2차 과제

블록 암호 ARIA에 대한 마스킹 부채널 대응기술 설계 및 구현

2020270103 임현성

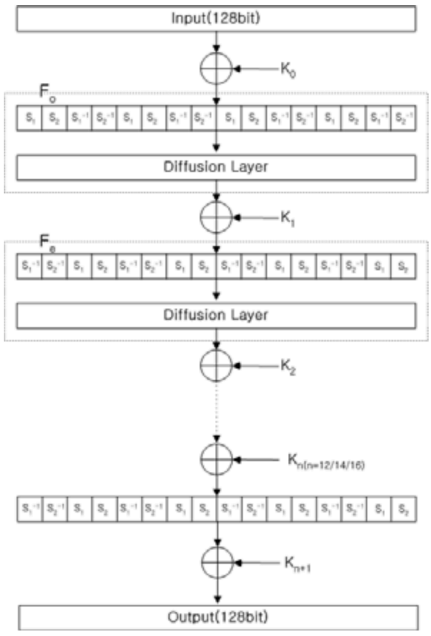
2024년 12월 6일

1. ARIA 알고리즘 설명

1.1 ARIA의 동작 구조

ARIA 알고리즘은 SPN 구조(Substitution Permutation Network)를 기반으로 설계되었습니다.

암호화 과정은 다음과 같은 단계로 이루어집니다:



1. 입력 블록 처리

입력 평문은 128비트 크기의 블록 단위로 처리됩니다. 이는 키와 XOR 연산을 통해 치환 연산을 수행합니다.

2. 라운드 반복

ARIA는 키 크기에 따라 12, 14, 16 라운드가 수행됩니다. 각 라운드는 치환 계층, 확산 계층, 키 스케줄링으로 구성됩니다.

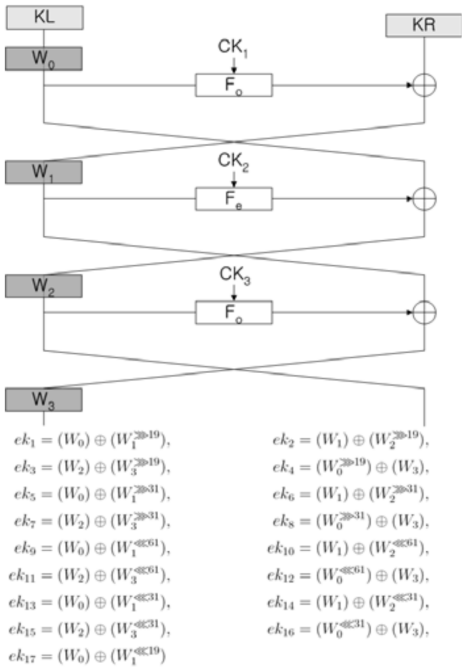
- **치환 계층**
  - 두 개의 S-Box(S1, S2)와 역치환 S-Box($S1^{-1}$, $S2^{-1}$)를 사용하여 비선형 변환을 수행합니다.
  - S1과 S2는 8비트 입력을 받아 8비트 출력을 반환하며, $GF(2^8)$ 상의 수학적 변환을 이용합니다.
- **확산 계층**
  - 입력 16바이트를 특정 행렬 연산을 통해 섞는 과정으로, 암호문의 혼돈과 확산을 증가시킵니다.

- **키 스케줄링**
  - 각 라운드에서 고유한 라운드 키를 평문과 XOR 연산하여 치환 계층과 확산 계층을 결합합니다.

3. 최종 라운드 출력

마지막 라운드에서는 추가적인 치환과 확산 없이, 최종 라운드 키와 XOR 연산하여 암호화된 128 비트의 암호문을 출력합니다.

1.2 ARIA 키 스케줄링



키 스케줄링은 주어진 암호 키로부터 라운드 키를 생성하는 과정입니다. ARIA는 키 크기에 따라 다른 고정값(CK_0, CK_1 , CK_2)을 사용하며, 아래와 같은 과정으로 키를 생성합니다:

1. 초기 키를 두 개의 블록(KL, KR)으로 분할한다.
2. KL과 KR에 대해 F_0, F_1 , F_2 연산을 수행한다.
3. 각 라운드 키(W_0, W_1 , ...)를 계산하여 암호화 및 복호화 과정에서 사용한다.

