암호알고리즘 1부: 대칭 암호알고리즘

NOTE 08

한국기술교육대학교 컴퓨터공학부 김상진

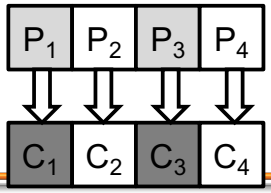
sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

교육목표

- 대칭 암호알고리즘
  - 암호화 모드: ECB, CBC, CTR
    - 채우기에 대한 이해
      - 암호문 훔침 기법
  - 인증 암호화 모드: GCM, CCM, EAX
  - DES
  - Triple-DES
  - AES
  - Salsa20

암호화 모드 (1/3)

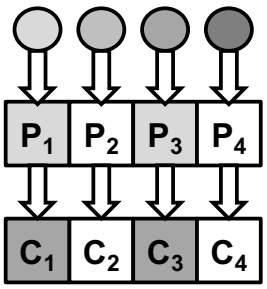
- 대칭 암호알고리즘은 크게 블록과 스트림 암호방식으로 분류함
  - 블록 암호방식: 메시지를 일정한 크기로 나누어 각 블록을 암호화하는 방식
    - 반드시 정해진 크기의 입력을 암호화 또는 복호화 함수에 제공해야 함
    - 보통 결정적 알고리즘임
  - 스트림 암호방식: 한 번에 XOR 연산을 이용하여 한 바이트를 암호화하는 방식
- 블록 암호방식에서 블록보다 큰 메시지를 암호화하는 방법이 필요함
  - 이 방법을 암호화 모드(cryptographic mode)라 함
  - 가장 기본적인 방법: 블록 단위로 메시지를 나누어 각 블록을 독립적으로 암호화하는 방법
    - 이 모드를 ECB(Electronic CodeBook) 모드라 함
    - 문제점. 이 모드를 사용하면 같은 평문 블록은 항상 같은 암호문 블록으로 암호화됨



평문 패턴이 암호문에 나타남

암호화 모드 (2/3)

- 이 문제를 극복하기 위해 현재 사용하는 대부분의 암호화 모드는 암호알고리즘, 피드백(feedback), 단순 연산으로 구성됨
  - 요구사항. 암호화 모드를 도입하여도 기존 알고리즘의 성능에 큰 영향을 주어서는 안 됨
  - 피드백: 한 블록을 독립적으로 암호화하지 않고, 추가적인 요소를 사용하여 암호화하여 평문 블록이 같더라도 결과가 달라지도록 해주는 요소
    - 이를 통해 ECB 모드의 문제점을 극복할 수 있음



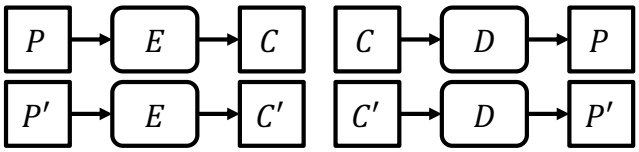
각 블록마다 다른 피드백

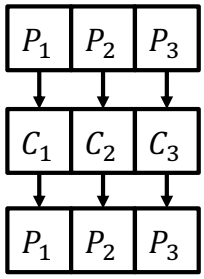
피드백과 평문 블록을 결합(?)하는 비용 발생
이 비용은 매우 저렴해야 함

복호화할 때 제거할 수 있어야 함

암호화 모드 (3/3)

- 암호화 모드의 분석
  - 평문오류: 암호화하기 전에 평문에 오류가 있을 때 암호문에 미치는 파급효과
    - 유사 평문의 암호화 결과를 보기 위함
    - 암호 블록(↑)에 미치는 영향과 전체 암호문(↑)에 미치는 영향을 분석





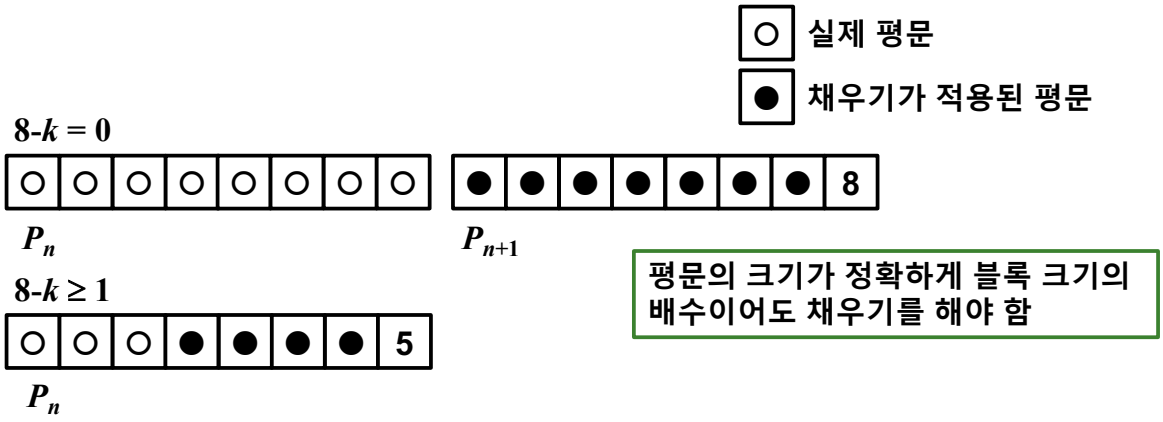
  - 암호문 조작(블록교체, 암호문 오류): 암호화된 암호문에서 두 암호문 블록을 바꾸거나 암호문 블록을 조작한 경우에 복호화한 결과
    - 평문 블록(↑)에 미치는 영향과 전체 평문(↓)에 미치는 영향을 분석
    - 공격자가 의미있는 변화를 줄 수 있는지 여부 (NM 특성)
  - 추가 보안 문제: 모드 사용에 따른 추가적인 보안 문제의 유무
  - 효율성: 모드 연산 비용, 병행 수행 가능 여부 등

채우기 (1/3)

