128 비트 블록 암호 LEA 및 운영모드 표준 해설서

128-bit Block Cipher LEA(Lightweight Encryption Algorithm) and its Modes of Operation

TTA 한국정보통신기술협회
Telecommunications Technology Association

128 비트 블록 암호 LEA 및 운영모드 표준 해설서

128-bit Block Cipher LEA and its Modes of Operation

발 간 한국정보통신기술협회(TTA)

저 자 박제홍(국가보안기술연구소, TTA 정보보호기반 프로젝트 그룹(PG501) 부의장)

도움주신분 김지홍(세명대학교, TTA 정보보호 기술위원회(TC5) 부의장)

남기효((주)유엠로직스)

128 비트 블록 암호 LEA 및 운영모드 표준 해설서

128-bit Block Cipher LEA and its Modes of Operation

목 차

1. 개요

스마트 그리드(Smart Grid), 사물인터넷(IoT, Internet of Things), 헬스케어(Healthcare) 등의 서비스가 점차 현실화되어감에 따라, 메모리 크기, 프로세서 성능, 소비 전력 등의 계산 자원이 제한된 특성을 가지는 기기들의 활용성이 점차 확대되고 있다. 특정 기능에 특화된 이러한 기기들은 상호 무선으로 연결되어 연동됨으로써 서비스의 다양한 요구사항을 충족시키고, 나아가 서비스 이용에 있어 사용자 편의성을 제공한다. 그러나 편리함을 더하기 위해 서비스 환경에서 유통되는 정보의 규모가 확대됨에 따라, 이들 정보의 노출에 의한 침해사고를 대응ㆍ예방할 수 있는 기술의 중요성이 점차 강조되고 있다.

정보의 보호를 위해서는 기본적으로 암호 알고리즘의 운용이 필수이다. 그러나 현재 사용 가능한 암호 알고리즘은 대다수가 PC나 서버와 같은 고사양ㆍ범용 환경에서의 운용을 고려하여 설계되었기 때문에, 계산 자원이 제약된 기기에 탑재하는 데 한계를 가진다. 실제 가장 많이 사용되고 있는 블록 암호 AES를 규격화시킨 미국 국립표준기술연구소(NIST, National Institute of Standards and Technology)에서도 이러한 한계를 인정하고 세계 시장에서의 암호 기술 주도권을 유지하기 위해, 새로운 환경에 적합한 경량 암호 알고리즘의 개발과 도입, 그리고 표준화 착수 계획을 밝힌 바 있다[20].

이러한 암호 기술의 새로운 개발 이슈에 대응하여 국가보안기술연구소는 미래창조과학부의 지원으로 2010년부터 3년 동안의 설계 기간을 거쳐 블록 암호 LEA(Lightweight Encryption Algorithm)를 개발하였다[33]. LEA는 하드웨어 환경을 주요 적용 대상으로 설계되던 기존의 경량 블록 암호[18]와 다르게, 자원이 제약된 기기에 탑재되는 프로세서의 특성을 고려하여 소프트웨어 구현 시 충분한 성능 제공이 가능하게 설계되었다.

LEA는 경량 환경에서 충분한 성능을 발휘하기 위해, AES를 비롯한 기존 대다수 블록 암호[19,16,18]의 설계에 사용된 비선형 치환 테이블(S-box, Substitution box) 기반 구조와는 차별화된 설계 기술인 ARX(Addition, Rotation, XOR) 구조를 채택하였다. ARX 구조는 프로세서가 제공하는 기본 연산을 그대로 사용할 수 있기 때문에 속도가 빠르고, 비선형 치환 테이블 저장에 필요한 메모리가 불필요하여 코드 크기나 구현 면적 대비 속도를 향상시킬 수 있는 장점을 가지고 있다. LEA는 이러한 ARX 구조의 장점을 유지하면서 주요 블록 암호 공격 방법에 대한 정량적인 안전성 분석이 가능할 뿐만 아니라[33,43], SIMD(Single Instruction Multiple Data) 구현과 같이 고성능 프로세서에서 제공하는 병렬 처리 인스트럭션의 활용 또한 용이하도록 설계되었다[33,41,40,42].

2012년 개발된 LEA는 국제 학회 WISA 2013을 통해 공개되었고 당해 TTA 단체 표준(TTAK.KO-12.0223, 이하 LEA 표준)으로 제정되었다[1]. 2015년 6월부터 한국 암호모듈 검증제도(KCMVP, Korea Cryptographic Module Validation Program)의 검증 대상 알고리즘으로 LEA가 포함되면서, LEA 표준은 LEA 구현을 위한 참조 규격으로 활용되고 있다. 또한 LEA를 기반 함수로 기밀성(Confidentiality)이

나 무결성(Integrity)과 같은 정보보호 서비스를 제공하는 방법을 정의한 TTA 단체 표준(TTAK.KO-12.0246, 이하 LEA 운영 모드 표준)이 2014년 제정되고[2] LEA 표준과 함께 KCMVP의 참조 규격으로 채택됨에 따라, 이들 LEA 관련 TTA 단체 표준의 활용 가치가 점차 높아지고 있다.
본 해설서는 블록 암호 LEA를 탑재한 암호 제품 개발을 위해 TTA 단체 표준[1,2]을 활용함에 있어 표준 사용자들의 이해를 돕기 위해 작성되었다. 따라서 상기 TTA 단체 표준들에 대한 상세 설명에 앞서 관련 일반 사항을 먼저 간략하게 정리한다. 그리고 블록 암호 LEA의 세부 함수를 정의한 LEA 표준의 주요 내용을 표준의 순서에 따라 설명한 후, LEA 운영 모드 표준의 내용을 설명한다. 특히 LEA 운영 모드 표준은 LEA를 기반으로 하여 기밀성과 인증을 제공할 수 있는 8종의 방식을 정의하고 있으므로, 각 방식이 가지는 특징과 요구조건을 가급적 상세하게 정리하여 개발하고자 하는 암호 제품의 기능에 적합한 운용 모드 방식의 선택에 도움이 되도록 한다.

참고로 TTA의 정보보호 기술위원회(TC5) 산하 정보보호기반 프로젝트그룹(PG501)에서는 TTA 단체 표준으로 제정된 블록 암호인 SEED, HIGHT, LEA에 대해 개별적으로 제정、관리되고 있던 블록 암호 운영 모드 표준을 하나의 표준으로 통합 관리하는 작업을 2015년부터 진행하고 있다. 이와 관련하여 일반 블록 암호에 적용 가능한 운영 모드 규격을 정의한 단체 표준[3]과 기반 블록 암호로 LEA를 적용할 경우 활용 가능한 참조 구현 값(Test vector)을 정리한 단체 표준[4]을 패밀리 표준으로 제정하였다. 본 해설서는 LEA 운영 모드 표준[2]뿐만 아니라 일반 블록 암호 운영 모드 표준 패밀리[3,4]의 참고 자료로도 활용 가능하다.

1.1 블록 암호

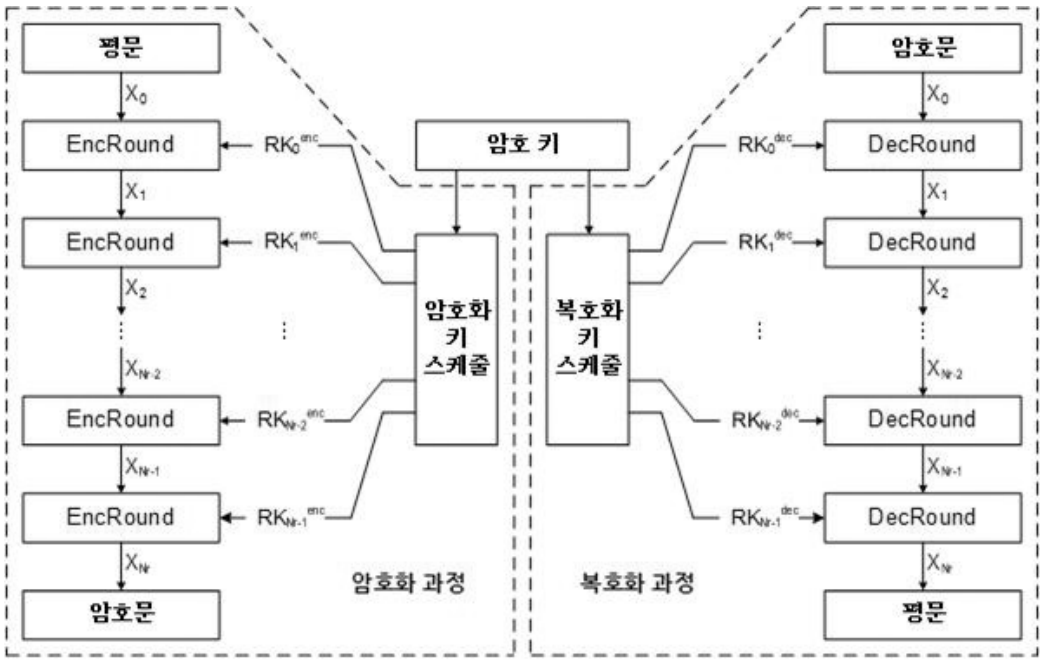
블록 암호는 고정된 짧은 길이를 가지는 데이터에 대한 암、복호화 연산을 처리하는 대칭키 암호(Symmetric key cryptography) 기술이다. 여기에서 대칭키 암호는 암호화(Encryption)와 복호화(Decryption)에 동일한 암호 키를 사용하는 암호 기술로, 블록 암호뿐만 아니라 스트림 암호(Stream cipher), 메시지 인증 코드(Message authentication code) 등의 암호 기술이 이에 해당한다. 대칭키 암호 기술의 안전성은 암호 키의 비밀성에 기반하므로, 일반적으로 '암호 키' 대신 '비밀 키'로 표기하기도 한다. 대칭키 암호 기술은 일반적으로 속도가 빠르기 때문에 데이터 보안을 위한 핵심 기술로 사용된다. 이는 인터넷 뱅킹, VoIP 통신과 같은 인터넷 기반 서비스의 보안을 위해 적용되는 SSL/TLS[9], IPsec[7,8], SRTP[6] 등의 암호 프로토콜이나, 패스워드 기반의 저장 데이터 보안을 위한 표준 규격[5] 등을 통해서 확인할 수 있다.

대다수 블록 암호는 고정된 암호 키에 대하여 라운드 키(Round key)를 생성하는 키 스케줄 함수(Key schedule function), 평문으로부터 암호문을 생성하는 암호화 함수(Encryption function), 그리고 암호문으로부터 평문을 복구하는 복호화 함수(Decryption function)로 구성되며, (그림 1-1)과 같은 구조로 동작한다.

블록 암호의 암、복호화 함수는 라운드 함수(Round function)라고 하는 특정 알고리즘을 주어진 라운드 수만큼 반복하는 구조로 되어 있다. 라운드 함수는 평문 또는 암호문으로부터 직전 라운드까지 변화되어온 결과인 내부 상태 값과 라운드 키를 입력으로 하여 새로운 내부 상태 값을 생성한다. 라운드 함수는 각 라운드마다 동일하게 동작하지만, 라운드마다 암호 키로부터 생성되는 개별 라운드 키가 반영되어 독립성이 부여된다.

대표적인 블록 암호 알고리즘으로는 미국 연방정부 표준인 AES(Advanced encryption standard)가 있다

[19]. AES는 90년대 중후반에 걸쳐 진행된 미국 연방정부 표준 블록 암호 공모사업을 통해 채택된 암호 알고리즘으로, 미국 연방정부에서 사용이 승인된 이후, SSL/TLS[9], IPsec/IKE[7,8,12], SRTP[6], CMS[10]와 같은 주요 암호 프로토콜에서 기본(default) 암호 알고리즘으로 채택되었을 뿐만 아니라 ISO 표준[16]에도 포함되어 현재 가장 광범위하게 사용되고 있다. 국내에서는 전자 상거래 시스템의 보안성 강화와 이용 활성화를 위해 자체 개발하여 적용한 블록 암호 SEED[16]와 국가ㆍ공공기관의 독자적인 정보보호 체계 구축을 위한 목적으로 개발한 블록 암호 ARIA[37,11]가 많이 사용되고 있다



(그림 1-1) 블록 암호 전체 구조

1.2 블록 암호와 정보보호 서비스

블록 암호는 대칭키 암호 기술을 이용한 정보보호 서비스 제공에 있어 기반 함수로 사용 가능하다. 블록 암호를 기반 함수로 특정 정보보호 서비스를 제공할 수 있는 방법을 정의한 것을 블록 암호 운영 모드(Block cipher mode of operation)라고 하며, 특정 블록 암호 운영 모드 방식을 '모드'라고 부른다. 본 소절에서는 블록 암호 운영 모드의 설명에 앞서 블록 암호를 이용하여 제공 가능한 정보보호 서비스를 먼저 소개한다. 이러한 정보보호 서비스는 사용되는 암호 키의 비밀성을 기반으로 한다.

1.2.1 기밀성

기밀성은 블록 암호 운영 모드에 대해 요구되는 가장 기본적인 안전성 요구조건으로, 비 인가된 개체에 의한 정보의 노출을 방지하는 것이다. 여기에서 '인가'는 암호 키의 소유 여부에 의해 결정되며, 기밀성의 보장 범위는 전송 또는 저장 데이터를 포함한다.

기밀성 보장을 위한 암호화 기술은 크게 공개키 암호화(Public key encryption)와 대칭키 암호화(Symmetric key encryption) 방식으로 구분할 수 있다. 공개키 암호화 방식은 각 사용자가 비밀로 관리하는 개인키와 사용자에게 공개하는 공개키로 이루어진 암호 키 쌍을 가지는 공개키 체계에 한정하여 적용할 수 있다. 주요 공개키 암호화 방식으로는 RSA-OAEP와 ECIES를 들 수 있다[15]. 일반적으로 공개키 암호 연산은 효율성이 떨어지기 때문에 암호 키 전송과 같이 길이가 짧은 데이터를 간헐적으로 암호화하는 경우에 사용한다. 반면 대칭키 암호화 방식은 송、수신자만이 비밀리에 공유하여 관리하는 암호 키를 이용하며, 효율성이 우수하여 대칭키 체계는 물론 공개키 체계에서도 일반적인 데이터 처리(암、복호화)에 사용된다.

대칭키 암호화 방식은 블록 암호를 이용하는 것과 스트림 암호를 이용하는 것으로 다시 구분할 수 있다. 블록 암호는 특정 길이의 데이터에 대한 암、복호화 처리만 정의하며, 운영 모드와 결합하여 가변 길이의 데이터에 대한 처리가 가능하다. 반면 스트림 암호는 별도의 운영 모드 없이 가변 데이터에 대한 암、복호화가 가능하다. 스트림 암호는 매우 단순한 구조로 동작하기 때문에 블록 암호에 비해 상대적으로 작은 면적(하드웨어)에 구현 가능하며 고속(소프트웨어)으로 동작한다. 그러나 스트림 암호의 안전성과 관련한 이론적인 연구 성숙도가 블록 암호에 비해 상대적으로 미흡하고, 기능 측면의 확장성 또한 제한적이다. 그리고 블록 암호에 대한 지속적인 경량화 및 고속화 설계、구현 기술 연구로 인해 스트림 암호가 가지는 상대적인 강점이 점차 희석되고 있는 실정이다. 따라서 현재는 블록 암호가 스트림 암호보다 많이 사용되고 있다.

1.2.2 무결성

무결성은 비 인가된(unauthorized) 방법이나 우연적으로(accidental) 발생하는 정보의 변조를 방지하는 것이다. 여기에서 변조는 데이터의 삽입, 삭제, 수정을 포함한다. 무결성을 확인하기 위한 기본적인 구조는 데이터에 대한 확인 값(Checking value)을 붙이는 것이다. 통신 과정에서 송신자에 의해 생성된 확인 값은 데이터와 함께 수신자에게 전송된다. 이를 전달받은 수신자는 송신자와 동의한 절차에 따라 자체 생성한 확인 값과 수신된 값이 일치하는지 여부를 확인함으로써 수신된 데이터가 올바른지 여부를 검증한다. 여기에서 송、수신자가 동의한 절차로 사용될 수 있는 암호 기술로 메시지 인증 코드나 전자 서명(Digital signature) 방식이 있다.

전자 서명 방식은 무결성뿐만 아니라 부인 방지를 보장할 수 있는 유일한 암호 기술이지만, 공개키 체계에 한정하여 적용할 수 있고 연산 효율성이 떨어지기 때문에 실시간 처리가 중요한 환경에서는 사용하기가 쉽지 않다. 반면 메시지 인증 코드 방식은 효율성이 우수하여 키 체계에 관계없이 일반적인 데이터 처리(확인 값 생성 및 검증)에 사용된다.

1.2.3 인증

인증의 개념은 크게 데이터 근원 인증(Data-origin authentication)과 개체 인증(Entity authentication)으로 구분할 수 있다. 데이터 근원 인증은 수신된 데이터의 출처를 보증하는 개념이라면, 개체 인증은 한 개체가 통신 과정에 참여하고 있는 상대방의 신원을 실시간으로 확인하는 개념으로 볼 수 있다.

데이터 근원 인증은 메시지 인증(Message authentication)이라고도 하며 소절 1.2.2에서 정리한 무결성과 밀접한 연관성을 갖는다. 무결성을 보장하기 위해서는 암호 키가 사용되어야 한다. 따라서 무결성 검증을 통해 수신자는 확인 값을 생성한 데이터의 출처에 대한 정보를 알 수 있게 된다. 그러나 무결성과 구분

해야 할 점은 데이터 근원 인증이 통신(communication) 상황을 내포하고 있다는 것이다. 무결성은 통신의 개념과 관계없이 성립하지만, 데이터 근원 인증은 통신 과정에서 수신자가 송신자를 확인하는 과정인 차이점을 가진다.

개체 인증은 한 개체가 프로토콜에 참여한 다른 개체의 신원을 확인하는 것과 함께, 실제로 프로토콜에 참여하고 있는지 여부(Aliveness)를 확인하는 절차이다. 암호 프로토콜에서는 신원 확인 대상이 되는 개체의 신원 정보 자체를 프로토콜 메시지로 사용하는 경우가 많다. 이때 증명 대상 개체의 신원과 실시간 참여에 대한 확인을 위해 데이터 근원 인증 방법을 적용하는 것이 일반적이다. 참고로 개체 인증만을 위한 방법으로는 패스워드나 질의-응답 구조의 프로토콜 등이 있다.

1.3 운영 모드

소절 1.1에서 소개한 바와 같이 블록 암호는 고정된 블록 길이의 데이터를 암、복호화한다. 여기에서 블록의 길이는 보통 128 비트이며 경량 환경을 대상으로 설계된 경우 구현 면적을 고려하여 이보다 작은 길이를 블록 단위로 설정한다. 그러나 암호화의 대상이 되는 데이터의 길이는 최소 수십 바이트에서 메가 또는 기가 바이트까지 유동적이기 때문에 블록 암호의 단위 블록을 가볍게 넘어간다.

따라서 가변 길이 데이터에 대한 암、복호화를 위한 방법을 블록 암호와는 별도로 정의할 필요가 있었으며, 이에 따라 등장한 것이 블록 암호 운영 모드이다. 가장 단순하고 직관적인 운영 모드로는 데이터를 블록 단위로 분할한 후 순차적으로 블록 암호의 암호화 함수에 입력하여 생성되는 암호문 (블록)을 동일한 순서로 연접시키는 방법이 있다. 이러한 방식을 ECB(Electronic code book) 모드라 하며 (그림 3-1)과 같이 동작한다. 그러나 블록 암호는 암호 키가 고정되어 있는 경우 동일한 평문 블록에 대해 암호화를 반복 수행하더라도 항상 동일한 암호문 블록을 생성하는 특성을 가지고 있다. 따라서 이미지 데이터와 같이 평문이 시각적인 특징을 가질 경우 암호문에서 평문과 유사한 패턴이 드러나게 된다.[1)]

상기 문제점은 블록 암호 자체가 안전하다고 하더라도 운용하는 방식에 따라서 적절한 정보보호 서비스를 제공하지 못할 수 있다는 것을 보여준다[39,38,44]. 따라서 블록 암호를 이용하여 가변 길이 데이터를 안전하게 보호하기 위한 방법은 생각보다 복잡하게 정의되고 있다. 또한 기밀성뿐만 아니라 다른 정보보호 서비스를 제공할 수 있는 방식으로도 정의가 가능하며, 이를 기준으로 기밀성 전용(Confidentiality only), 메시지 인증 전용(Message authentication only), 그리고 인증 암호화(Authenticated encryption)로 세분화할 수 있다. 아래에서는 기술의 편의를 위해 기밀성 전용 운영 모드를 '기밀성 운영 모드', 메시지 인증 전용 운영 모드를 '인증 운영 모드'로 표기한다.
기밀성 운영 모드는 암호화를 통해 데이터에 대한 노출을 방지할 수 있도록 설계되었으며, 초기 값(IV, Initialization Vector)을 사용하여 동일한 평문에 대해 암호화를 반복 수행할 때 마다 서로 다른 암호문을 생성할 수 있다. 인증 운영 모드는 블록 암호를 이용하여 암호화 방식이 아닌 메시지 인증 코드 방식을 정의한 것으로, 무결성과 메시지 인증을 제공한다.
기밀성 운영 모드는 데이터의 노출은 방지할 수 있지만 변조를 막을 수는 없다. 이를 보완하기 위해서는 별도의 메시지 인증 코드 방식을 사용해야 한다. 그러나 이러한 경우 메시지 인증 코드 방식에 사용할 암호

1)ECB로 암호화된 이미지 파일의 예로 'https://blog.filippo.io/the-ecb-benguim/' 를 참조하기 바란다.

키를 별도로 생성해야 한다. 이에 대응하여 하나의 암호 키로 메시지에 대한 기밀성과 인증을 동시에 보장할 수 있는 방식이 인증 암호화 운영 모드이다.