2. ARIA 마스킹 기법의 설계 방식

2.1 개념 정의

논문에 제시된 S-box 마스킹 설계는 S-box의 입력값과 출력값 모두에 마스킹 값을 적용하여 부채널 공격에 노출되는 정보를 제거하는 방식입니다. 이를 위해 마스킹된 S-box는 다음과 같이 정의됩니다:

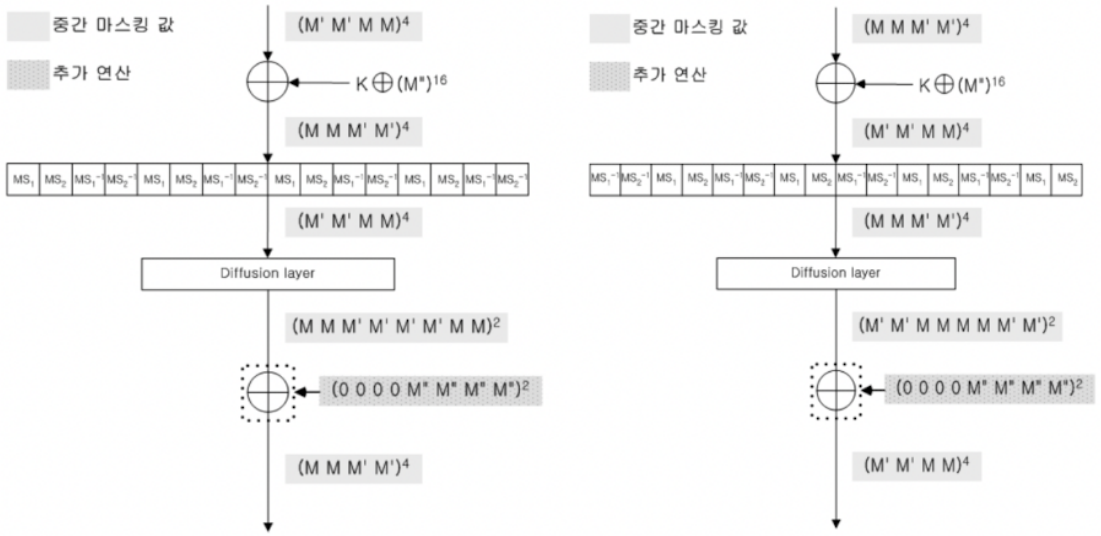
$$MS_1(x \oplus m) = S_1(x) \oplus m'$$
$$MS_2(x \oplus m) = S_2(x) \oplus m'$$
$$MS_3(x \oplus m') = S_1^{-1}(x) \oplus m$$
$$MS_4(x \oplus m') = S_2^{-1}(x) \oplus m$$

이 설계는 다음 4단계를 통해 구현됩니다:

1. 입력 마스킹 : 입력값 x에 마스킹 값 m을 XOR 연산으로 적용한다.
2. S-box 변환 : 마스킹이 적용된 입력값을 S-box에 전달한다.
3. 출력 마스킹 : S-box 출력값에 출력 마스킹 m을 XOR 연산으로 적용한다.
4. 역방향 변환 : MS가 양방향 변환(암호화 및 복호화)을 지원하도록 역방향 매핑한다

2.2 암호화 과정에서의 마스킹

암호화 과정에서는 라운드별로 데이터를 변환하며 마스킹을 유지합니다. 이 과정은 각 라운드에서 데이터, 키, 마스크가 통합적으로 적용되며, 논문에서 제안된 방식은 다음과 같습니다:



