정보보호개론

제8장 암호알고리즘: 대칭 암호알고리즘

1. 암호화 모드

대칭 암호알고리즘은 크게 스트림 암호와 블록 암호로 나뉜다. 이 중 블록 암호는 정해진 크기의 입력만 받도록 설계되어 있으며. 대부분의 블록 암호는 결정적 알고리즘이다. 따라서 블록 크기보다 큰 메시지나 작은 메시지를 암호화하기 위한 방법이 필요하며, 이를 **암호화 모드**라 한다.

가장 기본적인 모드는 2장에서 이미 살펴본 ECB 모드이다. 이 모드에서는 평문을 블록 크기로 나눈 후 각 블록을 독립적으로 암호화한다. 이 때문에 같은 평문 블록이 있으면 같은 암호문 블록으로 암호화된다. 따라서 평문의 패턴이 암호문에 나타나는 문제점이 있다. 이 때문에 일반적으로 사용하는 암호화 모드는 평문 패턴이 암호문에 나타나지 않도록 설계한다. 보통 평문 블록을 단독으로 암호화하지 않고, 다른 것에 영향을 받도록 하여 평문 블록이 같더라도 다른 암호문 블록으로 암호화되도록 하고 있다. 이 과정을 피드백이라고 한다. 하지만 피드백 과정으로 인하여 암호화 성능에 무시할 수 없을 정도로 많은 영향을 주면 해당 모드는 실용적으로 사용하기 어렵다. 따라서 피드백 과정이 매우 효율적이어야 한다.

암호화 모드를 분석할 때 보통 다음 4가지를 검토하게 된다.

- (평문 오류) 암호화하기 전에 한 평문 블록에 오류가 발생하였을 때 암호문 블록과 전체 암호문에 미치는 영향
- (암호문 조작) 암호화한 후에 암호문을 조작하였을 때 평문 블록과 전체 평문에 미치는 영향
- (추가 보안 문제) 암호화 모드 사용에 따른 추가적인 보안 문제의 유무
- (효율성) 모드 연산의 비용, 다중 프로세서(코어)의 활용 가능 여부 등을 분석

암호문 조작의 경우에는 두 암호문 블록을 바꾼 후 복호화하였을 때 평문에 미치는 영향과 암호문 블록에 오류가 발생하였을 때 복호화한 평문에 미치는 영향을 보통 살펴본다.

첫 번째 분석은 유사한 평문들을 암호화한 결과를 보기 위한 것이다. 너무나 당연하지만 평문에 오류(일부 비트 값이 바뀐 경우)가 있을 때 나중에 암호문을 복호화하면 그 오류는 평문에 그대로 남아 있다. 따라서 여기서 분석하고자 하는 것은 암호문의 변화이다. 이때 대응되는 암호문 블록의 변화와 전체 암호문의 변화를 나누어 살펴볼 수 있다. 둘 다 미치는 영향이 클수록 바람직하다. 그 이유는 암호문 블록을 통해 평문 블록을 유추할 수 없어야 하기 때문이다. 그렇지 않을 경우 두 암호문 블록이 유사하면 그것의 평문 블록도 유사하다는 것을 알 수 있다. 특히, 평문 블록의 변화가 암호문 블록에 미치는 영향은 예측할 수 없어야 한다. 이것은 평문 전체와 암호문 전체에 대해서도 마찬가지이다.

암호문 조작의 경우에는 두 가지 세부적 검토가 필요하다. 첫째, 암호문 조작을 통해 평문에 의미 있는 변화를 줄 수 있는지를 분석하게 된다. 이것은 2장에서 검토한 NM 특성과 관련된 것이다. 둘째, 암호문이 조작되었을 때 복호화된 평문에 미치는 영향을 분석하게 된다. 이 영향은 앞서 평문의 조작과 마찬가지로 복호화된 평문 블록과

전체 평문 블록에 미치는 영향을 나누어 살펴보게 된다. 복호화된 평문 블록에 미치는 영향은 크고 예측할 수 없어야 NM 특성이 제공되며, 전체 평문에 미치는 영향은 반대로 적어야 정상적인 사용자가 원 평문을 얻기 위한 비용이 최소화될 수 있다. 하지만 복호화한 사용자가 어떤 블록이 문제가 되었는지 판단하기 어렵고, 이 특성을 오히려 공격에 활용할 수 있기 때문에 이것이 중요하게 요구되는 특성이라고 보기는 어렵다.

세 번째 분석은 원래는 없던 보안 문제가 모드의 도입으로 새로 생길 수 있다. 이와 같은 문제점들을 살펴보는 것이다. 네 번째 효율성은 여러 블록을 동시에 암호화하거나 복호화할 수 있는지를 분석하는 것이다. 이것이 가능할 경우 다중 프로세서를 활용하여 암호화 속도를 높일 수 있다. 또한 어떤 암호화 모드의 경우에는 암호화와 복호화 함수가 모두 필요하지 않은 경우도 있다. 이 경우 암호 모듈을 더 소형화할 수 있는 이점도 있다.

1.1 채우기

평문 메시지가 블록 크기보다 작거나 평문 메시지의 크기가 정확하게 블록 크기의 배수가 아니면 마지막 평문 블록을 블록 크기로 만들기 위해 임의의 데이터를 추가하는 것을 **채우기**(padding)란 한다. 보통 일련의 비트(예: 모두 0, 모두 1)를 추가하여 완전한 블록을 만드는 방법을 사용한다. 채우기는 복호화한 사용자가 채워진 부분을 정확하게 제거할 수 있어야 한다. 이를 제공하는 방법은 다음과 같다.

- 방법 1. 암호문과 별도로 원 평문의 크기를 전달
- 방법 2. 채우기를 한 부분에 채운 부분을 제거할 수 있는 요소를 포함

보통 통신 프로토콜에서는 이번에 받아야 하는 메시지의 크기가 결정되어 있기 때문에 수신자가 메시지에 포함된 암호문의 평문 크기를 알고 있다. 따라서 보통 원 평문의 크기를 별도 전달하지 않아도 알 수 있다. 하지만 이 경우에도 방법 2을 사용하면 채우기 데이터를 통해 메시지를 검증할 수 있는 효과가 있으므로 임의의 데이터로 채우기보다는 약속된 데이터로 채우는 것이 바람직하다. 물론 채우기 데이터가 명백한 여분 정보가 되는 문제점은 있다.

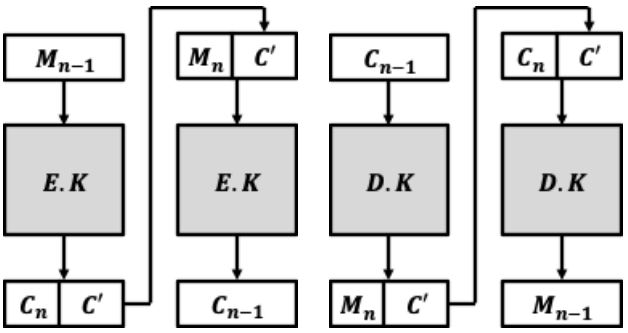
방법 2는 크게 비트 채우기와 바이트 채우기로 나누어진다. 보통 비트 채우기는 채우기의 첫 비트만 1로 하고 나머지를 0으로 채운다. 복호화한 사용자는 마지막 블록 끝에 있는 연속된 0과 그다음 1 하나를 제거함으로써 채워진 데이터를 제거한다. 바이트 채우기에서는 마지막 바이트에 채우기 해야 하는 바이트 수를 기록하고 나머지 부분은 일련의 비트로 채우는 방법을 사용한다. 비트 채우기와 바이트 채우기는 모두 평문이 정확하게 블록 크기의 배수이면 한 블록 전체를 채우기 해야 한다. 이렇게 하지 않으면 원래 데이터를 채우기 데이터로 오해할 수 있는 문제점이 발생할 수 있다.

x	x	x	x	x	x	x	x	x	x	x	x	4	4	4	4

<그림 8.1> PKCS7 채우기: 블록크기 16바이트, 채우기 크기 4바이트

RFC 5652 PKCS 7에 정의된 표준 채우기 방법은 채우는 모든 바이트를 채우는 바이트 크기 값으로 채운다[1]. 이를 통해 채운 데이터에 대한 추가 확인을 할 수 있도록 한다. 예를 들어 블록 크기가 128bit(16 byte)일 때 4byte의 채우기가 필요하면 그림 8.1과 같이 마지막 4byte에 각각 4로 채우는 방법이다.

