SHA-3(Keccak)

유튜브 주소 : https://youtu.be/epRV1bWklaE

SHA-3 개요

SHA-3 동작 과정

코드 분석(tiny_sha3)

SHA-3 개요

- SHA-3: 2015년 8월 NIST가 SHA-2 대체를 위해 발표
  - SHA-1의 충돌쌍 공격이 발견되면서 SHA-2의 안전성에 문제가 생길 것을 대비해 SHA-3 공모 진행

- SHA-1과 SHA-2는 NIST에서 자체적으로 디자인 했으나, SHA-3는 공개 경쟁을 통해 후보를 모집
  - 총 64개 알고리즘 등록
  - 1차 후보: 51개 알고리즘 선정
  - 2차 후보: 14개 알고리즘 선정
  - 최종 후보: 5개 알고리즘 발표

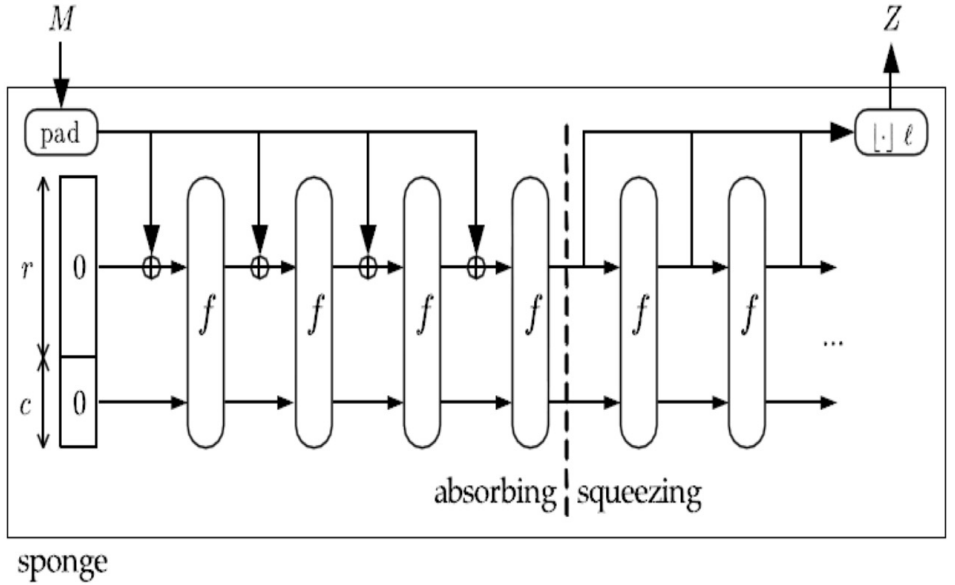
- 2012년 10월 1일 KECCAK 알고리즘이 SHA-3 알고리즘으로 선정

SHA-3 개요

- KECCAK은 4개의 해시 함수와 2개의 XOF(Extendable Output Function)으로 구성
  - 해시 함수: 다이제스트의 길이가 정해져있음
    - SHA3-224, SHA3- 256, SHA3-384, SHA3-512
    - SHA3 뒤의 숫자는 출력 해시값 길이를 의미
  - 출력 확장 함수(XOF): 다이제스트 길이가 정해져 있지 않고 출력 해시를 임의 길이로 확장 가능
    - SHAKE128, SHAKE256
    - SHAKE 뒤의 숫자는 보안 강도를 의미
- KECCAK은 스펀지 구조로 구성
  - 흡수 과정(absorbing)과 압착과정(squeezing)을 거쳐 해시 값이 만들어짐
- SHA-1, SHA-2와 달리 KECCAK은 길이 확장의 약점이 없음
  - HMAC 구간 설계가 필요하지 않음(MAC 계산에서 키를 메시지 앞에 붙임으로 수행 가능)
- 하드웨어 연산 시 고속 연산이 가능
  - 스펀지 구조는 병렬 구현이 가능하도록 설계되었음
- 부채널 공격에 강점을 지님
  - 스펀지 구조는 랜덤 액세스 패턴을 사용하기에 외부의 추측, 측정에 의존하지 않음

