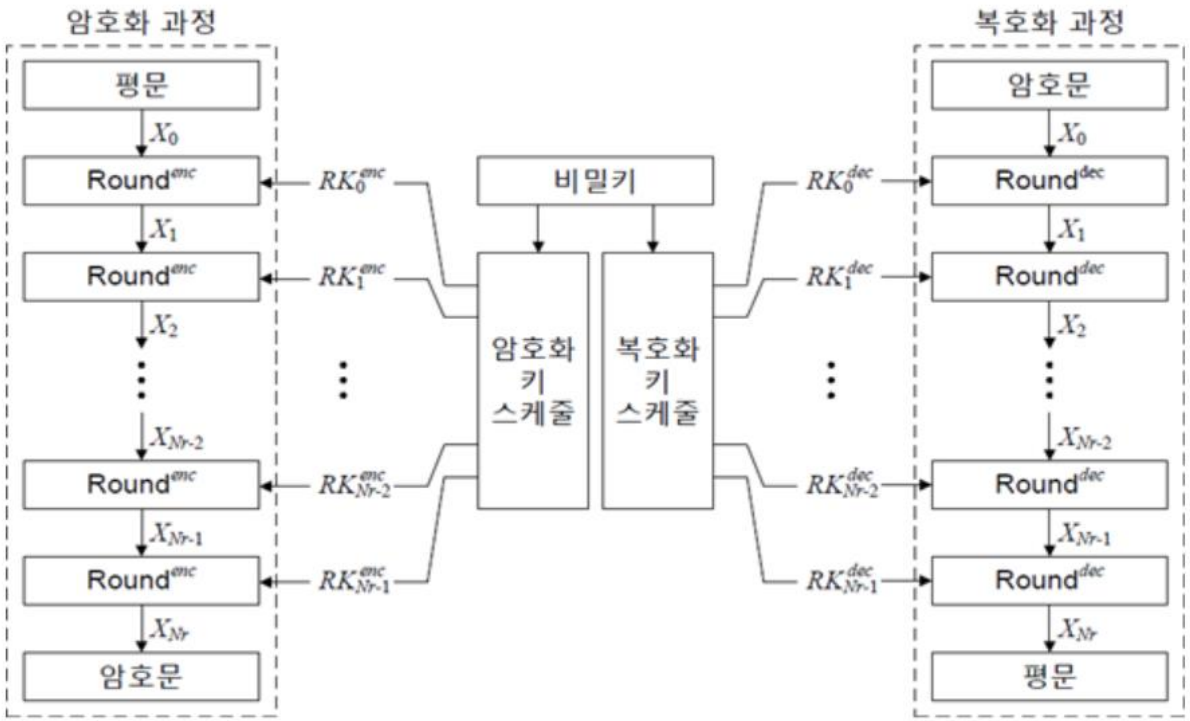
LEA에 대한 CPA 공격

https://youtu.be/h_LfVdYtpGw

LEA 알고리즘

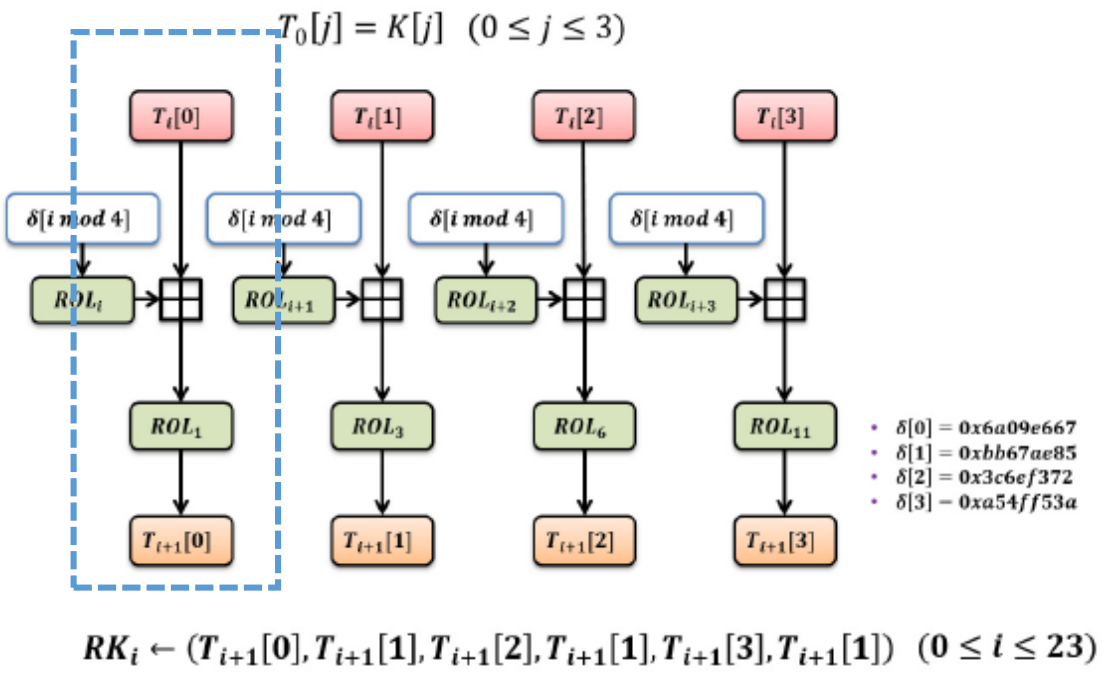
- 2013년 한국 국가암호기술연구소에서 개발한 블록 암호 알고리즘
- 경량 환경에서 기밀성을 제공하기 위해 개발
- 128bit 데이터 블록을 암호화
- LEA의 라운드 함수는 32bit 단위의 ARX(Addition, Rotation, XOR) 연산만으로 구성
- 범용 32bit 소프트웨어 플랫폼에서 고속 동작 동작 가능
- 안전성 보장 & S-box의 사용 배제하여 경량 구현 가능

LEA의 구조도



LEA-k	Size of block	Key length	Number of rounds
LEA-128	**128**	**128**	**24**
LEA-192	128	192	28
LEA-256	128	256	32