- 평문 메시지의 크기가 정확하게 블록 크기의 배수가 아닌 경우에 필요
  - 채우기(padding): 완전하지 않은 마지막 블록 끝에 규칙적인 일련의 비트(예: 모두 0, 모두 1)를 추가하여 완전한 블록을 만드는 것
- 암호화할 때 사용하는 채우기의 요구사항
  - 복호화한 사용자는 채워진 부분을 정확하게 제거할 수 있어야 함
- 위 요구사항을 충족하는 두 가지 방법
  - 방법 1. 암호문과 별도로 원 평문의 크기를 전달
    - 보통 프로토콜에서 교환하는 메시지의 크기가 고정되어 있고, 참여자가 알고 있음
    - 보통 채우기(여분 정보)를 통해 메시지의 조작 여부도 확인함
  - 방법 2. 채우기를 한 부분에 채운 부분을 제거할 수 있는 요소를 포함
    - 예) 비트 채우기
      - 평문 뒤에 첫 비트는 1로 하고 그 다음부터는 모두 0으로 채움
    - 채울 필요가 없어도 항상 채워야 함

채우기 (2/3)

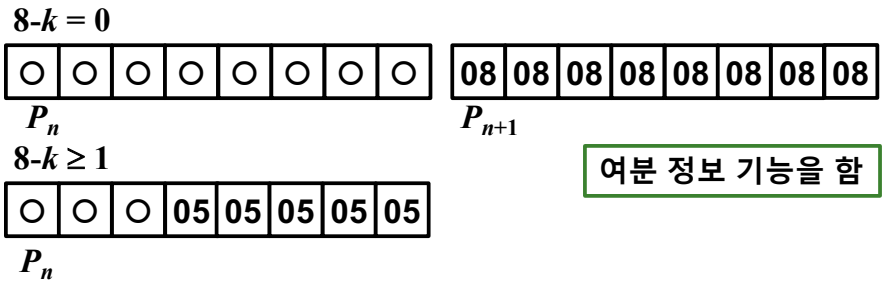
- 일반적인 바이트 단위 채우기 방법
  - P_n: 평문의 마지막 블록(k 바이트)
  - 블록 크기: 예) 8 바이트



평문의 크기가 정확하게 블록 크기의 배수이어도 채우기를 해야 함

채우기 (3/3)

- PKCS #7, RFC5652에 정의된 방법



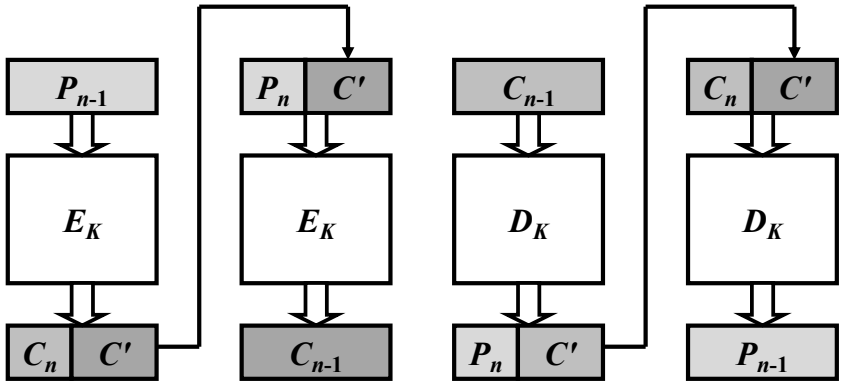
여분 정보 기능을 함

  - 채우기를 해야 하는 모든 부분을 채워야 하는 바이트 수의 값으로 채움
  - 마지막 바이트의 값이 1과 8 사이의 수인지 검사한 후에 그 수만큼 앞의 바이트들도 동일한 수로 채워져 있는지 검사한 다음에 제거함
- 일반적인 채우기 방법이나 이 방법 모두 모든 메시지에 대해 항상 채우기를 하여야 함
  - 문제점. 암호문이 평문보다 항상 큼
  - 채운 데이터는 여분 정보 역할을 할 수 있음

PKCS5, PKCS7 padding은 고려하는 블록 크기가 다를 뿐 동일함

암호문 훔침 기법

- 암호문 훔침 기법(ciphertext stealing): 바로 전 암호문 블록 이용하여 채우기를 하는 방식
  - 장점. 암호화된 메시지의 크기와 평문의 크기가 동일함
  - 비고. 표준 채우기에서는 새 데이터를 이용하여 채우기를 함



블록크기: 16byte
평문 크기: 36byte
P_1, P_2, P_3(4byte)
$C_1 = E.K(P_1)$
$C_3 \| C' = E.K(P_2)$
$C_2 = E.K(P_3 \| C')$
C_3(4byte), C'(12byte)

$P_1 = D.K(C_1)$
$P_3 \| C' = D.K(C_2)$
$P_2 = D.K(C_3 \| C')$

ECB(Electronic CodeBook) 모드

- 피드백을 사용하지 않는 모드
  - 암호화: $C_i = E.K(P_i)$, 복호화: $P_i = D.K(C_i)$
- 평문 오류: 한 평문 블록의 오류는 해당 암호문 블록에만 영향을 줌. 유사한 평문은 유사한 암호문으로 암호화되는 문제점이 있음

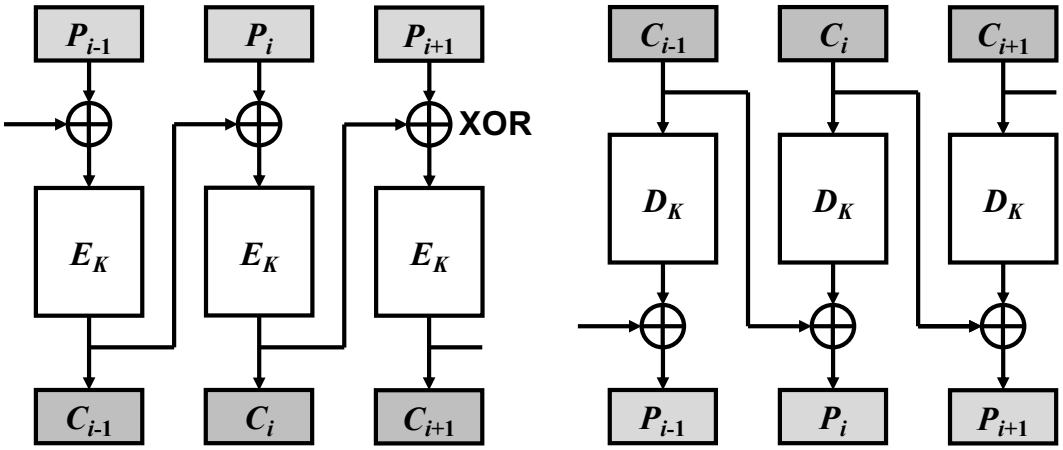
- 암호화된 메시지의 블록들의 위치를 바꾸면 복호화된 평문 메시지의 블록 위치들도 동일하게 바뀌게 됨. (의미있는 조작 가능)