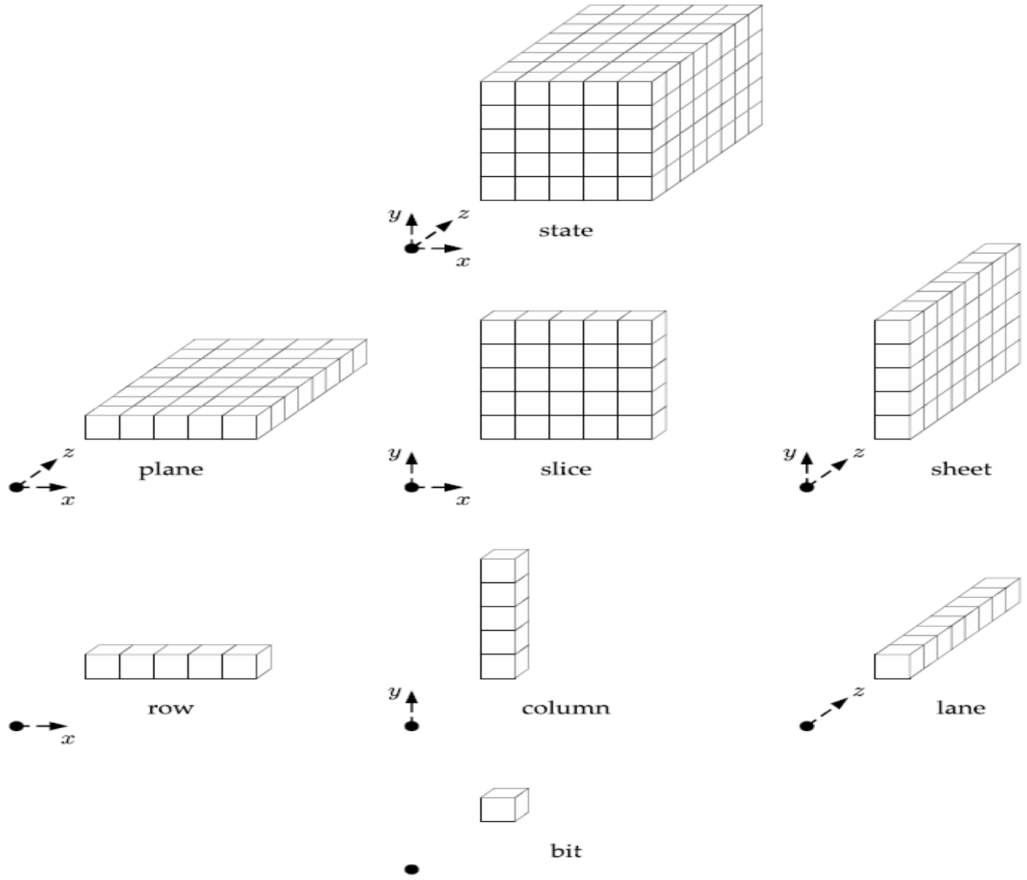
SHA-3 동작 과정

- SHA-3 구조: 스펀지 구조 ->SPONGE[f, pad, r](M,d)
  - f 함수와 패딩 함수를 이용하여 메시지 다이제스트를 출력
    - f 함수: b 비트의 순열을 가지고 r비트의 크기를 입력으로 하는 함수(b = r + c)
      - b: b ∈ {25, 50, 100, 200, 400, 800, 1600}
      - r: b보다 작은 양의 정수로 f 함수의 입력 비트를 의미
      - c: b - r 값을 갖는 양의 정수
    - 패딩 함수: 메시지를 입력 비트인 r의 크기의 배수로 만드는 함수
  - 평문인 메시지 M과 다이제스트 길이인 d를 파라미터로 사용