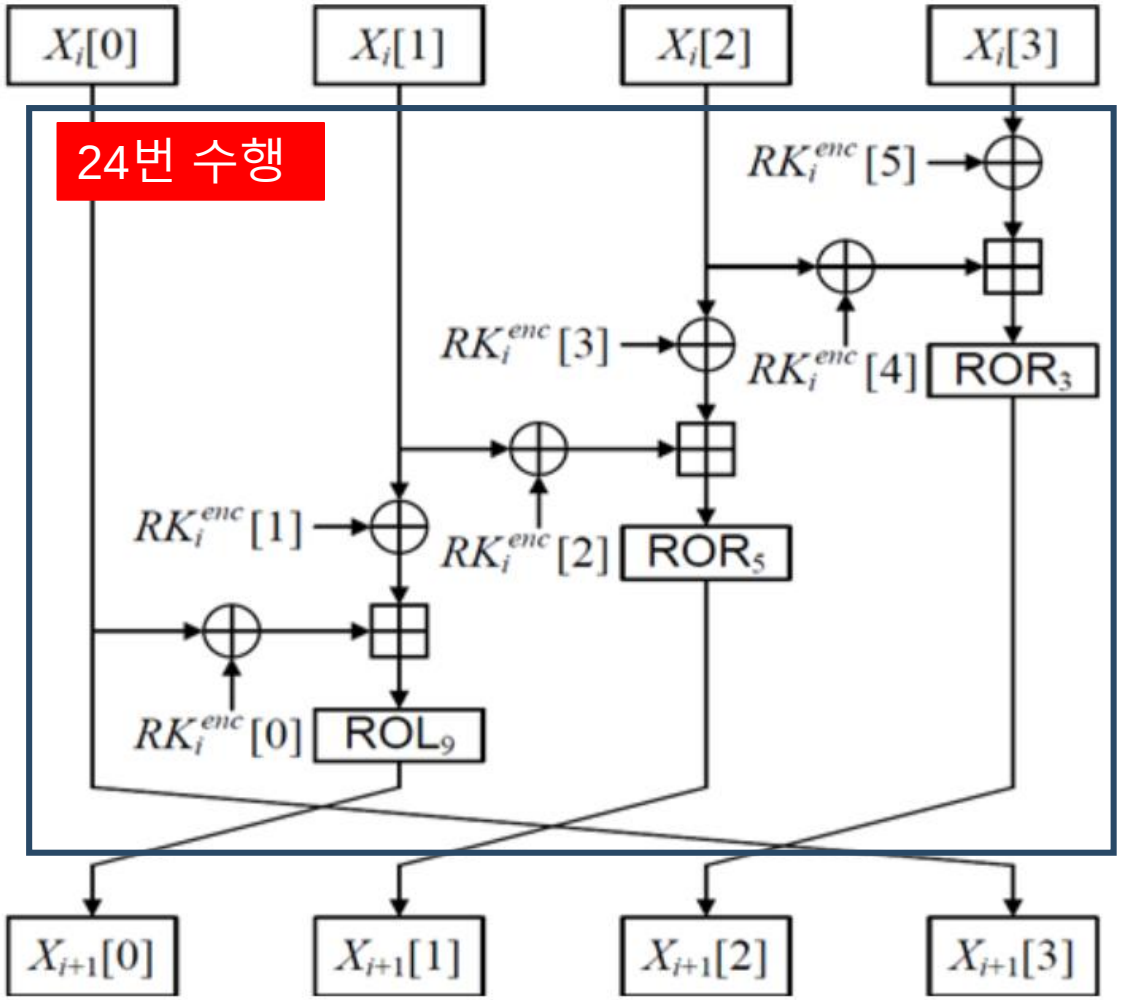
LEA의 키스케줄



- 각 라운드 키 $RK_r (0 \le r \le 23)$ 는 128비트의 비밀키로부터 유도
- 각 라운드에는 32비트로 된 6개의 $RK_r[i] (0 \le i \le 5)$ 사용
- LEA-128에서 $RK_r[1]$, $RK_r[3]$, $RK_r[5]$ 가 동일
- $T_{i+1}[j]$ 를 알면 $T_i[j]$ 유추 가능
- 위와 같은 특징으로 모든 라운드 키 유추 가능

박진학, 김태종, 안현진, 원유승, 한동국. (2015). LEA에 대한 부채널 분석 및 대응 방법. 정보보호학회논문지, 25(2), 449-456.

