2020 암호분석경진대회 답안제출

2020. 08. 31

참가자 1	**성 명**	이창원
	소속	서울시립대학교 수학과
	휴대폰	
	E-mail	

참가자 2	**성 명**	장호빈
	소속	서울시립대학교 수학과
	휴대폰	
	E-mail	

참가자 3	**성 명**	김인성
	소속	서울시립대학교 수학과
	휴대폰	
	E-mail	

2번 문제 답안

풀이 : 개요

구현 타겟 플랫폼은 아두이노 우노 (8비트 AVR 프로세서)로써 제한된 메모리 내에서 마이크로컨트롤러를 최적화해야 한다. key_gen, enc 내부만을 수정하기 때문에 두 함수 내부의 불필요한 연산을 제거, 함수 최적화를 통해 실행 속도를 증가시킬 수 있다. 우선 전체적인 과정을 살펴보면, key_gen 함수를 이용하여 입력 인자 rnd, key를 통해 rnd를 업데이트한 후, 업데이트된 rnd와 text를 enc 함수를 이용하여 새로운 text 값으로 업데이트를 한다. 그러므로 key_gen 에서는 불필요한 연산을 하지 않고 기존에 주어진 함수와 rnd 값이 일치하도록 만든다. 또한 enc 내부에서도 기존의 업데이트된 text 값과 일치하는 text 값을 생성하도록 만들어야 한다.

문제에 주어진 그대로 실행하였을 때 벤치마크 결과는 3909ms이다. 아래의 최적화 구현 방법을 통하여 기존보다 79.5% 가량 속도가 증가한 **801ms**의 벤치마크 결과를 얻었다. 아두이노 코드는 num2_UOS_mathematics.ino 파일을 첨부하였다.

1. 구현 기법 상세

1) key_gen

기존의 원리는 128번의 라운드 동안 key_in[0], key_in[1] = key[0], key[1]을 이용하여, 짝수 라운드에서는 key_in[0] = ROL8(key_in[0], 1) + ROL8(key_in[0], 5), key_in[1] = ROL8(key_in[1], 3) + ROL8(key_in[1], 7)을 계산한다. 그리고 key_in[0], key_in[1] = (key_in[0]+key_in[1]), (key_in[0] ^ key_in[1])을 계산한다.

홀수 라운드에서는 key_p=(u16*) key_in을 이용하여 *key_p = ROL16(*key_p, 1) + ROL16(*key_p, 9) + ROL16(con, (i%16))을 계산하여 새로운 key_in[0], key_in[1]에 대해 위와 마찬가지로,
key_in[0], key_in[1] = (key_in[0]+key_in[1]), (key_in[0] ^ key_in[1])을 계산한다.

불필요한 연산을 살펴보면, 우선 변수 key_in 에 값을 저장할 필요가 없다. 입력 인자인 key를 이용하여 라운드를 진행하면서 rnd를 업데이트하는 것이므로 key_in이라는 변수 대신 입력 인자인 key를 그대로 활용할 수 있다. 또한 tmp1, tmp2의 경우 계산을 위한 변수이지만, 결과적으로 매 라운드를 진행하면서 업데이트된 key[0], key[1]과 rnd[2i+0], rnd[2i+1] 가 같은 값을 가지는 것이므로 tmp1, tmp2 대신 rnd[2i+0], rnd[2i+1]을 사용하여 key[0]+key[1], key[0]^key[1]을 계산할 수 있다.

또한, for문 내의 if문의 경우, 비교 연산을 통하여 홀수, 짝수 라운드를 나누는 것에 불과하기에 이를 사용 하지 않고 2개의 라운드를 하나로 볼 수 있다. 그리고 for문의 경우 128번의 라운드를 모두 for 문을 이용하는 것보다 모든 라운드를 풀어쓰거나, 일정 라운드 별로 for문을 새로 묶을 수 있다. 실험 결과, key_gen 과 enc 모두 기존의 라운드 16개를 묶어 for문을 새로 만들었을 때 시간이 최소화가 되었다.
(rnd[32i+0] ~ rnd[32i+31]을 사용한 것이 for문의 한 라운드)
그리고 매 라운드마다 ROL16(con, (i%16))을 계산하지 않고 미리 계산하여 불필요한 연산을 없앨 수 있다.

