2024 암호분석경진대회 - 비밀분쇄기팀 6번 문제 풀이

Table 1: 최적화 결과.

	Original	Improved	Speedup
Run 1	25400	6951	3.65
Run 2	25302	6902	3.67
Run 3	25164	6883	3.66
Average	**25288.7**	**6912.0**	**3.66**

1 프로그램 분석

주어진 프로그램은 N-byte의 평문을 받아 N-byte의 암호문과 8-byte의 auth tag를 생성하는 Galois/Counter Mode의 변형된 구현이다. 함수 `CTR_mode`는 먼저 `key_scheduling`을 호출하여 8-byte의 마스터 키로부터 79개의 8-byte 키를 추가로 생성한다. 그 후 평문의 각 8-byte 블록에 대해 `block_encryption`을 호출하고, 해당 함수는 80라운드의 `ROUND_FUNC`을 적용하여 암호문을 생성한다. 생성한 암호문은 `AUTH_mode` 함수에 넘겨지는데, 해당 함수는 평문 길이를 기반으로 한 초기값으로부터 $GF(2^{64})$에서의 곱셈 및 암호문과의 xor 연산을 통해 최종적으로 8-byte auth tag를 생성한다.

Table 2: 실험에 사용된 시스템.

CPU	2 x Intel Xeon Gold 6130
Memory	8 x DDR4-2666 32GB
Compiler	GCC 11.4.0

표 2의 시스템에서 실험한 결과 주어진 구현은 192-byte 입력에 대해 약 25k 사이클이 소요되었으며, 그 중 `CTR_mode`가 약 16k, `AUTH_mode`가 약 9k 사이클이 걸렸다. 따라서 두 함수 모두 최적화가 필요하다.

2 적용한 최적화 기법

이번 절에서는 적용한 최적화 기법들을 서술하고자 한다.

2.1 `CTR_mode` 함수 최적화

주어진 구현에서 `CTR_mode` 함수는 8-bit xor, 덧셈 등의 연산을 통해 각 라운드를 처리하고 있다. 그러나 64-bit 시스템은 64-bit 연산 인스트럭션을 지원하기 때문에 8-bit 연산 인스트럭션을 주로 사용하면 프로세서의 IPC(Instruction Per Cycle)을 낭비하는 셈이다. **따라서 평문을 8블록(64-byte)씩 모아 64-bit 연산으로 처리하여 최적화할 여지가 있다.** 최적화에 앞서 먼저 다음과 같은 보조 함수들을 정의하였다.

```
static inline uint64_t pack64(uint8_t x0, uint8_t x1, uint8_t x2, uint8_t x3, uint8_t x4, uint8_t x5,
    uint8_t x6, uint8_t x7) {
  return x0 | ((uint64_t)x1 << 8) | ((uint64_t)x2 << 16) | ((uint64_t)x3 << 24) | ((uint64_t)x4 << 32) |
    ((uint64_t)x5 << 40) | ((uint64_t)x6 << 48) | ((uint64_t)x7 << 56);
}

static inline uint64_t dup8(uint8_t a) {
  return a * 0x0101010101010101;
}

static inline uint64_t rol64(uint64_t x, uint8_t r) {
  switch (r) {
    case 0: return x;
    case 1: return ((x << 1) & 0xFEFEFEFEFEFEFEFE) | ((x >> 7) & 0x0101010101010101);
    case 2: return ((x << 2) & 0xFCFCFCFCFCFCFCFC) | ((x >> 6) & 0x0303030303030303);
    case 3: return ((x << 3) & 0xF8F8F8F8F8F8F8F8) | ((x >> 5) & 0x0707070707070707);
    case 4: return ((x << 4) & 0xF0F0F0F0F0F0F0F0) | ((x >> 4) & 0x0F0F0F0F0F0F0F0F);
    case 5: return ((x << 5) & 0xE0E0E0E0E0E0E0E0) | ((x >> 3) & 0x1F1F1F1F1F1F1F1F);
    case 6: return ((x << 6) & 0xC0C0C0C0C0C0C0C0) | ((x >> 2) & 0x3F3F3F3F3F3F3F3F);
    case 7: return ((x << 7) & 0x8080808080808080) | ((x >> 1) & 0x7F7F7F7F7F7F7F7F);
```

```
  }
}

static inline uint64_t bytewise_add(uint64_t a, uint64_t b) {
  uint64_t c;
  for (int i = 0; i < 8; ++i) {
    ((uint8_t*)&c)[i] = ((uint8_t*)&a)[i] + ((uint8_t*)&b)[i];
  }
  return c;
}
```