- 오류 확산: 암호화된 메시지의 한 블록의 오류는 그 블록의 복호화에만 영향을 줌

- 효율성: 암호화, 복호화를 모두 병렬 처리 가능

CBC(Cipher Block Chaining) 모드 (1/6)

- 같은 평문 블록들도 서로 다른 암호문 블록으로 암호화됨



$$C_i = E.K(P_i \oplus C_{i-1})$$

$$P_i = D.K(C_i) \oplus C_{i-1}$$

병렬 수행 가능

C_i는 C_{i-1}를 계산한 후에 계산할 수 있음
하지만 C_i의 복호화는 독립적으로 계산할 수 있음

CBC 모드 (2/6)

- C_0: 초기 벡터(IV, Initalization Vector)

$$C_1 = E.K(P_1 \oplus IV) = E.K(P_1 \oplus C_0)$$
$$P_1 = D.K(C_1) \oplus IV = P_1 \oplus C_0 \oplus C_0$$

  - 보통 랜덤 블록을 사용함. 비밀성을 유지할 필요는 없음
  - 장점. 같은 메시지를 다시 암호화하여도 결과가 다름
    - 비고. 확률적 암호알고리즘(probabilistic encryption)
- 채우기
  - 일반적인 방법이나 끝 문자 표시 방법을 사용할 수 있음
  - 암호문 훔침 기법

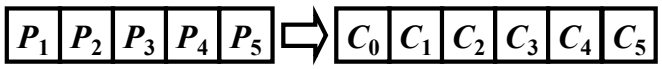
$$C_n||C' = E.K(P_{n-1} \oplus C_{n-2}) \qquad P_n||C' = D.K(C_{n-1}) \oplus \{C_n||0\}$$
$$C_{n-1} = E.K(\{P_n||0\} \oplus \{C_n||C'\}) \qquad P_{n-1} = D.K(C_n||C') \oplus C_{n-2}$$

0으로 채우기

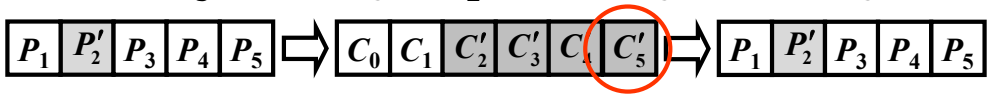
  - 암호문 훔칩 기법을 사용하지 않고 채우기를 하면 암호문의 길이는 평문보다 총 두 블록(초기 벡터 때문)이 커짐

CBC 모드 (3/6)

- 평문 오류
  - 암호화하기 전에 P_i에 오류가 발생하면 C_i에 영향을 줄 뿐만 아니라 C_i 이후 모든 암호문 블록에 영향을 줌
    - C_i: P_i와 P_i 앞에 있는 모든 평문 블록들에 의해 결정됨



$$C_2' = E.K(P_2' \oplus C_1) \neq E.K(P_2 \oplus C_1) = C_2$$
$$C_3' = E.K(P_3 \oplus C_2') \neq E.K(P_3 \oplus C_2) = C_3$$



$$P_2' = D.K(C_2') \oplus C_1 = P_2' \oplus C_1 \oplus C_1 = P_2'$$
$$P_3' = D.K(C_3') \oplus C_1 = P_3 \oplus C_2' \oplus C_2' = P_3$$

    - 마지막 블록을 MAC(Message Authentication Code)으로 사용할 수 있음
    - IV 때문에 유사 평문이더라도 전혀 다른 암호문을 얻게 됨

CBC 모드 (4/6)

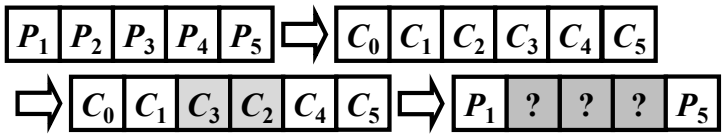
- 암호문 조작
  - 암호화된 이후 C_i에 오류가 발생하면 복호화할 때 P_i와 P_{i+1}만 영향을 받음
    - 자체 회복(self-recovering) 기능을 지니고 있다고 함
      - 하나의 암호문 블록에 오류가 발생하면 나머지 모든 블록의 복호화에 영향을 주지 않음

$$P_2' = D.K(C_2') \oplus C_1 = ? \oplus C_1 = ?$$
$$P_3' = D.K(C_3) \oplus C_2' = P_3 \oplus C_2 \oplus C_2' = ?$$
$$P_4' = D.K(C_4) \oplus C_3 = P_3 \oplus C_3 \oplus C_3 = P_4$$

    - C_i의 j번째 비트 오류
      - P_i는 쓰레기, P_{i+1}는 j번째 비트만 잘못됨
      - CBC 모드는 NM 특성을 만족하지 못함

CBC 모드 (5/6)

- 암호문 블록들의 위치를 변경하면 올바르게 복호화되지 않음



$$P'_2 = D.K(C_3) \oplus C_1 = P_3 \oplus C_2 \oplus C_1 = ?$$
$$P'_3 = D.K(C_2) \oplus C_3 = P_2 \oplus C_1 \oplus C_3 = ?$$
$$P'_4 = D.K(C_4) \oplus C_2 = P_4 \oplus C_3 \oplus C_2 = ?$$
$$P'_5 = D.K(C_5) \oplus C_4 = P_5 \oplus C_4 \oplus C_4 = P_5$$