지금까지 살펴본 채우기 방법은 평문의 크기가 정확하게 블록 배수이면 한 블록 전체를 채우기해야 하는 문제점이 있다. 따라서 암호문의 크기가 평문보다 최대 한 블록이 커질 수 있다. 이 문제는 **암호문 훔침 기법**(ciphertext stealing)이라는 채우기 기법을 사용하면 암호문의 크기와 평문의 크기를 일치시킬 수 있다. 총 n개의 평문 블록이 있을 때 $n-1$번째 암호문의 일부를 채우기 값으로 사용하는 기법이다. 암호문 훔침 기법은 그림 8.2와 같이 동작하며,



(1) 암호화 과정 (2) 복호화 과정

<그림 8.2> 암호문 훔침 기법

마지막 두 평문 블록을 암호화하는 것을 수식으로 나타내면 다음과 같고,

$$\begin{aligned} C_n||C' &= E.K(M_{n-1}) \\ C_{n-1} &= E.K(M_n||C') \end{aligned}$$

이 두 블록을 복호화하는 것은 다음과 같다.

$$\begin{aligned} M_n||C' &= D.K(C_{n-1}) \\ M_{n-1} &= D.K(C_n||C') \end{aligned}$$

위 식에서 C_n를 복호화하기 위해서는 C'이 필요한데, C'은 C_{n-1}를 복호화하여 얻을 수 있다. 참고로 마지막 두 블록의 위치가 바뀌게 된다. 또한 암호화할 평문이 한 블록보다 작은 경우에는 이 기법을 사용할 수 없다.

1.2 ECB 모드

ECB 모드는 평문을 블록 크기로 나누어 각 블록을 독립적으로 암호화하는 방식으로 식으로 표현하면 다음과 같다.

$$\begin{aligned} C_i &= E.K(M_i) \\ M_i &= D.K(C_i) \end{aligned}$$

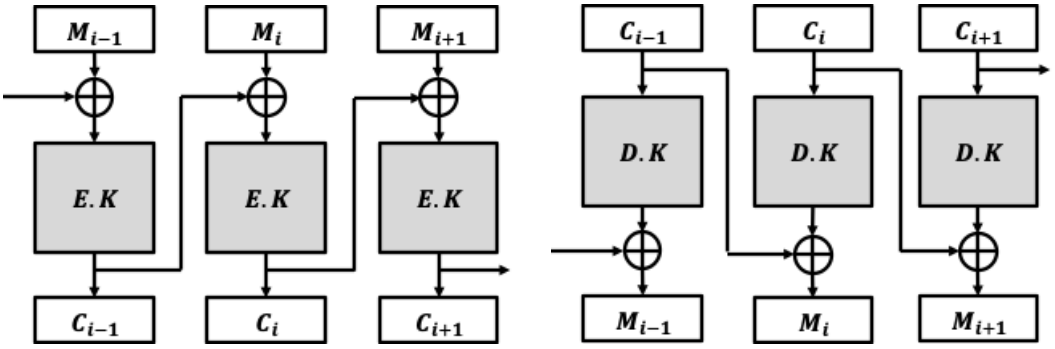
ECB 모드는 채우기가 필요한 모드이며, 암호문 크기와 평문 크기를 일치시키고 싶으면 암호문 훔침 기법을 사용할 수 있다. ECB 모드를 분석하면 다음과 같다.

- 평문 오류: M_i 블록에 오류가 있으면 C_i에만 오류가 발생한다. 따라서 유사한 평문을 암호화하면 암호문도 유사해진다.
- 암호문 블록 교체: C_i와 C_j를 교체하면 M_i와 M_j의 위치가 바뀐다. 평문에 내용에 따라 의미 있는 변화를 줄 수 있다. 이 때문에 같은 키로 암호화된 또 다른 암호문의 블록과 교체하여 의미있는 변화를 주는 공격도 가능하다.
- 암호문 오류: C_i에 오류가 발생하면 M_i는 예측하지 못하게 변한다. 하지만 다른 블록에는 영향을 주지 않는다.
- 추가 보안 문제: CBC 모드에서 자세히 살펴보지만 같은 키로 암호화된 여러 암호문 블록을 조합하여 새 암호문을 만들 수 있으며, 그것의 복호화하는 오류 없이 대응되는 평문 블록으로 복호화된다.

- 효율성: 암호화와 복호화 과정을 모두 병렬 처리할 수 있다.

분석 결과 ECB 모드는 평문 패턴이 암호문에 나타나는 심각한 문제를 가지고 있으며, NM 특성(암호문 블록 교체)도 제공하지 못한다. 따라서 일반 메시지는 ECB 모드로 절대 암호화하지 않는다. 하지만 암호화할 데이터가 한 블록 이하이면 암호문의 크기를 최소화하기 위해 ECB 모드를 사용하는 경우도 종종 있다.

1.3 CBC 모드



(1) 암호화 과정 (2) 복호화 과정

<그림 8.3> CBC 모드

CBC 모드는 이전 암호문 블록을 평문 블록과 XOR하여 그림 8.3과 같이 암호화하는 방법이다. 이것을 수식으로 나타내면 다음과 같다.

$$\begin{aligned} C_i &= E.K(M_i \oplus C_{i-1}) \\ M_i &= D.K(C_i) \oplus C_{i-1} \end{aligned}$$

이전 암호문 블록이 있어야 현재 평문 블록을 암호화할 수 있으므로 평문은 순차적으로 암호화할 수밖에 없다. 따라서 CBC 모드에서 암호화는 병렬 수행이 가능하지 않다. 복호화는 모든 암호문을 다 받은 상태에서는 병렬 수행이 가능하다.

CBC 모드는 이전 암호문 블록을 이용하기 때문에 첫 평문 블록은 활용할 암호문 블록이 없다. 따라서 랜덤한 블록을 생성하여 사용하며, 이를 **초기 벡터**(IV, Initialization Vector)라 한다. 초기 벡터의 장점은 같은 평문을 같은 암호키로 암호화하더라도 다른 초기 벡터를 이용하면 결과가 달라진다. 따라서 내부적으로 사용하는 암호화 함수는 결정적 알고리즘이지만 전체적인 측면에서 보면 암호화는 확률적 알고리즘이다. 하지만 초기 벡터 때문에 암호문의 길이는 한 블록이 증가하게 되며, 채우기까지 생각하면 최대 두 블록이 커질 수 있다. 초기 벡터는 암호문의 일부로 전달하는 방식을 주로 사용하며, 초기 벡터의 비밀성은 요구되지 않는다. 하지만 예측이 가능하지 않아야 한다.

CBC 모드도 앞서 언급한 네 가지에 대해 분석해 보자. 첫째, 평문 블록 M_i에 오류가 있으면 C_i 이후 모든 암호문이 달라진다. 이것은 다음 식을 통해 이해할 수 있다.

$$\begin{aligned} C_i &= E.K(M_i \oplus C_{i-1}) & \qquad C'_i &= E.K(M'_i \oplus C_{i-1}) \\ C_{i+1} &= E.K(M_{i+1} \oplus C_i) & \qquad C'_{i+1} &= E.K(M_{i+1} \oplus C'_i) \end{aligned}$$

CBC 모드는 평문이 조금 다르더라도 그 영향이 다른 부분 이후 모든 블록에 영향을 준다. 더욱이 이와 같은 특성이 없더라도 초기 벡터를 사용하기 때문에 같은 평문을 다시 암호화하면 결과 암호문은 달라진다. 평문이 조금만 달라져도 마지막 암호문 블록은 달라지는 특성 때문에 CBC로 암호화한 암호문에서 마지막 암호문 블록은 평문 전체를 대표하는 값으로 활용할 수 있다. 즉, 마지막 블록을 MAC으로 활용할 수 있다. 이에 대해서는 9장에서 자세히

다룬다.