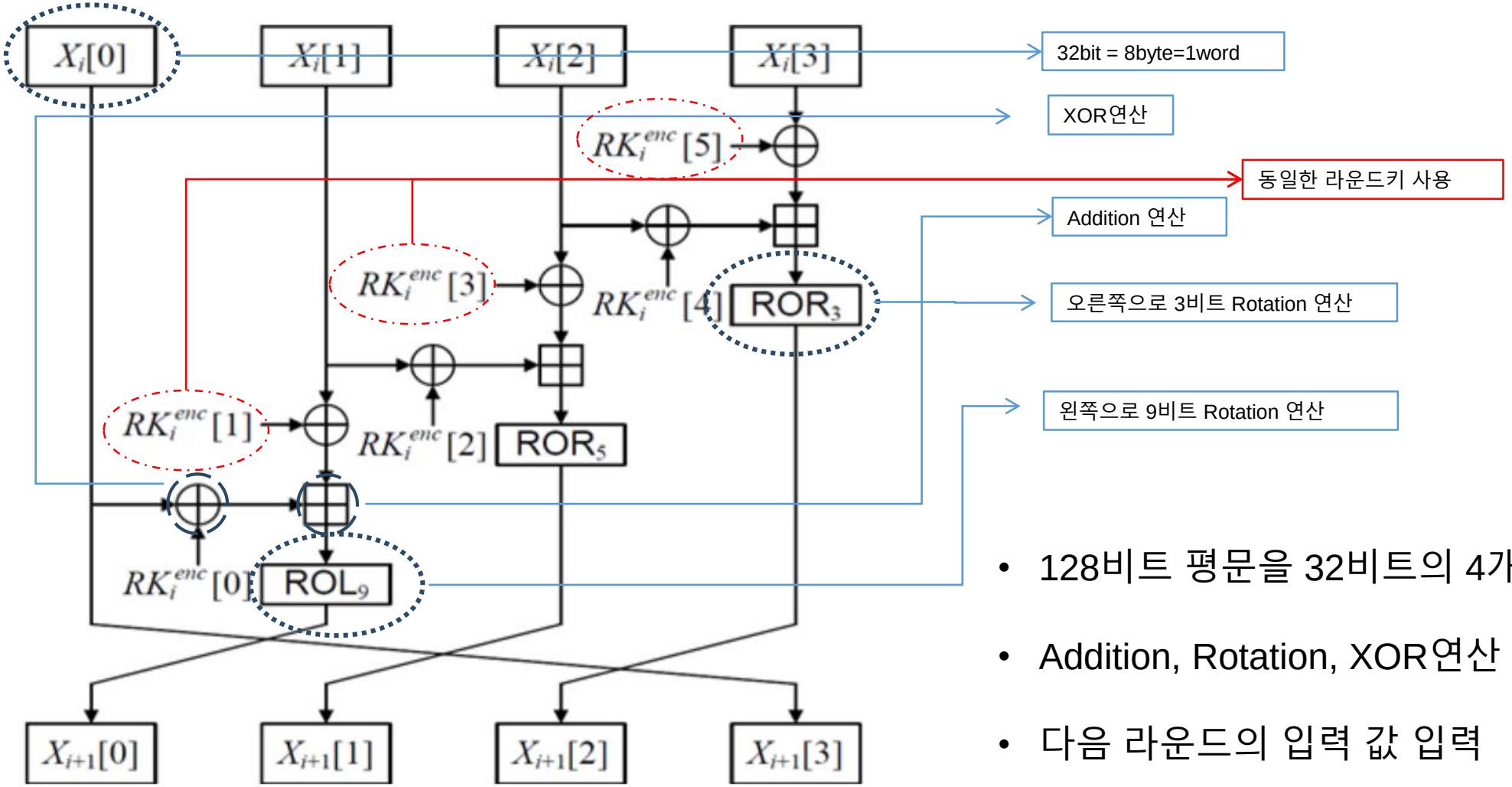
LEA의 암호화



- 128비트 평문을 32비트의 4개의 워드로 나눔
- Addition, Rotation, XOR연산 수행
- 다음 라운드의 입력 값 입력

박진학, 김태종, 안현진, 원유승, 한동국. (2015). LEA에 대한 부채널 분석 및 대응 방법. 정보보호학회논문지, 25(2), 449-456.