- 추가 보안 문제
  - 암호화된 메시지 끝에 임의의 블록을 추가할 수 있음

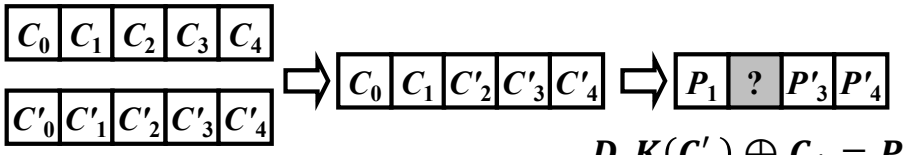
마지막 블록은 채우기 때문에
이 예처럼 결합하면 복호화하는
측이 문제가 있다는 것을 알게 됨

$$D.K(C_1) \oplus C_4 = P_1 \oplus C_0 \oplus C_4 = ?$$
$$D.K(C_2) \oplus C_1 = P_2 \oplus C_1 \oplus C_1 = P_2$$

CBC 모드 (6/6)

ECB 모드도 같은 보안 문제를 가지고 있음
더 심각한 것은 모든 블록이 오류 없이 복호화됨

- 두 개의 암호문을 조합하여 새 암호문을 만들 수 있음



$$D.K(C'_2) \oplus C_1 = P'_2 \oplus C'_1 \oplus C_1 = ?$$
$$D.K(C'_3) \oplus C'_2 = P'_3 \oplus C'_2 \oplus C'_2 = P'_3$$

- 암호문 블록을 변경하여 예측할 수 있는 변경을 도입할 수 있음
  - C_i의 한 비트를 변경하여 P_{i+1}의 특정 비트를 변경할 수 있음
- C_i와 C_j가 같으면 다음이 성립함 (충돌 발생)

$$P_i \oplus P_j = C_{i-1} \oplus C_{j-1}$$

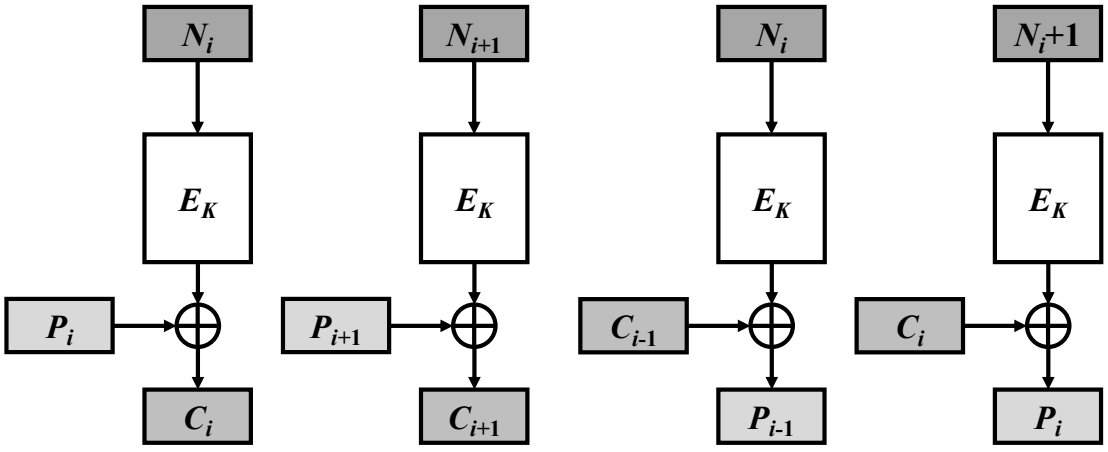
  - 우연히 같아질 확률은 매우 적지만 같은 키로 암호화되는 데이터가 많아지면 이 확률이 높아지기 때문에 시기적절한 키 갱신이 필요함

CBC모드 Padding Oracle 공격

- CBC 모드 PKCS #7 padding을 사용하여 메시지를 암호화하여 상대방에게 보내는 프로토콜이 있는데,
  - 상대방은 메시지를 복호화한 다음 먼저 채우기를 확인하고 채우기의 정확성 여부에 대한 알려줌
- 공격자가 $IV||C_1||C_2||\dots||C_n$ 암호문을 가지고 있을 경우 C_2에 대응되는 평문 M_2을 얻기 위해 M_2의 마지막 바이트를 0xXY로 예측한 다음 암호문 $IV||C_1 \oplus 0xXY \oplus 0x01||C_2$을 전송하여 공격할 수 있음
  - $D.K(C_2) \oplus C_1 \oplus 0xXY \oplus 0x01 = M_2 \oplus C_1 \oplus C_1 \oplus 0xXY \oplus 0x01 = M_2[15 \dots 1]||M_2[0] \oplus 0xXY \oplus 0x01$
  - 256번 공격하면 반드시 마지막 바이트를 정확하게 예측할 수 있음
- TLS 초기 버전은 mac-then-encrypt 방식을 사용하였으며, CBC 모드로 메시지를 암호화하였음
  - 복호화 절차는 채우기를 확인하고, 그다음 mac을 확인하는 방식이었으며, 두 종류의 오류 여부를 상대방에게 회신하였음
  - 암호세계에서는 오류의 종류를 구분하여 알려주는 것은 매우 위험함

CTR(Counter) 모드 (1/2)

- CBC보다 많이 사용하고 있는 암호모드
  - 암호화 연산만 사용 (하드웨어 소형화, 소프트웨어 모듈 크기에 유리함)



$C_i = P_i \oplus S_i$

$S_i = E.K(N_i)$

실제 암호화 연산을 사용하지 않아도 됨.
$MAC.K(N_i)$

$P_i = C_i \oplus S_i$

$S_i = E.K(N_i)$

CTR 모드 (2/2)

중첩 가능성 때문에
- 96비트 랜덤값 + 32비트 카운터 사용
- 키를 자주 교체 (2^{32} 메시지 이하)
- SIV: IV = MAC.K(M)
  ⇒ 인증 암호화 요소 (RFC 5297)

- 초기 N_0과 증가함수는 공개되어 있음
  - 증가함수는 보통 "+1"을 사용함
  - 하지만 초기 N_0은 항상 다른 것을 사용해야 함 (중첩의 가능성?)