둘째, 특정 암호문 블록 C_i의 오류는 두 개의 평문 블록에 영향을 준다. 이것은 다음 식을 통해 확인할 수 있다.

$$\begin{array}{lcllcl} M_i & = & D.K(C_i) \oplus C_{i-1} & M'_i & = & D.K(C'_i) \oplus C_{i-1} \\ M_{i+1} & = & D.K(C_{i+1}) \oplus C_i & M'_{i+1} & = & D.K(C_{i+1}) \oplus C'_i \\ M_{i+2} & = & D.K(C_{i+2}) \oplus C_{i+1} & M_{i+2} & = & D.K(C_{i+2}) \oplus C_{i+1} \end{array}$$

하나의 암호문 블록에만 오류가 발생하더라도 두 블록 이후에는 다시 오류 없이 복호화되기 때문에 CBC는 자체 회복(self recovering) 기능을 가지고 있다고 한다. 그런데 여기서 좀 더 면밀히 분석하면 M'_i은 어떤 값이 될지 예측할 수 없지만 M'_{i+1}과 M_{i+1}은 C_i와 C'_i의 차이에 의해 그 차이가 결정된다. C_i의 j번째 비트만 토글한 것이 C'_i이면 M'_{i+1}과 M_{i+1}은 j번째 비트만 차이가 난다. 따라서 공격자가 이를 이용하여 예측된 변화를 줄 수 있으며, 이를 활용한 공격을 bitflip 공격이라 한다. 이 때문에 CBC 모드도 NM 특성을 만족하지 못한다.

CBC 모드는 몇 가지 추가적인 보안 문제를 가지고 있다. 첫째, 암호문 끝에 동일 키로 암호화된 암호문을 추가할 수 있다. 예를 들어 두 개의 암호문 $C_0, C_1, \ldots, C_n$와 $C'_0, C'_1, \ldots, C'_m$이 있을 때, 이를 $C_0, C_1, \ldots, C_n, C'_1, C'_2 \ldots, C'_m$처럼 결합하면 C'_1를 복호화한 부분($M^*_1 = D(C'_1) \oplus C_n$)만 오류가 발생한다. 이와 같은 문제는 ECB 모드에도 나타나는 문제이다.

둘째, 두 개의 암호문을 조합하여 새로운 암호문을 만들 수 있다. 예를 들어 위 예제의 두 암호문을 다음과 같이

$$C_0, C_1, \ldots, C_j, C'_k, C'_{k+1}, \ldots$$

조합하여 하나의 암호문을 만들면 결합된 위치의 암호문 블록인 C'_k만 오류가 발생한다. 이 문제도 이전 문제와 마찬가지로 ECB에서도 나타나며, 심지어 ECB는 뒤에 추가하거나 조합하여도 엉뚱하게 복호화되는 평문 블록이 없다.

표준 채우기를 사용할 경우에는 항상 맨 마지막 블록은 유효한 채우기가 있어야 한다. 따라서 이와 같이 결합하거나 조합할 때 유효한 채우기가 포함된 블록이 마지막 블록이 되도록 하지 않으면 복호화하는 측에서 쉽게 메시지가 조작되었다는 것을 알 수 있다. 또 인증 암호화를 하면 이와 같은 결합, 조합한 암호문에 대한 MAC 값을 공격자가 제시할 수 없으므로 쉽게 알아챌 수 있다.

셋째, 암호문 블록 오류에서 살펴보았듯이 하나의 암호문 블록 C_i을 조작하면 두 개의 평문 블록에 영향을 주게 되는데, 이 중 M_{i+1}에는 의도된 변화를 줄 수 있다. 넷째, 만약 C_i와 C_j가 같으면 다음 식에 알 수 있듯이 두 개의 평문을 XOR한 값을 얻을 수 있다.

$$C_{i-1} \oplus C_{j-1} = M_i \oplus D.K(C_i) \oplus M_j \oplus D.K(C_j) = M_i \oplus M_j$$

보통 평문은 예측이 가능하기 때문에 두 개의 평문이 XOR된 값으로부터 각 평문을 구하는 것은 어렵지 않다. 두 암호문 블록이 우연히 일치할 확률은 매우 낮기 때문에 걱정할 문제점은 아니다. 하지만 같은 키로 많은 양의 데이터를 암호화하면 이 확률이 점점 높아지기 때문에 시기적절한 키 갱신은 필요하다.

지금까지 시기적절한 키 갱신을 요구한 기술적 이유를 다시 정리해보면 다음과 같다.

- 이유 1. 난스 기법의 사용에서 난스를 랜덤하게 생성할 때 우연히 과거에 사용한 것과 같은 것을 사용할 가능성을 줄이기 위해 키 갱신이 필요하다.
- 이유 2. CBC 모드 사용에서 우연히 두 암호문 블록이 같아지는 것의 발생을 줄이기 위해 필요하다.
- 이유 3. CBC 모드에서는 필요하지 않지만 다음에 소개할 CTR 모드를 사용할 경우에는 우연히 IV가 같아지는 것의 발생을 줄이기 위해 필요하다.

보통 구체적인 시점은 생일 파라독스를 이용하여 계산할 수 있다. 예를 들어 64bit 난스이면 2^{32}개의 난스를 생성하여 사용하기 전에 키를 갱신할 필요가 있다. 2^{32}개의 난스를 생성하면 50% 확률로 같은 것을 생성하게 되므로 실제는 훨씬 이전에 키 갱신을 해야 한다.

보통 CBC 모드는 표준 채우기 방법을 사용하지만 CBC 모드의 경우에도 암호문 훔침 기법을 통해 평문의 크기와 암호문의 크기를 일치시킬 수 있다. 다만, 암호문 훔침 기법을 사용하더라도 IV 때문에 암호문이 한 블록만큼은 커지게 된다.

1.4 CBC Padding Oracle 공격

블록 크기가 16바이트인 CBC 모드 PKCS #7 padding을 사용하여 메시지를 암호화하여 상대방에게 보내는 프로토콜이 있다고 가정하자. 수신자가 메시지를 복호화한 다음 먼저 채우기를 확인하고 채우기 값이 올바른지 여부를 송신자에게 알려주는 프로토콜을 사용한다면 공격자는 이 회신을 이용한 다음과 같은 공격을 할 수 있다.

공격자가 $IV||C_1||\ldots||C_n$ 암호문을 가지고 있으면 C_i에 대응되는 평문 M_i을 다음과 같이 수신자의 채우기에 대한 회신을 통해 알아낼 수 있다. 예를 들어 공격자가 C_2에 대응되는 M_2의 마지막 바이트를 $0xXY$로 예측하였을 때, 자신의 예측이 맞았는지 암호문 $IV||C_1 \oplus 0xXY \oplus 0x01||C_2$을 수신자에게 전송하여 확인할 수 있다. 예측이 정확했다면 다음과 같이 수신자가 복호화한 M_2의 마지막 바이트는 0x01이 되며, 수신자는 채우기가 1바이트로 인식하고 채우기가 문제없다고 메시지를 회신하게 된다.

$$\begin{aligned} M_1 &= D.K(C_1 \oplus 0xXY \oplus 0x01) \oplus IV \\ M_2 &= D.K(C_2) \oplus C_1 \oplus 0xXY \oplus 0x01 \\ &= M_2 \oplus C_1 \oplus C_1 \oplus 0xXY \oplus 0x01 \\ &= M_2[15\ldots1]||0x01 \end{aligned}$$

반대로 정확하게 예측하지 못하였으면 엉뚱한 값이 되며, 채우기 방법에 의해 마지막 바이트가 연속으로 같아야 하므로 채우기가 잘못되었다고 회신하게 된다.