아두이노 우노의 경우 flash 메모리 32KB, SRAM 2KB를 사용가능 하다. 그러므로 짝수 라운드의 경우 ROL8(key[0], 1) + ROL8(key[0], 5) 와 ROL8(key[1], 3) + ROL8(key[1], 7)를 look-up table을 활용하여 가용 메모리 내에서 처리할 수 있다. (기존의 key_in 대신 key 사용) 이를 각각 key_table1, key_table2 라 하였다. 그리고 홀수 라운드의 경우 ROL16(*key_p, 1) 과 ROL16(*key_p, 9)는 서로 key[0]와 key[1]이 swap된 형태이다. 이를 이용하여 *key_p=ROL16(*key_p, 1)을 구한 후, 해당 결과의 key[0], key[1]을 swap 하여 새로운
*(u16 *) key 값을 반환하는 함수 swap_arr을 새로 만들어 사용하였다. 그리고 일반적으로 함수를 호출하는 것보다 직접 적용하는 것이 빠름을 이용하여 ROL16(*key_p, 1)을 함수 호출이 아닌 직접 적용하였다.

2번 문제 답안

그리고 rnd[0] ~ rnd[255]를 모두 출력한 결과, 특정 부분에서 반복이 일어남을 발견하여 이를 활용하였다. 모든 key ({0x00, 0x00} ~ {0xFF, 0xFF})에 대해 측정한 결과, 일반적으로 rnd[0] ~ rnd[95] 까지 구하였을 때, rnd[64] ~ rnd[95] = rnd[96] ~ rnd[127] = rnd[128] ~ rnd[159] = rnd[160] ~ rnd[191] = rnd[192] ~ rnd[223] = rnd[224] ~ rnd[255]을 만족함을 확인하였다. 그러므로 rnd[96] ~ rnd[255] 는 key를 이용한 연산이 아닌, rnd[64] ~ rnd[95] 결과를 복사하는 연산만을 활용하였다.

2) enc

기존의 원리는 key_gen을 이용하여 업데이트된 rnd를 활용하여 text를 업데이트 하는 것이다. key_gen 과 마찬가지로 128번의 라운드를 실행하며, 짝수라운드의 경우 text_p=(u16*) text_in을 이용하여, *text_p = ROL16(*text_p, 4)를 계산한다. 그리고 text_in[0] = text_in[0] + rnd[2i+0], text_in[1] = text_in[1] ^ rnd[2i+1]을 계산한다. 홀수 라운드의 경우 *text_p = ROL16(*text_p, 8)을 계산한다. 그리고 위와 마찬가지로 text_in[0] = text_in[0] + rnd[2i+0], text_in[1] = text_in[1] ^ rnd[2i+1]을 계산한다. 그리고 text를 업데이트한다.

불필요한 연산을 살펴보면, text_in의 결과와 업데이트된 text의 결과가 같기 때문에 text_in 이라는 변수는 필요없는 변수이다. 또한 key_gen에서와 마찬가지로 for문 내의 if문은 비교 연산을 통해 짝수, 홀수 라운드를 구분하는 용도이므로 비교 연산을 사용하지 않고 for문을 활용할 수 있다. 그리고 key_gen과 마찬가지로 기존의 라운드 16개를 for문으로 묶었을 때 시간이 가장 빨랐다.
(rnd[32i+0] ~ rnd[32i+31]을 사용한 것이 for문의 한 라운드)

그리고 *text_p=ROL16(*text_p, 8)의 경우, text[0], text[1]를 swap 한 것이므로 이 과정 없이 rnd와의 연산 순서를 조정하여 같은 결과를 얻을 수 있다. 순서는 다음과 같다. (기존의 text_in 대신 text 사용)

① *text_p = ROL16(*text_p,4)
② text[0] = (text[0] + rnd[8k+0]) ^ rnd[8k+3], text[1] = (text[1] ^ rnd[8k+1]) + rnd[8k+2]
③ *text_p = ROL16(*text_p,4)
④ text[0] = (text[0] ^ rnd[8k+5]) + rnd[8k+6], text[1] = (text[1] + rnd[8k+4]) ^ rnd[8k+7]
⑤ rnd와의 연산을 위 과정처럼 반복한다.