$$C_i = P_i \oplus S_i \quad C_i' = P_i' \oplus S_i \quad C_i \oplus C_i' = P_i \oplus P_i'$$

- 채우기가 필요 없음
  - 마지막 블록 크기만큼 XOR하여 암호화함
- 평문 오류: 암호화하기 전에 평문 블록 P_i에 오류
  - P_i에 오류가 발생하면 C_i에 같은 위치에 오류가 발생하며, 다른 암호문 블록에는 영향을 주지 않음
  - IV를 사용하기 때문에 유사 평문도 다른 암호문으로 암호화됨
- 암호문 조작: 암호문 블록 C_i에 오류
  - P_i에 같은 위치에 오류가 발생하지만 다른 암호문 블록에는 영향을 주지 않음
    - NM 특성을 만족하지 못함

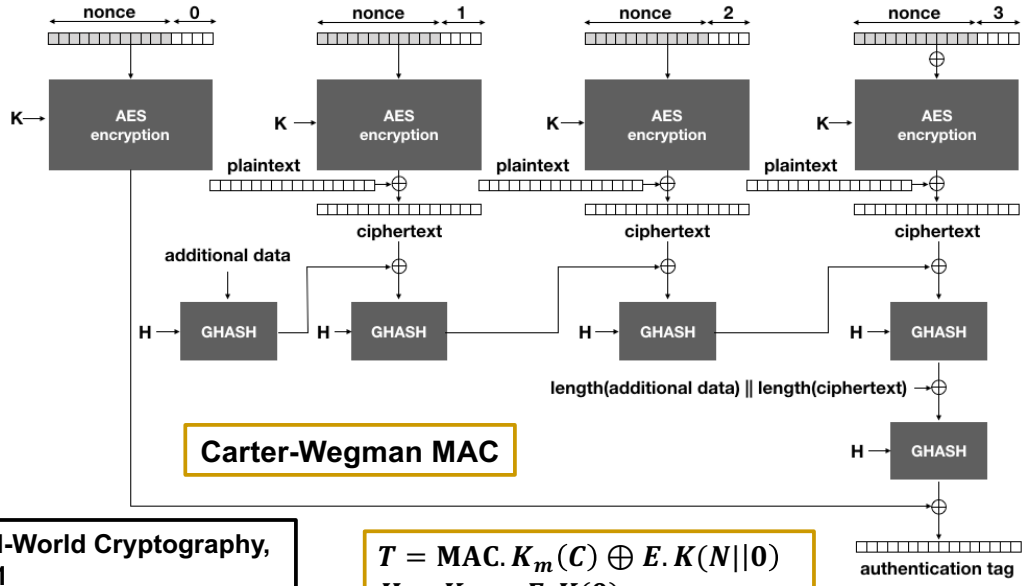
인증 암호화 모드 (1/3)

- 인증 암호화 모드는 크게 mac-then-encrypt 방식과 encrypt-then-mac 두 가지 방식이 있음
  - encrypt-then-mac이 더 안전하고 효과적인 방식임
  - 초기에는 mac-then-encrypt 방식도 사용되었음
- 기본적으로 두 개의 키가 필요함
- CCM (Counter Mode with CBC-MAC)
  - 메시지에 대한 CBC-MAC을 계산한 후에 메시지와 MAC 값을 CTR 모드로 암호화함
    - 2-pass mac-then-encrypt 모드
      - 2-pass: 암호화와 MAC 계산이 순차적으로 이루어짐
- EAX(encrypt-then-authenticate-then-translate)
  - 2-pass encrypt-then-mac 모드
    - CCM 대체용으로 제안

인증 암호화와 암호 모드 (2/3)

- **AEAD**(Authenticated Encryption with Associated Data)
  - 암호문과 별도 평문을 함께 전달하고, 암호문과 별도 평문의 무결성을 제공함
- **GCM**(Galois Counter Mode) 모드
  - 1-pass encrypt-then-mac 모드

일반적인 CTR 모드

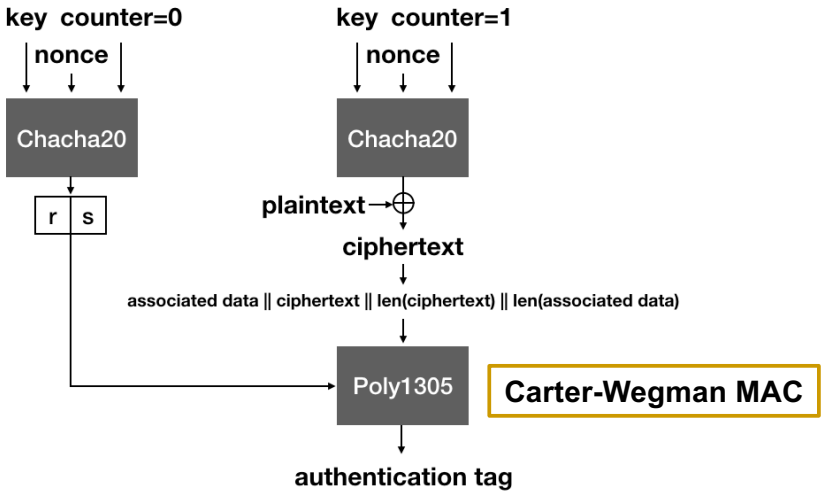


D. Wong, Real-World Cryptography, Manning, 2021

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

인증 암호화와 암호 모드 (3/3)

- **ChaCha20-Poly1305**
  - 스트림 암호를 이용한 인증 암호화 (AEAD 지원)
  - GCM과 유사 하지만 2-pass



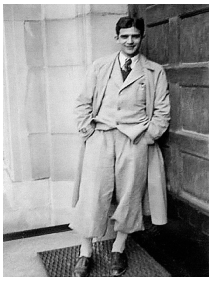
D. Wong, Real-World Cryptography, Manning, 2021