운영 모드는 적용 환경을 고려해서 선택해야 한다. LEA 운영 모드 표준에서는 다양한 적용 환경에서 사용 가능한 8종의 운영 모드를 정의하고 있다. 이들은 5종의 기밀성 운영 모드, 1종의 인증 운영 모드, 그리고 2종의 인증 암호화 운영 모드로 구성되어 있으며, KCMVP의 검증 대상 암호 기술로도 포함되어 있다. 운영 모드의 적용과 관련하여 고려해야 하는 요소를 정리하면 [표 1-1]과 같다. 각 열의 의미는 아래에 기술하며, 개별 운영 모드와 관련한 상세 설명은 3,4,5절의 해당 부분에 기술한다.

[표 1-1] 운영 모드별 주요 특징

운영 모드	오류 전파	순방향성	패딩	초기 값 조건	병렬 처리	사전 계산
ECB	1 블록	N	필요	–	Y	N
CBC	2 블록	N	필요	난수	암호화: N 복호화: Y	N
CFB	2 블록	Y	불필요	난수	암호화: N 복호화: Y	N
OFB	1 블록	Y	불필요	난수	N	Y
CTR	1 블록	Y	불필요	Nonce	Y	Y
CCM	–	Y	불필요	Nonce	N	N
GCM	–	Y	불필요	Nonce	Y	Y
CMAC	–	Y	불필요	–	N	N

- 오류 전파(Error propagation) 항목은 암호문의 1개 블록에서 발생한 오류가 복호화 과정에서 평문에 얼마나 영향을 주는지를 나타낸 것이다.
- 순방향성 항목은 복호화 시 LEA 복호화 함수의 사용 여부를 나타낸 것이다. 'N'은 LEA 복호화 함수를 사용하는 것을 의미하고 'Y'는 LEA 암호화 함수를 사용하는 것을 의미한다. 암호화 시에는 LEA의 암호화 함수만 사용하므로, 순방향성을 충족시킬 경우 하드웨어 구현 시 면적을 줄일 수 있는 장점을 가진다.
- 패딩 항목은 데이터가 LEA의 단위 블록 길이인 128 비트 단위로 처리되어야 하는지 여부를 나타낸 것이다.
- 초기 값 조건 항목은 운영 모드가 보장 가능한 안전성을 위해 초기 값(IV)이 가져야 하는 특성의 의미한다. '난수'는 초기 값이 난수 특성을 가져야 함을 의미하며, 'Nonce'는 난수 특성을 가질 필요 없이 한번 사용된 값이 재사용되지 않아야 함을 의미한다.
- 병렬 처리 항목은 가속화를 위해 LEA 암ㆍ복호화 함수를 병렬로 배치하여 운용 가능한지 여부를 정리한 것이다. 'Y'는 가능, 'N'은 불가능을 의미한다.
- 사전 계산 항목은 평문이 입력되기 전에 계산을 일부 수행하여 암ㆍ복호화 과정의 속도를 향상시킬 수 있는 지 여부를 나타낸 것이다. 'Y'는 가능, 'N'은 불가능을 의미한다.

1.4 기호와 표기

1.4.1 기호

BitToInt(x)	32 비트 열 $x = x_{31}\|x_{30}\|\cdots\|x_0$ 입력에 대하여 정수 $n=x_0 \cdot 2^0+x_1 \cdot 2^1+\cdots+x_{31} \cdot 2^{31}$을 출력하는 함수
C	암호문
C_i	암호문(C)의 i번째 블록
D_K	암호 키 K를 적용한 LEA 복호화 함수: $D_K(C):=LEA.Decrypt(C,LEA.DecKeySchedule_{\|K\|}(K))$
E_K	암호 키 K를 적용한 LEA 암호화 함수: $E_K(P):=LEA.Encrypt(P,LEA.EncKeySchedule_{\|K\|}(K))$
IntToBit(n)	정수 $n=x_0 \cdot 2^0+x_1 \cdot 2^1+\cdots+x_{31} \cdot 2^{31} \in [0,2^{32})$ $(xi \in \{0,1\}, 0 \le i \le 31)$을 입력받아 32 비트 열 $x = x_{31}\|x_{30}\|\cdots\|x_0$을 출력하는 함수
IV	운영 모드 초기 값(Initialization Vector)
K	암호 키
k	암호 키(K)의 비트 길이 (k=|K|). LEA의 경우 k = 128, 192, 또는 256으로 설정
LEA.Decrypt	LEA의 복호화 함수
LEA.Encrypt	LEA의 암호화 함수
LEA.DecKeyScheduleⅰk	암호 키 길이 k에 따른 LEA의 복호화 키 스케줄 함수
LEA.EncKeyScheduleⅰk	암호 키 길이 k에 따른 LEA의 암호화 키 스케줄 함수
LEA.EncRound	LEA의 암호화 라운드 함수
LEA.DecRound	LEA의 복호화 라운드 함수
$LSB_i(x)$	주어진 비트 열 x의 하위(오른쪽) i개 비트로 이루어진 비트 열
M	메시지 인증 코드의 입력 데이터
M_i	메시지 인증 코드 입력 데이터의 i번째 블록
$MSB_i(x)$	주어진 비트 열 x의 상위(왼쪽) i개 비트로 이루어진 비트 열
N	Nonce 특성을 가지는 인증 암호화 운영 모드의 초기 값
Nb	LEA 암ㆍ복호화 함수 입력 데이터 블록의 바이트 길이로 16으로 고정
Nk	LEA에 적용 가능한 암호 키의 바이트 길이로 $Nk \in \{16,24,32\}$
Nr	LEA의 라운드 수로 Nk에 따라 결정됨. $Nr \in \{24, 28, 32\}$
P	암호화를 위한 입력 평문
P_i	평문의 i번째 블록
RK	라운드 키
$ROL_i(x)$	32 비트 열 x의 i비트 좌측 순환이동
$ROR_i(x)$	32 비트 열 x의 i비트 우측 순환이동
RK_i^{enc}	암호화 과정의 i번째 라운드 함수에 사용되는 192 비트 라운드 키
RK_i^{dec}	복호화 과정의 i번째 라운드 함수에 사용되는 192 비트 라운드 키

T	인증 값(Authentication tag)
Tlen	인증 값 T의 비트 길이
U→V	비트 열 U를 V로 표현
U←V	비트 열 V를 U로 표현. 또는 V가 함수일 경우, V의 출력 값을 U에 대입
x mod y	정수 x와 정수 y(◊0)에 대하여 x를 y로 나눈 나머지. z=x mod y이면 z는 다음을 만족하는 유일한 정수 - $0 \le z < y$ - (x-z)는 y의 배수
$x \| y$	두 비트 열 x와 y의 연접
$x \oplus y$	같은 길이를 갖는 두 비트 열 x와 y의 XOR(eXclusive OR)
$x \boxplus y$	두 32 비트 열 x와 y에 대해 다음과 같이 32 비트 열 z를 계산하는 연산z ←IntToBit((BitToInt(x)+BitToInt(y)) mod 2^{32})
$x \boxminus y$	두 32 비트 열 x와 y에 대해 다음과 같이 32 비트 열 z를 계산하는 연산z ←IntToBit((BitToInt(x)-BitToInt(y)) mod 2^{32})
$x \ll r$	비트 열 x를 왼쪽으로 r비트만큼 이동시킨 결과 값 $LSB_{\|x\|}(x \| 0^r)$
$x \gg r$	비트 열 x를 오른쪽으로 r비트만큼 이동시킨 결과 값 $MSB_{\|x\|}(0^r \| x)$
0^s	비트 0을 s개 연접한 비트 열
1^s	비트 1을 s개 연접한 비트 열
|x|	주어진 비트 열 x의 비트 길이
|x|	주어진 수 x보다 크거나 같은 정수 중에서 최소 값
$[x]_s$	주어진 음이 아닌 정수 x의 s 비트를 사용한 이진수 표현

1.4.2 표기

1.4.2.1 알고리즘 변수 표기

블록 암호 LEA의 평문과 암호문은 각각 128 비트 길이의 비트 열이며, 암호 키는 128, 192, 또는 256 비트 길이의 비트 열이다. LEA의 암호 키, 평문, 암호문은 바이트 배열로 표기되며, 암호화, 복호화 및 키 스케줄 과정에 사용되는 내부 상태 변수 및 라운드 키는 워드 배열로 표기된다.

1.4.2.2 입력 비트 열의 바이트 배열 변환

LEA 규격 표준에서 바이트는 8 비트 길이의 비트 열을 의미한다. 길이 8n의 입력 비트 열 $input_0$, $input_1$, $input_2$,…, $input_{8n-1}$로부터 바이트 배열 A[0],A[1],A[2],…,A[n-1]은 다음과 같이 주어지며, 순서는 [표 1-2]와 같다.

$$A[i] := input_{8i} \| input_{8i+1} \| \cdots \| input_{8i+7} \ (0 \le i \le (n-1)).$$

[표 1-2] 입력 비트 수열과 바이트 배열의 관계

입력 비트 열	0	1	…	7	8	9	…	15	…	8n-8	8n-7	…	8n-1
바이트 배열	0				1				…	(n-1)			

1.4.2.3 바이트 배열과 워드 배열 간의 변환

LEA 규격 표준에서 워드 x는 32 비트 길이의 비트 열 $x_{31}\|x_{30}\|\cdots\|x_0$을 의미하며, 이것은 함수 BitToInt에 의해 0과 $(2^{32}-1)$ 사이의 어떤 정수에 대응된다. 반대로, 0과 $(2^{32}-1)$ 사이의 정수 n은 함수 IntToBit에 의해 어떤 워드에 대응된다. LEA 규격 표준에서 바이트 또는 워드 값은 16진법을 이용하여 표기되는데, 이 표기는 바이트 또는 워드의 비트들을 4 비트씩 묶고 그 4 비트를 [표 1-3]과 같이 단일 문자에 대응시켜 표기한다.

[표 1-3] 비트 패턴의 16진수 표현

비트 패턴	문자	비트 패턴	문자	비트 패턴	문자	비트 패턴	문자
0000	0	0100	4	1000	8	1100	c
0001	1	0101	5	1001	9	1101	d
0010	2	0110	6	1010	a	1110	e
0011	3	0111	7	1011	b	1111	f

소절 1.4.1에 소개된 연산 'B ← A'를 구체적으로 다음과 같이 정의한다.

① A와 B가 같은 형의 비트, 바이트, 또는 워드인 경우: B는 A와 동일한 비트 열을 가짐

② A가 바이트 배열이고 B가 워드인 경우: B에 A=(A[0],A[1],A[2],A[3])의 비트 열들이

$$B=A[3]\|A[2]\|A[1]\|A[0]$$

과 같이 대입됨

③ A와 B가 같은 형의 배열인 경우: B=(B[0],B[1],…,B[n−1])에 A=(A[0],A[1],…,A[n−1])의 비트 열들이

$$B[i]=A[i]\ \ (0\le i\le(n-1))$$

을 만족하도록 대입됨

④ A가 바이트 배열이고 B가 워드 배열인 경우: B=(B[0],B[1],…,B[n−1])에 A=(A[0],A[1],…,A[4n−1])의 비트 열들이

$$B[i]=A[4i+3]\|A[4i+2]\|A[4i+1]\|A[4i]\ \ (0\le i\le(n-1))$$

을 만족하도록 대입됨

⑤ A가 워드 배열이고 B가 바이트 배열인 경우: B=(B[0],B[1],…,B[4n−1])에 A=(A[0],A[1],…,A[n−1])의 비트 열들이

$$A[i]=B[4i+3]\|B[4i+2]\|B[4i+1]\|B[4i]\ \ (0\le i\le(n-1))$$

을 만족하도록 대입됨

바이트 배열, 워드 배열 및 워드 내 비트 색인의 관계를 정리하면 [표 1-4]와 같다.

[표 1-4] 바이트 배열, 워드 배열, 워드 내 비트 색인의 관계

바이트 배열	3	2	1	0	7	6	5	4	…	4n-1	4n-2	4n-3	4n-4
워드 배열	0				1				…	(n-1)			
워드 내 비트 색인	31 … 0				31 … 0				…	31 … 0			

바이트 배열과 워드 배열의 변환 규약의 적용 예는 다음과 같다.

바이트 배열	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
워드 배열	13121110					17161514				1b1a1918				1f1e1d1c		

2. 128 비트 블록 암호 LEA

LEA 규격 표준[1]은 블록 암호 LEA의 규격과 암ㆍ복호화 함수, 그리고 키 스케줄 함수를 정의하고 있다. 좀 더 상세하게는 암호화 라운드 과정을 구성하는 암호화 함수와 암호화 키 스케줄 함수, 그리고 복호화 과정을 구성하는 복호화 함수와 복호화 키 스케줄 함수를 정의하고 있다.

LEA는 128 비트 데이터 블록을 암호화하도록 설계된 블록 암호 알고리즘으로, 요구되는 안전성 기준에 따라 128, 192, 또는 256 비트 길이의 암호 키를 사용할 수 있다. LEA의 라운드 함수는 32 비트 단위의 Addition, Rotation, XOR 연산만으로 구성되어 있어, 이들 연산을 지원하는 32 비트 소프트웨어 플랫폼에서 고속으로 동작한다. 또한 라운드 함수 내부의 ARX 연산 배치는 충분한 안전성을 보장하는 것과 동시에 비선형 치환 테이블 사용을 배제하여 경량 구현이 가능하도록 하였다[33].

본 절에서는 LEA를 지원하는 암호 키 길이에 따라 각각 LEA-128, LEA-192, LEA-256으로 구분하여 표기한다. 암ㆍ복호화 과정에서 라운드 함수는 암호 키 길이에 관계없이 동일한 구조를 가지지만, 암호 키 길이에 따라 라운드 함수의 반복 횟수와 키 스케줄 함수가 독립적으로 정의된다.
블록 암호 LEA의 규격은 [표 2-1]과 같이 정리할 수 있다. [표 2-1]에서 블록 길이를 Nb 바이트, 암호 키 길이를 Nk 바이트, 라운드 수를 Nr로 나타내었다.

[표 2-] LEA 규격

구분	Nb	Nk	Nr
LEA-128	16	16	24
LEA-192		24	28
LEA-256		32	32

2.1 암호화

블록 암호 LEA의 암호화 과정은 다음의 두 가지 함수로 구성된다.

- 비트 길이가 k(=8×Nk)인 암호 키 K로부터 Nr개의 192 비트 암호화용 라운드 키 $RK_i^{enc}(0 \le i \le (Nr-1))$를 생성하는 암호화 키 스케줄 함수 $LEA.EncKeySchedule_k$
- 라운드 함수 LEA.EncRound와 암호화용 라운드 키 $RK_i^{enc}(0 \le i \le (Nr-1))$를 이용하여 128 비트 평문 P를 128 비트 암호문 C로 변환하는 암호화 함수 LEA.Encrypt

2.1.1 암호화 함수(LEA.Encrypt)

블록 암호 LEA의 암호화 함수 LEA.Encrypt는 비트 길이가 k인 암호 키 K에 대해 암호화 키 스케줄 함수

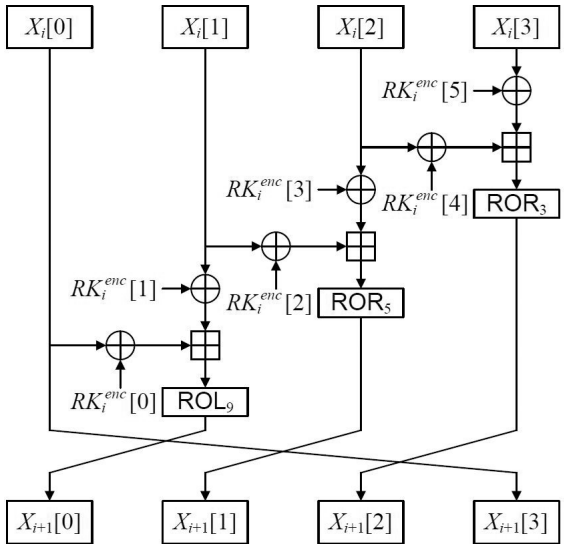
LEA.EncKeySchedule$_k$로부터 생성된 Nr개의 192 비트 라운드 키 RK_i^{enc} ($0 \le i \le (Nr-1)$)와 128 비트 평문 P를 입력받아 알고리즘 1을 수행하여 128 비트 암호문 C를 출력한다.

알고리즘 1 암호화 함수 : $C \leftarrow LEA.Encrypt(P,(RK_0^{enc},RK_1^{enc},\cdots,RK_{Nr-1}^{enc}))$

입력 : 128 비트 평문 P, Nr개의 192 비트 라운드 키 $RK_0^{enc},RK_1^{enc},\cdots,RK_{Nr-1}^{enc}$
출력 : 128 비트 암호문 C

1: $X_0 \leftarrow P$
2: for i = 0 to (Nr-1) do
3: $X_{i+1} \leftarrow LEA.EncRound(X_i,RK_i^{enc})$
4: end for
5: $C \leftarrow X_{Nr}$

알고리즘 1에서 $i(0 \le i \le (Nr-1))$번째 라운드에 동작하는 라운드 함수 LEA.EncRound는 128 비트 내부 상태 값 $X_i=(X_i[0],X_i[1],X_i[2],X_i[3])$와 192 비트 라운드 키 $RK_i^{enc} = (RK_i^{enc}[0], RK_i^{enc}[1],\cdots,RK_i^{enc}[5])$로부터 알고리즘 2를 실행하여 새로운 128 비트 내부 상태 값 $X_{i+1}=(X_{i+1}[0],X_{i+1}[1],X_{i+1}[2],X_{i+1}[3])$을 생성한다. (그림 2-1)은 암호화 과정의 i번째 라운드 함수를 도시한 것이다.



(그림 2-1) 암호화 과정의 i번째 라운드 함수($0 \le i \le (Nr-1)$)

알고리즘 2 암호화 과정의 i번째 라운드 함수 : $X_{i+1} \leftarrow LEA.EncRound(X_i,RK_i^{enc})$

입력 : 128 비트 내부 상태 값 X_i, 192 비트 라운드 키 RK_i^{enc}
출력 : 128 비트 내부 상태 값 X_{i+1}

1: $X_{i+1}[0] \leftarrow ROL_9((X_i[0] \oplus RK_i^{enc}[0]) \boxplus (X_i[1] \oplus RK_i^{enc}[1]))$
2: $X_{i+1}[1] \leftarrow ROR_5((X_i[1] \oplus RK_i^{enc}[2]) \boxplus (X_i[2] \oplus RK_i^{enc}[3]))$
3: $X_{i+1}[2] \leftarrow ROR_3((X_i[2] \oplus RK_i^{enc}[4]) \boxplus (X_i[3] \oplus RK_i^{enc}[5]))$
4: $X_{i+1}[3] \leftarrow X_i[0]$

2.1.2 암호화 키 스케줄 함수(LEA.EncKeySchedule)

비트 길이가 k(=8×Nk)인 암호 키 K로부터 암호화 과정에 필요한 Nr개의 192 비트 라운드 키 $RK_0^{enc}, RK_1^{enc}, \cdots, RK_{Nr-1}^{enc}$를 생성하는 암호화 키 스케줄 과정을 설명한다. 암·복호화 과정에서 라운드 함수는 암호 키 길이에 관계없이 동일한 구조를 가지므로, 각 라운드 키 $RK_i^{enc}(0 \le i \le (Nr-1))$의 길이는 192 비트로 동일하다. 그러나 암호 키 길이에 따른 라운드 함수의 반복 횟수(Nr)를 고려한 라운드 키 생성이 필요하다. 본 소절의 알고리즘 3,4,5는 각각 LEA-128, LEA-192, LEA-256에서 사용하는 암호화 키 스케줄 함수이다.

키 스케줄 함수에서 사용되는 32 비트 상수들 $\delta[i](0 \le i \le 7)$는 [표 2-2]와 같다.

[표 2-2] 키 스케줄 상수

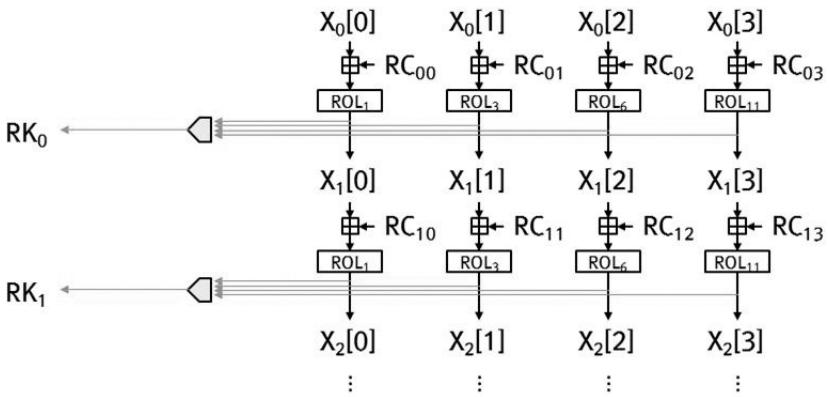
δ[0] = c3efe9db,	δ[1] = 44626b02,	δ[2] = 79e27c8a,	δ[3] = 78df30ec,
δ[4] = 715ea49e,	δ[5] = c785da0a,	δ[6] = e04ef22a,	δ[7] = e5c40957.

LEA-128의 암호화를 위해 사용되는 키 스케줄 함수 $LEA.EncKeySchedule_{128}$은 128 비트 암호 키 K로부터 24개의 192 비트 암호화 라운드 키 $RK_i^{enc}=(RK_i^{enc}[0], RK_i^{enc}[1], \cdots, RK_i^{enc}[5])$ $(0 \le i \le 23)$를 알고리즘 3과 같이 생성하며, 이 과정에서 128 비트 내부 상태 변수 X=(X[0],X[1],X[2],X[3])가 사용된다. (그림 2-2)는 LEA-128의 암호화 라운드 키 생성 과정을 도시한 것이다.