pack64는 주어진 8개의 8-bit 값을 하나의 64-bit 값으로 이어붙이는 함수이다. dup8의 경우 주어진 1개의 8-bit 값을 8번 복사하여 64-bit 값으로 만드는 함수인데 위와 같이 곱셈 한번으로 구현할 수 있다. rol64는 64-bit 값에서 바이트 단위로 left circular shift를 적용하는 함수인데, 각 바이트를 추출하여 적용하는 대신 bitmask를 활용하여 64-bit를 한번에 처리하도록 하였다. bytewise_add는 64-bit 값 두개를 받아 byte 단위로 더하는 함수인데, 보기에는 각 byte를 더하는 것과 성능 차이가 없을 것 같지만 실제로는 paddb 인스트럭션 하나로 컴파일되기 때문에 매우 빠르다. 위 보조 함수들을 이용하여 CTR_mode 함수의 80 라운드 처리 루프를 다음과 같이 수정하였다.

```
uint64_t tmp[8];
tmp[0] = pack64(i * 8 + 0, i * 8 + 1, i * 8 + 2, i * 8 + 3, i * 8 + 4, i * 8 + 5, i * 8 + 6, i * 8 + 7);
tmp[1] = dup8(NONCE1);
tmp[2] = dup8(NONCE2);
tmp[3] = dup8(NONCE3);
tmp[4] = dup8(NONCE4);
tmp[5] = dup8(NONCE5);
tmp[6] = dup8(NONCE6);
tmp[7] = dup8(NONCE7);
for (int r = 0; r < NUM_ROUND; r++) {
  uint64_t tmp0 = tmp[0];
  tmp[0] = rol64(dup8(RK[r][1]) ^ bytewise_add(tmp[0], dup8(RK[r][1]) ^ tmp[1]), 1);
  tmp[1] = rol64(dup8(RK[r][2]) ^ bytewise_add(tmp[1], dup8(RK[r][2]) ^ tmp[2]), 2);
  tmp[2] = rol64(dup8(RK[r][3]) ^ bytewise_add(tmp[2], dup8(RK[r][3]) ^ tmp[3]), 3);
  tmp[3] = rol64(dup8(RK[r][4]) ^ bytewise_add(tmp[3], dup8(RK[r][4]) ^ tmp[4]), 4);
  tmp[4] = rol64(dup8(RK[r][5]) ^ bytewise_add(tmp[4], dup8(RK[r][5]) ^ tmp[5]), 5);
  tmp[5] = rol64(dup8(RK[r][6]) ^ bytewise_add(tmp[5], dup8(RK[r][6]) ^ tmp[6]), 6);
  tmp[6] = rol64(dup8(RK[r][7]) ^ bytewise_add(tmp[6], dup8(RK[r][7]) ^ tmp[7]), 7);
  tmp[7] = tmp0;
}
```

기존에는 길이 8의 8-bit 중간 결과 배열을 사용하였으나, 길이 8의 64-bit 배열(tmp)로 수정하였다. 초기값은 pack8 및 dup8 보조함수를 통해 기존과 같게 초기화해 주었다. 각 라운드 역시 rol64, dup8, bytewise_add 보조함수들을 통해 최대한 64-bit 연산으로 처리해 주었다. **최적화 후 실행시간은 약 5k cycle로 기존 16k cycle에서 약 3배의 성능 향상이 있었다.**

2.2 AUTH_mode 함수 최적화

AUTH_mode 함수의 경우 POLY_MUL_RED 함수에서 대부분의 시간을 소모하는데 이 함수는 기본적으로 64-bit carryless multiplication을 수행하는 함수이다. Carryless multiplication은 pclmulqdq 등의 인스트럭션으로 아주 빠르게 구현할 수 있지만 해당 명령어를 사용하지 않는 것이 문제의 조건이므로 다른 방법을 생각해 보아야 한다.