암호화 모드의 선택

- 일반 암호화 모드
  - ECB는 가장 빠른 모드이지만 근본적인 문제점을 가지고 있는 모드임
    - 일반 메시지를 ECB 모드로 암호화하는 것은 바람직하지 않지만 암호키와 같이 한 블록보다 작은 크기의 값은 문제가 없음
  - 보통 파일과 같은 많은 양의 데이터를 암호화할 때에는 CBC나 CTR 모드를 사용함
    - 둘다 NM 특성을 만족하지 못함
- 인증 암호화 모드
  - 최근에는 대칭키를 이용한 암호화는 기본적으로 인증 암호화 모드를 이용함
    - 인증 암호화 모드 중 encrypt-then-mac 모드를 사용함
      - EAX보다 GCM 모드 선호

DES(Data Encryption Standard) (1/2)

- 1972: NIST(National Institute of Standards)에서 표준화 작업 착수
  - IBM에서 제안한 암호알고리즘(Lucifer)에 기반함
    - H. Feistel이 제안함
  - NSA(National Security Agency)에서 자문
    - IBM이 원래 제안했던 키 길이(128비트)를 56비트로 축소함
    - 내부 동작 메커니즘 중 일부를 교체
      - NSA가 DES에 트랩도어를 포함했다고 믿는 사람들의 근거
      - 반대로 IBM이 만들어 놓았을지 모르는 트랩도어를 제거하기 위해 교체했다는 소문도 있음
      - NSA가 공표하지는 않았지만 어떤 허점을 알고 있다는 근거



- 1977년에 표준으로 공식 채택 (FIPS PUB 46)
  - 유효기간을 10년으로 정함. 그러나 20년 동안 표준으로 사용함
- 1997년에 새 표준안 AES 작업 착수
  - 2000년 10월에 Rijndael 알고리즘을 새 표준으로 채택함

DES (2/2)

- 블록 방식의 대칭 암호알고리즘
  - 블록 크기: 64비트
  - 키의 길이: 56비트(패리티를 추가하여 64비트로 표현)
  - 16 라운드로 구성
    - 각 라운드에서 치환과 자리바꿈 연산을 수행 (product cipher)
  - 현재는 키 길이 때문에 안전하지 않음
    - 어떤 허점이 발견된 것은 아니고 현재의 컴퓨팅 능력을 고려하였을 때 전사 공격을 통해 키를 얻을 수 있음
  - 소프트웨어보다는 하드웨어에 적합한 알고리즘

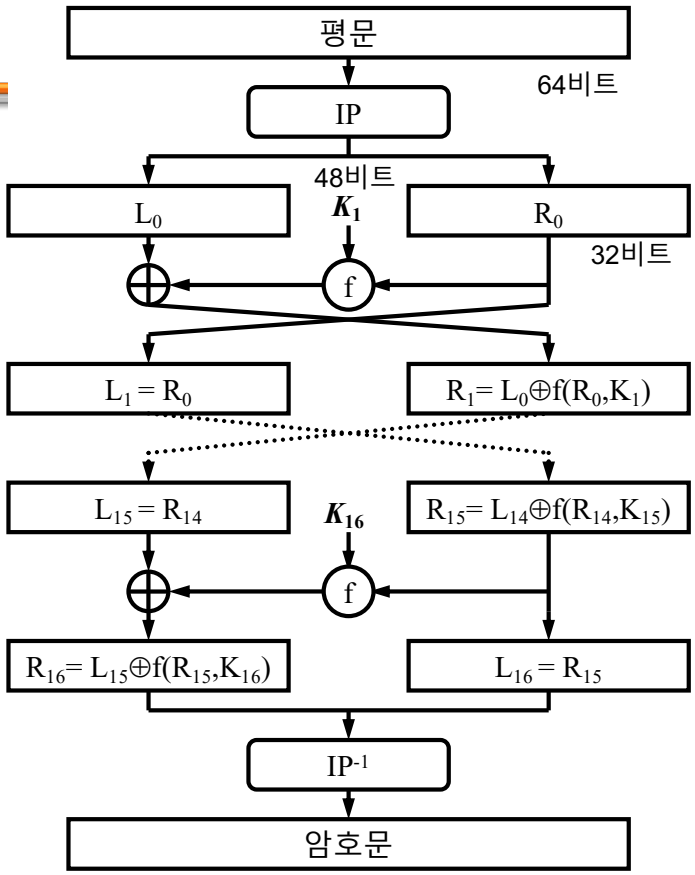
DES

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

$$\mathrm{IP}(M) = L_0 R_0$$
$$C = \mathrm{IP}^{-1}(R_{16} L_{16})$$

$$J_i^{K_i}(L_{i-1} || R_{i-1}) = L_i || R_i$$
where, $L_i = R_{i-1}, R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$

마지막 라운드는 기존과 다르게 계산됨을 나타내기 위해 C를 위와 같이 표현하고 있다.



DES – IP와 IP^{-1}

Initial Permutation(IP)

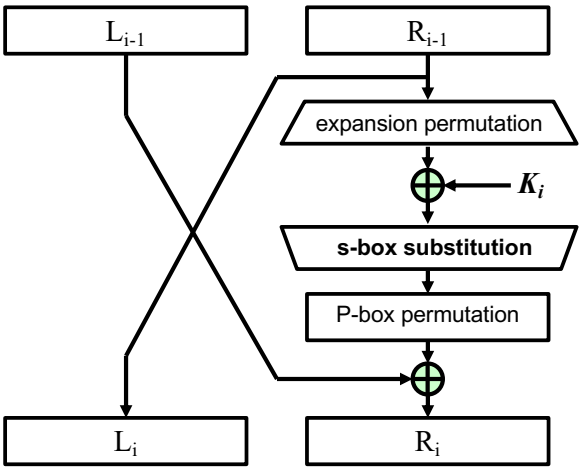
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Final Permutation(IP^{-1})

