블록암호 S-box 소프트웨어 구현 방법

IT정보공학과 신명수

목차

2021 암호경진대회
6번 문제 : 암호구현
Timing Attack은 비밀정보에 의존하는 암호화 연산 수행 시 발생하는 시간 정보를 이용하여 비밀정보를 추출하는 공격 기법을 의미한다. 대칭키에 대한 Timing Attack을 방어하기 위해서는 비트슬라이싱 기법을 통해 시간 의존적 암호화 연산을 제거하는 것이 중요하다. 아래에는 비트슬라이싱이 적용되지 않은 대칭키 암호화에 대한 레퍼런스 코드가 제시되어 있다. 해당 암호에 비트슬라이싱 기법을 적용하고 8-비트 저전력 프로세서에 맞게 최적화 고속구현하시오.

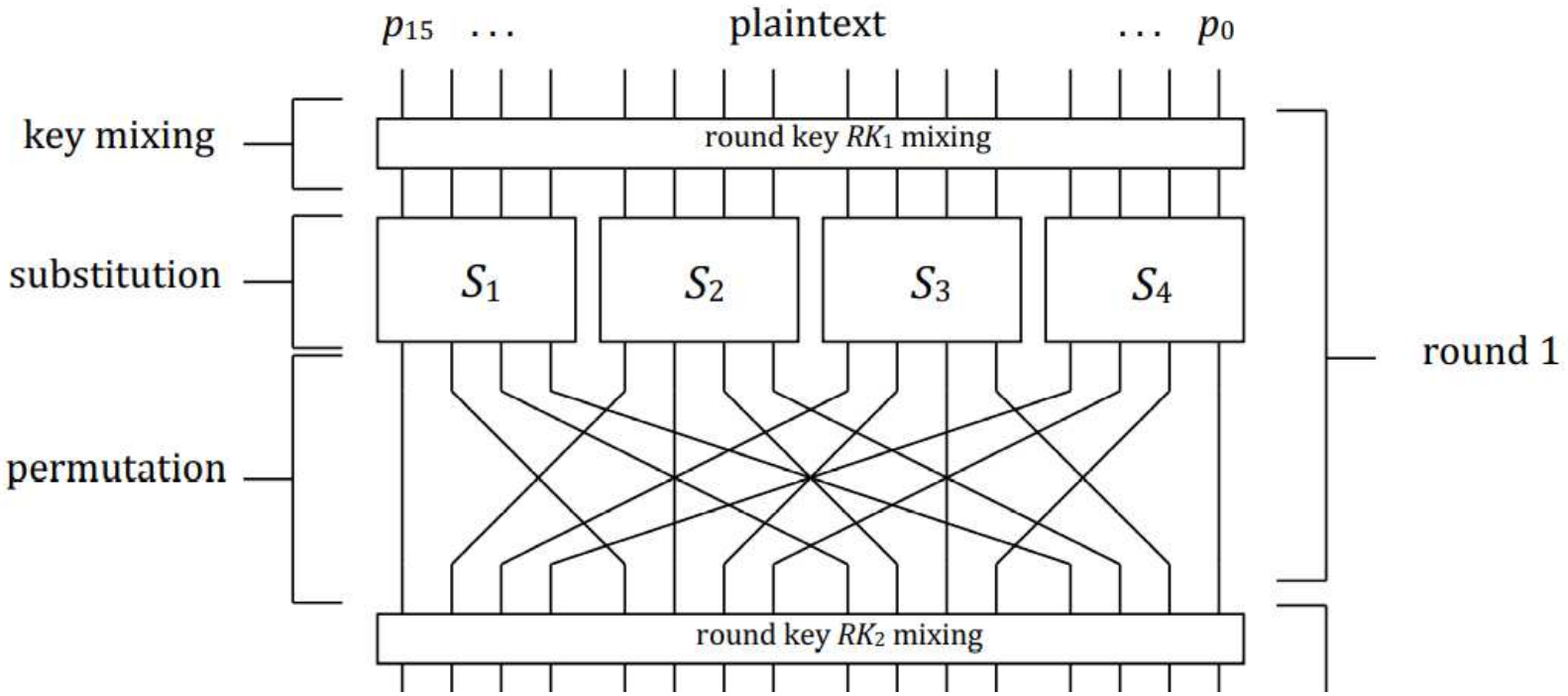
1. SPN ?

- Substitution Permutation Network
- Substitution과정과 Permutation과정을 반복하는 구조.
- Shannon’s product cipher의 특성이 잘 드러난 구조. (confusion, diffusion)

1. SPN

- Substitution Cipher와 Permutation Cipher를 중첩하는 형태로 개발.
- Substitution : 일정한 규칙에 따라 다른 값으로 치환한다.
- Permutation : 일정한 규칙에 따라 재배열 한다.