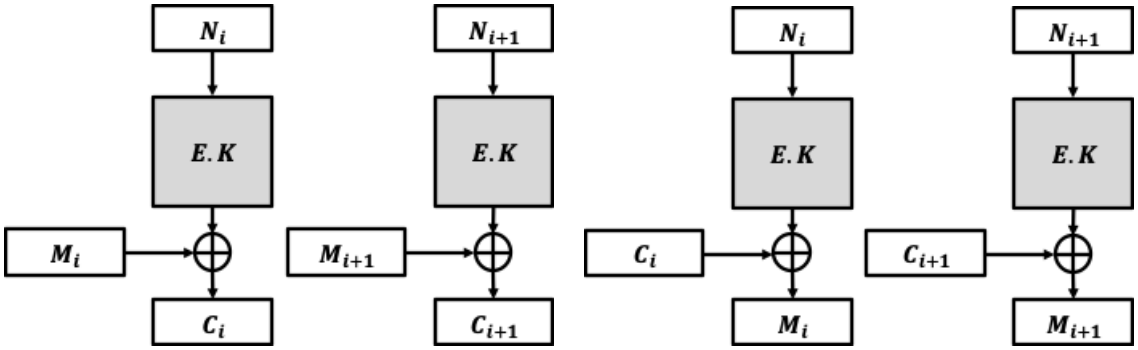
따라서 256번을 공격하면 한 블록의 마지막 바이트를 무조건 정확하게 알 수 있다. 마지막 바이트를 알게 되면 그것을 이용하여 같은 방법으로 다음 바이트에 대한 채우기 오라클 공격을 할 수 있다. TLS 초기 버전은 mac-then-encrypt 방식을 사용하였으며, CBC 모드로 메시지를 암호화하였다. 복호화 절차는 채우기를 확인하고, 그다음 MAC 값을 확인하는 방식이었으며, 두 종류의 오류 여부를 상대방에게 알려주었다. 여기서 교훈은 암호세계에서는 오류의 종류를 구분하여 알려주는 것도 매우 위험하다는 것이다. 현재의 TLS는 이 공격에 안전하지만, 이전 버전을 사용하도록 하여 공격하는 경우도 있다.

1.5 CTR 모드

CTR 모드는 사용하는 암호화 함수를 이용하여 평문을 암호화하지 않는다. 대신 암호화 함수를 이용하여 랜덤 스트림을 만들어 XOR 연산을 통해 평문을 암호화 방식을 사용하고 있으며, 병렬 수행이 가능하도록 하기 위해 카운터를 이용하여 스트림을 생성한다. CTR 모드는 그림 8.4와 같이 동작하며, 이 모드를 수식으로 나타내면 다음과 같다.

$$\begin{aligned} C_i &= M_i \oplus E.K(N_i) \\ M_i &= C_i \oplus E.K(N_i) \end{aligned}$$

위 식에서 알 수 있듯이 CTR 모드는 암호화 함수만 필요할 뿐만 아니라 꼭 암호화 함수를 이용할 필요도 없다. 암호화 함수 대신에 MAC 함수를 사용할 수 있다. 따라서 하드웨어로 제작할 경우 복호화 함수가 필요 없기 때문에



(1) 암호화 과정 (2) 복호화 과정

<그림 8.4> CTR 모드

더 간편한 회로로 제작이 가능하다. 또 카운터의 특성상 $N_i = N_{i-1} + 1$이므로 N_0만 알면 모든 N_i를 계산할 수 있기 때문에 암호화와 복호화 과정을 모두 병렬로 수행할 수 있다. 이 모드는 XOR을 이용하여 평문을 암호화하기 때문에 마지막 블록이 완전한 블록이 아니더라도 채우기가 필요 없다. 이것은 마지막 블록 크기만큼의 스트림을 사용하여 암호화하면 되기 때문이다.

CTR 모드도 CBC 모드와 마찬가지로 초기 벡터가 필요하며, 이 벡터값을 카운터의 시작 값으로 사용한다. 그런데 동일한 초기 벡터를 사용하면 XOR하는 스트림이 같아진다. 이 경우 두 암호문을 XOR하여 평문 블록간 XOR한 값을 얻을 수 있기 때문에 심각한 보안 문제가 발생한다. IV가 같지 않아도 두 암호문에 사용된 카운터가 일부 중첩될 수 있다. 이 때문에 IV를 랜덤하게 생성하지 않고, IV의 일부만 랜덤하게 생성하는 방식을 주로 사용한다. 예를 들어 블록 크기가 128비트이면 첫 96비트만 랜덤하게 생성하고 나머지 32비트는 카운터 역할을 하게 된다. 이와 같은 중첩 가능성 때문에 시기적절한 키 갱신(2^{48}개 이상은 암호화하지 않아야 함)이 CTR 모드를 사용할 경우에도 매우 중요하다.

IV 문제를 극복하기 위해 사용하는 또 다른 방법은 SIV(Synthetic IV)이다. SIV는 랜덤 IV 대신에 평문을 이용하여 IV를 계산한다. 이와 같은 방법을 사용하면 서로 다른 평문을 암호화할 때 같은 IV를 사용할 확률이 더 낮아진다. 같은 평문을 다시 암호화하면 IV가 바뀌지 않으므로 IV를 통한 확률 암호알고리즘으로의 전환 효과는 없어진다.

CTR 모드도 앞서 언급한 네 가지에 대해 분석해 보자. 첫째, 평문 블록 M_i의 오류는 C_i 암호문 블록에만 영향을 준다. 하지만 평문이 유사하더라도 암호화할 때 사용하는 스트림이 다르면 결과가 다르기 때문에 이것이 문제가 되지는 않는다.

둘째, 특정 암호문 블록에 오류가 발생하면 해당 평문 블록에만 영향을 주기 때문에 CBC 모드와 마찬가지로 자체 회복 기능을 가지고 있다. 하지만 XOR 연산을 사용하여 암호화하기 때문에 공격자는 원하는 변화를 줄 수 있다. 즉, C_i의 j번째 비트만 토글하면 M_i의 j번째 비트만 원래와 다르게 된다. 따라서 CTR 모드도 NM 특성을 만족하지 못한다. 두 개의 암호문 블록을 교체한 경우 CTR 모드의 특성에 따라 해당 블록의 평문들만 영향을 받는다.

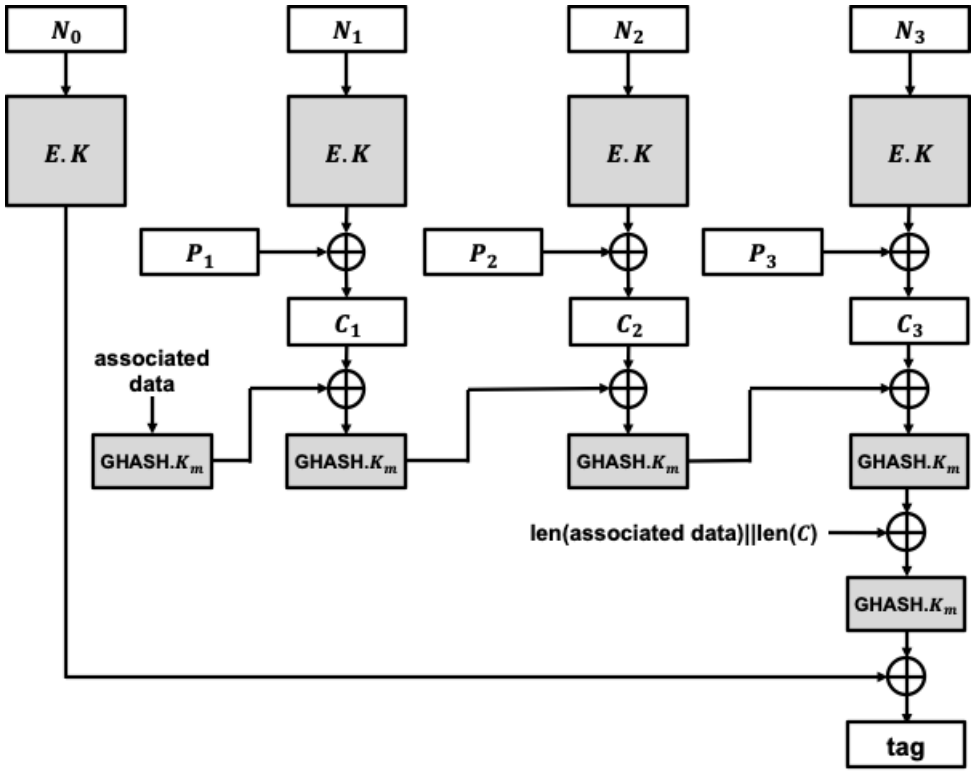
$$\begin{aligned} M_i &= C_i \oplus E.K(N_i) & \qquad M_i' &= C_j \oplus E.K(N_i) \\ M_j &= C_j \oplus E.K(N_j) & \qquad M_j' &= C_i \oplus E.K(N_j) \end{aligned}$$

셋째, CTR은 XOR 연산을 이용하여 암호화하기 때문에 발생하는 같은 초기 벡터의 사용 문제와 공격자가 암호문 블록을 조작하여 CBC보다 더 쉽게 의도된 변화를 줄 수 있다는 문제가 있다. CTR은 CBC의 효율 문제를 극복하기 위해 제안된 것이기 때문에 효율성 측면에서는 다른 모드들보다 우수하며, 암호화, 복호화 과정을 모두 병렬 처리할 수 있다.

