BLOWFISH

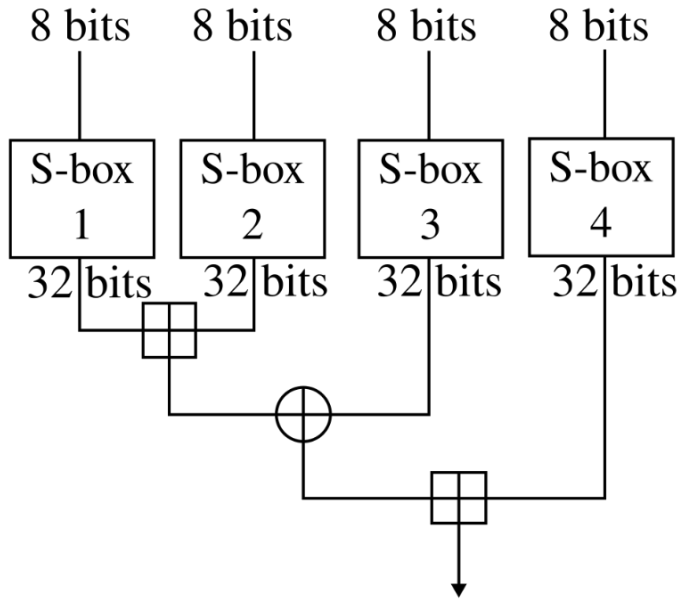
CUDA 구현

https://youtu.be/L-Ne4t4t2QE

BLOWFISH

- 블로피시(blowfish)는 1993년 브루스 슈나이어가 설계한 키(key) 방식의 대칭형 블록 암호이다.
- 소프트웨어에서 좋은 암호화 속도를 제공하며 현재까지 효과적인 암호화 분석 이 발견되지 않음
- 공개 도매인으로 누구나 사용가능, 수많은 암호화 제품군에 포함
- 키 길이 32~448bit, 블록 크기 64bit, Feistel 네트워크 16라운드
- 키 종속 S- 박스, 복잡한 키 스케줄의 특징