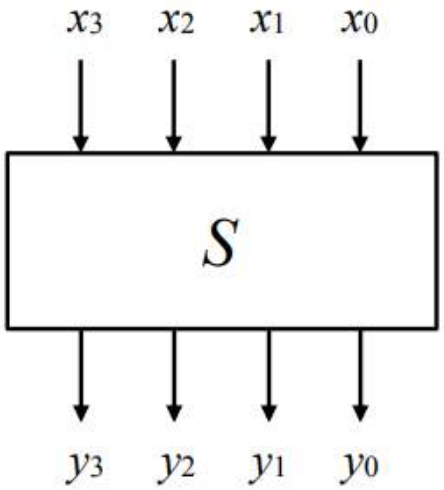
1. SPN



1 block 이 16-bit인 블록암호 예시.

2. S-box

- Substitution : 일정한 규칙에 따라 다른 값으로 치환한다.
- 비선형적 성질을 만들어 준다.



input	0	1	2	3	4	5	6	7
output	C	5	6	B	9	0	A	D
input	8	9	A	B	C	D	E	F
output	3	E	F	8	4	7	1	2

4-bit S-box Mapping table.

2. S-box

- S-box의 특성은 암호의 보안성과 매우 밀접한 연관성을 가짐.
- S-box는 일대일 대응이어야 한다.

input	0	1	2	3	4	5	6	7
output	C	5	6	B	9	0	A	D
input	8	9	A	B	C	D	E	F
output	3	E	F	8	4	7	1	2

4-bit S-box Mapping table.

3. S-box Implementations

- Table Lookup Implementations
- Bit-slice Implementations

3. Table Lookup Implementations

- 구현 효율성을 높이기 위해 사용하는 방법 중 하나.

```
for (byte = 0; byte < CRYPT_SIZE; byte++)
{
    high_= (p_text[byte] & 0xF0) >> 4;
    high_ = s_box[high_];

    low_ = p_text[byte] & 0x0F;
    low_ = s_box[low_];

    p_text[byte] = (high_ << 4) | low_;
}
```

```
static uint8_t s_box[16] = {
    0xC, 0x5, 0x6, 0xB,
    0x9, 0x0, 0xA, 0xD,
    0x3, 0xE, 0xF, 0x8,
    0x4, 0x7, 0x1, 0x2
};
```

3. Bit-slice Implementations

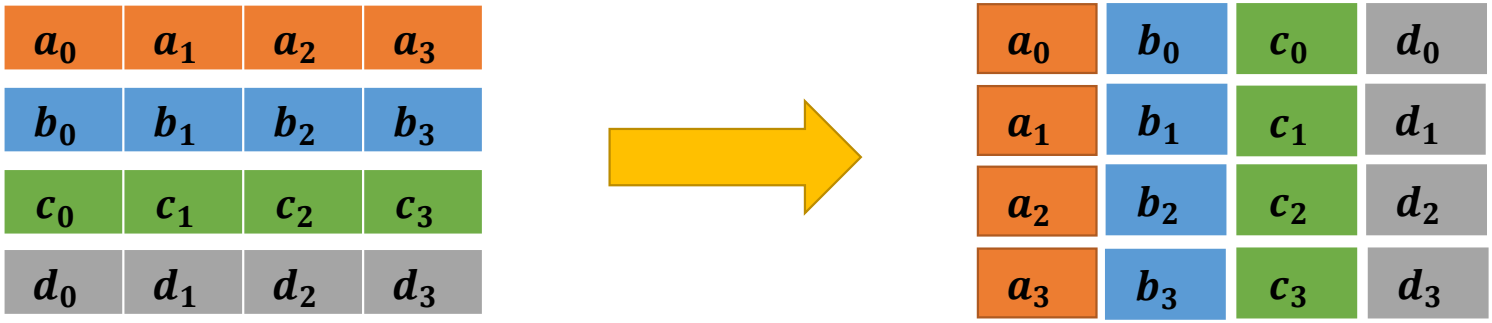
- Biham에 의해 DES 구현으로 처음으로 제시되었다.
- S-box 연산을 Boolean 함수로 구현하는 방법이다.
- Bit-slice approach 3 phase process
  1. 평문 블록을 bit-slicing 포맷으로 변경한다.
  2. 비트연산자를 통해 데이터를 처리한다.
  3. Bit-slice 데이터를 암호문 데이터로 변환한다.

3. Bit-slice Implementations

- 캐시 및 타이밍 관련 부채널 공격에 영향을 미치지 않는 암호화 알고리즘을 빠르고 상수 시간에 구현할 수 있도록 하는 구현 전략.
- Biham에 의해 DES 구현으로 처음으로 제시되었다.
- S-box 연산을 Boolean 함수로 구현하는 방법이다.
- Bit-slice approach 3 phase process
  1. 평문 블록을 bit-slicing 포맷으로 변경한다.
  2. 비트연산자를 통해 데이터를 처리한다.
  3. Bit-slice 데이터를 암호문 데이터로 변환한다.