알고리즘 3 LEA-128 암호화 키 스케줄 함수 : $(RK_0^{enc}, \cdots, RK_{23}^{enc}) \leftarrow LEA.EncKeySchedule_{128}(K)$

입력 : 128 비트 암호 키 K
출력 : 24개의 192 비트 암호화 라운드 키 RK_i^{enc} $(0 \le i \le 23)$

1: X ← K
2: for i = 0 to 23 do
3: X[0] ← $ROL_1(X[0] \boxplus RC_{i0})$ // $RC_{i0} = ROL_i(\delta[i \bmod 4])$
4: X[1] ← $ROL_3(X[1] \boxplus RC_{i1})$ // $RC_{i1} = ROL_{i+1}(\delta[i \bmod 4])$
5: X[2] ← $ROL_6(X[2] \boxplus RC_{i2})$ // $RC_{i2} = ROL_{i+2}(\delta[i \bmod 4])$
6: X[3] ← $ROL_{11}(X[3] \boxplus RC_{i3})$ // $RC_{i3} = ROL_{i+3}(\delta[i \bmod 4])$
7: RK_i^{enc} ← (X[0],X[1],X[2],X[1],X[3],X[1])
8: end for



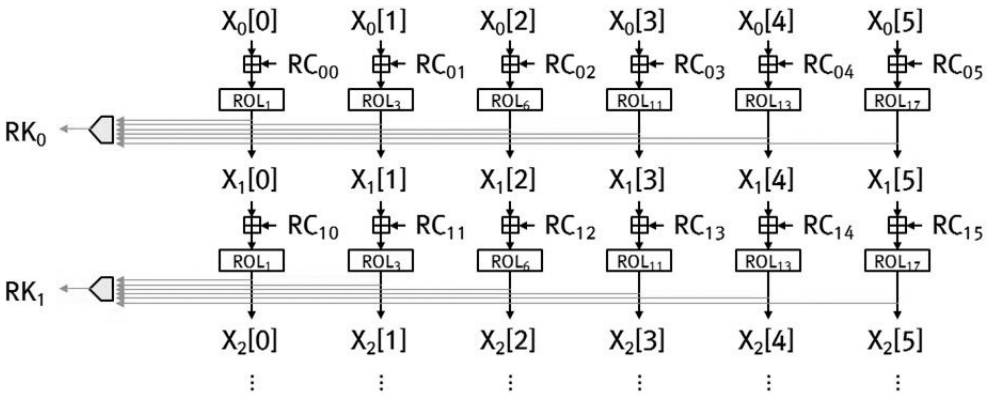
(그림 2-2) LEA-128 암호화 키 스케줄 함수

LEA-192의 암호화를 위해 사용되는 키 스케줄 함수 $LEA.EncKeySchedule_{192}$는 192 비트 암호 키 K로부터 28개의 192 비트 암호화 라운드 키 $RK_i^{enc}=(RK_i^{enc}[0],RK_i^{enc}[1],\cdots,RK_i^{enc}[5])$ $(0\le i\le 27)$를 알고리즘 4와 같이 생성하며, 이 과정에서 192 비트 내부 상태 변수 $X=(X[0],X[1],\cdots,X[5])$가 사용된다. (그림 2-3)은 LEA-192의 암호화 라운드 키 생성 과정을 도시한 것이다.

알고리즘 4 LEA-192 암호화 키 스케줄 함수 : $(RK_0^{enc},\cdots,RK_{27}^{enc}) \leftarrow LEA.EncKeySchedule_{192}(K)$

입력 : 192 비트 암호 키 K
출력 : 28개의 192 비트 암호화 라운드 키 RK_i^{enc} $(0\le i\le 27)$

1: $X \leftarrow K$
2: for i = 0 to 27 do
3: $X[0] \leftarrow ROL_1(X[0] \boxplus RC_{i0})$ // $RC_{i0} = ROL_i(\delta[i \bmod 6])$
4: $X[1] \leftarrow ROL_3(X[1] \boxplus RC_{i1})$ // $RC_{i1} = ROL_{i+1}(\delta[i \bmod 6])$
5: $X[2] \leftarrow ROL_6(X[2] \boxplus RC_{i2})$ // $RC_{i2} = ROL_{i+2}(\delta[i \bmod 6])$
6: $X[3] \leftarrow ROL_{11}(X[3] \boxplus RC_{i3})$ // $RC_{i3} = ROL_{i+3}(\delta[i \bmod 6])$
7: $X[4] \leftarrow ROL_{13}(X[4] \boxplus RC_{i4})$ // $RC_{i4} = ROL_{i+4}(\delta[i \bmod 6])$
8: $X[5] \leftarrow ROL_{17}(X[5] \boxplus RC_{i5})$ // $RC_{i5} = ROL_{i+5}(\delta[i \bmod 6])$
9: $RK_i^{enc} \leftarrow (X[0],X[1],X[2],X[3],X[4],X[5])$
10: end for



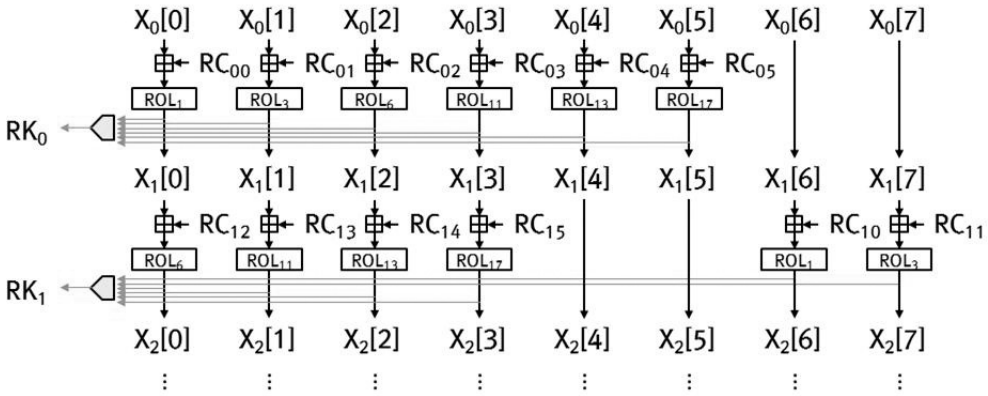
(그림 2-3) LEA-192 암호화 키 스케줄 함수

LEA-256의 암호화를 위해 사용되는 키 스케줄 함수 $LEA.EncKeySchedule_{256}$은 256 비트 암호 키 K로부터 32개의 192 비트 암호화 라운드 키 $RK_i^{enc}=(RK_i^{enc}[0],RK_i^{enc}[1],\cdots,RK_i^{enc}[5])$ $(0\le i\le 31)$를 알고리즘 5와 같이 생성하며, 이 과정에서 256 비트 내부 상태 변수 $X=(X[0],X[1],\cdots,X[7])$가 사용된다. (그림 2-4)는 LEA-256의 암호화 라운드 키 생성 과정을 도시한 것이다.

알고리즘 5 LEA-256 암호화 키 스케줄 함수 : $(RK_0^{enc},\cdots,RK_{31}^{enc}) \leftarrow LEA.EncKeySchedule_{256}(K)$

입력 : 256 비트 암호 키 K
출력 : 32개의 192 비트 암호화 라운드 키 RK_i^{enc} $(0 \le i \le 31)$

1: $X \leftarrow K$
2: for i = 0 to 31 do
3: $X[6i \bmod 8] \leftarrow ROL_1(X[6i \bmod 8] \boxplus RC_{i0})$ // $RC_{i0} = ROL_1(\delta[i \bmod 8])$
4: $X[6i+1 \bmod 8] \leftarrow ROL_3(X[6i+1 \bmod 8] \boxplus RC_{i1})$ // $RC_{i1} = ROL_{i+1}(\delta[i \bmod 8])$
5: $X[6i+2 \bmod 8] \leftarrow ROL_6(X[6i+2 \bmod 8] \boxplus RC_{i2})$ // $RC_{i2} = ROL_{i+2}(\delta[i \bmod 8])$
6: $X[6i+3 \bmod 8] \leftarrow ROL_{11}(X[6i+3 \bmod 8] \boxplus RC_{i3})$ // $RC_{i3} = ROL_{i+3}(\delta[i \bmod 8])$
7: $X[6i+4 \bmod 8] \leftarrow ROL_{13}(X[6i+4 \bmod 8] \boxplus RC_{i4})$ // $RC_{i4} = ROL_{i+4}(\delta[i \bmod 8])$
8: $X[6i+5 \bmod 8] \leftarrow ROL_{17}(X[6i+5 \bmod 8] \boxplus RC_{i5})$ // $RC_{i5} = ROL_{i+5}(\delta[i \bmod 8])$
9: $RK_i^{enc} \leftarrow (X[6i \bmod 8], X[6i+1 \bmod 8], X[6i+2 \bmod 8], X[6i+3 \bmod 8], X[6i+4 \bmod 8], X[6i+5 \bmod 8])$
10: end for



(그림 2-4) LEA-256 암호화 키 스케줄 함수

2.2 복호화

블록 암호 LEA의 복호화 과정은 다음의 두 가지 함수로 구성된다.

- 비트 길이가 k(=8×Nk)인 암호 키 K로부터 Nr개의 192 비트 복호화용 라운드 키 $RK_i^{dec}(0 \le i \le (Nr-1))$를 생성하는 복호화 키 스케줄 함수 $LEA.DecKeySchedule_k$
- 라운드 함수 LEA.DecRound와 복호화용 라운드 키 $RK_i^{dec}(0 \le i \le (Nr-1))$를 이용하여 128 비트 암호문 C를 128 비트 평문 P로 변환하는 복호화 함수 LEA.Decrypt

2.2.1 복호화 함수(LEA.Decrypt)

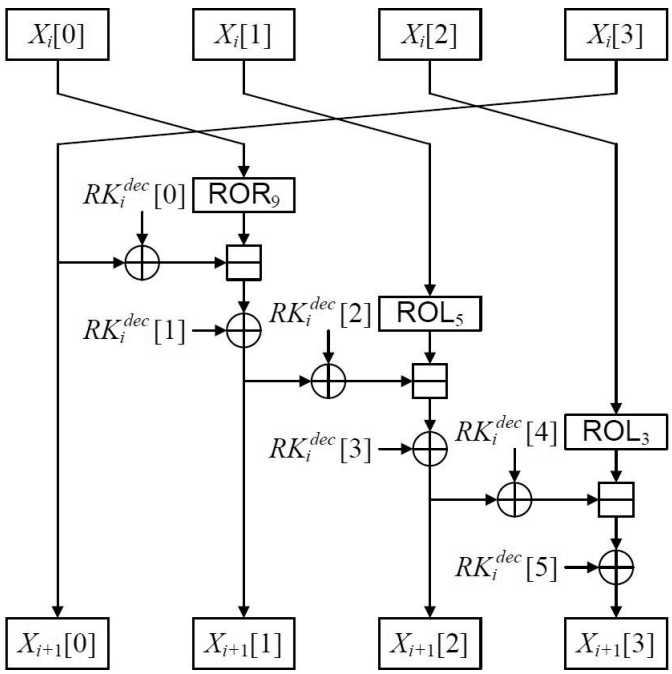
블록 암호 LEA의 복호화 함수 LEA.Decrypt는 비트 길이가 k인 암호 키 K에 대해 복호화 키 스케줄 함수 LEA.DecKeySchedulek로부터 생성된 Nr개의 192 비트 라운드 키 $RK_0^{dec}(0 \le i \le (Nr-1))$와 128 비트 암호문 C를 입력받아 알고리즘 6을 수행하여 128 비트 평문 P를 출력한다.

알고리즘 6 복호화 함수 : $P \leftarrow LEA.Decrypt(C,(RK_0^{dec},RK_1^{dec},\cdots,RK_{Nr-1}^{dec}))$

입력 : 128 비트 암호문 C, Nr개의 192 비트 라운드 키 $RK_0^{dec},RK_1^{dec},\cdots,RK_{Nr-1}^{dec}$
출력 : 128 비트 평문 P

1: $X_0 \leftarrow C$
2: for i = 0 to (Nr-1) do
3: $X_{i+1} \leftarrow LEA.DecRound(X_i,RK_i^{dec})$
4: end for
5: $P \leftarrow X_{Nr}$

알고리즘 6에서 i (0≤i≤(Nr-1))번째 라운드에 동작하는 라운드 함수 LEA.DecRound는 128 비트 내부 상태 값 $X_i=(X_i[0],X_i[1],X_i[2],X_i[3])$와 192 비트 라운드 키 $RK_i^{dec} = (RK_i^{dec}[0],RK_i^{dec}[1],\cdots,RK_i^{dec}[5])$로부터 알고리즘 7을 수행하여 새로운 128 비트 내부 상태 값 $X_{i+1}=(X_{i+1}[0],X_{i+1}[1],X_{i+1}[2],X_{i+1}[3])$을 생성한다. (그림 2-5)는 복호화 과정의 i번째 라운드 함수를 도시한 것이다.



(그림 2-5) 복호화 과정의 i번째 라운드 함수 (0≤i≤(Nr-1))

알고리즘 7 복호화 과정의 i번째 라운드 함수 : $X_{i+1} \leftarrow LEA.DecRound(X_i,RK_i^{dec})$

입력 : 128 비트 내부 상태 값 X_i, 192 비트 라운드 키 RK_i^{dec}
출력 : 128 비트 내부 상태 값 X_{i+1}

1: $X_{i+1}[0] \leftarrow X_i[3]$
2: $X_{i+1}[1] \leftarrow (ROR_9(X_i[0]) \boxminus (X_{i+1}[0] \oplus RK_i^{dec}[0])) \oplus RK_i^{dec}[1]$
3: $X_{i+1}[2] \leftarrow (ROL_5(X_i[1]) \boxminus (X_{i+1}[1] \oplus RK_i^{dec}[2])) \oplus RK_i^{dec}[3]$
4: $X_{i+1}[3] \leftarrow (ROL_3(X_i[2]) \boxminus (X_{i+1}[2] \oplus RK_i^{dec}[4])) \oplus RK_i^{dec}[5]$

2.2.2 복호화 키 스케줄 함수(LEA.DecKeySchedule)

비트 길이가 k(=8×Nk)인 암호 키 K로부터 복호화 과정에 필요한 Nr개의 192 비트 라운드 키 $RK_0^{dec}, RK_1^{dec}, \cdots, RK_{Nr-1}^{dec}$를 생성하는 복호화 키 스케줄 과정을 설명한다. 암호화 라운드 키와 복호화 라운드 키는

$$RK_i^{dec} = RK_{Nr-i-1}^{enc} \quad (0 \le i \le (Nr-1))$$

인 관계를 제외하면 동일한 방법으로 생성된다.

LEA-128의 복호화를 위해 사용되는 키 스케줄 함수 $LEA.DecKeySchedule_{128}$은 128 비트 암호 키 K로부터 24개의 192 비트 복호화 라운드 키 $RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5])$ $(0 \le i \le 23)$를 알고리즘 8과 같이 생성하며, 이 과정에서 128 비트 내부 상태 변수 X=(X[0], X[1], X[2], X[3])가 사용된다.

알고리즘 8 LEA-128 복호화 키 스케줄 함수 : $(RK_0^{dec}, \cdots, RK_{23}^{dec}) \leftarrow LEA.DecKeySchedule_{128}(K)$

입력 : 128 비트 암호 키 K
출력 : 24개의 192 비트 복호화 라운드 키 RK_i^{dec} $(0 \le i \le 23)$

```
1:  X ← K
2:  for i = 0 to 23 do
3:      X[0] ← ROL_1(X[0] ⊞ ROL_i(δ[i mod 4]))
4:      X[1] ← ROL_3(X[1] ⊞ ROL_{i+1}(δ[i mod 4]))
5:      X[2] ← ROL_6(X[2] ⊞ ROL_{i+2}(δ[i mod 4]))
6:      X[3] ← ROL_11(X[3] ⊞ ROL_{i+3}(δ[i mod 4]))
7:      RK_{23-i}^dec ← (X[0],X[1],X[2],X[1],X[3],X[1])
8:  end for
```

LEA-192의 복호화를 위해 사용되는 키 스케줄 함수 $LEA.DecKeySchedule_{192}$는 192 비트 암호 키 K로부터 28개의 192 비트 복호화 라운드 키 $RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5])$ $(0 \le i \le 27)$를 알고리즘 9와 같이 생성하며, 이 과정에서 192 비트 내부 상태 변수 X=(X[0],X[1],…,X[5])가 사용된다.

알고리즘 9 LEA-192 복호화 키 스케줄 함수 : $(RK_0^{dec}, \cdots, RK_{27}^{dec}) \leftarrow LEA.DecKeySchedule_{192}(K)$

입력 : 192 비트 암호 키 K
출력 : 28개의 192 비트 복호화 라운드 키 RK_i^{dec} $(0 \le i \le 27)$

```
1:  X ← K
2:  for i = 0 to 27 do
3:      X[0] ← ROL_1(X[0] ⊞ ROL_i(δ[i mod 6]))
4:      X[1] ← ROL_3(X[1] ⊞ ROL_{i+1}(δ[i mod 6]))
5:      X[2] ← ROL_6(X[2] ⊞ ROL_{i+2}(δ[i mod 6]))
6:      X[3] ← ROL_11(X[3] ⊞ ROL_{i+3}(δ[i mod 6]))
7:      X[4] ← ROL_13(X[4] ⊞ ROL_{i+4}(δ[i mod 6]))
8:      X[5] ← ROL_17(X[5] ⊞ ROL_{i+5}(δ[i mod 6]))
9:      RK_{27-i}^dec ← (X[0],X[1],X[2],X[3],X[4],X[5])
10: end for
```

LEA-256의 복호화를 위해 사용되는 키 스케줄 함수 $LEA.DecKeySchedule_{256}$은 256 비트 암호 키 K에 대해 32개의 192 비트 복호화 라운드 키 $RK_i^{dec}=(RK_i^{dec}[0],RK_i^{dec}[1],\cdots,RK_i^{dec}[5])$ $(0\le i\le 31)$를 알고리즘 10과 같이 생성하며, 이 과정에서 256 비트 내부 상태 변수 $X=(X[0],X[1],\cdots,X[7])$가 사용된다.

알고리즘 10 LEA-256 복호화 키 스케줄 함수 : $(RK_0^{dec},\cdots,RK_{31}^{dec}) \leftarrow LEA.DecKeySchedule_{256}(K)$

입력 : 256 비트 암호 키 K
출력 : 32개의 192 비트 암호화 라운드 키 RK_i^{dec} $(0\le i\le 31)$

1: $X \leftarrow K$
2: for i = 0 to 31 do
3: $X[6i \bmod 8] \leftarrow ROL_1(X[6i \bmod 8] \boxplus ROL_i(\delta[i \bmod 8]))$
4: $X[6i+1 \bmod 8] \leftarrow ROL_3(X[6i+1 \bmod 8] \boxplus ROL_{i+1}(\delta[i \bmod 8]))$
5: $X[6i+2 \bmod 8] \leftarrow ROL_6(X[6i+2 \bmod 8] \boxplus ROL_{i+2}(\delta[i \bmod 8]))$
6: $X[6i+3 \bmod 8] \leftarrow ROL_{11}(X[6i+3 \bmod 8] \boxplus ROL_{i+3}(\delta[i \bmod 8]))$
7: $X[6i+4 \bmod 8] \leftarrow ROL_{13}(X[6i+4 \bmod 8] \boxplus ROL_{i+4}(\delta[i \bmod 8]))$
8: $X[6i+5 \bmod 8] \leftarrow ROL_{17}(X[6i+5 \bmod 8] \boxplus ROL_{i+5}(\delta[i \bmod 8]))$
9: $RK_{31-i}^{dec} \leftarrow (X[6i \bmod 8], X[6i+1 \bmod 8], X[6i+2 \bmod 8], X[6i+3 \bmod 8], X[6i+4 \bmod 8], X[6i+5 \bmod 8])$
10: end for

3. 기밀성 운영 모드

기밀성 운영 모드는 블록 암호를 사용하여 가변 길이 데이터에 대한 기밀성을 보장할 수 있는 방법을 정의한 것으로, 주요 방식으로는 ECB(Electronic code book), CBC(Cipher block chaining), CFB(Cipher feedback), OFB(Output feedback), CTR(Counter) 등이 있다[21,14,28,39]. 이들은 개발된 지 30년이 넘었지만 현재까지도 다양한 암호 응용 환경에서 사용되고 있다. 예를 들어 CBC 모드는 SSL/TLS[9]와 IPsec[8,12]의 기본 운영 모드이고, CTR 모드는 SRTP[6]의 기본 운영 모드이다.

ECB 모드는 암호 키를 고정할 경우 동일한 평문이 동일한 암호문에 대응하므로, 범용성을 가지는 기밀성 운영 모드는 아니다. 그러나 ECB 모드는 CTR 모드나 OCB 모드[39]와 같은 다른 블록 암호 운영 모드의 주요 설계 도구로 활용된 바 있다[2]. 따라서 향후의 확장성을 고려한 적용은 가능하지만, 일반적인 사용에는 한계가 있음을 주의해야 한다.

CBC, CFB, OFB, CTR 모드는 암호 키와 함께 초기 값(IV)을 사용하여, 암호 키를 고정해서 사용하더라도 초기 값을 변경하여 동일 평문에 대해 암호화를 반복 수행할 때 마다 서로 다른 암호문이 생성될 수 있도록 한다. 이때 초기 값은 암호 키와 마찬가지로 암호화에 사용된 것과 동일한 값이 복호화에 사용되어야 하므로, 복호화 이전에 이를 공유할 수 있는 방법이 별도로 정의되어 있어야 한다. 그리고 기밀성을 보장하기 위해서 초기 값은 CBC, CFB, OFB 모드의 경우 난수 특성을 가져야 하며, CTR 모드는 Nonce 특성을 만족해야 한다[28,39].

CBC 모드와 ECB 모드는 암호화 과정에서 LEA 암호화 함수를 이용하고, 복호화 과정에서 LEA 복호화 함수를 이용한다. 그리고 LEA의 단위 블록인 128 비트의 배수 길이를 가지는 데이터에 대한 처리만 가능하다. 그러나 암호 응용에서 다루는 데이터의 비트 길이가 항상 128의 배수가 된다는 보장이 없기 때문에, CBC 모드를 사용할 경우 입력 데이터의 비트 길이가 항상 128의 배수가 되도록 보정하는 방법이 별도로 제공되어야 한다. 이를 덧붙이기 방법(Padding method)이라고 하며, 관련 예시는 3.2절에 구체적으로 제시한다.