1.6 인증 암호화와 암호화 모드

인증 암호화가 기본 암호화 방법이 되어 암호 모드에도 변화가 필요하게 되었다. 인증 암호화 중에 가장 안전하고 효과적인 방식이 encrypt-then-mac 방법이지만 처음에는 mac-then-encrypt 방식도 널리 사용하였다. 실제 TLS도 1.3 이전까지는 이 방식을 사용하였다. mac-then-encrypt 방식의 대표적인 인증 암호화 모드가 CCM(Counter Mode with CBC-MAC)이다. CCM 모드는 메시지에 대한 CBC-MAC을 계산한 후에 메시지와 MAC 값을 결합하여 CTR 모드로 암호화를 진행한다.

이 모드를 encrypt-then-mac 방식으로 대체하기 위해 처음 제안된 것이 EAX(encrypt-then-authenticate-then-translate) 모드이다. 이 방식은 CTR 모드로 암호화를 다 수행한 후에 결과 암호문을 이용하여 MAC 값을 계산한다. 이렇게 하는 것을 2-pass 인증 암호화 방식이라 한다. 더 빠르게 이 과정을 수행하기 위해 암호화와 MAC 계산을 동시에 수행하는 1-pass 암호화 모드가 제안되었으며, 현재는 이 방식을 더 많이 사용한다.



<그림 8.5> GCM 인증 암호화 모드

대표적인 1-pass 암호화 모드가 **GCM**(Galois-Counter-Mode)이다[2]. GCM 모드는 그림 8.5와 같이 동작한다. 기본적으로 카운터 모드를 이용하여 메시지를 암호화하지만 각 C_i를 이용하여 병렬로 MAC 값도 함께 계산한다. 이 과정에서 갈로아 필드에서 곱셈 연산을 사용하며, 이 연산은 비트 연산을 이용하여 효율적으로 계산할 수 있다. 최종적 MAC 값은 다음과 같이 계산한다.

$$T = MAC.K_m(C) \oplus E.K(N||0)$$

여기서 $K_m = E.K(0)$이며, $MAC.K_m(C)$는 암호화 과정에서 출력되는 C_i와 갈로아 필드에서 곱셈 연산을 이용하여 계산된다. 보통 encrypt-then-mac을 할 때 암호키와 MAC키는 서로 독립적인 키이어야 한다. 이들을 독립적으로 생성하여 암호화 모드에 전달할 수 있지만, 제시된 것처럼 GCM은 MAC키를 모두 0인 평문을 암호키로 암호화하여 생성한다.

GCM 모드는 내부적으로 블록 암호화를 사용한다. 내부적으로 블록 암호화 대신에 스트림 암호화를 이용하여

인증 암호화를 할 수 있다. 현재 대표적으로 사용하는 것이 ChaCha20-Poly1305이다. 하지만 ChaCha20-Poly1305은 1-pass 인증 암호화는 아니다. ChaCha20-Poly1305은 내부적으로 ChaCha20 스트림 암호화[3]를 사용하고 MAC은 Poly1305[4]를 사용한다.

인증 암호화 관련하여 AEAD(Authenticated Encryption with Associated Data)라는 모드도 있다. 지금까지 소개한 CCM, EAX, GCM, ChaCha20-Poly1305은 모두 AEAD를 지원한다. AEAD는 암호화되는 평문 외에 암호화되지 않는 평문 데이터(associated data)의 무결성까지 보장하여 준다. AEAD를 encrypt-then-mac 형태로 표현하면 다음과 같다.

$$A, C = E.K_1(M), T = MAC.K_2(A||C)$$

여기서 A가 암호화되지 않는 연관 데이터이다.

1.7 암호화 모드의 선택

ECB는 가장 빠른 모드이지만 근본적인 문제점을 가지고 있는 모드이다. 따라서 일반 메시지를 ECB 모드로 암호화하는 것은 바람직하지 않지만 암호키와 같이 한 블록보다 작은 크기의 랜덤한 값은 ECB 모드를 사용하여 암호화할 수 있다. 특히, 다른 모드를 사용하면 초기화 벡터가 필요하기 때문에 암호문의 크기가 평문에 비해 상대적으로 커지므로 응용에 따라 블록보다 작은 데이터는 ECB 모드를 사용하는 것이 효과적일 수 있다.

반면에 파일과 같은 많은 양의 데이터를 암호화할 때에는 초기화 벡터에 의한 크기의 증가를 무시할 수 있으므로 CBC나 CTR 모드를 사용한다. 하지만 CBC나 CTR 모드는 둘 다 NM 특성을 만족하지 못하기 때문에 지금은 encrypt-then-mac 방식의 인증 암호화 모드를 사용하는 것이 가장 바람직하다.

2. DES 대칭 암호알고리즘

2.1 DES의 역사

1972년 미국 표준화 기구인 NIST(National Institute of Standards)는 대칭 암호알고리즘에 대한 표준화 작업을 시작하여 1977년에 IBM에서 제안한 Lucifer라는 알고리즘을 일부 수정하여 표준으로 공식 채택하였다. 공식 채택된 알고리즘을 DES(Data Encryption Standard)라 하며, 표준번호는 FIPS 46이다[5]. Lucifer는 IBM의 암호학자 Feistel이 제안한 알고리즘이며, DES의 내부 구조는 Lucifier와 동일이다. DES 이후에 제안된 많은 대칭 암호알고리즘도 동일한 내부 구조로 되어 있다. 이 때문에 DES의 내부 구조를 Feistel 구조(structure)라 하며, Feistel 구조를 가지는 대칭 암호알고리즘을 Feistel 암호알고리즘이라고도 한다.

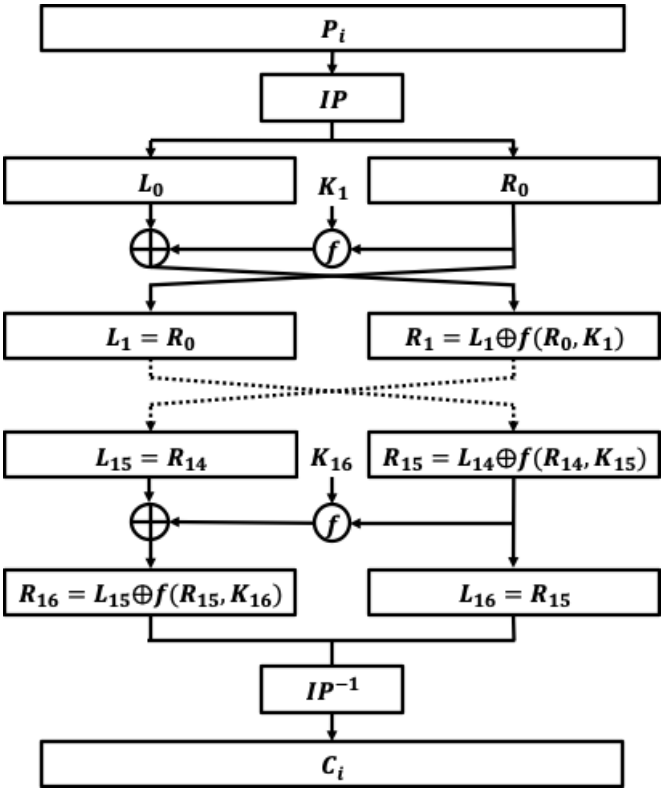
표준화 과정에서 NIST는 미국 국가안보국인 NSA(National Security Agency)에서 자문을 받아 IBM이 원래 제안했던 키 길이(128bit)를 56bit로 축소하였으며, 내부 동작 메커니즘 중 교체 연산에 해당하는 S-박스를 수정하였다. 이 때문에 NSA가 DES에 트랩도어를 포함하였다는 음모론이 있었지만 2009년 비밀 문서의 공개로 NSA가 IBM과 Lucifier 암호를 강건하게 만들기 위해 상호 협력한 사실이 밝혀졌으며, 학문적 연구 결과도 수정된 S-박스가 차분 암호해독(differential cryptoanalysis) 공격에 매우 강건하다는 것도 밝혀졌다. 하지만 축소된 키 길이가 오늘날 컴퓨팅 능력을 고려하였을 때 너무 짧다는 문제점을 가지고 있으며, 소프트웨어보다는 하드웨어에 더 적합한 알고리즘이라는 단점도 있다.