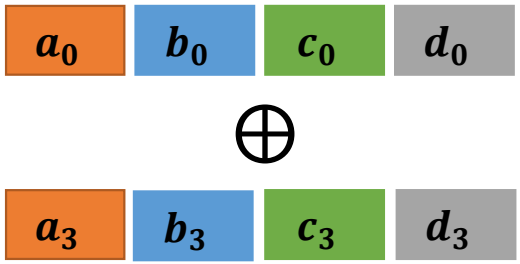
3. Bit-slice Implementations

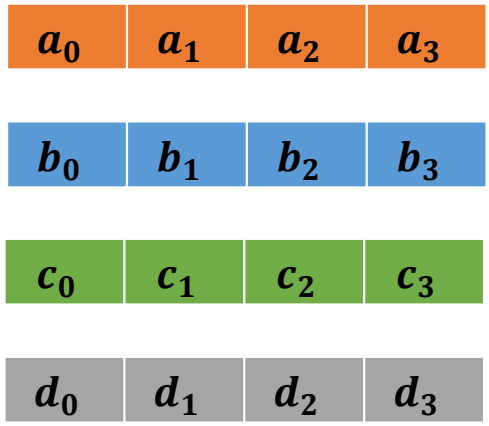
1. 평문 블록을 bit-slicing 포맷으로 변경한다.



3. Bit-slice Implementations

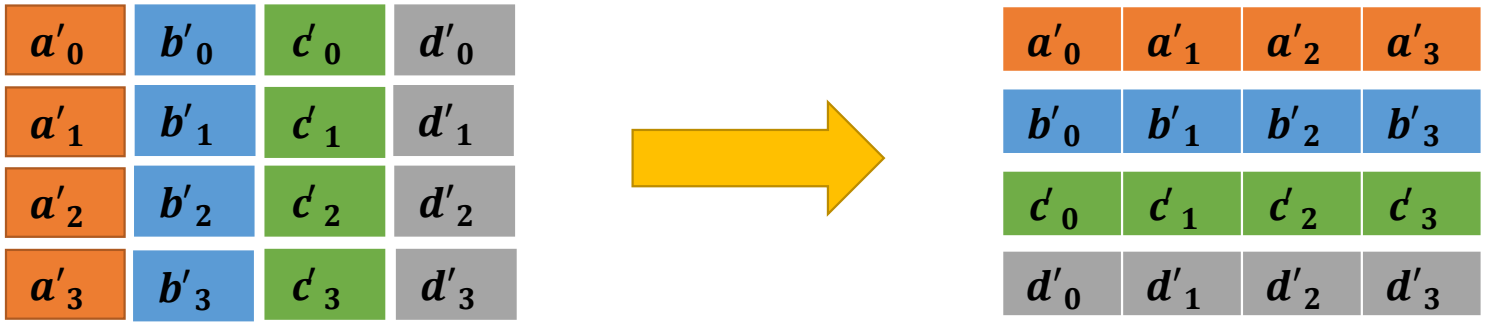
2. 비트 연산자를 통해 데이터를 처리한다.





3. Bit-slice Implementations

3. bit-slice 데이터를 암호문 데이터로 변환한다.



3. Bit-slice Implementations

Input : 4-bit				Output : S-box 4-bit			
a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3
0	0	0	0	1	1	0	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
0	1	0	0	1	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1
1	0	0	0	0	0	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	1	1
1	1	1	0	0	0	0	1
1	1	1	1	0	0	1	0

input	0	1	2	3	4	5	6	7
output	C	5	6	B	9	0	A	D
input	8	9	A	B	C	D	E	F
output	3	E	F	8	4	7	1	2

4-bit S-box Mapping table.

3. Bit-slice Implementations

Input : 4-bit				Output : S-box 4-bit			
a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3
0	0	0	0	1	1	0	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
0	1	0	0	1	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1
1	0	0	0	0	0	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	1	1
1	1	1	0	0	0	0	1
1	1	1	1	0	0	1	0

a_0a_1 \ a_2a_3	00	01	11	10
00	1	0	1	0
01	1	0	1	1
11	0	0	0	0
10	0	1	1	1

b_0에 대한 카르노 맵

3. Bit-slice Implementations

a_0a_1 \ a_2a_3	00	01	11	10
00	1	0	1	0
01	1	0	1	1
11	0	0	0	0
10	0	1	1	1

b_0에 대한 카르노 맵

$$b_0 = \overline{a_0}\overline{a_2}\overline{a_3} + \overline{a_1}a_2a_3 + \overline{a_0}a_1a_2 + a_0\overline{a_1}a_3 + a_0\overline{a_1}a_2$$