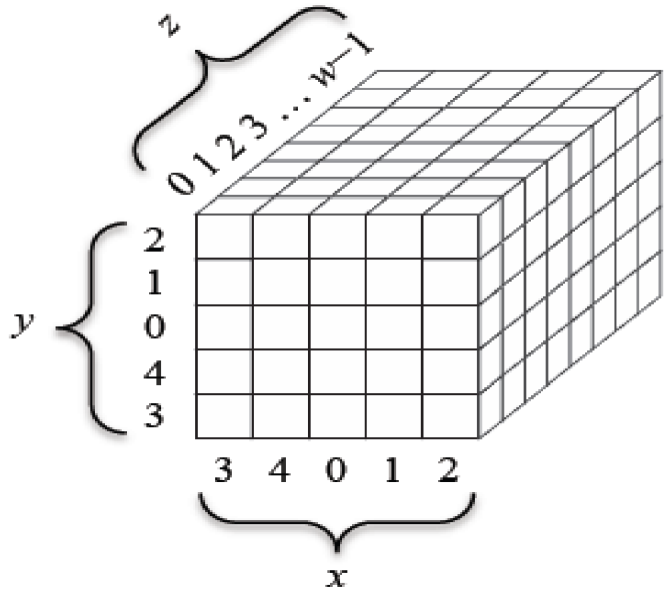
Keccak 동작 과정

- SHA-3의 f 함수
  - b의 입력 비트에 따라 정해지는 w값으로 5*5*w의 3차원 행렬(state)로 이루어짐
  - 3차원 행렬 내에서 총 5가지의 함수를 통해 f 함수의 출력이 정해짐



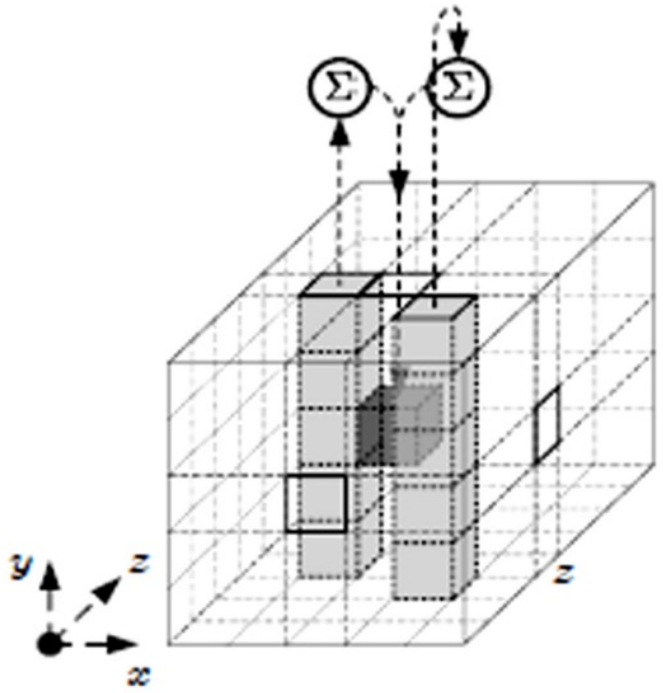
Keccak 동작 과정≤,≥,

- Keccak의 state
  - b의 값을 25로 나눈 w값과 w의 이진로그 값인 l로 구성
  - w는 f함수 내에서 치환되는 5*5*w의 3차원 행렬의 state를 생성
  - state는 0 ≤ x ≤ 5, 0 ≤ y ≤ 5, 0 ≤ z ≤ 5 값을 지님
  - 3차원 배열의 형태로 f 함수에 입력되어 메시지를 치환하는 과정을 거침
  - x, y 축 중간 값을 (0, 0)으로 잡고 z는 0부터 w-1의 값을 지님