LEA의 암호화

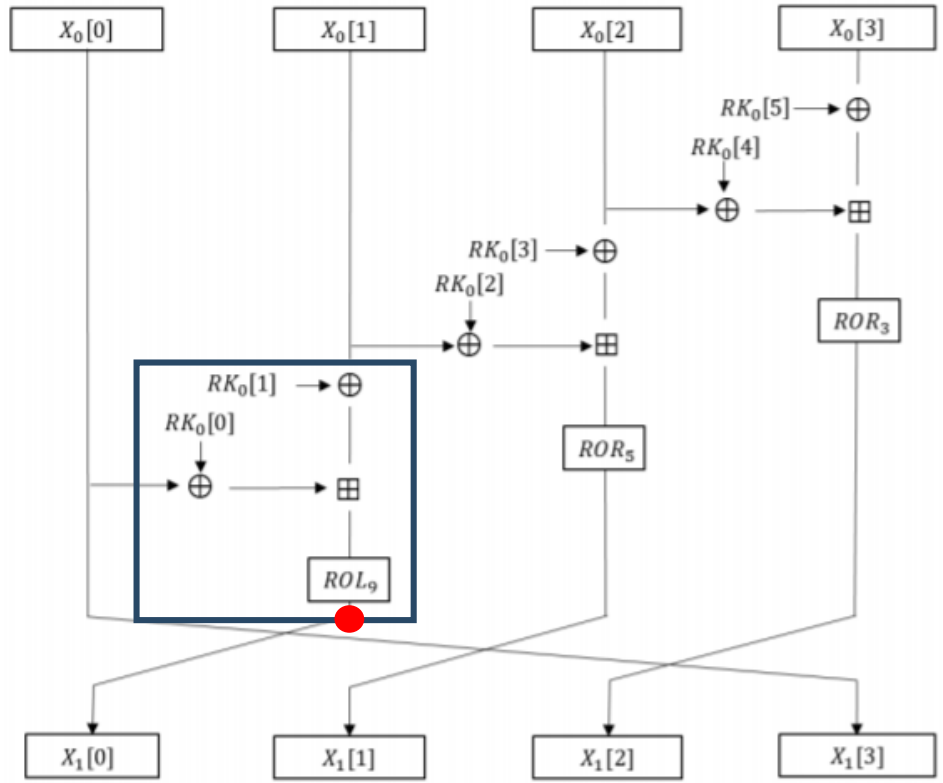


- 128비트 평문을 32비트의 4개의 워드로 나눔
- Addition, Rotation, XOR연산 수행
- 다음 라운드의 입력 값 입력

박진학, 김태종, 안현진, 원유승, 한동국. (2015). LEA에 대한 부채널 분석 및 대응 방법. 정보보호학회논문지, 25(2), 449-456.

LEA에 대한 CPA공격

- 처음 제시된 LEA에 대한 CPA공격(한 라운드에 사용되는 서브 키의 두쌍을 찾아내는 방법 제시)



- 예를 들어 $(RK_r[0], RK_r[1])$ 찾아내는 방법
- 위의 두 라운드 키를 추출하기 위해서

  $X_0[0]$와 $X_1[0]$ 의 해밍 거리를 계산
- 32비트의 키를 한번에 추측 x → 8비트씩 나눠 사용

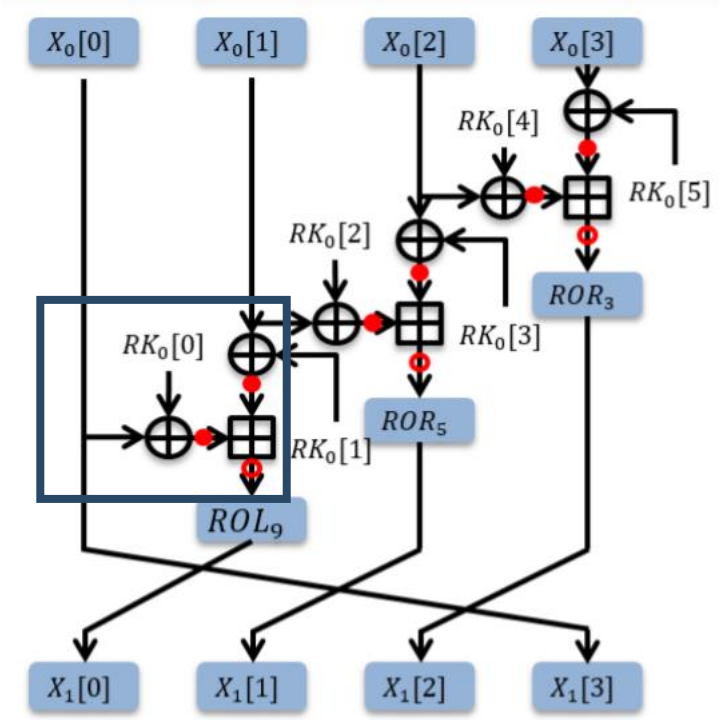
* 2^32의 값이 너무 크기 때문에 2^8 * 4로 계산 (비밀키 추출 연산 감소)

$$2^{32} = 2^8 * 2^8 * 2^8 * 2^8 = 4 * 2^8$$

박은수, 오수현, 하재철. (2017). 부채널 공격에 대응하는 마스킹 기반의 블록 암호 LEA. 정보보호학회논문지, 27(5), 1023-1032.