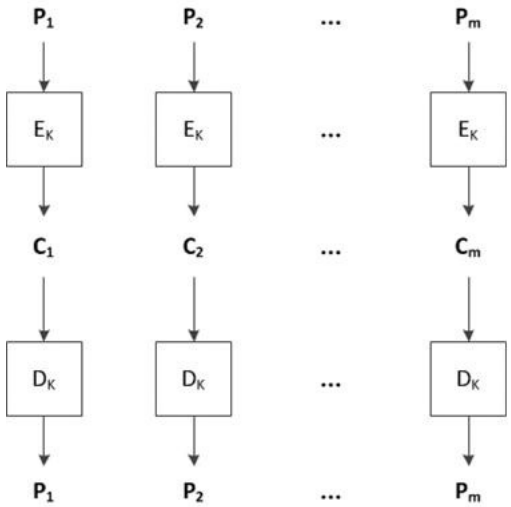
CBC 모드는 그 자체가 기밀성을 제공하는 운영 모드이면서, 4절에서 다루는 CMAC을 비롯한 다양한 인증 운영 모드 설계의 핵심 요소로 사용되었다. 따라서 CBC 모드 특징의 상당수가 CMAC에도 그대로 나타난다.
CFB, OFB, CTR 모드는 암ㆍ복호화 과정 모두 LEA의 암호화 함수만을 이용한다. 또한 평문을 직접 LEA의 암호화 함수에 입력하는 것이 아니라, 초기 값으로부터 유도된 값을 이용하여 키 수열을 생성한 후 이를 평문과 XOR하여 암호문을 생성한다. 이는 스트림 암호의 동작 방식과 동일하므로, CFB, OFB, CTR 모드를 스트림형 운영 모드라고 분류하기도 한다.

스트림형 운영 모드에서는 암ㆍ복호화 과정에서 사용되는 키 수열을 평문이나 암호문 길이에 맞춰서 생성할 수 있다. 따라서 스트림형 운영 모드는 평문과 암호문의 비트 길이가 128의 배수가 되도록 강제하지 않는다.

2) 알고리즘 20 참조

3.1 ECB(Electronic code book) 모드

ECB 모드는 각 평문 블록을 암호문 블록으로 독립적으로 암호화한다. 따라서 암호문 블록을 재배치할 경우 이에 대응하는 평문 또한 재배치되는 특성을 가지며, 128 비트 배수 단위의 데이터에 대한 암ㆍ복호화 처리만 가능하다. ECB 모드의 동작 방식은 (그림 3-1)과 같다.



(그림 3-1) ECB 모드의 암호화 및 복호화

ECB 모드는 동일한 평문에 대해 같은 암호 키를 적용하여 암호화할 때 마다 항상 동일한 암호문이 생성되므로 디지털 이미지와 같은 데이터는 패턴이 노출되기 쉽다. 따라서 안전성을 고려할 때, ECB 모드는 평문 블록이 개별 암호 키를 사용하여 독립적으로 처리되거나 평문이 전체적으로 균일하게 난수 특성을 갖는 경우와 같이 매우 제한적인 환경에 한하여 적용을 고려할 수 있다.
ECB 모드의 경우 응용 환경에 따라 평문의 덧붙이기 방법이 요구되기도 한다. 덧붙이기 방법의 예는 소절 3.2.3에 제시하며, 아래 암호화 과정과 복호화 과정의 설명에서는 덧붙이기가 이미 반영되어 평문과 암호문의 비트 길이가 128의 배수인 것으로 가정한다.

3.1.1 ECB 모드 암호화(ECB_LEA.Encrypt)

ECB 모드 암호화 과정의 구체적인 절차는 알고리즘 11과 같다.

알고리즘 11 ECB 암호화 과정 : $C \leftarrow$ ECB_LEA.Encrypt(K,P)
입력 : 암호 키 K, 평문 $P=P_1 \| P_2 \| \cdots \| P_m$ ($\|P_i\|=128$, $1 \le i \le m$) 출력 : 암호문 $C=C_1 \| C_2 \| \cdots \| C_m$
1: $RK \leftarrow LEA.EncKeySchedule_{\|K\|}(K)$ 2: for i = 1 to m do 3: $C_i \leftarrow LEA.Encrypt(P_i,RK)$ 4: end for

ECB 모드의 암호화 과정은 입력된 평문 P를 128 비트 블록 $P_1, P_2, \cdots, P_m$으로 분할한 후, P_i를 각각 LEA 암호화 함수에 입력하여 얻은 출력 값을 암호문 블록 C_i로 설정한다. 그리고 평문 블록의 순서대로 암호문 블록을 연접한 결과인 C를 암호문으로 출력한다. 따라서 암호화는 평문 블록 단위로 독립적으로 이루어진다.

3.1.2 ECB 모드 복호화(ECB_LEA.Decrypt)

ECB 모드 복호화 과정의 구체적인 절차는 알고리즘 12와 같다.

알고리즘 12 ECB 복호화 과정 : P ← ECB_LEA.Decrypt(K,C)

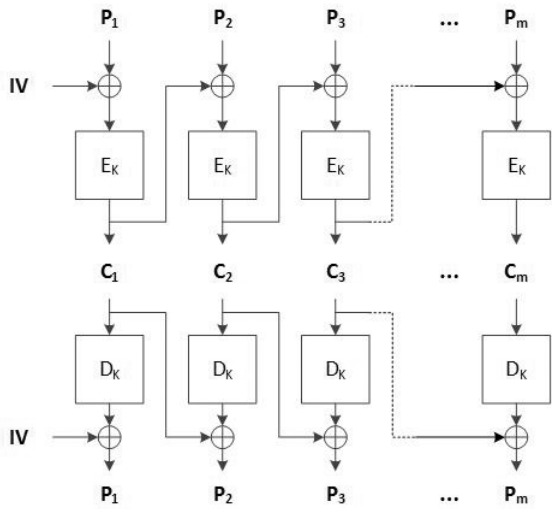
입력 : 암호 키 K, 암호문 $C=C_1 \| C_2 \| \cdots \| C_m$ ($|C_i|=128$, $1 \le i \le m$)
출력 : 평문 $P=P_1 \| P_2 \| \cdots \| P_m$

```
1: RK ← LEA.DecKeySchedule|K|(K)
2: for i = 1 to m do
3:     Pi ← LEA.Decrypt(Ci,RK)
4: end for
```

ECB 모드가 가지는 블록 단위의 독립적인 암·복호화 특성에 의해, 특정 암호문 블록에서 발생한 오류는 해당 순서의 평문 블록에만 영향을 미친다. 구체적으로 살펴보면, 알고리즘 12의 단계 3에 의해 특정 암호문 블록 C_i에서 오류가 발생할 경우 복호화된 평문 블록 P_i의 각 비트는 50%의 확률로 오류가 발생한다.

3.2 CBC(Cipher block chaining) 모드

CBC 모드는 직전 암호문 블록(최초 평문 블록일 경우 초기 값)과 평문 블록의 XOR 연산 결과를 LEA 암호화 함수에 입력하여 암호문 블록을 생성한다. 따라서 동일한 평문에 대해 같은 암호 키와 초기 값을 적용하여 암호화를 할 때 마다 항상 동일한 암호문이 생성되며, 초기 값을 변경함으로써 동일한 평문에도 다른 암호문이 생성되도록 운용한다. CBC 모드의 동작 방식은 (그림 3-2)와 같다.



(그림 3-2) CBC 모드의 암호화 및 복호화

기밀성을 최대한 보장하기 위해서는 평문마다 독립적으로 선택되며 균일한 난수성을 가진 초기 값을 사용해야 하며[39], 초기 값의 길이는 LEA의 블록 단위 길이와 동일한 128 비트이다.
CBC 모드의 경우 응용 환경에 따라 평문의 덧붙이기 방법이 요구되기도 한다. 덧붙이기 방법의 예는 소절 3.2.3에 제시하며, 아래 암호화 과정과 복호화 과정의 설명에서는 덧붙이기가 이미 처리되어 평문과 암호문의 비트 길이가 128의 배수인 것으로 가정한다.

3.2.1 CBC 모드 암호화(CBC_LEA.Encrypt)

CBC 모드 암호화 과정의 구체적인 절차는 알고리즘 13과 같다.

알고리즘 13 CBC 암호화 과정 : C ← CBC_LEA.Encrypt(K,IV,P)

입력 : 암호 키 K, 초기 값 IV, 평문 $P=P_1 \| P_2 \| \cdots \| P_m$ ($|P_i|=128$, $1 \le i \le m$)
출력 : 암호문 $C=C_1 \| C_2 \| \cdots \| C_m$

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $C_0 \leftarrow IV$
3: for i = 1 to m do
4: $X \leftarrow P_i \oplus C_{i-1}$
5: $C_i \leftarrow LEA.Encrypt(X,RK)$
6: end for

알고리즘 13의 단계 4는 (그림 3-2)에 도시된 LEA 암호화 함수 입력 값 사이의 관계를 보여준다. 구체적으로 살펴보면, 평문 블록 P_i와 암호문 블록 C_{i-1}의 XOR 연산 결과를 LEA 암호화 함수의 입력 값 X로 설정하는 것이다. 이때, 첫 번째 LEA 암호화 함수 입력 값 생성에는 단계 2에 의해 초기 값 IV가 사용된다. 이러한 순차적인 연결 관계에 인해, LEA 암호화 함수의 병렬 배치를 통한 CBC 모드 암호화 속도의 향상은 기대하기 어렵다.

3.2.2 CBC 모드 복호화(CBC_LEA.Decrypt)

CBC 모드 복호화 과정의 구체적인 절차는 알고리즘 14와 같다.

알고리즘 14 CBC 복호화 과정 : P ← CBC_LEA.Decrypt(K,IV,C)

입력 : 암호 키 K, 초기 값 IV, 암호문 $C=C_1 \| C_2 \| \cdots \| C_m$ ($|C_i|=128$, $1 \le i \le m$)
출력 : 평문 $P=P_1 \| P_2 \| \cdots \| P_m$

1: $RK \leftarrow LEA.DecKeySchedule_{|K|}(K)$
2: $C_0 \leftarrow IV$
3: for i = 1 to m do
4: $X \leftarrow LEA.Decrypt(C_i,RK)$
5: $P_i \leftarrow X \oplus C_{i-1}$
6: end for

알고리즘 14의 단계 4와 단계 5는 (그림 3-2)에서 도시한 바와 같이 특정 암호문 블록 C_i의 복호화를 위해서 이전 암호문 블록 C_{i-1}이 필요함을 보여준다. 이러한 구조적인 특징으로 인해, CBC 모드는 임의의 암호문 블

록에서 시작하여 복호화가 가능하다. 또한 특정 암호문 블록 C_i에서 비트 오류가 발생할 경우 복호화 시 평문 블록 P_i와 P_{i+1}에 영향을 미친다. 평문 블록 P_i의 경우 알고리즘 14의 단계 4에 의해 각 비트별로 50% 확률로 오류가 발생하며, 평문 블록 P_{i+1}은 알고리즘 14의 단계 5에 의해 C_i에서 발생한 오류와 동일한 위치의 비트에서만 오류가 발생한다.
암호화 과정과 다르게, CBC 모드의 복호화 과정에서는 이전 LEA 복호화 함수의 출력 값을 사용하지 않는다. 따라서 LEA 복호화 함수를 병렬로 배치하여 CBC 모드의 복호화 성능을 향상시킬 수 있다.

3.2.3 덧붙이기 방법(Padding method)

ECB, CBC 모드는 평문 블록을 LEA 암호화 함수의 입력 값으로 사용하기 때문에, 평문의 비트 길이가 128의 배수가 되도록 덧붙이기 방법을 적용해야 한다. 이를 위해 여러 국제 표준 및 암호 응용 프로토콜 규격을 통해 다양한 덧붙이기 방법이 제안된 바 있다[5,10,14]. 그러나 CBC 모드를 사용하는 다양한 암호 응용 프로토콜에 대한 패딩 오라클 공격(Padding oracle attack) 결과, 현재 알려진 대다수의 덧붙이기 방법이 공격에 활용될 수 있는 것으로 밝혀졌다[45,46,30,25,31,26,27]. 성공적인 패딩 오라클 공격은 공격자가 CBC 모드로 암호화된 데이터로부터 덧붙이기가 올바르게 적용되었는지 여부를 확인할 수 있다는 가정에 기반하여, 암호 키 없이 평문 정보를 알 수 있다. 따라서 데이터 암호화를 위해 CBC 모드를 사용할 경우, 패딩 오라클 공격의 여지를 없애도록 구현하거나 패딩 오라클 공격에 내성을 가지는 덧붙이기 방법[36]을 적용하는 것이 필요하다. 또한 암호문에 대한 인증 값 계산을 추가하여 공격자의 유효한 암호문 생성을 방지해야 한다.

아래에는 LEA 운영 모드 표준에서 제시한 세 가지 덧붙이기 방법(방법 1/2/3)을 소개한다. 이들 방법의 적용 예는 16진법 바이트 단위로 표기한다.

3.2.3.1 덧붙이기 방법 1

평문의 길이가 $(128\times n+m)$ 비트 $(0\le m<128,\ 0 \le n)$이고 m이 0이 아닐 경우, 평문의 비트 길이가 128의 배수가 되도록 평문의 끝에 $(128-m)$개의 비트 0을 덧붙인다. 방법 1의 경우 복호화 되는 평문과 추가로 덧붙여진 값과의 구분이 모호할 수 있으므로 평문의 길이가 명확히 알려져 있는 경우에 사용함을 권고한다.

예) 평문 (48 비트) : 4F 52 49 54 48 4D
적용 결과 (128 비트) : 4F 52 49 54 48 4D 00 00 00 00 00 00 00 00 00 00

평문 (128 비트) : 53 45 45 44 41 4C 47 A8 3E D1 80 F1 29 DC 4A 78
적용 결과 (128 비트) : 53 45 45 44 41 4C 47 A8 3E D1 80 F1 29 DC 4A 78

3.2.3.2 덧붙이기 방법 2

평문의 길이가 $(128\times n+m)$ 비트 $(0\le m<128,\ 0\le n)$이고 m이 0이 아닐 경우, 평문의 비트 길이가 128의 배수가 되도록 평문의 끝에 비트 1을 추가한 후 $(128-m-1)$개의 비트 0을 덧붙인다. 또한 m이 0인 경우에는 덧붙이기 방법이 사용됨을 표기하기 위해, 128 비트 블록 $1 \| 0^{127}$을 추가한다.

예) 평문 (48 비트) : 4F 52 49 54 48 4D
적용 결과 (128 비트) : 4F 52 49 54 48 4D 80 00 00 00 00 00 00 00 00 00

평문 (128 비트) : 53 45 45 44 41 4C 47 A8 3E D1 80 F1 29 DC 4A 78
적용 결과 (256 비트) : 53 45 45 44 41 4C 47 A8 3E D1 80 F1 29 DC 4A 78
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

3.2.3.3 덧붙이기 방법 3

덧붙이기 방법 3은 바이트 단위로만 적용 가능하다. 평문의 길이가 $(16\times n+m)$ 바이트 $(0\leq m<16,\ 0\leq n)$ 이고 m이 0이 아닐 경우, 평문의 바이트 길이가 16의 배수가 되도록 평문의 끝에 덧붙이기가 필요한 바이트 수 $(16-m)$을 덧붙인다. m이 0인 경우에는 덧붙이기 방법이 사용됨을 표기하기 위해, 16 바이트 블록 '0x10…10' 을 추가한다.

예) 평문 (48 비트) : 4F 52 49 54 48 4D
적용 결과 (128 비트) : 4F 52 49 54 48 4D 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A

평문 (128 비트) : 53 45 45 44 41 4C 47 A8 3E D1 AF 07 4A 73 12 2C
적용 결과 (256 비트) : 53 45 45 44 41 4C 47 A8 3E D1 AF 07 4A 73 12 2C
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

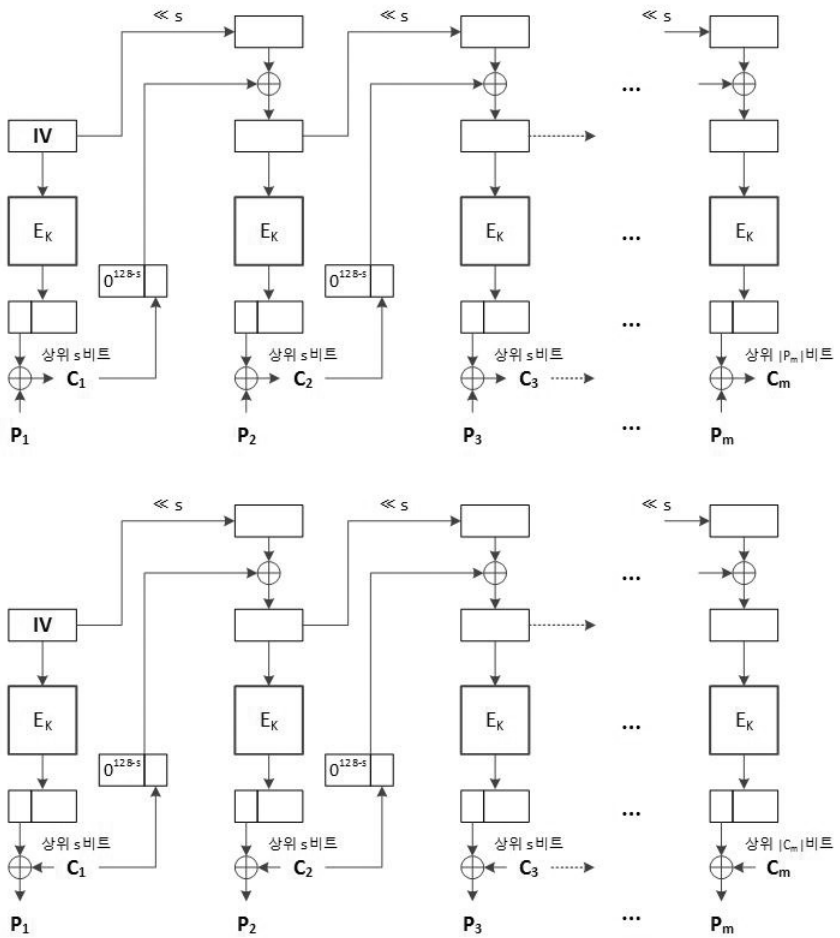
3.3 CFB(Cipher feedback) 모드

CFB 모드는 LEA 암호화 함수의 출력 값을 키 수열로 사용하는 스트림형 운영 모드이다. CFB 모드에서 현재 단계 평문 블록을 암호화하기 위해 필요한 키 수열 블록은 직전 단계 암호문 블록(최초 평문의 경우 초기 값)을 LEA 암호화 함수의 입력으로 사용하여 얻는다. 따라서 동일한 평문에 대해 같은 암호 키와 초기 값을 적용하여 암호화할 때 마다 항상 동일한 암호문이 생성되며, 초기 값을 변경함으로써 동일한 평문에도 다른 암호문이 생성되도록 운용한다. CFB 모드의 동작 방식은 (그림 3-3)과 같다.

CFB 모드의 정의에는 평문과 암호문 블록의 길이를 나타내는 매개변수 s가 사용되는데, LEA 운영 모드 표준에서는 s를 1,8,16,32,64,128로 제한한다.

매개변수 s가 작을수록 특정 길이의 데이터를 암호화할 경우에 필요한 LEA 암호화 함수의 동작 횟수가 증가함을 알 수 있다. 구체적으로 매개변수 s에 대해 128 비트 데이터 암호화를 위한 LEA 암호화 함수의 동작 횟수는 128/s이다.

기밀성을 최대한 보장하기 위해서는 평문마다 독립적으로 선택되며 균일한 난수성을 가진 초기 값을 사용해야 하며[39], 초기 값의 길이는 LEA의 블록 단위 길이와 동일한 128 비트이다.



(그림 3-3) CFB 모드의 암호화 및 복호화

3.3.1 CFB 모드 암호화(CFB_LEA.Encrypt)

CFB 모드 암호화 과정의 구체적인 절차는 알고리즘 15와 같다.

알고리즘 15 CFB 암호화 과정 : C ← CFB_LEA.Encrypt(K,IV,P)

입력 : 암호 키 K, 초기 값 IV, 평문 $P=P_1 \| \cdots \| P_m$ ($|P_i|=s$ $(1 \le i \le (m-1))$, $0 < |P_m| \le s$)
출력 : 암호문 $C=C_1 \| \cdots \| C_m$

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $X \leftarrow IV$
3: for i = 1 to (m−1) do
4: $Y \leftarrow LEA.Encrypt(X,RK)$
5: $C_i \leftarrow P_i \oplus MSB_s(Y)$
6: $X \leftarrow LSB_{128-s}(X) \| C_i$ // $|X|=128$
7: end for
8: $Y \leftarrow LEA.Encrypt(X,RK)$
9: $Cm \leftarrow P_m \oplus MSB_{|Pm|}(Y)$

알고리즘 15의 단계 5는 s 비트 키 수열 블록 MSBs(Y)와 평문 블록 P_i의 XOR 연산으로 암호문 블록 C_i을 생성하는 과정을 보여준다. 그리고 단계 9는 동일한 방식으로 마지막 평문 블록 P_m으로부터 암호문 블록 C_m을 생성하는 과정을 보여준다.

단계 6은 다음 s 비트 키 수열 블록 생성을 위한 LEA 암호화 함수 입력 값 X의 생성 방법을 보여준다. 구체적으로 살펴보면, 현재 키 수열 블록 생성을 위해 사용된 LEA 암호화 함수 입력 값의 하위 (128-s) 비트와 암호문 C_i를 연접하여 새로운 LEA 암호화 함수의 입력 값 X를 생성하는 것이다. 이러한 순차적인 연결 관계에 인해, LEA 암호화 함수의 병렬 배치를 통한 CFB 모드 암호화 속도의 향상은 기대하기 어렵다.

