        constantMultiplication(ahSquare, t0);
        //t1 = ah * al
        multiplication(ah, al, t1);
        //alSquare = al2
        square(al, alSquare);
        addition(t0, t1, t2);
        //t3 = t0 + t1 + alSquare
        addition(t2, alSquare, t3);
        //d = t3(-1)
        inverse(t3, d);

        multiplication(ah, d, t4);
        addition(ah, al, t5);
        multiplication(t5, d, t6);

        bit8[7] = t4[3];
        bit8[6] = t4[2];
        bit8[5] = t4[1];
        bit8[4] = t4[0];
        bit8[3] = t6[3];
        bit8[2] = t6[2];
        bit8[1] = t6[1];
        bit8[0] = t6[0];
}


The SubBytes transformation is illustrated as following functions.
void sbox(uint64_t* a)
{
        uint64_t x[8];
        uint64_t y[8];

        x[0] = a[0] ^ a[1] ^ a[3];
        x[1] = a[1] ^ a[5];
        x[2] = a[1] ^ a[3] ^ a[4] ^ a[7];
        x[3] = a[1] ^ a[2] ^ a[5];
        x[4] = a[4] ^ a[7];
        x[5] = a[1] ^ a[3] ^ a[4];
        x[6] = a[2] ^ a[5] ^ a[6];
        x[7] = a[5];

        G256_inv(x);

        a[0] = x[0] ^ x[2] ^ x[4] ^ 0x0000000000000000;
        a[1] = x[1] ^ x[7] ^ 0x0000000000000000;
        a[2] = x[1] ^ x[3] ^ 0xffffffffffffffff;
        a[3] = x[0] ^ x[1] ^ x[7] ^ 0x0000000000000000;
        a[4] = x[0] ^ x[1] ^ x[5] ^ x[7] ^ 0x0000000000000000;
        a[5] = x[3] ^ 0xffffffffffffffff;
        a[6] = x[1] ^ x[2] ^ x[4] ^ x[6] ^ 0xffffffffffffffff;
        a[7] = x[1] ^ x[2] ^ x[7] ^ 0x0000000000000000;
}

void subBytes(uint64_t* r)
{
        uint64_t t[8];

        for(int i = 0; i < 128; i += 8)
        {
                t[0] = r[i];
                t[1] = r[i + 1];
                t[2] = r[i + 2];
                t[3] = r[i + 3];
                t[4] = r[i + 4];
                t[5] = r[i + 5];
                t[6] = r[i + 6];
                t[7] = r[i + 7];

                sbox(t);

                r[i] = t[0];
                r[i + 1] = t[1];
                r[i + 2] = t[2];
                r[i + 3] = t[3];
                r[i + 4] = t[4];
                r[i + 5] = t[5];
                r[i + 6] = t[6];
                r[i + 7] = t[7];
        }
}
```

```
The ShiftRows transformation is illustrated as following function.
void shiftRows(uint64_t* r)
{
        uint64_t t0[8];
        uint64_t t1[8];

        for(int i = 0; i < 8; i++)
        {
                t0[i] = r[i + 8];
                r[i + 8] = r[i + 40];
                r[i + 40] = r[i + 72];
                r[i + 72] = r[i + 104];
                r[i + 104] = t0[i];

                t0[i] = r[i + 16];
                t1[i] = r[i + 48];
                r[i + 16] = r[i + 80];
                r[i + 48] = r[i + 112];
                r[i + 80] = t0[i];
                r[i + 112] = t1[i];

                t0[i] = r[i + 120];
                r[i + 120] = r[i + 88];
                r[i + 88] = r[i + 56];
                r[i + 56] = r[i + 24];
                r[i + 24] = t0[i];
        }
}


The MixColumns transformation is illustrated as following function.
void mixColumns(uint64_t* r)
{
        uint64_t t[128];

        for(int i = 0; i < 128; i += 8)
        {
                t[i] = r[i % 32 +  i / 32 * 32] ^ r[(i + 7) % 32 +  i / 32 * 32] ^ r[(i + 15) % 32 + i / 32 * 32]
                ^ r[(i + 16) % 32 + i / 32 * 32] ^ r[(i + 24) % 32 +  i / 32 * 32];
                t[i + 1] = r[i % 32 +  i / 32 * 32] ^ r[(i + 1) % 32 +  i / 32 * 32] ^ r[(i + 8) % 32 +  i / 32 * 32]
                ^ r[(i + 17) % 32 + i / 32 * 32] ^ r[(i + 25) % 32 + i / 32 * 32];
                t[i + 2] = r[(i + 1) % 32 + i / 32 * 32] ^ r[(i + 2) % 32 +  i / 32 * 32] ^ r[(i + 7) % 32 +  i / 32 * 32]
                ^ r[(i + 9) % 32 +  i / 32 * 32] ^ r[(i + 15) % 32 +  i / 32 * 32] ^ r[(i + 18) % 32 + i / 32 * 32]
                ^ r[(i + 26) % 32 + i / 32 * 32];
                t[i + 3] = r[(i + 2) % 32 + i / 32 * 32] ^ r[(i + 3) % 32 +  i / 32 * 32] ^ r[(i + 7) % 32 +  i / 32 * 32]
                ^ r[(i + 10) % 32 +  i / 32 * 32] ^ r[(i + 15) % 32 +  i / 32 * 32]^ r[(i + 19) % 32 + i / 32 * 32]
                ^ r[(i + 27) % 32 + i / 32 * 32];
                t[i + 4] = r[(i + 3) % 32 + i / 32 * 32] ^ r[(i + 4) % 32 +  i / 32 * 32] ^ r[(i + 7) % 32 +  i / 32 * 32]
                ^ r[(i + 11) % 32 +  i / 32 * 32] ^ r[(i + 15) % 32 +  i / 32 * 32] ^ r[(i + 20) % 32 + i / 32 * 32]
                ^ r[(i + 28) % 32 + i / 32 * 32];
                t[i + 5] = r[(i + 4) % 32 + i / 32 * 32] ^ r[(i + 5) % 32 +  i / 32 * 32] ^ r[(i + 12) % 32 +  i / 32 * 32]
                ^ r[(i + 21) % 32 + i / 32 * 32] ^ r[(i + 29) % 32 + i / 32 * 32];
                t[i + 6] = r[(i + 5) % 32 + i / 32 * 32] ^ r[(i + 6) % 32 + i / 32 * 32] ^ r[(i + 13) % 32 + i / 32 * 32]
                ^ r[(i + 22) % 32 + i / 32 * 32] ^ r[(i + 30) % 32 + i / 32 * 32];
                t[i + 7] = r[(i + 6) % 32 + i / 32 * 32] ^ r[(i + 7) % 32 + i / 32 * 32] ^ r[(i + 14) % 32 + i / 32 * 32]
                ^ r[(i + 23) % 32 + i / 32 * 32] ^ r[(i + 31) % 32 + i / 32 * 32];
        }

        for(int i = 0; i < 128; i++)
        {
                r[i] = t[i];
        }
}


The AddRoundKey transformation is illustrated as following function.
void addRoundKey(uint64_t* r, uint64_t* key)
{
        for(int i = 0; i < 128; i++)
        {
                r[i] = r[i] ^ key[i];
        }
}
```