LEA에 대한 CPA공격

- 평문을 이용한 1차 부채널 분석



LEA 1라운드

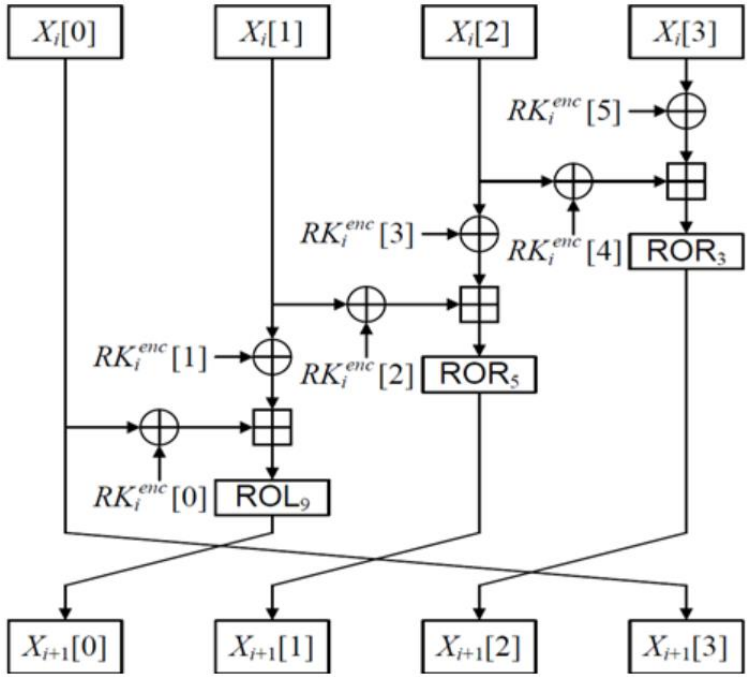
- 공격지점으로 활용될 수 있는 부분
  1) 평문 $X_0[i]\{i\in 1,2,3,4\}$과 라운드키 $RK_0[j]\{j\in 1,2,3,4,5,6\}$ XOR연산하는 부분
  2) 두 가지 키가 사용된 Addition 연산이 일어나는 부분

- 평문을 이용하여 Addition 연산이 사용되는 부분
  8비트씩 추측 $2^8 * 2^8 * 2^8 * 2^8 = 2^{32} = 2^8 * 4$

- Addition연산을 효율적으로 하기 위한 방법
  xor연산이 일어난 지점을 중간값으로 분석한 뒤,
  Addition 연산이 일어난 지점을 중간값으로 하여 분석

박진학, 김태종, 안현진, 원유승, 한동국. (2015). LEA에 대한 부채널 분석 및 대응 방법. 정보보호학회논문지, 25(2), 449-456.

LEA 코드




```
# encrypts a 128 bit input block
def encrypt(self, pt):
    if len(pt) != 16:
        raise AttributeError('length of pt should be 16 not %d'%len(pt))

    #pt = LEA.to_bytearray(pt)
    temp = list(struct.unpack('<LLLL',pt))

    for i in range(0, self.rounds, 4):
        temp[3] = self.ROR(((temp[2] ^ self.rk[i][4]) + (temp[3] ^ self.rk[i][5])) & 0xffffffff, 3)
        temp[2] = self.ROR(((temp[1] ^ self.rk[i][2]) + (temp[2] ^ self.rk[i][3])) & 0xffffffff, 5)
        temp[1] = self.ROL(((temp[0] ^ self.rk[i][0]) + (temp[1] ^ self.rk[i][1])) & 0xffffffff, 9)
        i += 1
        temp[0] = self.ROR(((temp[3] ^ self.rk[i][4]) + (temp[0] ^ self.rk[i][5])) & 0xffffffff, 3)
        temp[3] = self.ROR(((temp[2] ^ self.rk[i][2]) + (temp[3] ^ self.rk[i][3])) & 0xffffffff, 5)
        temp[2] = self.ROL(((temp[1] ^ self.rk[i][0]) + (temp[2] ^ self.rk[i][1])) & 0xffffffff, 9)
        i += 1
        temp[1] = self.ROR(((temp[0] ^ self.rk[i][4]) + (temp[1] ^ self.rk[i][5])) & 0xffffffff, 3)
        temp[0] = self.ROR(((temp[3] ^ self.rk[i][2]) + (temp[0] ^ self.rk[i][3])) & 0xffffffff, 5)
        temp[3] = self.ROL(((temp[2] ^ self.rk[i][0]) + (temp[3] ^ self.rk[i][1])) & 0xffffffff, 9)
        i += 1
        temp[2] = self.ROR(((temp[1] ^ self.rk[i][4]) + (temp[2] ^ self.rk[i][5])) & 0xffffffff, 3)
        temp[1] = self.ROR(((temp[0] ^ self.rk[i][2]) + (temp[1] ^ self.rk[i][3])) & 0xffffffff, 5)
        temp[0] = self.ROL(((temp[3] ^ self.rk[i][0]) + (temp[0] ^ self.rk[i][1])) & 0xffffffff, 9)

    ct = bytearray(struct.pack('<LLLL',temp[0], temp[1], temp[2], temp[3]))
    return ct
```

Q & A