POLY_MUL_RED 함수는 크게 3군데서 호출되는데 이 중 두번은 피연산자 중 하나(AUTH_nonce)가 모든 블록에 대해 같다. 따라서 두 경우를 나누어 최적화를 생각해 볼 수 있다. **아래 두 최적화를 적용한 후 실행시간은 약 2.5k cycle로 기존 9k cycle에서 약 4.8배의 성능 향상이 있었다.**

2.2.1 POLY_MUL_RED 최적화 - 피연산자 중 하나가 상수인 경우

피연산자 중 하나가 상수인 경우 아래와 같이 lookup table을 생성하는 전처리를 통해 최적화할 수 있다.

```
uint64_t CMUL_DB[DB_SIZE][2];
uint64_t H = pack64(num_enc_auth, num_enc_auth ^ NONCE1, num_enc_auth & NONCE2, num_enc_auth | NONCE3,
    num_enc_auth ^ NONCE4, num_enc_auth & NONCE5, num_enc_auth | NONCE6, num_enc_auth ^ NONCE7);

CMUL_DB[0][0] = 0;
CMUL_DB[0][1] = 0;
for (int i = 1, j = 0; i < DB_SIZE; i *= 2, ++j) {
  CMUL_DB[i][0] = H << j;
  CMUL_DB[i][1] = j > 0 ? H >> (64 - j) : 0;
  for (int k = i + 1; k < 2 * i; ++k) {
    CMUL_DB[k][0] = CMUL_DB[k - i][0] ^ CMUL_DB[i][0];
    CMUL_DB[k][1] = CMUL_DB[k - i][1] ^ CMUL_DB[i][1];
  }
}
```

H는 기존 코드에서 AUTH_nonce의 초기값이고, 전처리를 통해서 CMUL_DB[i][0]에 $H * i$의 하위 64-bit, CMUL_DB[i][1]에 상위 64-bit를 채우려고 한다. 2의 승수에 대해서 CMUL_DB[2^j]는 H<<j이므로 먼저 이 값들을 채우고, 나머지 값은

CMUL_DB[k]=CMUL_DB[k-i] $\oplus$ CMUL_DB[i]라는 점을 활용하여 선형시간 내에 CMUL_DB 전체를 초기화할 수 있다. DB_SIZE로는 다양한 값이 가능한데 여기서는 256으로 하여 H와 1바이트 값의 carryless multiplication 결과를 모두 저장할 수 있게 하였다. 다음은 위 전처리 결과를 활용한 POLY_MUL_RED 함수의 최적화 결과이다.

```
static inline void POLY_MUL_RED_IMP_DB(uint8_t *INOUT, uint64_t (*db)[2]) {
  uint64_t p2 = *(uint64_t *)INOUT;
  uint64_t result0 = 0, result1 = 0;
  for (int i = 0; i < 64; i+= DB_SIZE_LOG) {
    uint64_t L = db[(p2 >> i) & (DB_SIZE - 1)][0];
    uint64_t H = db[(p2 >> i) & (DB_SIZE - 1)][1];
    result0 ^= L << i;
    if (i > 0) result1 ^= L >> (64 - i);
    result1 ^= H << i;
  }
  result0 ^= result1;
  result0 ^= result1 << 9;
  result0 ^= result1 >> 55;
  result0 ^= (result1 >> 55) << 9;
  *(uint64_t*)INOUT = result0;
}
```

기존 루프는 64번 실행되며 첫번째 피연산자의 비트 하나를 확인하고 두번째 피연산자로 xor를 적용하였으나, 개선된 구현에서는 루프를 8번만 실행하며 첫번째 피연산자의 8비트 값을 인덱스로 하여 전처리 결과를 불러와서 xor을 적용하고 있다.