1977년에 표준으로 채택될 당시 이 표준의 유효기간은 10년이었으며, 유효기관이 만료되기 전에 10년을 더 연장하였지만 90년대 후반에는 컴퓨팅 기술의 발달과 비용 감소로 56bit 키로는 전사공격에 취약해짐에 따라 미국은 1997년에 새 표준을 마련하는 작업을 시작하였다. 새 표준으로 2000년 10월에 Rijndael 알고리즘이 채택되었으며, 이 표준을 AES(Advanced Encryption Standard)라 한다.

2.2 DES의 특징

DES는 블록 방식의 대칭 암호알고리즘으로 블록 크기는 64bit이고, 키 길이는 56bit이다. 키 길이는 원래 64bit이었지만 패리티 비트의 필요성 때문에 56bit로 축소했다. DES도 내부적으로 치환, 자리바꿈, 키와 XOR, 3가지 기본 연산을 사용하는 합성 암호(product cipher)이며, 기본 연산으로 구성된 라운드를 총 16번 반복한다. 라운드마다 해당 라운드의 중간 결과 값과 키 값이 XOR되는데, 라운드마다 서로 다른 랜덤한 키 값을 사용하는 것이 안전성 측면에서 필요하다. 라운드마다 48bit의 키값이 필요하고 총 16라운드로 구성되어 있으므로 96byte가 필요한데, DES의 키값은 7byte다. 96byte의 랜덤키를 DES키로 사용할 수 있지만, 교환, 보관 등의 문제뿐만 아니라 유사 키 공격이 가능해지므로 좋은 방법이 아니다. 따라서 대부분의 암호알고리즘은 사용자가 유지하는 키를 확장하여 알고리즘이 내부적으로 필요로 하는 키를 만들어 사용하게 되며, 이때 사용하는 확장 알고리즘을 키 스케줄링(key scheduling) 알고리즘이라 한다.

2.2.1 DES 암호화와 복호화 함수



<그림 8.6> DES 암호화 과정: Feistal 구조

DES의 암호화 함수는 그림 8.6과 같이 동작하며, 암호화 함수와 복호화 함수를 식으로 표현하면 다음과 같다.

$$\begin{aligned} E.K(M) &= \mathrm{IP}^{-1} J_{16}^{K_{16}} \cdots J_2^{K_2} J_1^{K_1} \mathrm{IP}(M) \\ D.K(C) &= \mathrm{IP}^{-1} J_{16}^{K_1} \cdots J_2^{K_{15}} J_1^{K_{16}} \mathrm{IP}(C) \end{aligned}$$

여기서 J는 라운드 함수이며, 식에서 알 수 있듯이 총 16라운드로 구성되어 있다. DES의 라운드 함수는 64bit 입력을 두 개의 32bit로 나누어 처리하며, 마지막 라운드에서 동일한 과정을 거치지만 왼쪽과 오른쪽이 바뀐다. 또 식에서 알 수 있듯이 사용자 암호키는 내부적으로 총 16개의 라운드키로 확장되며, 암호화할 때와 복호화할 때 사용하는 순서가 거꾸로이다. 이것은 적용된 XOR 연산을 차례로 역하기 위한 것이다. IP(Initial Permutation)는 초기 자리바꿈으로 모든 비트 위치가 바뀌게 되며, IP를 통해 바뀐 출력을 IP^{-1}에 적용하면 원래 입력을 얻게 된다.

이들은 암호화와 복호화 과정에 서로 쌍으로 존재하기 때문에 서로 상쇄된다. 그리고 DES의 정확성 만족을 위해 암호화에 있는 $J_i^{K_i}$와 복호화에 있는 $J_{16-i+1}^{K_i}$은 서로 상쇄되는 특성이 있다.

DES의 라운드 함수는 64bit 입력을 두 개의 32bit로 나누어 다음 식과 같이 처리한다.

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$

두 개의 32bit 중 하나는 그대로 반대쪽으로 내려오고, 다른 하나는 라운드 키와 함께 f 함수를 통해 복합적으로 암호화한다. f 함수는 총 4개의 세부과정으로 구성되어 있다. 첫째는 확장 자리바꿈으로 32bit 입력을 48bit로 확장한다. 이때 일부 비트를 중복하여 확장한다. 이것은 입력의 각 비트가 선형적으로 영향을 주지 않고 고루 영향을 주도록 해주는 효과가 있다. 이것을 눈사태 효과(avalanche effect)라 한다. 실제 1bit만 다른 입력을 암호화하면 평균 반 이상이 다른 값으로 암호화된다. 둘째, 확장된 48bit와 라운드 키가 XOR된다. 따라서 DES의 라운드 키는 48bit이고, 총 16개가 필요하기 때문에 원래 7byte 키가 96byte로 확장되어야 한다.

셋째, 6bit를 4bit로 교체하여 주는 8개의 S-박스를 통해 다시 32bit로 바꾼다. DES의 모든 과정은 역이 가능하지만, S-박스만 역이 가능하지 않다. 6비트를 4비트로 변경하게 되므로 특정 결과 값을 줄 수 있는 경우의 수가 4개가 존재한다. 따라서 결괏값을 통해서는 입력을 예측할 수 없다. 마지막으로 32bit 자리바꿈을 한 번 더 하게 된다.

2.3 3중 DES

DES는 표준화된 이후 허점을 찾기 위한 많은 연구가 진행되었음에도 불구하고 키 길이가 짧은 문제를 제외하고는 심각한 문제가 발견되지 않았다. RSA사에서는 DES의 짧은 키 길이의 문제점을 알리기 위해 DES-Challenge라는 대회를 3차례 개최하였다. 1999년에 개최된 마지막 대회에서 22시간 15분만에 전사 공격에 성공을 하였다. 따라서 90년대 후반부터는 DES가 안전하지 않다는 것이 널리 인지된 사실이었으며, 이 때문에 이 시기에는 DES 대신에 다른 대칭 암호알고리즘들을 더 많이 사용하였다. 하지만 앞서 언급한 바와 같이 전사 공격에 의한 취약성을 제외하고는 다른 문제는 없으므로 DES를 활용할 방안도 연구가 되었으며, 그 결과로 등장한 것 중 하나가 3중 DES이다.

3중 DES는 한 평문을 DES를 이용하여 3번 암호화하는 것이다. 암호화할 때마다 다른 키를 사용하면 키 길이가 총 168bit가 된다. 하지만 그렇게 하지 않고 다음과 같이 한다.

$$\begin{aligned} C &= E.K_1(D.K_2(E.K_1(M))) \\ M &= D.K_1(E.K_2(D.K_1(C))) \end{aligned}$$

이렇게 한 이유는 키 길이가 112bit만 되어도 당시 컴퓨팅 수준을 고려하였을 때 전사 공격에 충분히 안전하였으며, 모두 다른 키를 사용하더라도 중간 만남(meet-in-the-middle) 공격 때문에 어차피 안전성은 112bit밖에 되지 않기 때문이다. 중간 만남 공격 때문에 2중 DES를 사용하지 않고 3중 DES를 사용하는 것이다. 또 $K_1 = K_2$로 설정하면 3중 DES가 단일 DES와 같아지기 때문에 3번 연속 암호화하는 대신에 중간에 복호화 연산을 사용하고 있다.