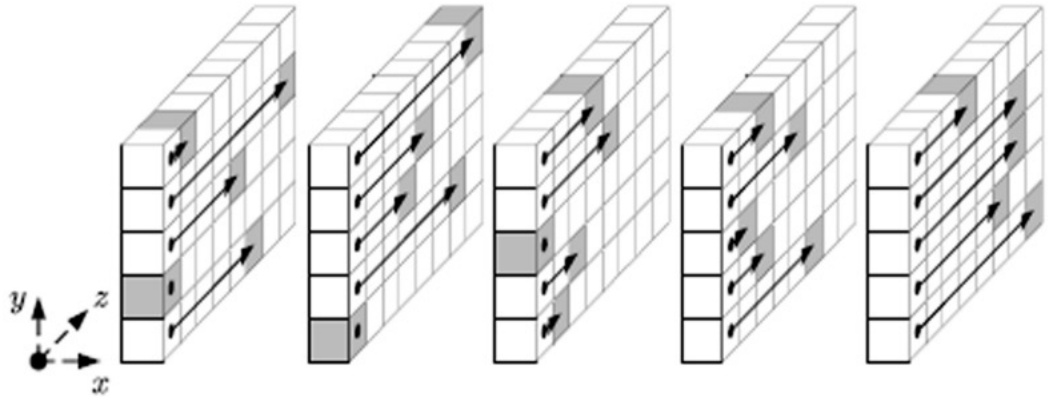
Keccak 동작 과정

- Θ(A) 함수(theta 함수)
  - ((x-1),z) Column 비트의 합과 ((x+1), (z-1))에 해당하는 Column 비트의 합을 XOR
  - 위 값을 A(x,y,z)의 값과 XOR하여 새로운 A'(x,y,z)에 저장하는 과정



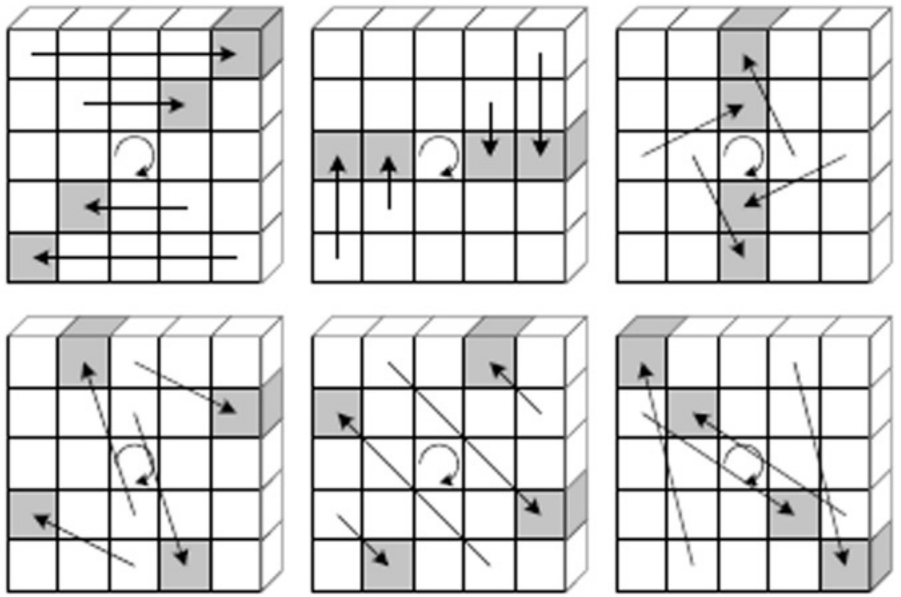
Keccak 동작 과정

- p(A) 함수(rho 함수)
  - 각각의 lane에서 정해진 offset 만큼 로테이션 하는 과정
  - (x,y)를 (1,0)으로 가정 후 A’[x,y,z] = A[x,y,(z-(t-1)(t+2)/2 mod w]를 계산
  - 위 계산을 통해 로테이션 가능
    - e.g. (x,y) = (y, (2x + 3y) mod 5) 식을 이용해 다음 로테이션 할 x, y를 계산
    - 이후 x, y값이 바뀔 때 마다 t의 값을 0부터 23까지 1씩 증가
    - x,y가 (0,0)을 제외한 24개의 lane을 로테이션