위 방법에서는 128-bit를 상위 64-bit와 하위 64-bit로 나누어서 처리하는 오버헤드가 있으므로 최종 버젼에서는 후술할 Karatsuba 알고리즘을 응용하여 64-bit 전처리 배열을 3개 만드는 방식으로 변경하였다. (제출 코드의 POLY_MUL_RED_IMP_DB3)

2.2.2 POLY_MUL_RED 최적화 - 일반적인 경우

피연산자가 상수가 아닌 경우에는 위와 같이 전처리 기반 최적화가 불가능하다. 따라서 이 경우에는 Karatsuba 알고리즘 [1]을 이용하여 다음과 같이 구현하였다.

```
static inline uint64_t clsq_32b(uint64_t a) {
  uint64_t c = 0;
  uint64_t DB[4] = {0, a, a << 1, a ^ (a << 1)};
  for (int i = 0; i < 32; i+= 2) {
    c ^= DB[(a >> i) & 3] << i;
  }
  return c;
}

static inline void POLY_MUL_RED_IMP_SQ(uint8_t *INOUT) {
  uint64_t p1 = *(uint64_t *)INOUT;
  uint32_t p1l = p1;
  uint32_t p1h = p1 >> 32;
  uint64_t z0 = clsq_32b(p1l);
  uint64_t z2 = clsq_32b(p1h);
  uint64_t z1 = clsq_32b(p1l ^ p1h) ^ z0 ^ z2;
  uint64_t result0 = z0 ^ (z1 << 32);
  uint64_t result1 = (z1 >> 32) ^ z2;
  result0 ^= result1;
  result0 ^= result1 << 9;
  result0 ^= result1 >> 55;
  result0 ^= (result1 >> 55) << 9;
  *(uint64_t*)INOUT = result0;
}
```

먼저, 이 경우에는 기존 구현에서 피연산자 2개가 같은 값(AUTH_inter)이였기 때문에 피연산자를 하나만 받아 제곱하는 방식으로 구현하였다. clsq_32b 함수는 위와 같은 아이디어로 전처리를 통해 크기 4의 lookup table을 생성하여 32-bit carryless multiplication을 실행하는 함수이다. 64-bit 곱셈을 32-bit 곱셈으로 구현하려면 일반적으로 clsq_32b 함수를 4번 호출해야 하지만 Karatsuba 알고리즘을 활용하여 해당 함수를 3번만 호출하는 방식으로 구현된 것을 볼 수 있다.

3 평가

임의 길이의 평문과 키를 랜덤으로 생성하여 ENC_AUTH와 ENC_AUTH_IMP의 결과값을 비교하여 여러가지 테스트 벡터에 대해 답이 맞음을 검증하였다. (validate 함수 참고)

벤치마크의 경우 기존에는 1000번 반복한 평균값을 출력하도록 되어있으나 측정 편차를 줄이기 위해 100000번 반복하여 측정하였다. 컴파일 및 실행 커맨드는 아래와 같이 사용하였으며 3회의 측정 결과를 풀이 첫 부분의 표 1에 정리하였다. 문제의 조건에 따라 Ubuntu 22.04 LTS에 기본으로 설치된 gcc-11을 사용하였으며 특정 명령어 셋을 활성화하는 옵션은 사용하지 않고 최적화 옵션(-O3)만을 사용하였다.

```
$ gcc-11 contest.c -O3 -o contest && ./contest
--- TEST VECTOR ---
test pass.
--- BENCHMARK ---
Original implementation runs in ..................    25164 cycles
Improved implementation runs in ..................     6883 cycles
```

References

[1] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.