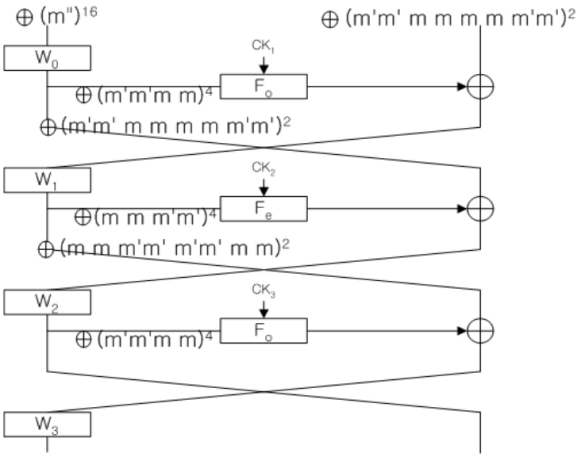
홀수 라운드와 짝수 라운드 경우

2.2.1 마스킹 과정

1. 초기 평문 마스킹
    - 평문 데이터에 입력 마스킹을 XOR 연산으로 적용한다

2. S-box 마스킹
    - 입력값에 마스킹을 적용한 후, S-box를 통해 변환한다
3. 확산 계층 마스킹
    - 확산 계층에서는 각 바이트에 다른 마스킹 값을 적용한다
4. 라운드 키와의 마스킹 연산
    - 라운드 키에도 마스크 값을 적용한다.

2.3 키 스케줄링에서의 마스킹



2.3.1 설계 개요

1. 마스킹된 키 생성
    - 기본 키 K에 마스킹 mk를 적용한다.
2. 라운드 키 분할
    - 라운드 키 생성 과정에서 각 부분에 서로 다른 마스킹을 적용한다.
3. 라운드별 마스킹
    - 각 라운드에서 마스킹된 라운드 키 Ki를 사용하여 암호화를 수행한다.

2.3.2 키 스케줄링 흐름

키 스케줄링 마스킹 흐름은 다음과 같습니다

1. 입력 키에 마스킹 값 m을 XOR 연산으로 적용한다
2. 각 라운드 키 생성 시, m, mp, mpp를 순차적으로 적용하여 보안 강화한다
3. 마지막 라운드에서 마스크를 제거하여 암호문 복구한다

3. ARIA 마스킹 구현

3.1 구현 방식

3.1.1 generateMaskingSbox 함수

generateMaskingSbox 함수는 ARIA 알고리즘에 사용할 마스킹된 S-Box를 생성합니다. 입력된 두 개의 마스킹 값 m_1과 m_2를 사용하여 S-Box를 변형합니다.

```
void generateMaskingSbox(Byte S[4][256], Byte m_1, Byte m_2, Byte masking_Sbox[3][256]) {
    for (int i = 0; i < 256; i++) {
        Byte masking_index = (Byte)i ^ m_1;
        Byte making_Sbox_m_1 = S[0][i] ^ m_2;
        Byte making_Sbox_m_2 = S[1][i] ^ m_2;

        masking_Sbox[0][masking_index] = making_Sbox_m_1;
        masking_Sbox[1][masking_index] = making_Sbox_m_2;
        masking_Sbox[2][making_Sbox_m_1] = masking_index;
        masking_Sbox[3][making_Sbox_m_2] = masking_index;
    }
}
```

- m_1 : S-box 입력값에 XOR 연산으로 적용되는 마스킹 값.
- m_2 : S-box 출력값에 XOR 연산으로 적용되는 마스킹 값.
- 생성된 masking_Sbox는 암호화 및 복호화 과정에서 마스킹된 상태의 S-box를 대체합니다.

3.1.2 EncKeySetup에서의 마스킹 적용

키 설정 과정에서, 입력 키와 중간 키에 마스킹 값을 XOR 연산으로 적용하여 부채널 공격으로부터 보호합니다.

```
for (i = 0; i < 16; i++) {
    w0[i] ^= masking_xor;  // KL 마스킹
}

for (i = 16; i < 32; i++) {
    if (i == 16 || i == 17 || i == 22 || i == 23 ||
        i == 24 || i == 25 || i == 30 || i == 31) {
        w0[i] ^= m_2;  // KR 마스킹 (짝수 바이트)
    } else {
        w0[i] ^= m_1;  // KR 마스킹 (홀수 바이트)
    }
}
```

- 첫 번째 16바이트는 KL에 마스킹을 적용하며, XOR 연산을 통해 보호합니다.
- 다음 16바이트는 KR에 대해 홀수 및 짝수 바이트별로 다른 마스킹 값을 XOR하여 적용합니다.

```
Byte s1 = (masking_xor >> 3) | (masking_xor << (8 - 3));
Byte s2 = (masking_xor >> 7) | (masking_xor << (8 - 7));
Byte s3 = (masking_xor << 5) | (masking_xor >> (8 - 5));
Byte s4 = (masking_xor << 7) | (masking_xor >> (8 - 7));

for (i = 0; i < 64; i++) e[i] ^= s1;
for (i = 64; i < 128; i++) e[i] ^= s2;
for (i = 128; i < 192; i++)e[i] ^= s3;
for (i = 192; i < 256; i++)e[i] ^= s4;
```

- 마스킹 값을 변형하기 위한 비트 이동 연산을 수행합니다.
- 마스킹 값을 생성한 뒤, 키의 구간에 XOR 연산을 적용하여 암호화 키를 마스킹합니다.