참고로 다음이 성립하면

$$E.K_i(D.K_j(E.K_i(M))) = E.K_k(M)$$

3중 암호화를 무시하고 56bit 키만 찾으면 된다. 하지만 DES는 이것이 성립하지 않는다는 것이 증명되었기 때문에 3중 DES는 112bit 안전성을 가지고 있다[6]. 3중 DES의 유일한 문제는 암호화 과정을 3번 하므로 속도가 기존 DES보다 약 3배 증가하였다는 것이다.

3. AES

3.1 AES의 역사

DES는 앞서 살펴본 바와 같이 키 길이가 너무 짧아 현재 컴퓨팅 수준을 고려하였을 때 안전하지 못하다는 것과 소프트웨어보다는 하드웨어를 고려한 설계이었기 때문에 소프트웨어로 구현하였을 때 성능이 떨어지는 문제도 있었다. 이에 NIST는 1997년에 차세대 암호화 표준에 대한 제안 요청(call for proposal)을 시작하였다. 이 요청의 중요 요구사항은 키 길이는 128bit, 196bit, 256bit를 모두 지원할 수 있어야 한다는 것과 블록 크키는 128bit를 지원해야 한다는 것이었다. 접수된 것 중 15개(CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, Rijndael, SAFER+, Serpent, Twofish)가 1차로 채택되었다. 이 중 CRYPTON은 한국에서 제안된 것이고, 이들 중 Feistel 구조로 되어 있는 것도 많다. 1999년에 이들 15개 중 5개(MARS, RC6, Rijndael, Serpent, Twofish)를 최종후보로 선택하였다. MARS는 IBM이, RC6는 RSA의 개발자 중 한 명인 MIT의 Rivest 교수가, Serpernt는 영국 케임브리지 대학 Anderson 교수 등이, Rijndael는 벨기에 석사 학생들이, Twofish는 유명한 암호학자인 Schneier가 제안한 것이다. 이 중 MARS, RC6, Twofish는 Feistel 구조이다. NIST가 2001년에 최종적으로 선택한 것은 아이러니하게도 유명한 기관이나 학자가 제안한 것이 아닌 석사 학생들이 제안한 Rijndael이었다[7].

3.2 AES 암호화와 복호화 함수

표준번호 FIPS 197인 AES는 벨기에 암호학자인 Daemen과 Rijmen이 개발한 알고리즘이며, 블록 크기는 128bit, 192bit, 256bit를 지원하고, 키 길이는 128bit, 192bit, 256bit를 지원한다. 하지만 블록 크기가 128bit인 것만 표준으로 채택되었다. 128bit 기준으로 키 길이가 증가함에 따라 알고리즘은 10라운드, 12라운드, 14라운드로 구성된다. 이 알고리즘은 DES와 달리 Feistel 구조가 아니며, SPN(Substitution-Permutation Network) 구조로 되어 있다. 즉, 꼬임이 없이 치환과 자리바꿈을 하는 구조이다.

AES의 각 라운드는 S-박스를 위한 치환, 행 이동(자리바꿈), 열 섞음(교체), 라운드키 XOR로 구성된다. 블록 크기가 128bit인 경우에는 11개의 128bit 라운드키가 사용된다. 각 세부 연산들 중 기존 대칭 암호알고리즘과 달리 다항식 링이라는 수학적 개념을 사용하는 것도 있지만 그 자체가 이동과 XOR 연산으로 구현 가능한 것이므로 AES는 소프트웨어로 구현하기에도 매우 적합하다는 이점도 있다.

AES 알고리즘에 대한 여러 형태의 최적화를 사용하고 있다. 특히, 각 라운드에서 하는 일을 매번 새롭게 계산하지 않고 다양한 사전 계산을 하여 테이블에 저장하여 사용하는 방법으로 성능을 높이는 방법도 있다. Intel은 Intel Westmere부터 AES 암호화와 복호화 수행 성능을 높이기 위한 명령어 집합 AES-NI(New Instrution)를 지원하고 있다. 이 명령어 집합은 총 7개의 명령어로 구성되어 있다.

4. Salsa20

스트림 암호방식은 키를 이용하여 생성하는 의사난수 스트림과 평문을 XOR하여 암호화하는 방식이다. 스트림 암호방식은 one-time pad를 실용적으로 사용할 수 있도록 만든 암호 방식이다. one-time pad는 수동 공격에 대해 완벽한 안전성을 제공하는 암호알고리즘이지만 키 길이가 평문 길이보다 커야 하며, 매번 다른 키를 사용해야 하므로 현장에서 실제 사용할 수 없는 알고리즘이다. 오래전부터 스트림 암호방식이 제안되어 사용되었지만, 그것의 안전성에 대해서는 확신이 없었다. 하지만 관련 암호 이론의 발전과 유럽에서 안전하고 효율적인 스트림 암호방식을 찾기 위한 2004년에 시작한 eStream 프로젝트의 결과로 지금은 블록 암호 방식 못지않게 현장에서 스트림 암호방식을 사용하고 있다.

eStream 프로젝트에서는 소프트웨어에 적합한 스트림 암호 방식과 자원이 제한된 하드웨어에 적합한 스트림 암호 방식 두 분야에 대한 제안 요청을 받아 3단계에 걸쳐 분야마다 4개의 스트림 암호 방식을 최종 선정하였다. Salsa20은 Bernstein이 제안한 스트림 방식으로 eStream 프로젝트에서 소프트웨어에 적합한 최종 4개의 스트림 암호 방식 중 하나이며, 자원 제한 하드웨어 분야에서도 2단계까지 선발된 알고리즘이다[8]. 지금은 같은 저자가 발표한 Salsa20을 조금 변형한 ChaCha20이 TLS 등 현장에서 널리 사용하고 있다.

Salsa20은 add-rotate-xor 연산에 기반한 의사난수 함수를 이용한다. Salsa20에서 키 길이는 128bit 또는 256bit 이다. Salsa20이 내부적으로 사용하는 핵심 함수 F는 512bit 입력을 받아 512bit를 출력하여 주는 의사난수 조합(PRP, PseudoRandom Permutation)이며, 입력은 키 K, 64bit 난스 r, 64bit 카운터를 사용하여 구성된다. 즉, Salsa20은 다음과 같이 필요한 만큼의 키 스트림을 생성한다.

$$F(K,(r,0))||F(K,(r,1))||...$$

여기서 F 함수는 내부적으로 또 다른 함수를 사용하며, 이 함수는 키 K, 난스 r, 카운터 값 등으로 구성된 입력을 이 내부 함수에 반복적으로 적용하여 결과값을 계산한다.

참고문헌

[1] R. Housley, “Cryptographic Message Syntax (CMS),” IETF RFC 5652, Sept. 2009.

[2] David A. McGrew, John Viega, “The Security and Performance of the Galois/counter mode (GCM) of Operation,” Progress in Cryptology, INDOCRYPT 2004, LNCS 3348, Springer, Dec. 2004.

[3] Daniel J. Bernstein, “ChaCha, a Variant of Salsa20,” Workshop of SASC 2008, Feb. 2008.

[4] Daniel J. Bernstein, ”The Poly1305-AES Message-Authentication Code,” Fast Software Encryption 2005, LNCS 3557. pp. 32–49, Springer, 2005.

[5] FIPS, Data Encryption Standard (DES), Federal Information Processing FIPS 46-3, 1999.

[6] Keith W. Campbell, Michael J. Wiener, “DES is not a Group,” Advances in Cryptology, Crypto 1992, LNCS 740 pp. 512–520, Springer, 1993.

[7] Joan Daemen, Vincent Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard, Springer, 2002.

[8] Daniel J. Bernstein, “The Salsa20 Family of Stream Ciphers,” New Stream Cipher Designs: The eStream Finalists, LNCS 4986, Springer, 2008.

퀴즈

1. CTR 모드와 관련된 다음 설명 중 틀린 것은?

   ① CTR 모드는 다른 암호화 모드와 마찬가지로 암호화, 복호화 함수가 모두 필요하다.

   ② CTR 모드는 이전에 사용한 초기화 벡터를 사용하면 심각한 보안 문제가 발생한다.