- 64 비트 블록 크기를 사용하므로
  4GB 이상의 파일을 암호화하는 데 사용되지 않을 것을 권장



관련 알고리즘

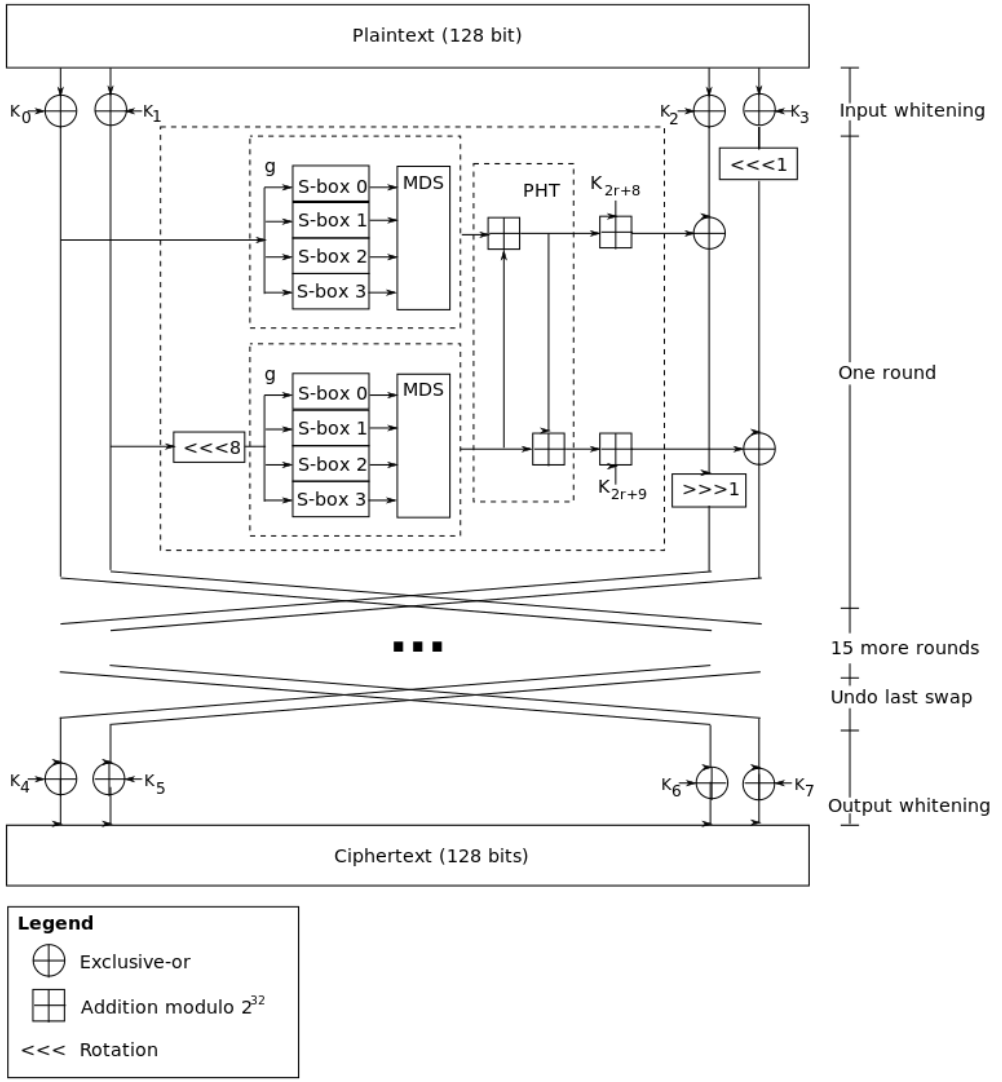
- Twofish
  Advanced Encryption Standard 콘테스트 의 5 개 결선 진출 자 중 하나
  알고리즘은 제한없이 누구나 무료로 사용하나 Blowfish보다는 사용하지 않음