3.1.3 암호화 함수 Crypt

암호화 과정에서 평문에 마스킹을 적용하여 부채널 정보 노출을 막습니다.

```
for (j = 0; j < 16; j++) c[j] = p[j];
for (j = 0; j < 16; j += 4) {
    c[j] ^= m_2;  // 짝수 바이트 마스킹
    c[j+1] ^= m_2;
    c[j+2] ^= m_1;  // 홀수 바이트 마스킹
    c[j+3] ^= m_1;
}
```

라운드 연산에서는 홀수/짝수 라운드별로 S-box와 확산 계층을 거치며 마스킹을 유지합니다.

```
 for (j = 0; j < 16; j++) {
     t[j] = masking_Sbox[j % 4][e[j] ^ c[j]];  // 홀수 라운드
 }
 DL(t, c);

 for (j = 0; j < 16; j++) {
     t[j] = masking_Sbox[(2 + j) % 4][e[j] ^ c[j]];  // 짝수 라운드
 }
 DL(t, c);
```

최종적으로 암호문에서 마스크를 제거하여 원래 암호문이 출력됩니다.

```
for (j = 0; j < 16; j += 4) {
    c[j] ^= m_2;
    c[j + 1] ^= m_2;
    c[j + 2] ^= m_1;
    c[j + 3] ^= m_1;
}
```

3.2 암호화 결과 검증

128비트 암호키에 대한 테스트 벡터를 이용하여 암호화 결과를 확인했습니다.

- 라운드 별 출력값

```
Encryption:
 1 round : 71 f2 58 e5 33 a1 25 79 48 29 48 8f 65 5d 8f f6
 2 round : d5 b0 6a 76 fb 8b 55 96 3f c4 4b 2f 03 f0 70 4d
 3 round : 8d 40 db a4 e1 86 bb 7b bf d9 c1 57 04 4b 24 74
 4 round : 6c 07 61 05 c3 1e 92 ac ab 19 8d 71 59 a3 04 6c
 5 round : ae 5c 56 34 83 ff 97 9e be e0 78 c6 94 3d 7f e8
 6 round : 83 4c bc 0e 00 c0 5e 66 d4 04 36 19 f6 6c 61 71
 7 round : 4f 6b f4 a8 2a 33 7b 1d e1 fb 5d 56 7b f7 01 42
 8 round : b8 34 ab 22 69 87 b9 99 f4 dc ba 5d 24 a5 c3 37
 9 round : 48 c9 b1 56 b1 f1 8d 37 73 19 57 f5 11 e9 c9 7b
10 round : 18 c0 6a 98 d0 d5 e3 4a 9f 63 a2 94 39 56 1c 6f
11 round : 4f f0 7f d1 78 83 28 84 29 3a 5f 91 5f f6 34 bb
12 round : c6 ec d0 8e 22 c3 0a bd b2 15 cf 74 e2 07 5e 6e
```

12 라운드 암호화 결과

c6 ec d0 8e 22 c3 0a bd b2 15 cf 74 e2 07 5e 6e 를 출력값으로 얻는다는 것을 확인했습니다.

```
Byte cryptResult[] = {
    0xc6, 0xec, 0xd0, 0x8e, 0x22, 0xc3, 0x0a, 0xbd,
    0xb2, 0x15, 0xcf, 0x74, 0xe2, 0x07, 0x5e, 0x6e
};
```

```
key       : 00112233 44556677 8899aabb ccddeeff
plaintext: 11111111 aaaaaaaa 11111111 bbbbbbbb
result is: c6ecd08e 22c30abd b215cf74 e2075e6e
should be: c6ecd08e 22c30abd b215cf74 e2075e6e
Okay.  The result is correct.
```

테스트 벡터와 일치하는 암호화 결과값을 얻어, 마스킹에 성공했음을 확인했습니다.