3.3.2 CFB 모드 복호화(CFB_LEA.Decrypt)

CFB 모드 복호화 과정은 알고리즘 16과 같다. 스트림형 운영 모드이므로 CFB 모드의 암·복호화 과정에서 키 수열 생성 방식은 동일하다.

알고리즘 16 CFB 복호화 과정 : P ← CFB_LEA.Decrypt(K,IV,C)

입력 : 암호 키 K, 초기 값 IV, 암호문 $C=C_1 \| \cdots \| C_m$ ($|C_i|=s$ $(1 \le i \le (m-1)$, $0 < |C_m| \le s)$
출력 : 평문 $P=P_1 \| \cdots \| P_m$

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $X \leftarrow IV$
3: for i = 1 to (m-1) do
4: $Y \leftarrow LEA.Encrypt(X,RK)$
5: $P_i \leftarrow C_i \oplus MSB_s(Y)$
6: $X \leftarrow LSB_{128-s}(X) \| C_i$ // $|X|=128$
7: end for
8: $Y \leftarrow LEA.Encrypt(X,RK)$
9: $P_m \leftarrow C_m \oplus MSB_{|Cm|}(Y)$

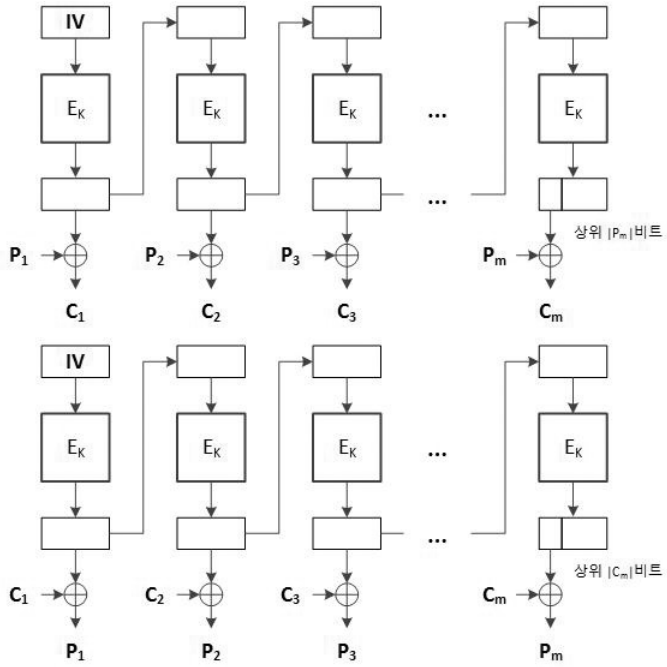
알고리즘 16의 단계 3부터 단계 7까지의 과정은 (그림 3-3)에서 도시한 바와 같이 특정 암호문 블록 C_i가 이후 128/s개 키 수열 블록 생성에 영향을 미치는 것을 보여준다. 이러한 구조적인 특징으로 인해, 암호문 블록 C_i에서 비트 오류가 발생할 경우 평문 블록 $P_i, P_{i+1}, \cdots, P_{i+128/s}$에 영향을 미친다. 평문 블록 P_i의 경우 알고리즘 16의 단계 5에 의해 C_i에서 발생한 오류와 동일한 위치의 비트에서만 오류가 발생하는 반면, 평문 블록 $P_{i+1}, \cdots, P_{i+128/s}$의 경우 알고리즘 16의 단계 4에 의해 각 비트별로 50%의 확률로 오류가 발생한다. 해당 과정은 또한 암호문 블록 C_i의 복호화를 위해 암호문 블록 $C_{i-128/s}, \cdots, C_{i-2}, C_{i-1}$이 필요함을 보여주며, 이는 임의의 암호문 블록에서 시작하여 복호화가 가능함을 의미한다.

암호화 과정과 다르게, CFB 모드의 복호화 과정에서는 이전 LEA 암호화 함수의 출력 값을 사용하지 않는다. 따라서 LEA 암호화 함수를 병렬로 배치하여 CFB 모드의 복호화 성능을 향상시킬 수 있다.

3.4 OFB(Output feedback) 모드

OFB 모드는 LEA 암호화 함수의 출력 값을 키 수열로 사용하는 스트림형 운영 모드이다. OFB 모드에서 현재 단계 평문 블록을 암호화하기 위해 필요한 키 수열 블록은 직전 단계 키 수열 블록(최초 평문의 경우 초기 값)을 LEA 암호화 함수의 입력으로 사용하여 얻는다. 따라서 키 수열이 평문과 독립적으로 생성되며,

동일한 평문에 대해 같은 암호 키와 초기 값을 적용하여 암호화할 때 마다 항상 동일한 암호문이 생성된다. 일반적으로는 암호 키를 고정한 상태에서 초기 값을 변경함으로써 동일한 평문에도 다른 암호문이 생성되도록 운용한다. OFB 모드의 동작 방식은 (그림 3-4)와 같다.



(그림 3-4) OFB 모드의 암호화 및 복호화

기밀성을 최대한 보장하기 위해서는 평문마다 독립적으로 선택되며 균일한 난수성을 가진 초기 값을 사용해야 하며[39], 초기 값의 길이는 LEA의 블록 단위 길이인 128 비트이다.

3.4.1 OFB 모드 암호화(OFB_LEA.Encrypt)

OFB 모드 암호화 과정의 구체적인 절차는 알고리즘 17과 같다.
알고리즘 17의 단계 5는 키 수열 블록 Y와 평문 블록 P_i의 XOR 연산으로 암호문 블록 C_i를 생성하는 것을 보여준다. 그리고 단계 9는 동일한 방식으로 마지막 평문 블록 P_m으로부터 암호문 블록 C_m을 생성하는 과정을 보여준다.

알고리즘 17의 단계 6은 암호화에 사용된 키 수열 블록이 다음 키 수열 블록 생성을 위한 LEA 암호화 함수의 입력 값으로 사용되는 것을 보여준다. 이러한 순차적인 연결 관계에 인해, LEA 암호화 함수의 병렬 배치를 통한 OFB 모드 암호화 속도의 향상은 기대하기 어렵다. 그러나 평문과 독립적으로 키 수열을 생성할 수 있기 때문에, 키 수열을 미리 생성한 후 평문이 입력될 때에 사용하는 방식을 적용할 수 있다.

알고리즘 17 OFB 암호화 과정 : C ← OFB_LEA.Encrypt(K,IV,P)

입력 : 암호 키 K, 초기 값 IV, 평문 $P=P_1 \| \cdots \| P_m$ ($|P_i|=128$ ($1 \le i \le (m-1)$), $0 < |P_m| \le 128$)
출력 : 암호문 $C=C_1 \| \cdots \| C_m$

```
1: RK ← LEA.EncKeySchedule|K|(K)
2: X ← IV
3: for i = 1 to (m−1) do
4:     Y ← LEA.Encrypt(X,RK)
5:     Ci ← Pi ⊕ Y
6:     X ← Y
7: end for
8: Y ← LEA.Encrypt(X,RK)
9: Cm ← Pm ⊕ MSB|Pm|(Y)
```

3.4.2 OFB 모드 복호화(OFB_LEA.Decrypt)

OFB 모드 복호화 과정의 구체적인 절차는 알고리즘 18과 같다. 스트림형 운영 모드이므로 OFB 모드의 암、복호화 과정에서 키 수열 생성 방식은 동일하다.

알고리즘 18 OFB 복호화 과정 : P ← OFB_LEA.Decrypt(K,IV,C)

입력 : 암호 키 K, 초기 값 IV, 암호문 $C=C_1 \| \cdots \| C_m$ ($|C_i|=128$ ($1 \le i \le (m-1)$), $0 < |C_m| \le 128$)
출력 : 평문 $P=P_1 \| \cdots \| P_m$

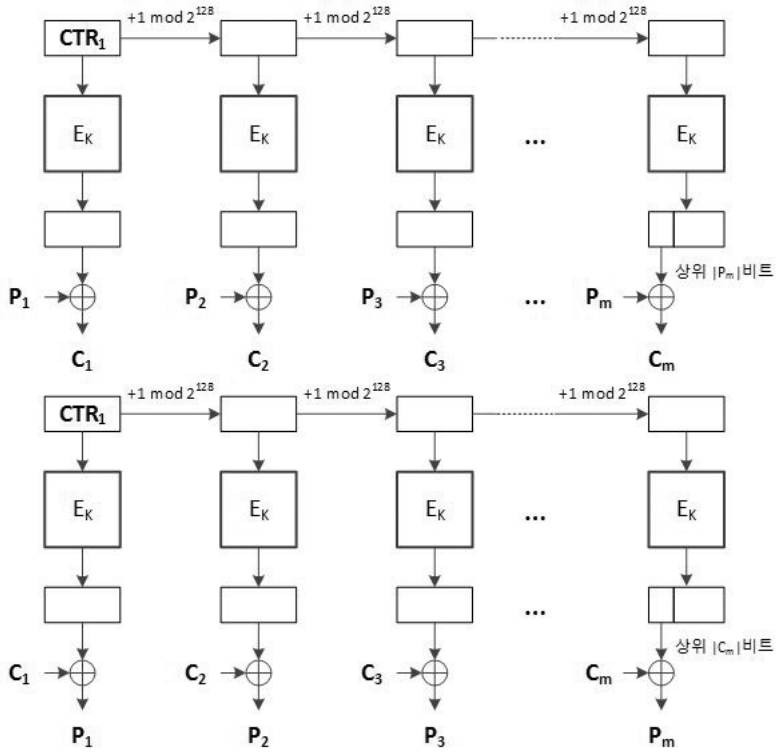
```
1: RK ← LEA.EncKeySchedule|K|(K)
2: X ← IV
3: for i = 1 to (m−1) do
4:     Y ← LEA.Encrypt(X,RK)
5:     Pi ← Ci ⊕ Y
6:     X ← Y
7: end for
8: Y ← LEA.Encrypt(X,RK)
9: Pm ← Cm ⊕ MSB|Cm|(Y)
```

OFB 모드는 암호문이 키 수열 생성에 관여하지 않고 초기 값으로부터 순차적으로 키 수열을 생성하기 때문에 임의의 암호문 블록에서 시작하여 복호화를 수행하는 것이 불가능하다. 순차적인 구조로 인해 암호화 과정에서 언급한 바와 같이 LEA 암호화 함수의 병렬 배치를 통한 성능 향상을 기대하기 어렵지만, 키 수열 사전 생성 방식을 적용할 수 있다.

OFB 모드에서 암호문의 비트 오류는 대응하는 평문 블록을 제외하고 다른 평문 블록으로 전파되지 않는다. 또한 해당 평문 블록은 암호문 블록에서 오류가 발생한 위치와 동일한 비트에서만 오류가 발생한다.

3.5 CTR(Counter) 모드

CTR 모드는 LEA 암호화 함수의 출력 값을 키 수열로 사용하는 스트림형 운영 모드이다. CTR 모드에서 키 수열 블록은 블록 단위로 변경되는 카운터 블록을 LEA 암호화 함수의 입력 값으로 사용하여 얻을 수 있다. 따라서 키 수열이 평문과 독립적으로 생성된다. CTR 모드의 동작 방식은 (그림 3-5)와 같다.



(그림 3-5) CTR 모드의 암호화 및 복호화

카운터 블록은 주어진 암호 키에 대해 이전에 LEA 암호화 함수의 입력 값으로 사용되지 않은 128 비트 블록을 사용해야 한다. 따라서 CTR 모드는 다른 기밀성 운영 모드와는 다르게 카운터 블록 생성 함수 CntGenFt를 별도로 고려할 필요가 있다. 실제로 CTR 모드를 기본 블록 암호 운영 모드로 채택한 암호 프로토콜에서는 프로토콜의 운용 방식을 고려하여 카운터 블록의 설정 방법을 제시하고 있으며, 5절에 제시하는 CTR 모드를 핵심 기능으로 사용하는 인증 암호화 운영 모드인 CCM이나 GCM에서도 카운터 블록 설정 예를 확인할 수 있다. 이러한 경우 초기 값은 암·복호화를 위해 공유해야 하는 최소 크기의 정보로 사용된다. (그림 3-5)에 도시된 카운터 블록 생성 함수는 다음과 같이 정의할 수 있다.

$$\mathrm{CntGenFt(IV,m)}\colon \mathrm{CTR1} \leftarrow \mathrm{IV},\ \mathrm{CTR}_{i+1} \leftarrow (\mathrm{CTR}_i+1) \bmod 2^{128}\ (1 \le i \le m).$$

이외에도 충분한 주기를 보장할 수 있는 LFSR의 출력을 128 비트 단위로 카운터 블록으로 설정하여 사용할 수도 있다. 이러한 카운터 블록 생성 함수는 공통적으로 암호 키의 유효 기간 내에서 Nonce 특성을 가지도록 카운터 블록을 생성할 수 있어야 한다. 또한 개별 초기 값에 따라 생성되는 카운터 블록 열 사이에서도 동일 카운터 블록이 발생할 가능성을 최소화할 수 있도록 정의되어야 한다.

이러한 카운터 블록의 특성에 의해 동일한 평문에 대해 같은 암호 키와 초기 값을 적용하여 암호화할 때 마다 항상 동일한 암호문이 생성된다. 일반적으로는 암호 키를 고정한 상태에서 초기 값을 변경하여 얻는 새로운 카운터 블록 열을 사용함으로써 동일한 평문에도 다른 암호문이 생성되도록 운용한다.

3.5.1 CTR 모드 암호화(CTR_LEA.Encrypt)

CTR 모드 암호화 과정의 구체적인 절차는 알고리즘 19와 같다. 여기에서 카운터 블록은 블록 단위로 1씩 증가하는 설정을 가정한다.

알고리즘 19 CTR 암호화 과정 : C ← CTR_LEA.Encrypt(K,IV,P)

입력 : 암호 키 K, 초기 값 IV, 평문 P=$P_1 \| \cdots \| P_m$ ($|P_i|$=128 ($1 \le i \le (m-1)$,$0 < |P_m| \le 128$)
출력 : 암호문 C=$C_1 \| \cdots \| C_m$

1: RK ← LEA.EncKeySchedule$_{|K|}$(K)
2: CTR_1 ← IV
3: for i = 1 to (m−1) do
4: Y ← LEA.Encrypt(CTR_i,RK)
5: C_i ← $P_i \oplus Y$
6: CTR_{i+1} ← (CTR_i+1) mod 2^{128}
7: end for
8: Y ← LEA.Encrypt(CTR_m,RK)
9: C_m ← $P_m \oplus MSB_{|Pm|}(Y)$

알고리즘 19의 단계 5는 128 비트 키 수열 블록 Y와 평문 블록 P_i의 XOR 연산으로 암호문 블록 C_i를 생성하는 과정을 보여준다. 그리고 단계 9는 동일한 방식으로 마지막 평문 블록 P_m으로부터 암호문 블록 C_m을 생성하는 과정을 보여준다.

단계 6은 키 수열 생성을 위한 카운터 블록의 갱신 과정을 보여준다. 이러한 구조로 인해, LEA 암호화 함수를 병렬로 배치할 경우 CTR 모드 암호화 속도를 향상시킬 수 있다. 그리고 평문과 독립적으로 키 수열을 생성할 수 있기 때문에, 키 수열을 미리 생성한 후 평문이 입력될 때에 사용하는 방식 또한 적용 가능하다.
알고리즘 20은 일반적인 카운터 블록 생성 함수 CntGenFt와 키 수열 사전 생성 방식을 적용하여 알고리즘 19를 재구성한 것이다.

알고리즘 20 CTR 암호화 과정 : C ← CTR_LEA.Encrypt(K,IV,P)

입력 : 암호 키 K, 초기 값 IV, 평문 P=$P_1 \| \cdots \| P_m$ ($|P_i|$=128 ($1 \le i \le (m-1)$,$0 < |P_m| \le 128$)
출력 : 암호문 C=$C_1 \| \cdots \| C_m$

1: CTR(=$CTR_1 \| CTR_2 \| \cdots \| CTR_m$) ← CntGenFt(IV,m)
2: $Y_1 \| Y_2 \| \cdots \| Y_m$ ← ECB_LEA.Encrypt(K, CTR)
3: for i = 1 to (m−1) do
4: C_i ← $P_i \oplus Y_i$
5: end for
6: C_m ← $P_m \oplus MSB_{|Pm|}(Y_m)$

3.5.2 CTR 모드 복호화(CTR_LEA.Decrypt)

CTR 모드 복호화 과정의 구체적인 절차는 알고리즘 21과 같다. 알고리즘 19와 같이 카운터 블록은 블록 단위로 1씩 증가하는 설정을 가정한다. 스트림형 운영 모드이므로 CTR 모드의 암、복호화 과정에서 키 수열 생성 방식은 동일하다.

알고리즘 21 CTR 복호화 과정 : $P \leftarrow CTR\_LEA.Decrypt(K,IV,C)$

입력 : 암호 키 K, 초기 값 IV, 암호문 $C=C_1 \| C_2 \| \cdots \| C_m$ ($|C_i|=128$ $(1 \le i \le (m-1), 0 < |C_m| \le 128)$
출력 : 평문 $P = P_1 \| P_2 \| \cdots \| P_m$

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $CTR_1 \leftarrow IV$
3: for i = 1 to (m−1) do
4: $Y \leftarrow LEA.Encrypt(CTR_i, RK)$
5: $P_i \leftarrow C_i \oplus Y$
6: $CTR_{i+1} \leftarrow (CTR_i+1) \bmod 2^{128}$
7: end for
8: $Y \leftarrow LEA.Encrypt(CTRm, RK)$
9: $P_m \leftarrow C_m \oplus MSB_{|Cm|}(Y)$

암호화 과정에서 언급한 바와 같이 LEA 암호화 함수의 병렬 배치로 인한 성능 향상과 키 수열 사전 생성 방식의 적용이 동시에 가능하다.

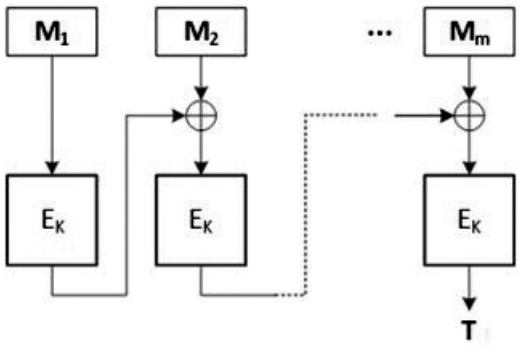
CTR 모드는 특정 위치에 대한 카운터 블록의 생성이 가능한 경우, 임의의 암호문 블록에서 시작하여 복호화가 가능하다. 그리고 암호문의 비트 오류는 대응하는 평문 블록을 제외하고 다른 평문 블록으로 전파되지 않는다. 또한 해당 평문 블록은 암호문 블록에서 오류가 발생한 위치와 동일한 비트에서만 오류가 발생한다.

4. 메시지 인증 운영 모드: CMAC (Cipher-based MAC)

메시지 인증 운영 모드는 일반적으로 블록 암호 기반 메시지 인증 코드(Message authentication code)로 잘 알려져 있다. 본 절에서도 '메시지 인증 운영 모드' 대신 '메시지 인증 코드'로 사용한다. 메시지 인증 코드는 암호 키와 메시지를 입력받아 주어진 길이의 인증 값을 생성한다. 대다수의 메시지 인증 코드 방식은 암호 키를 고정할 경우 동일한 메시지에 대해 동일한 인증 값을 생성하며, Nonce 성질을 가지는 초기 값을 부가 입력으로 사용하여 동일한 메시지에 대해서도 다른 인증 값을 생성하도록 설계된 방식도 일부 존재한다. 본 절에서 다루는 CMAC[34,22]은 전자에 해당하므로, 아래에서는 Nonce 성질을 가지는 초기 값 사용에 대해서는 고려하지 않는다. 참고로 5절에서 다루는 인증 암호화 운영 모드 GCM의 암호화 기능을 제외한 형태인 GMAC은 Nonce 성질의 초기 값을 사용하는 메시지 인증 코드이다[24].

메시지 인증 코드 방식의 실제 구현과 관련한 주요 파라미터로는 메시지 인증 코드가 생성하는 인증 값의 길이를 결정하는 '인증 값 길이(Tag length) 정보'가 있다. 응용 환경에서 통신량의 부하를 줄이기 위해 메시지 인증 코드 방식으로 생성된 인증 값을 절삭하여 사용하는 경우, 송·수신자는 인증 값 길이 정보를 사전에 공유해야 한다. 인증 값 길이는 메시지 인증 코드의 안전성과 관련된 주요 요소이다. 예를 들어 주어진 인증 값 길이를 Tlen 비트라 하면, 공격자는 2^{-Tlen}의 확률로 암호 키 없이 인증 값의 위조가 가능하다. 따라서 인증 값 길이 설정에 앞서 운용 환경을 고려한 안전성의 면밀한 검토가 필요하다.

CMAC은 3.2절에 정리한 CBC 모드로부터 파생된 메시지 인증 코드 CBC-MAC을 안전성 측면에서 개선시킨 것이다. CBC-MAC은 (그림 4-1)과 같이 CBC 모드의 암호화 과정을 그대로 사용하여 마지막 암호문 블록을 인증 값으로 설정하는 방식으로, 가장 대표적인 블록 암호 기반 메시지 인증 코드이다.