- Bcrypt
  Blowfish 암호를 암호 해싱1999 년 USENIX 에서 발표, 192비트 해시

  레인보우 테이블 공격 으로부터 보호하기 위해 솔트통합

  반복 횟수를 늘려 속도를 늦출 수 있어
  계산 능력이 증가하더라도 무차별 대입 검색 공격에 대한 내성을 유지

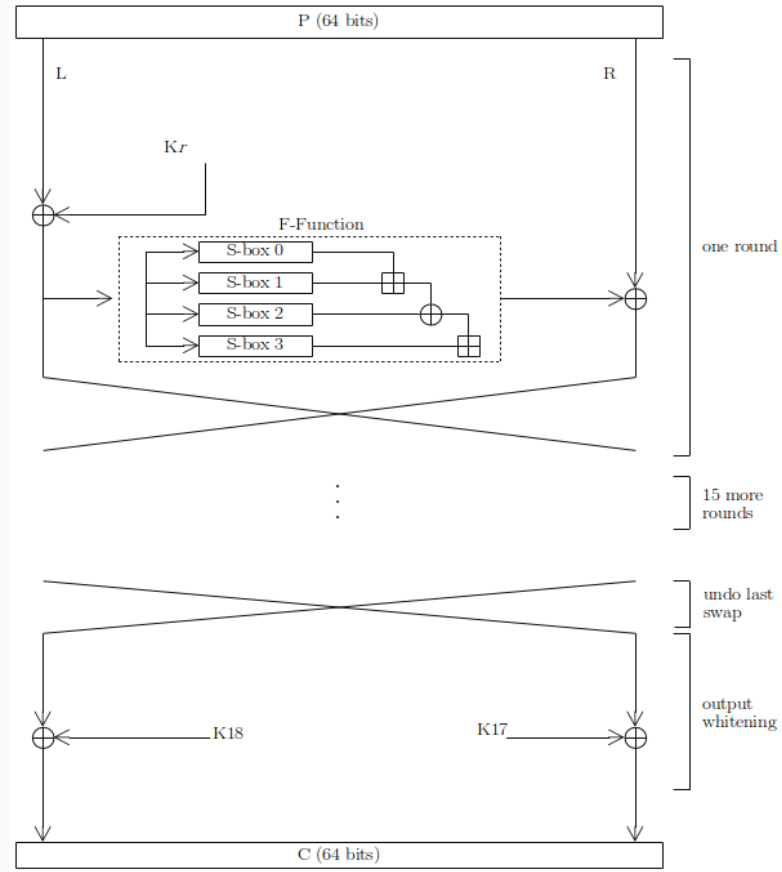
  Blowfish의 비싼 키 스케줄 단계를 발전하여 만듬



Encryption, Keyschdule

```
uint32_t P[18];
uint32_t S[4][256];

uint32_t f (uint32_t x) {
   uint32_t h = S[0][x >> 24] + S[1][x >> 16 & 0xff];
   return ( h ^ S[2][x >> 8 & 0xff] ) + S[3][x & 0xff];
}

void blowfish_encrypt(uint32_t *L, uint32_t *R) {
for (short r = 0; r < 16; r++) {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;
        swap(L, R);
    }
    swap(L, R);
    *R = *R ^ P[16];
    *L = *L ^ P[17];
}

void blowfish_decrypt(uint32_t *L, uint32_t *R) {
    for (short r = 17; r > 1; r--) {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;
        swap(L, R);
    }
    swap(L, R);
    *R = *R ^ P[1];
    *L = *L ^ P[0];
}
```

```
{
    /* initialize P box w/ key*/
    uint32_t k;
    for (short i = 0, p = 0; i < 18; i++) {
        k = 0x00;
        for (short j = 0; j < 4; j++) {
            k = (k << 8) | (uint8_t) key[p];
            p = (p + 1) % key_len;
        }
        P[i] ^= k;
    }

    /* blowfish key expansion (521 iterations) */
    uint32_t l = 0x00, r = 0x00;
    for (short i = 0; i < 18; i+=2) {
        blowfish_encrypt(&l, &r);
        P[i] = l;
        P[i+1] = r;
    }
    for (short i = 0; i < 4; i++) {
        for (short j = 0; j < 256; j+=2) {
            blowfish_encrypt(&l, &r);
            S[i][j] = l;
            S[i][j+1] = r;
        }
    }
}
```



CUDA 구현

```
#define BF_ENC(LL, R, S, P)                                    \
  (LL ^= P,                                                    \
   LL ^=                                                       \
   (((S[((int)(R >> 24) & 0xff)] + S[0x0100 + ((int)(R >> 16) & 0xff)]) ^ \
     S[0x0200 + ((int)(R >> 8) & 0xff)]) +                     \
    S[0x0300 + ((int)(R)&0xff)]) &                             \
   0xffffffffL)
```

```
void BF_decrypt(uint32_t *data,
    uint32_t l, r;
    const uint32_t *p, *s;
    p = P;
    s = &(S[0]);
    l = data[0];
    r = data[1];
    l ^= p[BF_ROUNDS + 1];
    BF_ENC(r, l, s, P: p[16]);
    BF_ENC(l, r, s, P: p[15]);
    BF_ENC(r, l, s, P: p[14]);
    BF_ENC(l, r, s, P: p[13]);
    BF_ENC(r, l, s, P: p[12]);
    BF_ENC(l, r, s, P: p[11]);
    BF_ENC(r, l, s, P: p[10]);
    BF_ENC(l, r, s, P: p[9]);
    BF_ENC(r, l, s, P: p[8]);
    BF_ENC(l, r, s, P: p[7]);
    BF_ENC(r, l, s, P: p[6]);
    BF_ENC(l, r, s, P: p[5]);
    BF_ENC(r, l, s, P: p[4]);
    BF_ENC(l, r, s, P: p[3]);
    BF_ENC(r, l, s, P: p[2]);
    BF_ENC(l, r, s, P: p[1]);
    r ^= p[0];
    data[1] = l & 0xffffffffL;
    data[0] = r & 0xffffffffL;
}
```

```
__device__ void BF_encrypt(uint32_t *data, const uint32_t *P, const uint32_t *S) {
    uint32_t l, r;
    const uint32_t *p, *s;
    p = P;
    s = &(S[0]);
    l = data[0];
    r = data[1];
    l ^= p[0];
    BF_ENC(r, l, s, P: p[1]);
    BF_ENC(l, r, s, P: p[2]);
    BF_ENC(r, l, s, P: p[3]);
    BF_ENC(l, r, s, P: p[4]);
    BF_ENC(r, l, s, P: p[5]);
    BF_ENC(l, r, s, P: p[6]);
    BF_ENC(r, l, s, P: p[7]);
    BF_ENC(l, r, s, P: p[8]);
    BF_ENC(r, l, s, P: p[9]);
    BF_ENC(l, r, s, P: p[10]);
    BF_ENC(r, l, s, P: p[11]);
    BF_ENC(l, r, s, P: p[12]);
    BF_ENC(r, l, s, P: p[13]);
    BF_ENC(l, r, s, P: p[14]);
    BF_ENC(r, l, s, P: p[15]);
    BF_ENC(l, r, s, P: p[16]);
    r ^= p[BF_ROUNDS + 1];
    data[1] = l & 0xffffffffL;
    data[0] = r & 0xffffffffL;
}
```