   ③ 우연히 일부 카운터 값이 중첩되는 것을 막기 위해 카운터의 초깃값을 랜덤하게 생성하지 않고, 카운터 크기의 앞 부분만 랜덤하게 생성한다. 예를 들어 카운터의 크기가 128비트이면 96비트만 랜덤하게 생성히고, 나머지 32비트는 0인 값을 초깃값으로 사용한다.

   ④ 암호문 블록을 조작하면 해당 평문 블록에 원하는 변화를 줄 수 있으므로 CTR 모드는 NM 특성을 만족하지 못한다.

2. CBC 모드의 특성과 관련된 다음 설명 중 틀린 것은?

   ① 특정 암호문 블록을 조작하면 그다음 평문 블록에 원하는 변화를 줄 수 있으므로 CBC 모드는 NM 특성을 만족하지 못한다.

② 표준 채우기를 하면 암호문의 크기는 평문에 비해 최대 2블록이 커질 수 있다.

③ 마지막 암호문 블록은 마지막 평문 블록에만 영향을 받기 때문에 MAC 값으로 사용하기 어렵다.

④ 이전 암호문 블록을 피드백으로 사용하기 때문에 순차적으로 암호화할 수밖에 없어 다중 프로세서가 있더라도 이를 이용하여 암호화 속도를 향상할 수 없다.

3. 블록 크기가 8바이트인 대칭 암호알고리즘을 이용하여 CTR 모드로 한 블록보다 작은 서로 다른 메시지를 암호화한 결과가 각각 "01 02 03 04 05 06 07 08 33 22 11 05 21 AA F2 F6", "01 02 03 04 05 06 07 08 12 34 55 42 77 04 28 F2"이다. 채우기를 포함한 두 평문의 마지막 바이트를 XOR한 값은?

① 00

② 02

③ 01

④ 04

4. 블록 길이가 8바이트인 대칭 암호알고리즘을 이용하여 한 블록보다 작은 메시지를 CBC 모드로 암호화한 결과가 16진수로 "01 02 03 04 05 06 07 0B 01 09 0A 0B 0C 0D 0E 0F"이었다. IV의 마지막 바이트 0B를 공격자 09로 바꾸었다. 그런데 복호화하는 사용자는 채우기 측면에서 아무런 문제를 발견하지 못하였다. 복호화한 사용자는 평문의 크기를 무엇이라 생각하는가? 그리고 원래 평문의 크기는 얼마인가?

① 7바이트, 5바이트

② 6바이트, 7바이트

③ 7바이트, 6바이트

④ 6바이트, 7바이트

5. 블록 길이가 8바이트인 대칭 암호알고리즘을 이용하여 6바이트 메시지를 CBC 모드로 암호화한 결과가 16진수로 "01 02 03 04 05 06 07 08 01 09 0A 0B 0C 0D 0E 0F"이었다. 표준 채우기를 하였기 때문에 평문의 마지막 두 바이트의 값은 02 02이다. $D.K$(01 09 0A 0B 0C 0D 0E 0F) = $X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$라 하면 16진수로 X_1과 X_0의 값은?

① 0A, 05

② 05, 0A

③ 07, 08

④ 02, 02

연습문제

1. 블록 길이가 8바이트인 암호알고리즘을 이용하여 7바이트 메시지를 CBC 모드, PKCS #7로 암호화하였다고 가정하자. 이때 결과 암호문이 16진수로 "07 06 05 04 03 02 01 01 08 09 0A 0B 0C 0D 0E 0F"라고. 가정하자. 또한 암호화한 데이터는 모르지만 7바이트이므로 16진수 01이 padding되었을 것이다. 만약 공격자가 IV 중 마지막 두 바이트 "01 01"을 "00 02"로 바꾸었지만 수신자가 복호화하는 과정에서 문제점을 발견하지 못하였다. 그러면 원래 메시지의 7번째 바이트 값을 계산하시오.

2. 블록 길이가 8바이트인 암호알고리즘을 이용하여 ASCII 코드 문자열 "sendme$2"를 CBC 모드로 암호화하여 얻은 암호문을 16진수로 표현하면 "FF 09 28 30 8C 93 81 06 39 D0 81 5A 07 79 17 43"이다. 이 암호문을 받은 수신자가 복호화하였더니 결과가 "sendme$4"가 되었다. 수신자가 실제로 받은 암호문을 구하시오. (채우기는 없다고 가정)

3. 블록 길이가 4바이트인 암호알고리즘을 이용하여 16진수 "00 00 01 01"을 암호화한 결과가 "01 01 01 01 05 06 07 08"이라고 가정하자. 여기서 "01 01 01 01"은 IV이다. 그러면 16진수 "01 01 01 01"을 암호화한 유효한 암호문을 제시하라.

4. CBC나 CTR 모드는 IV 벡터를 사용한다. 이것의 장점과 단점을 설명하시오.

5. CBC 모드를 암호화할 때 어떤 IV를 사용할지 예측이 된다고 가정하자. 또한 특정 메시지를 보내 암호화를 요청할 수 있는 공격자가 있다고 가정하자. 이 경우 공격자는 암호문 C_0, C_1이 평문 M을 암호화한 것인지 확인할 수 있다. 그 방법을 설명하시오.

6. CTR 모드에서 항상 다른 IV를 사용하여야 한다. 같은 IV를 사용하면 평문 블록과 XOR하는 값들이 같아지는 문제점이 있다. 다른 IV를 사용하더라도 평문 블록과 XOR하는 값들이 중첩될 수 있으며, 중첩되면 여전히 두 평문 블록을 XOR한 값을 얻어낼 수 있다. 이 때문에 매번 IV를 랜덤하게 생성하는 대신에 IV의 일정 부분만 랜덤하게 생성하는 방식을 더 많이 사용한다. 예를 들어 블록 크기가 128비트이면 앞 96비트만 랜덤하게 생성하고 나머지 32비트는 0으로 채운다. 랜덤하게 128비트 생성하는 것과 이렇게 생성하는 것을 중첩 가능성 측면에서 비교하시오.

7. CBC 암호화 모드에서 평문의 블록 수보다 암호문의 블록 수가 최대 2개 증가할 수 있다. 그 이유를 설명하시오.

8. CBC와 CTR 모드를 비교하였을 때 CTR 모드의 장점을 나열하시오.

9. CBC 모드의 마지막 블록은 MAC 값으로 충분히 활용할 수 있지만 CTR 모드를 이용하여 암호화하였을 때 마지막 블록은 MAC 값으로 사용하기 힘들다. 그 이유를 간단히 설명하시오.

10. CBC 모드를 이용하면 최종 암호문 블록을 MAC 값으로 충분히 활용할 수 있다고 하지만 실제 마지막 블록을 그대로 MAC으로 활용하면 공격자가 쉽게 위조할 수 있다. 공격자는 이 방식의 MAC을 위조하기 위해 한 블록 크기의 메시지 M에 대한 CBC-MAC 값 t을 얻으면 공격자는 t가 $(M||t \oplus M)$에 대한 MAC임을 주장할 수 있다. 그 이유를 설명하시오. CBC-MAC 값을 계산할 때 IV는 모든 비트가 0인 블록을 사용한다.

11. 디스크 단위 암호화(디스크 전체를 하나의 키로 투명하게 암호화함)에서 고려해야 하는 특성을 나열하고, 디스크 단위 암호를 위해 ECB, CBC, CTR 모드를 사용하는 것의 문제점을 설명하시오.

12. 3중 DES는 두 개의 키를 사용하지만 내부적으로 3번의 DES를 수행한다. 3번 대신에 2번만 DES를 수행하는 2중 DES 방식을 사용하지 않고 3중 DES 방식을 사용한 이유는 2중 DES는 중간 만남 공격(meet-in-the-middle-attack)에 취약하기 때문이다. 2중 DES는 $E.K_2(E.K_1(M)) = C$ 형태이지만 평문-암호문 쌍 (M, C)를 가지고 있으면 $D.K_2(C) = E.K_1(M)$을 이용하여 2^{112}가 아니라 그것의 반 정도의 비용으로 K_1과 K_2를 찾을 수 있다. 그 방법과 그것의 비용을 제시하고, 같은 방법으로 3개의 키를 이용하는 3중 DES를 공격하는 비용을 제시하시오.