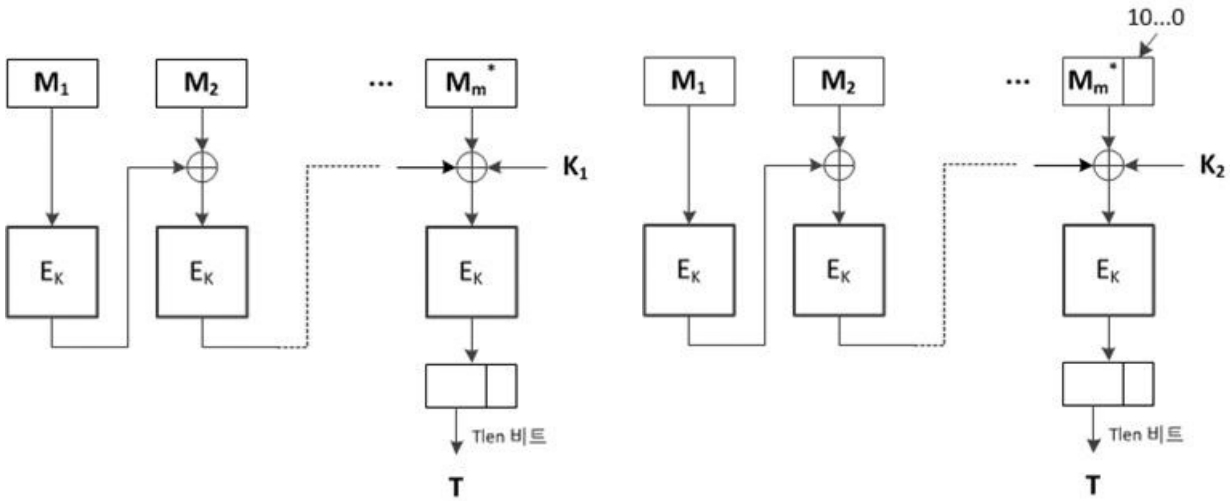
(그림 4-1) CBC-MAC 인증 값 생성 과정

그러나 CBC-MAC은 비트 길이가 128의 배수인 메시지에 대한 처리만 가능하며, 가변 길이 메시지의 처리 시 안전성에 문제점을 가지고 있다[29]. 단순한 예로 메시지 M에 대한 CBC-MAC의 인증 값을 T라 하면, T는 M ‖ (M ⊕T)의 인증 값도 된다. 이러한 특성은 암호 키를 모르는 공격자가 특정 메시지와 인증 값의

쌍을 알 경우 이를 이용하여 인증 값이 알려지지 않은 새로운 메시지에 대한 인증 값 생성할 수 있다는 것을 보여준다. 이를 보완하기 위해 다양한 개선 방식이 제안되었으며[29,34,39], CMAC도 그 중 하나이다.

4.1 CMAC 인증 값 생성(CMAC_LEA.TagGen)

CMAC은 인증 대상 메시지를 128 비트 단위 블록으로 분할한 후 (그림 4-1)에서 제시한 CBC-MAC과 동일하게 블록 단위로 메시지를 처리한다. 그러나 마지막 메시지 블록 처리 시 메시지 길이에 따라 두 가지 경우로 구분하여 LEA 암호화 함수의 입력 값을 설정한 후, 얻은 출력 값을 인증 값으로 반환한다. 동작 방식은 (그림 4-2)와 같다.



|Mm*|=128일 때 |Mm*|≠128일 때

(그림 4-2) CMAC 인증 값 생성

CMAC은 CBC-MAC과 비교하여 메시지 길이에 대한 제약이 없기 때문에 별도의 덧붙이기 방법은 필요 없지만, 마지막 메시지 블록 처리를 위해 보조 키 K_1과 K_2의 생성이 필요하다. 보조 키 K_1과 K_2를 암호 키 K로부터 유도하는 방법은 알고리즘 22와 같다.

알고리즘 22 CMAC 보조 키 생성 : $\{K_1,K_2\} \leftarrow$ CMAC_LEA.SubKeyGen(K)

입력 : 암호 키 K
출력 : 보조 키 K_1, K_2

1: $L \leftarrow E_K(0^{128})$
2: if $MSB_1(L)=0$ then
3: $K_1 \leftarrow L \ll 1$
4: else
5: $K_1 \leftarrow (L \ll 1) \oplus (0^{120} \| 10000111)$
6: end if
7: if $MSB_1(K_1)=0$ then
8: $K_2 \leftarrow K_1 \ll 1$
9: else
10: $K_2 \leftarrow (K_1 \ll 1) \oplus (0^{120} \| 10000111)$
11: end if

알고리즘 22는 유한체 $GF(2^{128})$상에 정의되는 2배 연산을 사용하여 간략하게 표현할 수 있다. 유한체 $GF(2^{128})$의 원소 표현을 위한 감산 다항식(Reduction polynomial) g는 다음과 같다.

$$g(u) = u^{128} + u^7 + u^2 + u + 1.$$

그러면 유한체 $GF(2^{128})$에서의 2배 연산 함수 dbl은 다음과 같이 정의할 수 있다.

$$dbl(X) = \begin{cases} (X \ll 1) & \text{if } MSB_1(X) = 0, \\ (X \ll 1) \oplus 0^{120}10000111 & \text{if } MSB_1(X) = 1. \end{cases}$$

2배 연산 함수 dbl를 이용하여 알고리즘 22를 표현하면 다음과 같다.

$$L \leftarrow E_K(0^{128}), \qquad K_1 \leftarrow dbl(L), \qquad K_2 \leftarrow dbl(K_1).$$

즉, 알고리즘 22에서 중간 값 L 계산(단계 1), 보조 키 K_1 계산(단계 2~단계 6), 그리고 보조 키 K_2 계산(단계 7~단계 11)이 순차적으로 이루어지는 것을 확인할 수 있다.
CMAC 인증 값 생성 과정의 구체적인 절차는 알고리즘 23과 같다.

알고리즘 23 CMAC 인증 값 생성 과정 : T ← CMAC_LEA.TagGen(K,M,Tlen)

입력 : 암호 키 K, 메시지 $M=M_1 \| \cdots \| M_m$ ($|M_i|=128$ $(1 \le i \le (m-1))$, $0 < |M_m| \le 128$), 인증 값 길이 Tlen
출력 : 인증 값 T ($|T|=Tlen$)

```
1:  if |M| = 0 then
2:      m ← 1
3:  end if
4:  {K1,K2} ← CMAC_LEA.SubKeyGen(K)          // 알고리즘 22
5:  if |Mm*|=128 then
6:      Mm←K1 ⊕ Mm*
7:  else
8:      Mm←K2 ⊕ (Mm* ‖ 1 ‖ 0^j), j=128−|Mm*|−1
9:  end if
10:  Y ← 0^128
11:  for i = 1 to m do
12:     X ← Y ⊕ Mi
13:     Y ← LEA.Encrypt(X,RK);
14:  end for
15:  T ← MSB_Tlen(Y)
```

참고로 LEA 운영 모드 표준에서는 알고리즘 22를 알고리즘 23에 포함시켜 하나의 알고리즘 형태로 기술하였으나, 본 해설서에서는 보조 키 생성에 대한 좀 더 상세한 설명을 위해 구분하였다.

단계 4에서 알고리즘 22를 이용하여 보조 키를 계산한 후, 단계 5에서 단계 9까지는 (그림 4-2)에 도시한 마지막 메시지 블록 처리 방식에 따라 전체 메시지의 비트 길이가 128의 배수가 되도록 보정하는 과정을 보여준다. 이어서 단계 11부터 단계 14까지의 과정은 CBC-MAC을 계산하여 128 비트 출력 값을 생성하며, 단계 15는 입력으로 주어진 인증 값 길이 정보 Tlen에 따라 필요한 길이만큼 절삭하여 CMAC의 인증 값으로 반환하는 과정을 보여준다. 참고로 단계 1부터 단계 3까지는 빈 메시지를 처리하기 위한 부분으로, 실제 운용

시 빈 메시지에 대한 인증 값 계산이 필요 없는 경우 생략 가능하다.

실제 구현 과정에서 메시지의 길이를 처음부터 확인하기 어려운 경우에는 마지막 블록이 입력되기 전까지 CBC-MAC의 연결 과정(단계 11~단계 14)을 수행하다가 마지막 블록에서 단계 5부터 단계 9까지의 메시지 블록 처리 과정을 적용한 후, CBC-MAC 계산의 마지막 메시지 블록으로 적용할 수 있다.

(그림 4-2)에서 직관적으로 알 수 있듯이, CMAC은 결국 CBC 모드로부터 파생된 구조로 볼 수 있다. 따라서 실제 구현과 관련한 특징(병렬 처리, 사전 계산 등)은 CBC 모드와 거의 동일하다. 효율성 측면에서는 보조 키 생성을 위해 LEA 암호화 함수를 1회 더 실행해야 하지만, 안전성 측면에서 CBC-MAC이 갖는 제약을 해소하였다.

4.2 CMAC 인증 값 검증(CMAC_LEA.TagVrfy)

수신자가 메시지 M의 인증 값으로 주어진 T의 유효성을 확인하기 위해서는, 암호 키 K, 메시지 M, 인증 값 길이 정보 Tlen을 입력으로 CMAC 인증 값 생성 과정을 수행하여 인증 확인 값 T′을 계산한 후 T′과 T가 동일한지 여부를 확인한다. 따라서 인증 값 검증 과정에서도 알고리즘 23을 사용하며, 이는 CMAC이 LEA 암호화 함수만을 사용하는 것을 의미한다.

5. 인증 암호화 운영 모드

인증 암호화(Authenticated encryption) 운영 모드는 데이터에 대한 기밀성과 인증을 동시에 제공하기 위해 기밀성 운영 모드와 인증 운영 모드에 독립된 암호 키를 적용하여 사용하는 대신, 하나의 암호 키로 동일한 안전성을 보장할 수 있도록 하는 방식이다. 인증 암호화 운영 모드가 가지는 편의성과 안전성 측면의 장점으로 인해, SSL/TLS, IPsec, SRTP 등 주요 암호 프로토콜은 규격을 정비하여 인증 암호화 운영 모드의 적용을 지원하고 있다. 또한 학계에서는 CAESAR 국제 공모(https://competitions.cr.yp.to/index.html)를 통해 다양한 신규 방식의 설계 및 안전성 분석 기술 연구를 활성화시키고 있다.

LEA 운영 모드 표준에서 제시하는 CCM(Counter with CBC-MAC) 모드[23,17]와 GCM(Galois/Counter mode)[24,17]은 기밀성 운영 모드인 CTR과 인증 운영 모드를 조합하여 하나의 암호 키를 사용할 수 있도록 설계된 인증 암호화 운영 모드이다. 기밀성 운영 모드와 동일하게, 인증 암호화 운영 모드는 암호 키와 함께 초기 값을 사용하며, 초기 값은 Nonce 특성을 가져야 한다. 이때 초기 값은 암호 키와 마찬가지로 암호화에 사용된 것과 동일한 값이 복호화에 사용되어야 하므로, 복호화 이전에 이를 공유할 수 있는 방법이 별도로 정의되어 있어야 한다. 또한 암호 키, 초기 값, 평문(암호문)과 함께, 암호화(복호화) 시 입력으로 부가 인증 데이터(Associated data)를 받을 수 있다. 부가 인증 데이터는 공개되어도 무방하지만 무결성은 보장되어야 하는 데이터로, 암호문에 포함되지 않으며 암호문으로부터 복구 가능할 필요도 없다. 부가 인증 데이터의 주요 용례로는 SSL/TLS나 IPsec 등 암호 프로토콜을 통해 보호하는 패킷이나 프레임의 헤더 정보를 들 수 있다. 헤더 정보는 해당 데이터의 구성 정보를 비롯하여 전송을 위한 네트워크 경로 배정 정보를 담고 있어 무결성은 보장되어야 하는 반면, 기밀성을 보장할 경우 오히려 처리가 어려워지는 문제가 발생할 수 있다.

CCM 모드와 GCM의 암호화 과정을 통해 출력된 암호문 C_{AE}는 CTR 모드에 의해 생성되는 값 C_{CTR}과 메시지 인증 코드에 의해 생성되는 인증 값 T를 연접시킨 구성으로 볼 수 있다. 따라서 입력 데이터 대비 출력 데이터의 크기 확장이 발생한다. 수신자는 주어진 암호 키 K, 초기 값 N, 부가 인증 데이터 A에 대해 수신된 암호문 C_{AE}로부터 평문 후보 P'을 복구하고 인증 확인 값 T'을 계산할 수 있다. 평문 후보 P'의 복구 과정에서 오류가 발생하지 않았을 경우, 이는 수신된 암호문 C_{AE}가 (K, N, A)로부터 생성되었다는 유력한 증거가 된다. 그러나 수신된 암호문 C_{AE}에 대한 엄밀한 인증은 수신자가 생성한 확인 값 T'과 수신된 인증 값 T가 같은 지 여부를 확인함으로써 결정되는 것으로, 두 값이 다를 경우 수신자는 P'을 무시하고 오류 코드를 반환한다. 따라서 복호화 시 인증 값 검증을 통과해야 평문을 얻는 것이 가능하다.

CCM 모드는 암호화 전에 모든 데이터의 정보가 정해지는 환경에서 사용하는 것을 가정하며, 스트리밍 서비스와 같이 온라인 처리는 지원하지 않는다. 암호화 과정에서는 평문과 부가 인증 데이터, 그리고 초기 값에 CBC-MAC을 적용하여 인증 값을 생성한 후, 평문과 인증 값에 CTR 모드를 적용하여 출력 값 C_{CCM}을 생성한다.

반면 GCM은 입력 데이터의 정보가 사전에 정해질 필요가 없기 때문에 온라인 처리가 가능하다. 암호화 과정에서는 평문에 CTR 모드를 적용하여 C_{CTR}을 계산한 후, 부가 인증 데이터와 함께 유한체 연산으로 정의된 함수에 입력하여 인증 값을 생성하고 두 값을 연접하여 출력 값 C_{GCM}을 생성한다. GCM은 데이터 처리 구조가 CCM 모드에 비해 좀 더 직관적이지만, 인증 값을 절삭해서 사용할 경우 취약성이 발생할 수 있다는 단점을 가진다[32,39]. 그리고 성능상의 장점을 극대화하기 위해서는 128 비트 유한체 곱셈 연산이 효율적으로 동작해야 한다.

5.1 CCM(Counter with CBC-MAC) 모드

CCM 모드는 3.5절에서 소개한 CTR 모드와 (그림 4-1)에 도시한 CBC-MAC을 조합한 구조를 가진다. CCM 모드는 기본적인 암ㆍ복호화 과정 이외에 입력 데이터를 구조화하는 함수와 카운터 블록 생성 함수를 고려해야 한다. 입력 데이터 구조화 함수 CCM.InputFormatFt는 입력 데이터인 초기 값, 부가 인증 데이터, 평문으로부터 인증 값 계산을 위한 128 비트 데이터 블록 열 $B=B_0 \| B_1 \| \cdots \| Br$을 구성하고, 카운터 블록 생성 함수 CCM.CntGenFt는 CTR 모드 동작 및 인증 값 계산을 위한 카운터 블록 열 $CTR=CTR_0 \| CTR_1 \| \cdots \| CTR_m$을 구성한다.

본 절에서는 LEA 운영 모드 표준에서 정의한 CCM 모드의 입력 데이터 구조화 함수와 카운터 블록 생성 함수를 먼저 소절 5.1.1에 정리하고, 암호화 과정과 복호화 과정은 각각 소절 5.1.2와 소절 5.1.3에 정리한다.

5.1.1 데이터 블록 열과 카운터 블록 설정

입력 데이터 구조화 함수 CCM.InputFormatFt는 초기 값, 부가 인증 데이터, 평문을 입력으로 하여 인증 값 계산에 사용될 데이터 블록 열 B를 생성한다. 초기 값 N, 부가 인증 데이터 A, 그리고 평문 P의 비트 길이는 모두 8의 배수이어야 한다. 따라서 이들은 모두 바이트 열로 표현될 수 있고 N, A, P의 바이트 길이를 각각 n, a, p로 표기한다. CCM.InputFormatFt는 추가로 CCM 모드에 의해 계산되는 인증 값의 바이트 길이 정보 t를 입력받는다.

입력 데이터 구조화 함수의 구체적인 절차는 알고리즘 24와 같다.

알고리즘 24 CCM 입력 데이터 구조화 함수 : B ← CCM.InputFormatFt(N,A,P,t)

입력 : 초기 값 N, 부가 인증 데이터 A, 평문 P, 인증 값 바이트 길이 t
출력 : 데이터 블록 열 $B=B_0 \| B_1 \| \cdots \| Br$, $|B_i|=128$ $(0 \le i \le r)$

1: if Alen = 0 then
2: $\quad Flag_B \leftarrow 0^2 \| [(t-2)/2]_3 \| [q-1]_3$ // q←(15−n)
3: else
4: $\quad Flag_B \leftarrow 0 \| 1 \| [(t-2)/2]_3 \| [q-1]_3$
5: end if
6: $B_0 \leftarrow Flag_B \| N \| [p]_{8q}$
7: if $0<a<65280$ then // $65280=2^{16}-2^8$
8: $\quad B \leftarrow B_0 \| [a]_{16} \| A$
9: else if $65280 \le a < 2^{32}$ then
10: $\quad B \leftarrow B_0 \| 1^{15} \| 0 \| [a]_{32} \| A$
11: else if $2^{32} \le a < 2^{64}$ then
12: $\quad B \leftarrow B_0 \| 1^{16} \| [a]_{64} \| A$
13: end if
14: $B \rightarrow B_0 \| B_1 \| \cdots \| B_u$, where $|B_0| = \cdots = |B_{u-1}| = 128$ and $0 < w = |B_u| \le 128$

15: $B \leftarrow B \| 0^{128-w} \| P$
16: $B \rightarrow B_0 \| B_1 \| \cdots \| B_r$, where $|B_0| = \cdots = |B_{r-1}| = 128$ and $0 < s = |B_r| \leq 128$
17: $B \leftarrow B \| 0^{128-s}$

알고리즘 24의 단계 1부터 단계 6까지 과정을 통해 입력 데이터 블록 열 B의 첫 번째 블록 B_0을 구성한다. 데이터 블록 B_0의 구성은 [표 5-1]과 같다.

[표 5-1] 데이터 블록 B_0 구성

바이트 색인	0	1 … (15-q)	(16-q) … 15
내용	$Flag_B$	N	Q

알고리즘 24의 단계 1부터 단계 5까지는 데이터 블록 B_0의 첫 번째 바이트인 데이터 블록 플래그($Flag_B$)를 설정하며, 단계 6은 $Flag_B$와 초기 값 N, 그리고 p를 비트 열로 표현한 Q를 [표 5-1]과 같이 차례로 연접하여 B_0을 구성한다. Q의 바이트 길이를 q로 표기하면, q는 N의 길이에 의해 결정됨을 알 수 있다. 예를 들어, N과 P의 비트 길이가 각각 96과 4,096이라 하자. 그러면 n=12이고 따라서 q=3이다. 가정에서 p=512이므로 Q는 다음과 같다.

Q=00000000 00000010 00000000.

즉, Q는 정수 p를 q개 바이트에 표현한 비트 열 $[p]_{8q}$이다.

데이터 블록 플래그는 세 가지 설정 정보를 담고 있다. 해당 설정 정보는 부가 인증 데이터의 존재 유무 정보(Adata) 한 비트, 인증 값의 바이트 길이 t를 인코딩한 3 비트, 그리고 q를 인코딩한 3 비트로 구성된다. 인증 값 바이트 길이의 인코딩은 $[(t-2)/2]_3$으로 정의하며, q의 인코딩은 $[q-1]_3$으로 정의한다. 예를 들어, t=8이면, 011로 인코딩된다. 만약 a=0이면 Adata는 0, 그렇지 않으면 1이다. 데이터 블록 플래그의 구성은 [표 5-2]와 같다.

[표 5-2] 데이터 블록 플래그(FlagB) 구성

비트 색인	7	6	5	4	3	2	1	0
내용	0	Adata	$[(t-2)/2]_3$			$[q-1]_3$		

입력 데이터 블록 열의 첫 번째 데이터 블록 B_0은 입력 데이터의 주요 정보를 담고 있다. 예를 들어, B_0이 다음과 같다고 하자.

01101110 00010011 11010100 10100011 01011101 01110001 10100101 00000000
00000000 00000000 00000000 00000000 00000000 00000000 01000100 00000001

이로부터 입력 데이터의 정보를 추출하면 다음과 같이 정리할 수 있다.
○ Adata=1이기 때문에 부가 인증 데이터의 길이는 0이 아니다.
○ $[(t-2)/2]_3$=101이기 때문에 인증 값의 바이트 길이는 12이다.

○ Q는 바이트 길이가 7($[q-1]_3$=110)이기 때문에 다음과 같다.

00000000 00000000 00000000 00000000 00000000 01000100 00000001.

○ 따라서 Q=$[17409]_{56}$이며, 평문의 바이트 길이는 17,409이다.
○ 초기 값(N)의 바이트 길이는 8이기 때문에 N은 다음과 같다.

00010011 11010100 10100011 01011101 01110001 10100101 00000000 00000000.

입력 데이터 블록 열 B의 첫 번째 블록 B_0을 구성한 후, 알고리즘 24의 단계 7부터 단계 13까지의 과정은 부가 인증 데이터 A를 B_0에 연접한다. 만약 별도의 부가 인증 데이터가 없을 경우(a=0), 현재까지 생성한 입력 데이터 블록 열 B(=B_0)에 변경이 발생하지 않는다. 반면 부가 인증 데이터가 있을 경우(a>0), 입력 데이터 블록 열 B(=B_0)에 a를 다음과 같이 인코딩하여 연접하고, 이어서 해당 부가 인증 데이터를 데이터 블록 열에 덧붙인다.
○ 만약 $0 < a < (2^{16}-2^8)$이면, a는 $[a]_{16}$으로 인코딩한다.
○ 만약 $(2^{16}-2^8) \le a < 2^{32}$이면, a는 0xFF‖0xFE‖$[a]_{32}$로 인코딩한다.
○ 만약 $2^{32} \le a < 2^{64}$이면, a는 0xFF‖0xFF‖$[a]_{64}$로 인코딩한다.

예를 들어, 만약 $a=2^{16}$이면 a의 인코딩 결과는 다음과 같다.

11111111 11111110 00000000 00000001 00000000 00000000.

알고리즘 24의 단계 14부터 단계 17까지의 과정은 부가 인증 데이터에 이어 평문을 덧붙여 전체 입력 데이터 블록 열을 구성한다. 이 과정에서 평문을 덧붙이기 전의 입력 데이터 블록과 덧붙인 후 데이터 블록 열의 바이트 길이가 각각 16의 배수가 될 수 있도록 최소 개수의 비트 0을 덧붙인다.
입력 데이터 구조화 함수에 의해 CCM 모드의 입력 값은 다음과 같은 몇 가지 제한 조건을 가진다.