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

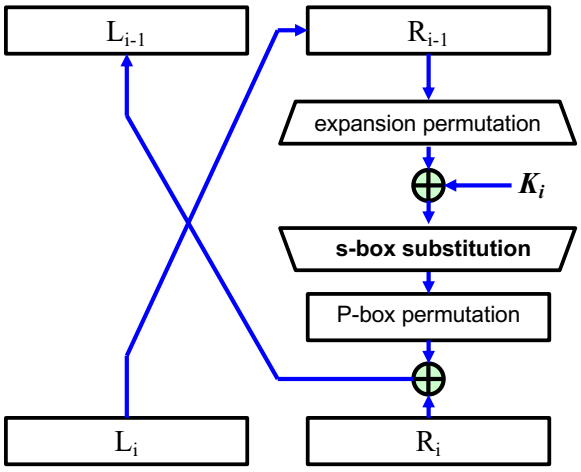
DES – ROUND

암호화



$$L_i = R_{i-1},\quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

복호화



$$L_{i-1} = R_i \oplus f(R_{i-1}, K_i),\quad R_{i-1} = L_i$$

Feistel Cipher

DES – Expansion/P-box Permutation

Expansion Permutation

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

avalanche effect

① 00 00 00 00 00 00 00 01
② 00 00 00 00 00 00 00 00
⇒ ①과 ②를 같은 키로 암호화

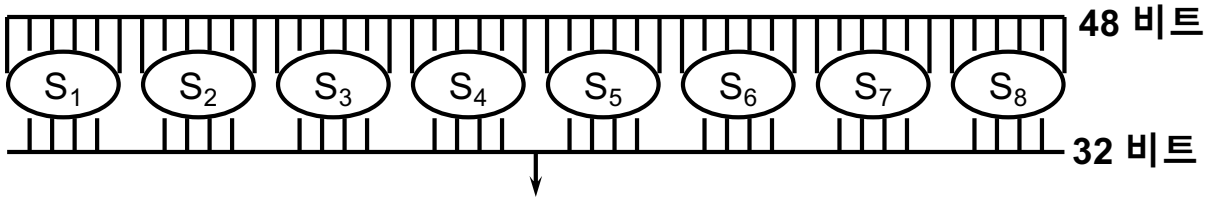
결과는 매우 다름
대략: 34 곳

이런 효과가 없으면
범위가 좁혀질 수 있음

P-box

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

DES – S-box (substitution-box)



S_1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

- $S_1(011011) = S_1(01, 1101) = S_1(1, 13) = 5 = 0101$
- $S_1(110011) = S_1(11, 1001) = S_1(3, 9) = 11 = 1011$
- S 박스의 각 행은 0부터 15의 permutation임

DES – Key Scheduling

- DES는 56 비트 키를 사용함
  - 하지만 표현은 64비트로 표현하며, 나머지 8비트는 parity 비트임
  - DES는 16라운드로 구성되어 있으며, 각 라운드에서 사용하는 키를 key scheduling 알고리즘을 통해 생성함
- DES의 key scheduling 알고리즘
  - 64비트로 표현된 키에서 parity를 제외하고 먼저 key permutation 과정을 통해 두 개의 28비트 값으로 나눔

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36

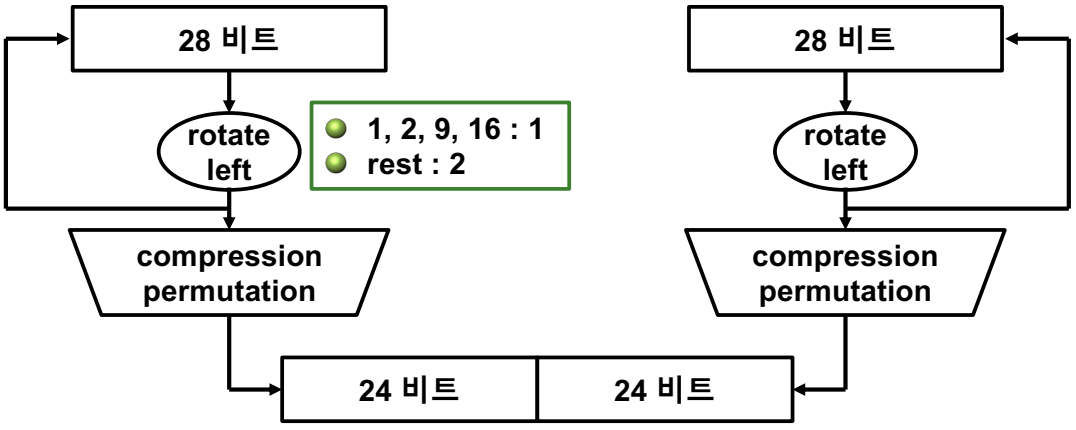
C_0

key permutation

63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

D_0

DES – Key Scheduling



14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2

41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

DES의 문제점

- DES의 가장 큰 문제점은 키 길이가 너무 짧다는 것임
  - DES의 전신은 128비트 ⟹ 원래 제안은 64비트 ⟹ 패리티의 필요성 때문에 56비트
  - 1977년: Diffie와 Hellman은 $20,000,000이면 하루 만에 DES 키를 찾아내는 특수 하드웨어를 제작할 수 있음을 보임
  - 1993년: Weiner는 $100,000이면 하루 반 만에 DES 키를 찾아내는 특수 하드웨어를 제작할 수 있음을 보임
  - Triple-DES 등장, 새 표준의 필요성 대두

1998	RSA – DES Challenge II 예산: $250,000 시간: 56 hours
1999	RSA – DES Challenge III 시간: 22 hours 15 minutes
2006	COPACOBANA(120 FPGAs): 7 days ($10,000)

Triple DES

- DES의 문제점을 극복하는 방안으로 일시적으로 사용함
- 한 블록을 3번 반복하여 암호화하지만 두 개의 키만 사용
  - 블록의 크기는 같지만 키의 길이는 112비트로 늘어남
  - 3개의 키를 사용할 수 있지만 키 길이는 2개의 키를 사용하는 것과 비교하여 증가하지 않음 (meet-in-the-middle 공격)
- DES 함수는 다음이 성립하지 않다는 것이 증명되어 있음

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

  - 이것이 성립하면 3중 암호화는 아무 의미가 없음
- 암호화: $C = E.K_1(D.K_2(E.K_1(M)))$
- 복호화: $M = D.K_1(E.K_2(D.K_1(M)))$
- 암호화 과정에서 $E.K_2$대신에 $D.K_2$를 사용한 이유
  - 단일 DES와 호환을 위해: $K_1 = K_2$이면 단일 DES와 같음

AES(Advanced Encryption Standard)