또한 함수 호출보다 함수를 직접 적용하는 것이 일반적으로 빠르므로, ROL16(*text_p, 4)을 함수 호출이 아닌 직접 적용하였다.

2번 문제 답안

따라서 최적화 과정을 요약하면 다음과 같다.

순서	key_gen	enc
1	key_in 제거 (key_in 대신 key 사용)	text_in 제거 (text_in 대신 text 사용)
2	tmp1, tmp2 제거 (rnd를 이용하여 계산)	*
3	if문 사용하지 않고, 기존 라운드 16개를 for문의 한 라운드로 묶음, ROL16(con, (i%16)) 미리 계산	if문 사용하지 않고, 기존 라운드 16개를 for문의 한 라운드로 묶음
4	ROL8 계산 부분은 table 활용 (key_table1 : ROL8(key[0],1)+ROL8(key[0],5) (key_table2 : ROL8(key[1],3)+ROL8(key[1],7)	ROL16(*text_p,8)을 계산하지 않고 rnd와의 연산 순서를 조정하여 같은 결과를 얻도록 수정
5	ROL16(*key_p,9)는 ROL16(*key_p,1) 계산 후 key[0], key[1]을 swap한 것임을 이용	*
6	ROL16(*key_p,1) 계산 시 함수 호출 대신 직접 적용	ROL16(*text_p,4) 계산 시 함수 호출 대신 직접 적용
7	모든 key ({0x00, 0x00} ~ {0xFF, 0xFF})에 대해 rnd[64]~rnd[95] = rnd[96]~rnd[127] = ... = rnd[224]~rnd[255]를 만족함을 이용	*

2. 테스트 벡터 확인 결과

key = { {0x12, 0x34}, {0x9A, 0xBD}, {0x11, 0x22} }, text = { {0x56, 0x78}, {0xDE, 0xF0}, {0x33, 0x44} }, out_text = { {0x50, 0x3F}, {0x88, 0x28}, {0x7F, 0x33} }에 대해 test vector를 통과하였다.

3. 벤치마크 결과

문제에 주어진 코드를 그대로 실행하였을 때, 벤치마크 결과 3909 ms 가 측정된다. 위 최적화 과정에 따른 시간 측정 변화는 다음과 같다. 최종적으로 약 79.5% 향상된 **801 ms** 의 최적화 결과를 얻었다.

순서	key_gen	enc	결과(ms)
1	key_in 제거 (key_in 대신 key 사용)	text_in 제거 (text_in 대신 text 사용)	3825
2	tmp1, tmp2 제거 (rnd를 이용하여 계산)	*	3825
3	if문 사용하지 않고, 기존 라운드 16개를 for문의 한 라운드로 묶음, ROL16(con, (i%16)) 미리 계산	if문 사용하지 않고, 기존 라운드 16개를 for문의 한 라운드로 묶음	1510
4	ROL8 계산 부분은 table 활용 (key_table1 : ROL8(key[0],1)+ROL8(key[0],5) (key_table2 : ROL8(key[1],3)+ROL8(key[1],7)	ROL16(*text_p,8)을 계산하지 않고 rnd와의 연산 순서를 조정하여 같은 결과를 얻도록 수정	1283
5	ROL16(*key_p,9)는 ROL16(*key_p,1) 계산 후 key[0], key[1]을 swap한 것임을 이용	*	1068
6	ROL16(*key_p,1) 계산 시 함수 호출 대신 직접 적용	ROL16(*text_p,4) 계산 시 함수 호출 대신 직접 적용	1068
7	모든 key ({0x00, 0x00} ~ {0xFF, 0xFF})에 대해 rnd[64]~rnd[95] = rnd[96]~rnd[127] = ... = rnd[224]~rnd[255]를 만족함을 이용	*	801

2번 문제 답안

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>3909
----------
```

(기본 결과)

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>3825
----------
```

(1,2단계 수행 후)

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>1510
----------
```

(3단계 수행 후)

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>1283
----------
```

(4단계 수행 후)

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>1068
----------
```

(5,6단계 수행 후)

```
----------------
 TEST_VECTOR
 ----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>>801
----------
```

(7단계 수행 후 최종 결과)