제한조건 1) $t \in \{4, 6, 8, 10, 12, 14, 16\}$
제한조건 2) $n \in \{7, 8, 9, 10, 11, 12, 13\}$
제한조건 3) $n + q = 15$
제한조건 4) $a < 2^{64}$

이로 인해 암호화 가능한 평문의 바이트 길이 p는 최대 $(2^{8q}-1)$로 제한됨을 알 수 있다. 참고로 n은 주어진 암호 키에 대해 CCM을 얼마나 실행시킬 수 있는지를 결정하는 요소이며, q는 초기 값이 결정될 때 마다 암호화 가능한 최대 평문의 길이를 결정하는 요소이다. 이들은 제한조건 3에 의해 균형 관계(tradeoff)를 가진다.
카운터 블록 생성 함수 CCM.CntGenFt는 초기 값 N과 블록 순서 값 i를 입력으로 받아 i번째 카운터 블록 CTR_i를 생성한다. 카운터 블록 CTR_i의 구성은 [표 5-3]과 같다.

[표 5-3] 카운터 블록 CTR_i 구성

바이트 색인	0	1 … (15-q)	(16-q) … 15
내용	$Flag_C$	N	$[i]_{8q}$

카운터 블록 구성에서 첫 번째 바이트에 위치하는 카운터 블록 플래그($Flag_C$)는 [표 5-4]와 같이 구성한다.

[표 5-4] 카운터 블록 플래그($Flag_C$) 구성

비트 색인	7	6	5	4	3	2	1	0
내용	0	0	0	0	0	$[q-1]_3$		

카운터 블록 플래그의 3, 4, 5번째 비트는 0으로 설정하여 [표 5-1]에서 정의한 데이터 블록 플래그($Flag_B$)와 구별될 수 있다.
카운터 블록을 128 비트 크기의 정수로 볼 때, 다음의 관계식이 성립함을 알 수 있다.

$$CTR_0=Flag_C \| N \| 0^{8q}, \quad CTR_{i+1}=(CTR_i+1) \bmod 2^{8q} \quad (0 \le i < 2^{8q}).$$

이러한 관계에 기반하여 구성한 카운터 블록 생성 함수는 알고리즘 25와 같다.

알고리즘 25 CCM 카운터 블록 생성 함수 : CTR ← CCM.CntGenFt(N,p)

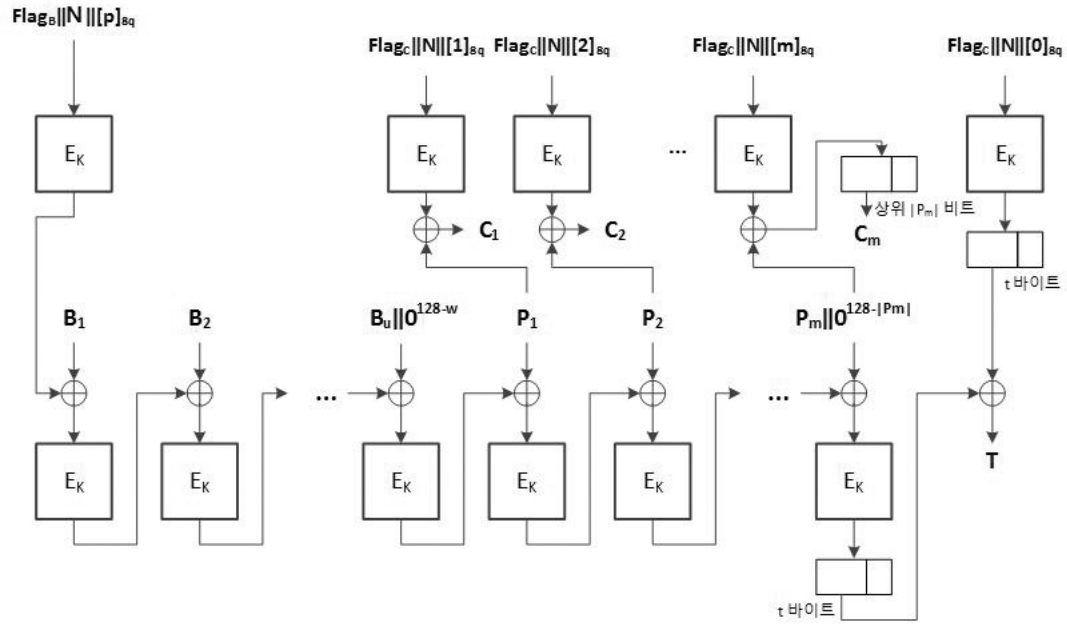
입력 : 초기 값 N, 평문 바이트 길이 p
출력 : 카운터 블록 $CTR=CTR_0 \| CTR_1 \| \cdots \| CTR_m$, $|CTR|=128$ $(0 \le i \le m(=|p/16|)$

```
1: Flag_C ← 0^5 ‖ [q−1]_3          // q←(15−n)
2: CTR_0 ← Flag_C ‖ N ‖ 0^8q
3: for i = 1 to m do
4:     CTR_i ← CTR_i−1 + 1
5: end for
```

변경 가능한 카운터 블록의 개수는 $(2^{8q}-1)$이다. 즉, 암호 키와 초기 값을 고정한 상태에서 최대 $(2^{8q}-1)\times 2^4$ 바이트 길이의 데이터를 암호화할 경우 카운터 블록의 중복이 발생하지 않는다. CCM 모드 적용 시 암호화 가능한 평문 길이는 $(2^{8q}-1)$ 바이트로 제한되므로, 충분한 개수의 중복되지 않는 카운터 블록을 생성함을 알 수 있다. 또한 초기 값을 변경할 경우 Nonce 특성에 의해 이전 카운터 블록 열과는 겹치는 블록이 없는 카운터 블록 열을 생성한다.

5.1.2 CCM 모드 암호화(CCM_LEA.Encrypt)

CCM 모드 암호화 과정은 입력 데이터 구조화 함수로부터 생성된 입력 데이터 블록 열 B에 대한 CBC-MAC((그림 4-1) 참조)과 평문 P에 대한 CTR 모드 계산의 조합으로 구성되어 있으며, 단순하고 직관적인 구조를 가지고 있다. CCM 모드 암호화 과정의 구체적인 구조를 도시하면 (그림 5-1)과 같다.



(그림 5-1) CCM 모드 암호화 과정

CCM 모드 암호화 과정의 구체적인 절차는 알고리즘 26과 같다.

알고리즘 26 CCM 암호화 과정 : $C_{CCM}(=C_{CTR} \| T) \leftarrow$ CCM_LEA.Encrypt(K,N,A,P,t)

입력 : 암호 키 K, 초기 값 N, 부가 인증 데이터 A, 평문 P, 인증 값 바이트 길이 t
출력 : C_{CTR}, 인증 값 T ($|C_{CTR}|=|P|$, $|T|=8\times t$)

1: $(B_0 \| B_1 \| \cdots \| B_r) \leftarrow$ CCM.InputFormatFt(N,A,P,t)
2: $(CTR_0 \| CTR_1 \| \cdots \| CTR_m) \leftarrow$ CCM.CntGenFt(N,|P|/8) // $m = \| p/16 \|$
3: RK ← LEA.EncKeyScheduleǀKǀ(K)
4: Y ← LEA.Encrypt(B_0,RK)
5: for i = 1 to r do
6: $X \leftarrow B_i \oplus Y$
7: Y ← LEA.Encrypt(X,RK)
8: end for
9: $S_0 \leftarrow$ LEA.Encrypt(CTR_0,RK)
10: $T \leftarrow MSB_{Tlen}(Y \oplus S_0)$ // Tlen = 8×t
11: for i = 1 to (m−1) do
12: $C_i \leftarrow P_i \oplus$ LEA.Encrypt(CTR_i,RK)
13: end for
14: $S_m \leftarrow$ LEA.Encrypt(CTR_m,RK)
15: $C_m \leftarrow P_m \oplus MSB_{|Pm|}(S_m)$
16: $C_{CTR} \leftarrow C_1 \| \cdots \| C_m$

알고리즘 26의 단계 4부터 단계 8까지 과정은 입력 데이터 블록 열 B에 대한 CBC-MAC 계산이다. 그리고 단계 9와 단계 10은 최초 카운터 블록 CTR_0을 암호화한 후 CBC-MAC 계산 결과와 XOR 연산한 결과를 인증 값으로 설정하는 과정이다. 이어서 단계 11부터 단계 15까지 과정은 CTR 모드로 평문 P를 암호화한다. CTR 모드에 의해 생성된 C_{CTR}과 CBC-MAC 계산을 통해 생성된 인증 값 T를 연접한 $C_{CTR} \| T$를 CCM 암호화 과정의 결과 값 C_{CCM}으로 반환한다.

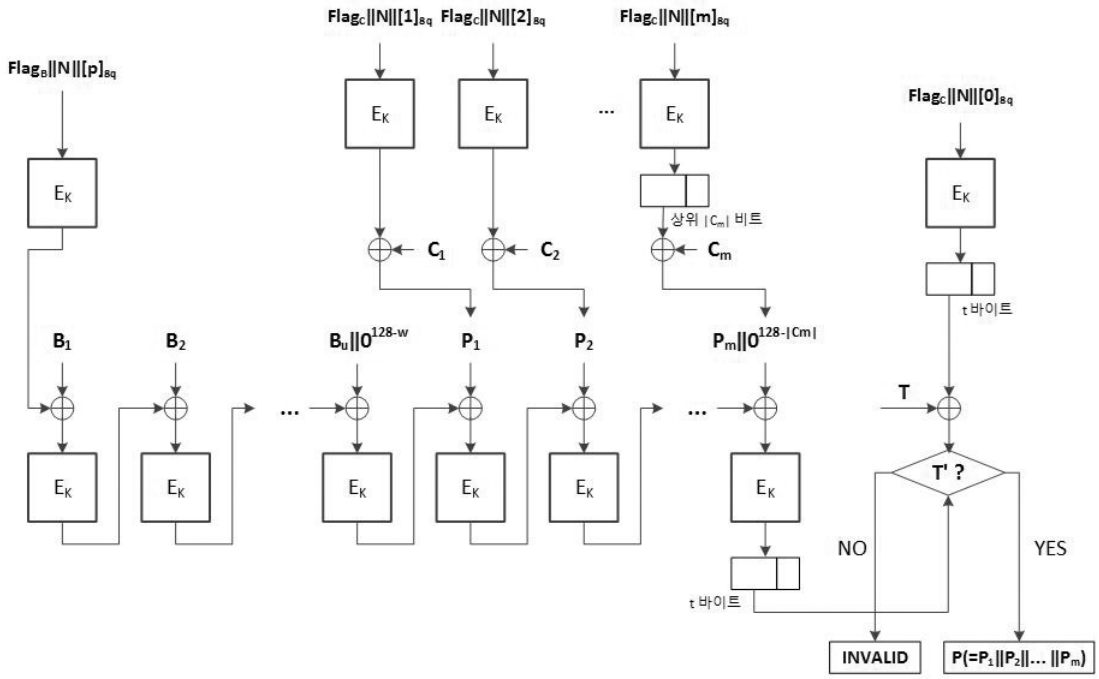
입력 데이터 구조화 함수 CCM.InputFormatFt를 CCM 모드의 암호화 과정에 적용하는 것과 관련하여 효율성 측면에서의 고려사항은 다음과 같다.

① 평문의 길이와 부가 인증 데이터의 길이를 사전에 결정하도록 강제하므로, 데이터 처리가 끝날 때 까지 데이터 길이 정보를 식별하기 어려운 환경(스트리밍 등)에는 CCM 모드를 적용하는 것이 적합하지 않다.
② 첫 번째 데이터 블록에 초기 값과 평문의 길이 정보가 포함되므로, 다수의 데이터에서 부가 인증 데이터가 고정된 값을 가지는 경우에 고려할 수 있는 CBC-MAC의 사전 계산을 적용하기 어렵다.

따라서 CCM 모드 적용에 있어서 운용 환경의 특성을 면밀히 파악할 필요가 있다. 참고로 소절 5.1.1에 정의된 입력 데이터 구조화 함수를 대체할 수 있는 방식은 아직 알려진 바 없다.

5.1.3 CCM 모드 복호화(CCM_LEA.Decrypt)

CCM 모드 복호화 과정을 도시하면 (그림 5-2)와 같다.



(그림 5-2) CCM 모드 복호화 과정

(그림 5-2)에서 볼 수 있듯이 CBC-MAC 계산을 위한 입력 데이터 블록 열은 평문을 포함하고 있다. 따라서 CTR 모드를 이용한 C_{CTR}의 복호화는 CBC-MAC 계산에 선행하거나 혹은 블록 단위로 CBC-MAC 계산보다 앞서 진행되어야 한다.

CCM 모드 복호화 과정의 구체적인 절차는 알고리즘 27과 같다.

알고리즘 27의 단계 5부터 단계 10까지 과정은 C_{CTR}을 CTR 모드로 복호화하여 평문 후보 P를 얻는다. 초기 값 N, 부가 입력 데이터 A와 함께 P를 입력으로 단계 15에서는 CBC-MAC 계산을 위한 데이터 블록 열 B를

구성한다. 이어 단계 16부터 단계 20까지 과정은 CBC-MAC을 계산하여 확인 값 T′을 생성하고, 수신된 인증 값 T와 일치 여부를 단계 22에서 확인한다. (이때 수신된 인증 값은 엄밀하게는 단계 4에서 최초 카운터 블록(CTR_0)의 LEA 암호화 함수 계산 결과와의 XOR 연산 결과 값이다.) 만일 두 값이 일치할 경우에는 평문 후보 P를 CCTR에 대응하는 평문으로 반환하고, 아닐 경우에는 오류 코드 ⊥를 반환한다.

알고리즘 27 CCM 복호화 과정 : $\{P,\perp\} \leftarrow$ CCM_LEA.Decrypt(K,N,A,C_{CCM})

입력 : 암호 키 K, 초기 값 N, 부가 인증 데이터 A, 암호문 $C_{CCM}(=C_{CTR} \| T)$
출력 : 평문 P 또는 오류 코드 ⊥

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $(CTR_0 \| CTR_1 \| \cdots \| CTR_m) \leftarrow CCM.CntGenFt(N,|C_{CTR}|/8)$ // $m = \| |C_{CTR}|/128 \|$
3: $S_0 \leftarrow LEA.Encrypt(CTR_0,RK)$
4: $T \leftarrow T \oplus MSB_{Tlen}(S_0)$ // Tlen = |T|
5: $C_{CTR} \rightarrow C_1 \| \cdots \| C_m$, where $|C_1| = \cdots = |C_{m-1}| = 128$ and $0 < |C_m| \leq 128$
6: for i = 1 to (m−1) do
7: $P_i \leftarrow C_i \oplus LEA.Encrypt(CTR_i,RK)$
8: end for
9: $S_m \leftarrow LEA.Encrypt(CTR_m,RK)$
10: $P_m \leftarrow C_m \oplus MSB_{|Cm|}(S_m)$
11: $P \leftarrow P_1 \| \cdots \| P_m$
12: if N, A or P is not valid, then // 소절 5.1.1 참조
13: halt and return '⊥'
14: end if
15: $(B_0 \| B_1 \| \cdots \| B_r) \leftarrow CCM.InputFormatFt(N,A,P,t)$
16: $Y \leftarrow LEA.Encrypt(B_0,RK)$
17: for i = 1 to r do
18: $X \leftarrow B_i \oplus Y$
19: $Y \leftarrow LEA.Encrypt(X,RK)$
20: end for
21: $T_1 \leftarrow MSB_{Tlen}(Y)$
22: if $T \neq T_1$ then
23: return '⊥'
24: else
25: return P
26: end if

5.2 GCM(Galois/Counter Mode)

GCM은 3.5절에서 소개한 CTR 모드에 유한체 곱셈 연산을 이용한 메시지 인증 코드를 결합한 구조를 가진다. 유한체 곱 연산은 $GF(2^{128})$상에서 이루어지며, 이와 같은 특성에 따라 특히 하드웨어 구현에 적합하다. GCM의 메시지 인증 코드는 CBC-MAC의 연결 구조와 유사한 구조로, LEA 암호화 함수 대신 특정 값을 곱하는 연산을 적용하며 이를 GHASH 함수라고 부른다.

본 절에서는 LEA 운영 모드 표준에서 정의하는 GCM의 GHASH 함수를 소절 5.2.1에 정리하고, 암호화 과정과 복호화 과정은 각각 소절 5.2.2와 소절 5.2.3에 정리한다. 그리고 GCM의 메시지 인증 코드 용례인 GMAC에 대해서 소절 5.2.4에서 간략하게 정리한다.

5.2.1 GHASH 함수

GHASH 함수는 GCM의 메시지 인증 과정을 구성하는 요소로, 다음과 같이 정의한다.

$$GHASH(H,M) = (M1 \bullet H^{m}) \oplus (M2 \bullet H^{m-1}) \oplus (M_3 \bullet H^{m-2}) \oplus \cdots \oplus (M_{m-1} \bullet H^{2}) \oplus (M_m \bullet H),$$
$$\text{where } M=M_1 \| M_2 \| \cdots \| M_m \ (|M_i|=128,\ 1 \le i \le m).$$

GHASH 함수에서 사용되는 유한체 $GF(2^{128})$상에서의 곱셈 연산 '•'는 감산 다항식

$$g(u) = u^{128} + u^{7} + u^{2} + u + 1$$

를 이용하여 알고리즘 28과 같이 표현할 수 있다.

알고리즘 28 128 비트 유한체 곱셈 $W \leftarrow U \bullet V$

입력 : U (128 비트), $V=v0 \| v_1 \| \cdots \| v_{127}$ (128 비트, 각 v_i는 1 비트)
출력 : W (128 비트)

```
1:  W ← 0^128
2:  Z ← U
3:  for i = 0 to 127 do
4:      if v_i=1 then
5:                  W ← W ⊕ Z
6:      end if
7:
    if z_127=0 then
8:                  Z ← (Z≫1)
9:      else
10:
                    Z ← (Z≫1) ⊕ (11100001 ‖ 0^120)
11:     end if
12:  end for
```

GHASH 함수는 알고리즘 29와 같으며, (그림 4-1)에 도시한 CBC-MAC의 연결 방식과 동일하게 표현 가능함을 확인할 수 있다.

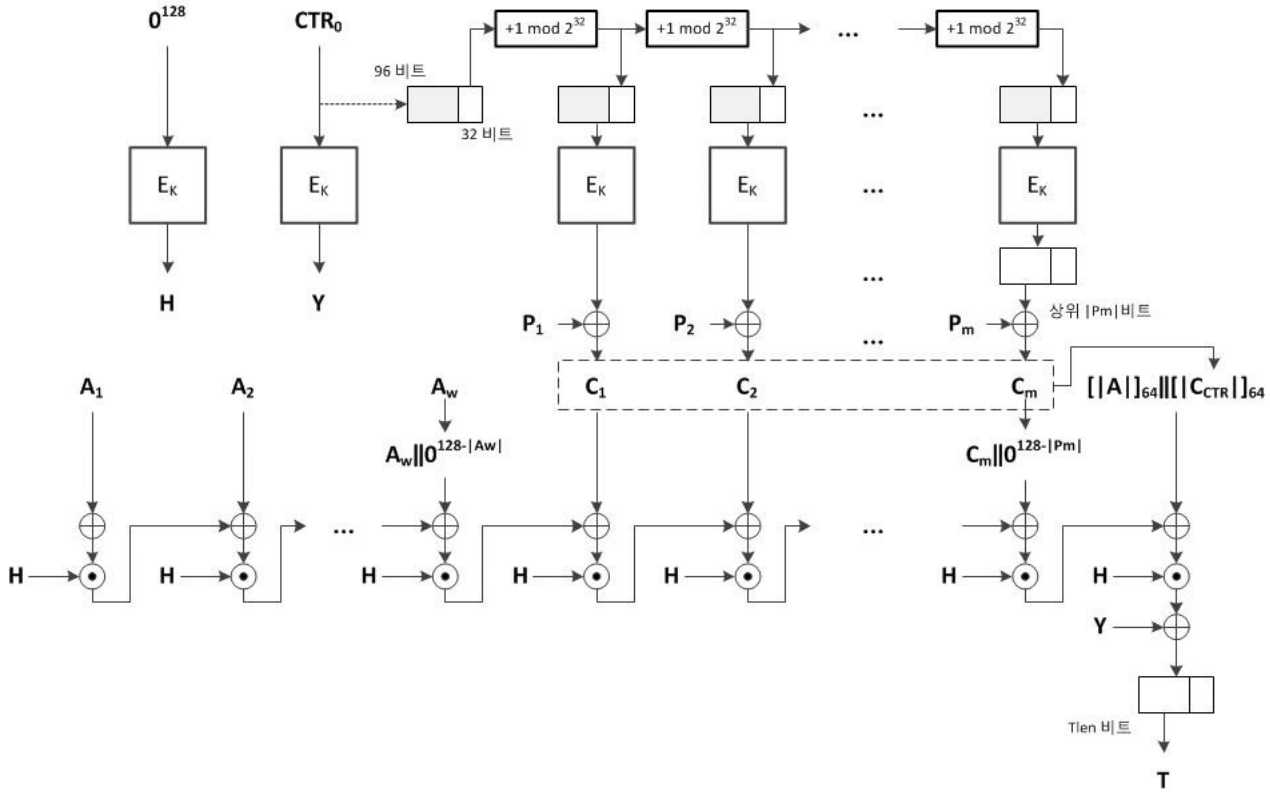
알고리즘 29 $Y \leftarrow GHASH(H,M)$

입력 : H (128 비트), $M=M_1 \| M_2 \| \cdots \| M_m$ ($|M_i|=128,\ 1 \le i \le m$)
출력 : Y (128 비트)

```
1:  Y ← 0^128
2:  for i = 1 to m do
3:      Y ← (Y ⊕ M_i) • H                    // 알고리즘 28
4:  end for
```

5.2.2 GCM 암호화(GCM_LEA.Encrypt)

GCM의 암호화 과정은 평문 P에 대한 CTR 모드 계산으로 CCTR을 생성하고, 부가 인증 데이터와 연접한 후 GHASH 함수를 거쳐 인증 값 T를 생성하는 구조이다. GCM 암호화 과정의 구체적인 구조를 도시하면 (그림 5-3)과 같다.