- DES의 키 길이는 현대의 기준으로 너무 짧음
- 3-DES를 사용할 수 있지만 DES보다 성능이 떨어지며, DES 또한 하드웨어 구현을 목표로 설계되었기 때문에 기존 다른 대칭 암호알고리즘에 비해 소프트웨어로 구현하였을 때에는 성능이 떨어짐

1997	call for proposal
1998	15 accepted
1999	reduced to 5
2001	Rijndael accepted

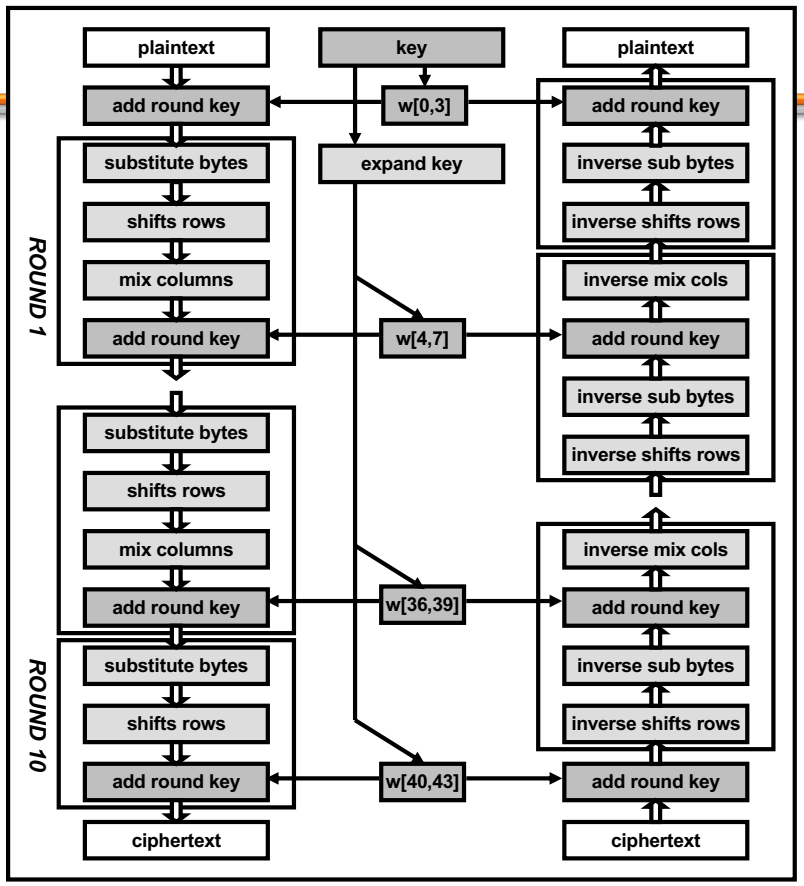
block size: 128 bits, key length support: 128, 196, 256
CAST-256, CRYPTON, DEAL, DFC , E2, FROG, HPC, LOK197, MAGENTA, SAFER+
MARS, RC6, Rijndael, Serpent, Twofish
developed by: J. Daemen, V. Rijmen FIBS PUB 197

Belgium Cryptographer

MARS (IBM): Feistel Structure
RC6 (R. Rivest, M. Robshaw, R. Sidney, Y. Yin) : Feistel Structure
Serpent (R. Anderson, E. Biham, L. Knudsen): SPN
Twofish (B. Schneier): Feistel Structure

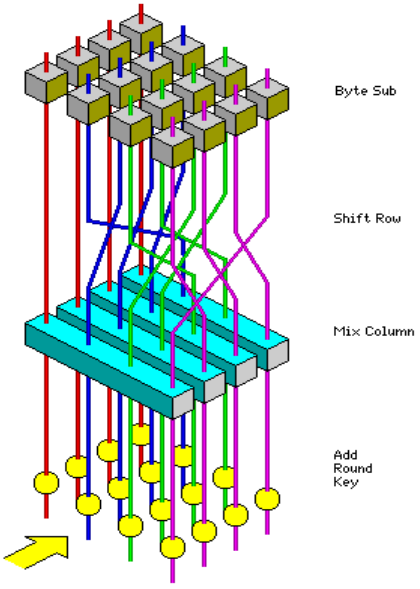
AES (2/4)

블록크기: 128비트
키길이: 128비트



AES (3/4)

- 각 라운드에서 하나의 자리바꿈과 3개의 치환 연산을 사용함
  - S-박스: GF(2^8)을 사용하여 구성함
  - shift row: 단순 자리바꿈
  - mix column: GF(2^8)을 이용한 치환
  - add round key: XOR 연산을 사용
- 구조가 매우 단순함



AES (4/4)

- 구현할 때 각 라운드에서 하는 일을 매번 새롭게 계산하는 형태가 아니라 다양한 사전 계산을 하여 테이블에 저장하는 방법으로 성능을 높이는 방법이 있음
- Intel은 Intel Westmere부터 AES 암/복호화의 수행 성능을 향상시키기 위한 명령어 집합 AES-NI(New Instructions)를 지원하기 시작함 (2008년 3월)
  - 총 7개 명령어로 구성되어 있음

Salsa20

eStream Profile 1에 선택된 4개 중 하나
같은 개발자가 만든 ChaCha20이 현재 널리 사용되고 있음

- 대칭 암호알고리즘: 스트림 방식
  - 키 길이: 128비트 또는 256비트
  - 암호화할 때 64비트 난스 r를 사용함
  - $\text{Salsa20}(K; r) := F(K, (r, 0))||F(K, (r, 1))|| \ldots$
  - F 함수: 64byte $\Longrightarrow$ 64byte
    - 초기 64byte $S := \tau_0||K||\tau_1||r||i||\tau_2||K||\tau_3$
      - τ_j: 4byte (알고리즘 자체 제시된 값)
      - i: 8byte counter
    - $F\big(K, (r, 0)\big) = f^{10}(S) + S$
      - f: 64byte $\Longrightarrow$ 64byte
        - SW 뿐만 아니라 HW로도 효율적으로 구현할 수 있으며, 역이 가능한 함수
  - 암호화: F함수의 결과를 평문과 XOR함