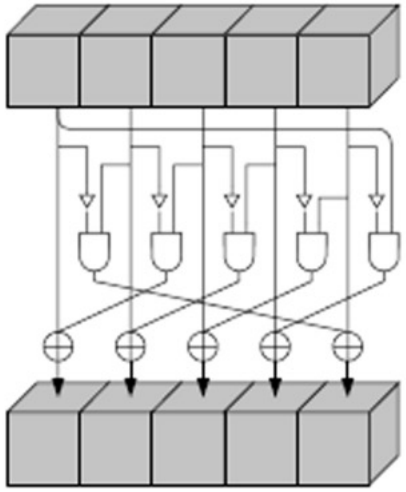
Keccak 동작 과정

- π(A) 함수(Pi 함수)
  - state내에서 lane의 자리를 재 배열하는 과정
  - 각 slice들의 부분들이 로테이션 되는 과정
  - A[(x+3y) mod 5,x,z] = A’ 의 식을 통해 로테이션 됨



Keccak 동작 과정

- x(A) 함수(chi 함수)
  - 각각의 row들의 값을 치환하는 과정
  - 오른쪽 2개의 비트가 곱셈 연산을 통해 비선형화된 후 XOR연산을 수행
  - (x+1,y,z) 값과 (x+2, y, z)값을 AND 연산
  - 위 값을 (x,y,z)값과 XOR 연산한 값을 (x,y,z)에 저장



코드 분석(tiny_sha3)

```
void sha3_keccakf(uint64_t st[25])
{
    // constants
    const uint64_t keccakf_rndc[24] = {
        0x0000000000000001, 0x0000000000008082, 0x800000000000808a,
        0x8000000080008000, 0x000000000000808b, 0x0000000080000001,
        0x8000000080008081, 0x8000000000008009, 0x000000000000008a,
        0x0000000000000088, 0x0000000080008009, 0x000000008000000a,
        0x000000008000808b, 0x800000000000008b, 0x8000000000008089,
        0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
        0x000000000000800a, 0x800000008000000a, 0x8000000080008081,
        0x8000000000008080, 0x0000000080000001, 0x8000000080008008
    };
    const int keccakf_rotc[24] = {
        1,  3,  6,  10, 15, 21, 28, 36, 45, 55, 2,  14,
        27, 41, 56, 8,  25, 43, 62, 18, 39, 61, 20, 44
    };
    const int keccakf_piln[24] = {
        10, 7,  11, 17, 18, 3, 5,  16, 8,  21, 24, 4,
        15, 23, 19, 13, 12, 2, 20, 14, 22, 9,  6,  1
    };

    // variables
    int i, j, r;
    uint64_t t, bc[5];
```

```
    // actual iteration
    for (r = 0; r < KECCAKF_ROUNDS; r++) {

        // Theta
        for (i = 0; i < 5; i++)
            bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];

        for (i = 0; i < 5; i++) {
            t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
            for (j = 0; j < 25; j += 5)
                st[j + i] ^= t;
        }

        // Rho Pi
        t = st[1];
        for (i = 0; i < 24; i++) {
            j = keccakf_piln[i];
            bc[0] = st[j];
            st[j] = ROTL64(t, keccakf_rotc[i]);
            t = bc[0];
        }

        //  Chi
        for (j = 0; j < 25; j += 5) {
            for (i = 0; i < 5; i++)
                bc[i] = st[j + i];
            for (i = 0; i < 5; i++)
                st[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
        }

        //  Iota
        st[0] ^= keccakf_rndc[r];
    }
```

Q & A