(그림 5-3) GCM 암호화 과정

GCM 암호화 과정의 입력인 초기 값은 하나의 암호 키에 대하여 중복되지 않도록 $(2^{64}-1)$ 비트 이하의 비트열로 선택하여 사용한다. 부가 인증 데이터의 길이도 $(2^{64}-1)$ 비트 이하가 되도록 하며, 평문의 길이는 카운터 블록의 개수3)를 고려하여 $(2^{39}-256)$ 비트 이하로 제한된다.

초기 값은 Nonce 특성을 가지는 것으로 충분하며, CTR 모드의 카운터 블록 생성에 사용된다. GCM에서는 초기 값의 비트 길이가 96인 경우와 아닌 경우에 대해 다르게 카운터 블록을 설정한다.

GCM의 카운터 블록 생성 함수 GCM.CntGenFt는 알고리즘 30과 같다.

알고리즘 30 GCM 카운터 블록 생성 함수 : CTR ← GCM.CntGenFt(H,N, m)

입력 : GHASH 키 H, 초기 값 N, 카운터 블록 개수 m
출력 : 카운터 블록 $CTR=CTR_0 \| CTR_1 \| \cdots \| CTR_m$, $|CTR_i|=128$ $(0 \le i \le m)$

1: if |N|=96 then
2: $CTR_0 \leftarrow N \| 0^{31} \| 1$
3: else
4: $s \leftarrow 128 \| |N|/128 \| -|N|$
5: $CTR_0 \leftarrow GHASH(H,N \| 0^{s+64} \| [|N|]_{64})$ // 알고리즘 29 참조
6: end if
7: for i = 1 to m do
8: $CTR_i \leftarrow MSB_{96}(CTR_{i-1}) \| [(BitToInt(LSB_{32}(CTR_{i-1}))+1) \bmod 2^{32}]_{32}$
9: end for

3) 알고리즘 30 참조

알고리즘 30의 단계 1부터 단계 6까지의 과정은 초기 카운터 블록 CTR_0을 설정한다. 초기 값의 비트 길이가 96일 경우 단계 2와 같이 단순하게 설정할 수 있지만, 이외의 경우는 단계 5와 같이 GHASH 함수의 결과 값으로 설정한다. 따라서 초기 값의 비트 길이를 96으로 제한하는 것이 효율성 측면에서 유리하다. 또한 초기 값의 비트 길이를 96으로 고정하는 것이 가변으로 사용하는 것에 비해 안전성이 높다는 것이 밝혀졌다[35].

알고리즘 30의 단계 8에서 볼 수 있듯이, 카운터 블록은 하위 32 비트에 대해서만 순차적으로 1씩 증가하며, 캐리가 발생할 경우 무시한다.

GCM의 카운터 블록 생성 함수에 기반한 GCM 암호화 과정은 알고리즘 31과 같다.

알고리즘 31 GCM 암호화 과정 : $C_{GCM}(=C_{CTR} \| T) \leftarrow$ GCM_LEA.Encrypt(K,N,A,P,Tlen)

입력 : 암호 키 K, 초기 값 N, 부가 인증 데이터 $A=A_1 \| \cdots \| A_w(|A_i|=128(1 \le i \le (w-1)), 0 < |Aw| \le 128)$, 평문 P, 인증 값 비트 길이 Tlen

출력 : C_{CTR}, 인증 값 T ($|C_{CTR}|=|P|$, $|T|=Tlen$)

1: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
2: $H \leftarrow LEA.Encrypt(0^{128},RK)$
3: $(CTR_0 \| CTR_1 \| \cdots \| CTR_m) \leftarrow GCM.CntGenFt(H,N,\ m)$ // $m=\| |P|/128 \|$
4: $Y \leftarrow LEA.Encrypt(CTR_0,RK)$
5: if $|P| > 0$ then
6: $P \rightarrow P_1 \| \cdots \| P_m$, where $|P_1| = \cdots = |P_{m-1}| = 128,\ 0 < |P_m| \le 128$
7: for i = 1 to (m-1) do
8: $C_i \leftarrow P_i \oplus LEA.Encrypt(CTR_i,RK)$
9: end for
10: $C_m \leftarrow P_m \quad MSB_{128-r}(LEA.Encrypt(CTR_m,RK))$ // $r=128-|P_m|$
11: end if
12: $C_{CTR} \leftarrow C_1 \| \cdots \| C_m$
13: $S \leftarrow GHASH(H,A \| 0^k \| C_{CTR} \| 0^r \| [|A|]_{64} \| [|C_{CTR}|]_{64})$ // $k = 128-|A_w|$
14: $T \leftarrow MSB_{Tlen}(Y \oplus S)$

알고리즘 31의 단계 2에서는 GHASH 키 H를 생성하며, H는 암호화 과정 전체에서 GHASH 함수의 고정 입력 값으로 사용된다. 알고리즘 31의 단계 3에서는 H와 초기 값 N을 입력으로 카운터 블록을 생성한다. 알고리즘 31의 단계 5부터 단계 11까지 과정은 평문 P를 CTR 모드로 암호화하고, 단계 4와 단계 13, 그리고 단계 14는 GCM의 인증 값 T를 생성한다. 단계 13에서 GHASH 함수에 입력된 메시지는 부가 인증 데이터 A와 C_{CTR}, 그리고 두 데이터의 길이 정보로 구성된다. 이때 (그림 5-3)에서와 같이 A와 C_{CTR}이 128 비트 블록 단위로 구분하여 처리될 수 있도록 필요한 최소 개수의 비트 0을 덧붙인다. 따라서 GHASH 함수 입력 메시지의 마지막 128 비트 블록은 A와 C_{CTR}의 길이 정보로 구성된 $[|A|]_{64} \| [|C_{CTR}|]_{64}$이다.

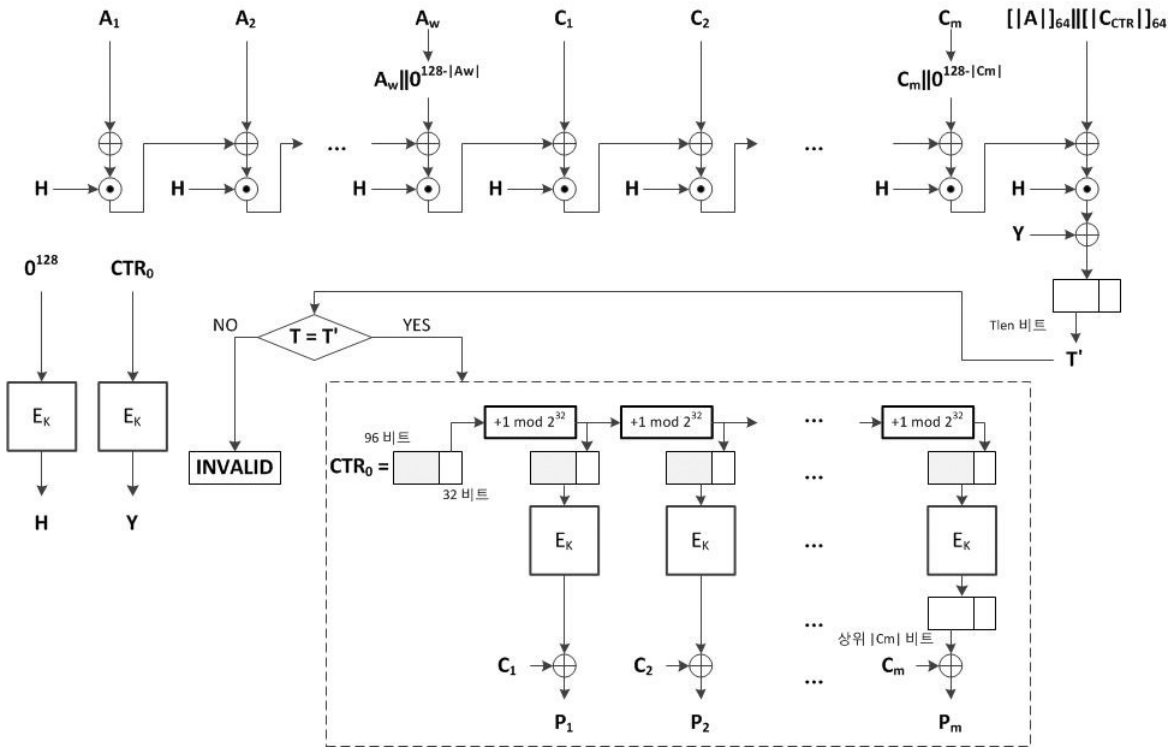
인증 값의 비트 길이인 Tlen은 96 이상 128 이하 중 8의 배수로 정한다. 짧은 길이의 인증 값 사용과 관련한 안전성 측면의 문제점[32,24]을 고려하여, LEA 운영 모드 표준에서는 96 비트 미만의 인증 값 사용은 권고하지 않는다.

GCM 암호화 과정에서는 사전에 평문과 부가 인증 데이터에 대한 길이 정보를 필요로 하지 않는다. 알고리즘 31의 단계 3에서는 평문 P의 길이 정보를 이용하여 카운터 블록을 사전에 생성하는 것으로 표현하였지만, 실제 구현에 있어서는 알고리즘 30과 알고리즘 31을 동시에 적용하여 알고리즘 19와 같이 평문 블록 처리

과정에서 카운터 블록을 생성하는 것이 일반적이다. 따라서 알고리즘 31의 단계 13에서만 평문과 부가 인증 데이터에 대한 길이 정보가 필요하며, 이는 입력 데이터의 크기 정보를 저장하는 내부 상태 변수를 이용해 계산할 수 있다. 반면 초기 값이 예측 가능하고 평문의 길이 정보가 사전에 알려질 경우, 알고리즘 31의 단계 3과 같이 키 수열을 미리 계산함으로써 암ㆍ복호화 처리 성능을 향상시킬 수 있다.
그리고 다수 데이터에서 부가 인증 데이터 A가 고정된 값을 가질 경우, $A \| 0^k$에 대해 GHASH 함수를 사전에 계산하여 그 결과를 내부 상태 값으로 저장할 수 있다. 이 값은 C_{CTR}이 결정될 때 알고리즘 31의 단계 13에 있는 GHASH 계산에 사용함으로써 인증 값 계산 속도를 향상시킬 수 있다.

5.2.3 GCM 복호화(GCM_LEA.Decrypt)

GCM 복호화 과정을 도시하면 (그림 5-4)와 같다.



(그림 5-4) GCM 복호화 과정

(그림 5-4)에서 볼 수 있듯이, 수신된 인증 값에 대한 검증 절차는 C_{CTR}의 복호화 과정과 독립적으로 진행될 수 있다. 이러한 특징은 GCM 복호화 과정의 구현에 있어 인증 값 검증에 실패할 경우 CCTR의 복호화 절차를 수행하지 않는 일종의 조기 중단 정책(Early-abort strategy)을 가능하게 하여 효율성 측면의 개선을 도모할 수 있다.
GCM 복호화 과정의 구체적인 절차는 알고리즘 32와 같다.

알고리즘 32 GCM 복호화 과정 : $\{P,\perp\} \leftarrow$ GCM_LEA.Decrypt(K,N,A,C_{GCM}, Tlen)

입력 : 암호 키 K, 초기 값 N, 부가 인증 데이터 $A=A_1 \| \cdots \| A_w(|A_i|=128(1\le i\le(w-1)),$ $0<|A_w|\le 128)$, 암호문 $C_{GCM}(=C_{CTR} \| T)$, 인증 값 길이 Tlen
출력 : 평문 P 또는 오류 코드 $\perp$

1: if |N|, |A| or $|C_{CTR}|$ are not supported, or $|T|\neq$Tlen then
2: halt and return '$\perp$'
3: end if
4: $RK \leftarrow LEA.EncKeySchedule_{|K|}(K)$
5: $H \leftarrow LEA.Encrypt(0^{128},RK)$
6: $(CTR_0 \| CTR_1 \| \cdots \| CTR_m) \leftarrow GCM.CntGenFt(H,N,m)$ // $m=\| |C_{CTR}|/128 \|$
7: $Y \leftarrow LEA.Encrypt(CTR_0,RK)$
8: if $|C_{CTR}|>0$ then
9: $C_{CTR} \rightarrow C_1 \| \cdots \| C_m$, where $|C_1|=\cdots=|C_{m-1}|=128,\ 0<|C_m|\le 128$
10: for i = 1 to (m-1) do
11: $P_i \leftarrow C_i \oplus LEA.Encrypt(CTR_i,RK)$
12: end for
13: $P_m \leftarrow C_m \oplus MSB_{128-r}(LEA.Encrypt(CTR_m,RK))$ // $r=128-|C_m|$
14: end if
15: $S \leftarrow GHASH(H,A \| 0^k \| C_{CTR} \| 0^r \| [|A|]_{64} \| [|C_{CTR}|]_{64})$ // $k = 128-|A_w|$
16: $T' \leftarrow MSB_{Tlen}(Y \oplus S)$
17: if $T' \neq T$ then
18: return '$\perp$'
19: else
20: return $P \leftarrow P_1 \| \cdots \| P_m$
21: end if

알고리즘 32의 단계 5에서는 GHASH 키 H를 생성하며, H는 복호화 과정 전체에서 GHASH 함수의 고정 입력 값으로 사용된다. 알고리즘 32의 단계 6에서는 H와 초기 값 N을 입력으로 카운터 블록을 생성한다. 알고리즘 32의 단계 8부터 단계 14까지 과정은 C_{CTR}을 CTR 모드로 복호화하고, 단계 7과 단계 15, 그리고 단계 16은 수신된 GCM의 인증 값 T의 유효성을 확인하기 위해 확인 값 T'을 생성한다. 단계 17에서 수신된 인증 값 T와 확인 값 T'의 일치 여부를 확인하여, 일치할 경우에는 평문 후보 P를 C_{CTR}에 대응하는 평문으로 반환하고, 아닐 경우에는 오류 코드 $\perp$를 반환한다.
암호화 과정에서도 설명한 바와 같이, 다수 데이터에서 부가 인증 데이터 A가 고정된 값을 가질 경우, $A \| 0^k$에 대해 GHASH 함수를 사전에 계산하여 확인 값 T' 계산에 활용할 수 있다.

5.2.4 GMAC

GCM은 CTR 모드를 이용한 암호화 과정을 생략하고 인증 값만 생성하는 형태로 운용할 수 있다. 이러한 운용 방식을 GCM과 구분하여 GMAC이라고 한다. 4절에서 언급한 바와 같이, GMAC은 Nonce 특성을 가지는 초기 값을 입력받아서 사용한다. 따라서 암호 키가 동일하더라도 초기 값이 다를 경우, 동일한 메시지에 대해서 다른 인증 값을 생성하게 된다.

참고 문헌

[1] TTA. 128 비트 블록 암호 LEA. TTAK.KO-12.0223, 2013.

[2] TTA. 128 비트 블록 암호 LEA 운영 모드. TTAK.KO-12.0246, 2014.

[3] TTA. n 비트 블록 암호 운영 모드 - 제1부 일반. TTAK.KO-12.0271-Part1/R1, 2016.

[4] TTA. n 비트 블록 암호 운영 모드 - 제2부 블록 암호 LEA. TTAK.KO-12.0271- Part2, 2015.

[5] IETF. PKCS #5: Password-Based Cryptography Specification Version 2.0. IETF RFC 2898, 2000.

[6] IETF. The Secure Real-time Transport Protocol (SRTP). IETF RFC 3711, 2004.

[7] IETF. IP Authentication Header. IETF RFC 4302, 2005.

[8] IETF. IP Encapsulating Security Payload (ESP). IETF RFC 4303, 2005.

[9] IETF. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[10] IETF. Cryptographic Message Syntax (CMS). IETF RFC 5652, 2009.

[11] IETF. A Description of the ARIA Encryption Algorithm. IETF RFC 5794, 2010.

[12] IETF. Internet Key Exchange Protocol Version 2 (IKEv2). IETF RFC 7296, 2014.

[13] ISO. Information technology - Security techniques - Message Authentication Codes (MACs) - Part 1: Mechanisms using a block cipher. ISO/IEC 9797-1, 2011.

[14] ISO. Information technology - Security techniques - Modes of operation for an n-bit block cipher. ISO/IEC 10116, 2006.

[15] ISO. Information technology - Security techniques - Encryption algorithms - Part 2: Asymmetric ciphers. ISO/IEC 18033-2, 2006.

[16] ISO. Information technology - Security techniques - Encryption algorithms - Part 3: Block ciphers. ISO/IEC 18033-3, 2010.

[17] ISO. Information technology - Security techniques - Authenticated encryption. ISO/IEC 19772, 2009.

[18] ISO. Information technology - Security techniques - Lightweight cryptography - Part 2: Block ciphers. ISO/IEC 29192-2, 2012.

[19] NIST. Advanced Encryption Standard. NIST FIPS 197, 2001.

[20] NIST. Report on Lightweight Cryptography. NIST IR 8114 (DRAFT), 2016.

[21] NIST. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. NIST SP 800-38A, 2001.

[22] NIST. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. NIST SP 800-38B, 2005.

[23] NIST. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST SP 800-38C, 2004.

[24] NIST. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST SP 800-38D, 2007.

[25] M.R. Albrecht, K.G. Paterson, and G.J. Watson. Plaintext recovery attacks against SSH. Proc. of IEEE S&P 2009, pp. 16-26, 2009.

[26] N.J. AlFardan and K.G. Paterson. Plaintext-recovery attacks against datagram TLS. Proc. of NDSS, 2012.

[27] N.J. AlFardan and K.G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. Proc. of IEEE S&P 2013, pp. 526-540, 2013.

[28] M. Bellare, A. Dasai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. Proc. of IEEE FOCS' 97, pp. 394-403, 1997.

[29] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. Journal of Computer and Systems Sciences, vol. 61(3): 362-399 (2000).

[30] J.P. Degabriele and K.G. Paterson. Attacking the IPsec standards in encryption-only configurations. Proc. of IEEE S&P 2007, pp. 335-349, 2007.

[31] J.P. Degabriele and K.G. Paterson. On the (in)security of IPsec in MAC-then-Encrypt configuration. Proc. of ACM CCS 2010, pp. 493-504, 2010.

[32] N. Ferguson. Authentication weaknesses in GCM. Manuscript, 2005.

[33] D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K.H. Ryu, and D.-G. Lee. LEA: A 128-bit block cipher for fast encryption on common processors. Proc. of WISA 2013, LNCS, vol. 8267, pp. 3-27, 2014.

[34] T. Iwata and K. Kurosawa. OMAC: One-key CBC MAC. Proc. of FSE 2003, LNCS, vol. 2887, pp. 129-153, 2003.

[35] T. Iwata, K. Ohashi, and K. Minematsu. Breaking and repairing GCM security proofs. Proc. of CRYPTO 2012. LNCS, vo. 7417, pp. 31-49, 2012.

[36] H. Kang, M. Park, D. Moon, C. Lee, J. Kim, K. Kim, and S. Hong. New efficient padding methods secure against padding oracle attacks. Proc. of ICISC 2015, LNCS, vol. 9558, pp. 329-342, 2016.

[37] D. Kwon, J. Kim, S. Park, S.H. Sung, Y. Sohn, J.H. Song, Y. Yeom, E.-J. Yoon, S. Lee, J. Lee, S. Chee, D. Han, and J. Hong. New block cipher: ARIA. Proc. of ICISC 2003, LNCS, vol. 2971, pp. 432-445, 2004.

[38] K. Minematsu, S. Lucks, H. Morita, and T. Iwata. Attacks and security proofs of EAX-prime. Proc. of FSE 2013, LNCS, vol. 8424, pp. 327-347, 2014.

[39] P. Rogaway. Evaluation of some blockcipher modes of operation. Evaluation Report of CRYPTREC, 2011.

[40] H. Seo, Z. Liu, J. Choi, T. Park, and H. Kim. Compact implementations of LEA block cipher for low-end microprocessors. Proc. of WISA 2015, LNCS, vol. 9501, pp. 28-40, 2016.

[41] H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi, and H. Kim. Parallel implementations of LEA. Proc. of ICISC 2013, LNCS, vol. 8565, pp. 256-276, 2014.

[42] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y. Nogami, Y. Zhu, and H.W. Kim. Parallel implementations of LEA, revisited. Proc. of WISA 2016, LNCS, to appear.

[43] L. Song, Z. Huang, and Q. Yang. Automatic differential analysis of ARX block ciphers: with application to SPECK and LEA. Proc. of ACISP 2016(2), LNCS, vol. 9723, pp. 379-394, 2016.

[44] H. Sui, W. Wu, L. Zhang, and P. Wang. Attacking and fixing the CS mode. Proc. of ICICS 2013, LNCS, vol. 8233, pp. 318-330, 2013.

[45] S. Vaudenay. Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS... Proc. of EUROCRYPT 2002, LNCS, vol. 2332, pp. 534-546, 2002.

[46] A.K.L. Yau, K.G Paterson, and C.J. Mitchell. Padding oracle attacks on the CBC-mode encryption with random and secret IVs. Proc. of FSE 2005, LNCS, vol. 3557, pp. 299-319, 2005

아래의 QR 코드는 본 표준 해설서에
대한 만족도 조사입니다.
많은 참여 부탁드립니다.

128 비트 블록 암호 LEA 및 운영모드 표준 해설서

2016년도 12월 20일 인쇄
2016년도 12월 30일 발행

발 행 소 : 한국정보통신기술협회 (TTA)
발 행 인 : 박 재 문
발간번호 : TTA-16133-SD
인 쇄 인 : [ADWORLD | 애드월드] Tel. 02-2271-0369

TTA 한국정보통신기술협회
Telecommunications Technology Association
[13591] 경기도 성남시 분당구 분당로 47
TEL (031)724-0114 http://www.tta.or.kr