CUDA 구현

```
__global__ void BF(int len, const uint32_t *Data, uint32_t *PT) {
    int k = blockDim.x * blockIdx.x + threadIdx.x;

    BF_KEY bf_init = {

    int i;
    uint32_t *p, *P, *S, ri, in[2];
    const uint8_t *d, *end;
    P=bf_init.P;
    S=bf_init.S;

    p = P;

    if (len > ((BF_ROUNDS + 2) * 4))
        len = (BF_ROUNDS + 2) * 4;
    uint8_t *data = (uint8_t *)Data;

    d = data;

    end = &data[len];
    for (i = 0; i < BF_ROUNDS + 2; i++) {
        ri = *(d++);
        if (d >= end) {
            d = data;
        }
        ri <<= 8;
        ri |= *(d++);
        if (d >= end) {
            d = data;
        }
        ri <<= 8;
        ri |= *(d++);
        if (d >= end) {
            d = data;
        }
        ri <<= 8;
        ri |= *(d++);
        if (d >= end) {
            d = data;
        }
        p[i] ^= ri;
    }
    in[0] = 0L;
    in[1] = 0L;
    for (i = 0; i < BF_ROUNDS + 2; i += 2) {
        BF_encrypt(in, P,S);
        p[i] = in[0];
        p[i + 1] = in[1];
    }
    p = S;
    for (i = 0; i < 4 * 256; i += 2) {
        BF_encrypt(in, P,S);
        p[i] = in[0];
        p[i + 1] = in[1];
    }
    uint32_t pt[2];
    pt[0]=PT[2*k];
    pt[1]=PT[2*k+1];
    BF_encrypt(pt, P,S);
    PT[2*k]=pt[0];
    PT[2*k+1]=pt[1];
}
```

CUDA 구현

```
cudaMalloc((void**)&d_text, size: sizeof(uint32_t) * number * 2);
cudaMalloc((void**)&d_key, size: sizeof(uint32_t) * number*len);
start = clock();
for (int i = 0; i < 1; i++) {
    cudaMemcpy(d_text, text, count: sizeof(uint32_t) * number * 2, kind: cudaMemcpyHostToDevice);
    cudaMemcpy(d_key, key, count: sizeof(uint32_t) * number*len, kind: cudaMemcpyHostToDevice);

    BF <<< gridDim: blocknum, blockDim: number/blocknum >>> (keylen, d_key, d_text);

    cudaMemcpy(text, d_text, count: sizeof(uint32_t) * number * 2, kind: cudaMemcpyDeviceToHost);
}
```

최적화

- 단순한 암호이므로 사용하는 메모리에 따라 속도 차이
  pinned memory, constant memory, shared memory

- streaming kernels

- 32~448bit의 키 길이 사용하므로

  작은 키의 경우 레인보우 테이블

Q & A