3. Bit-slice Implementations

Input : 4-bit				Output : S-box 4-bit			
a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3
0	0	0	0	1	1	0	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	0
0	0	1	1	1	0	1	1
0	1	0	0	1	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1
1	0	0	0	0	0	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	1	1
1	1	1	0	0	0	0	1
1	1	1	1	0	0	1	0

$$b_0 = \overline{a_0}\overline{a_2}\overline{a_3} + \overline{a_1}a_2a_3 + \overline{a_0}a_1a_2 + a_0\overline{a_1}a_3 + a_0\overline{a_1}a_2$$

$$b_1 = \overline{a_0}a_1a_2a_3 + \overline{a_0}\overline{a_1}\overline{a_3} + \overline{a_1}\overline{a_2}a_3 + \overline{a_1}a_2\overline{a_3} + a_0a_1\overline{a_2}$$

$$b_2 = \overline{a_0}\overline{a_1}a_2 + \overline{a_0}a_2\overline{a_3} + a_0a_2a_3 + a_0\overline{a_1}\overline{a_2} + a_0\overline{a_1}a_3$$

$$b_3 = \overline{a_0}a_1\overline{a_2}\overline{a_3} + a_0a_1\overline{a_2}a_3 + a_0\overline{a_1}\overline{a_3} + \overline{a_0}a_2a_3 + a_0a_2\overline{a_3} + a_0\overline{a_1}\overline{a_3}$$

3. Bit-slice Implementations

$$b_0 = \overline{a_0}\overline{a_2}\overline{a_3} + \overline{a_1}a_2a_3 + \overline{a_0}a_1a_2 + a_0\overline{a_1}a_3 + a_0\overline{a_1}a_2$$

$$b_1 = \overline{a_0}a_1a_2a_3 + \overline{a_0}\overline{a_1}\overline{a_3} + \overline{a_1}\overline{a_2}a_3 + \overline{a_1}a_2\overline{a_3} + a_0a_1\overline{a_2}$$

$$b_2 = \overline{a_0}\overline{a_1}a_2 + \overline{a_0}a_2\overline{a_3} + a_0a_2a_3 + a_0\overline{a_1}\overline{a_2} + a_0\overline{a_1}\overline{a_3}$$

$$b_3 = \overline{a_0}a_1\overline{a_2}\overline{a_3} + a_0a_1\overline{a_2}a_3 + a_0\overline{a_1}\overline{a_3} + \overline{a_0}a_2a_3 + a_0a_2\overline{a_3} + a_0\overline{a_1}a_3$$

```
u32 new_s_box_gen(u32 text)
{
    u32 output = 0;
    // transpose
    u8 num1 = (u8)(((text & 0x80000000) >> 24) | ((text & 0x8000000) >> 21) | ((text & 0x800000)
    >> 18) | ((text & 0x80000) >> 15) | ((text & 0x8000) >> 12) | ((text & 0x800) >> 9) | ((text & 0x80) >>
    6) | ((text & 0x8) >> 3));
    u8 num2 = (u8)(((text & 0x40000000) >> 23) | ((text & 0x4000000) >> 20) | ((text & 0x400000)
    >> 17) | ((text & 0x40000) >> 14) | ((text & 0x4000) >> 11) | ((text & 0x400) >> 8) | ((text & 0x40) >>
    5) | ((text & 0x4) >> 2));
    u8 num3 = (u8)(((text & 0x20000000) >> 22) | ((text & 0x2000000) >> 19) | ((text & 0x200000)
    >> 16) | ((text & 0x20000) >> 13) | ((text & 0x2000) >> 10) | ((text & 0x200) >> 7) | ((text & 0x20) >>
    4) | ((text & 0x2) >> 1));
    u8 num4 = (u8)(((text & 0x10000000) >> 21) | ((text & 0x1000000) >> 18) | ((text & 0x100000)
    >> 15) | ((text & 0x10000) >> 12) | ((text & 0x1000) >> 9) | ((text & 0x100) >> 6) | ((text & 0x10) >>
    3) | ((text & 0x1) ));
    // f, g, h, k with karnaugh map and bit slicing
    u8 f = (u8)((~(num1 | num2) & num4) | ((num1 | num2) & ~num4));
    u8 g = (u8)((num1 & ~(num2 | num3)) | (~num1 & (num2 | num3)));
    u8 h = (u8)(((~num1 | num2) & ~(num3 | num4)) | (~num2 & ((num1 & num4) | num3)));
    u8 k = (u8)((num2 & ((~num1 & num3 & ~num4) | (num1 & ~num3))) | (~num3 & num4) |
    (~(num1 | num2) & (~num3 | num4)));
    output = (((u32)f << 24) | ((u32)g << 16) | ((u32)h << 8) | ((u32)k));
    return output;
}
```