

암호학

Project #4

소프트웨어학부

2017012251

윤영훈

1. source code

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 all: test.o mRSA.o
5     $(CC) $(CFLAGS) -o test test.o mRSA.o -lbsd
6
7 test.o: test.c mRSA.h
8     $(CC) $(CFLAGS) -c test.c
9
10 mRSA.o: mRSA.c mRSA.h
11     $(CC) $(CFLAGS) -c mRSA.c
12
13 clean:
14     rm -rf *.o
15     rm -rf test
```

makefile -lbsd 추가

```
5  #include <stdlib.h>
6  #include "mRSA.h"
7
8  #include <bsd/stdlib.h>
```

mRSA.c #include <bsd/stdlib.h> 추가

```
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "mRSA.h"
8
9  #include <bsd/stdlib.h>
```

test.c #include <bsd/stdlib.h> 추가

source code를 설명하기에 앞서 주어진 makefile과 test.c, mRSA.c로는 linux 환경에서 arc4random 함수를 정상적으로 사용할 수 없으므로 다음과 같이 변경하였다.

① 지역 함수

```
80 uint64_t gcd(uint64_t a, uint64_t b){
81     uint64_t temp = 0, result = 0;
82
83     while (a != 0 && b != 0){
84         temp = a;
85         a = b;
86         b = temp % b;
87     }
88
89     if (a == 0)
90         result = b;
91     else
92         result = a;
93
94     return result;
95 }
```

gcd

```
97 uint64_t mul_inv(uint64_t a, uint64_t m){
98     uint64_t d0 = a, d1 = m;
99     uint64_t x0 = 1, x1 = 0;
100
101     uint64_t q = d0 / d1;
102     uint64_t d2 = d0 - q * d1;
103     uint64_t x2 = x0 - q * x1;
104
105     while (d2 > 1){
106         q = d0 / d1;
107         d2 = d0 - q * d1;
108         x2 = x0 - q * x1;
109
110         d0 = d1;
111         d1 = d2;
112         x0 = x1;
113         x1 = x2;
114     }
115
116     if (d2 == 1)
117         return (x2 > (uint64_t)1<<63 ? x2+m : x2);
118     else
119         return 0;
120 }
```

mul_inv

```
122 uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)
123 {
124     while (a >= m || b >= m){
125         if (a >= m) a -= m;
126         if (b >= m) b -= m;
127     }
128
129     return ((a >= m-b) ? a-(m-b) : a+b);
130 }
```

mod_add

```
132 uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m)
133 {
134     uint64_t r=0;
135
136     while (b > 0){
137         if (b & 1) r = mod_add(r, a, m);
138         b = b >> 1;
139         a = mod_add(a, a, m);
140     }
141
142     return r;
143 }
```

mod_mul

```

145 uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m)
146 {
147     uint64_t r=1;
148
149     while (b > 0){
150         if (b & 1) r = mod_mul(r, a, m);
151         b = b >> 1;
152         a = mod_mul(a, a, m);
153     }
154
155     return r;
156 }

```

mod_pow

```

158 int miller_rabin(uint64_t n)
159 {
160     if (n % 2 == 0 && n != 2) return COMPOSITE;
161
162     int k = 0;
163     uint64_t q = n-1;
164
165     while ((q % 2) == 0){
166         q /= 2;
167         k++;
168     }
169
170     for (int i=0; i<ALEN && a[i] < n-1; i++){
171         uint64_t x = mod_pow(a[i], q, n);
172         int count = 0;
173
174         if (x == 1) continue;
175
176         for (int j = 0; j < k; j++){
177             if (mod_pow(x, 1 << j, n) == n-1){
178                 count++;
179                 break;
180             }
181         }
182
183         if (count == 0) return COMPOSITE;
184     }
185     return PRIME;
186 }

```

miller_rabin

이전 프로젝트들에서 구현했던 gcd, mul_inv, mod_add, mod_mul, mod_pow, miller_rabin 총 6개의 지역 함수들을 사용했다. 이후 mRSA.h에서 지역 함수들 선언 및 PRIME, ALEN 등 필요한 변수를 define했다.

```

5  #ifndef mRSA_H
6  #define mRSA_H
7
8  #include <stdint.h>
9
10 #define PRIME 1
11 #define COMPOSITE 0
12 #define MINIMUM_N 0x8000000000000000
13 #define ALEN 12
14
15 void mRSA_generate_key(uint64_t *e, uint64_t *d, uint64_t *n);
16 int mRSA_cipher(uint64_t *m, uint64_t k, uint64_t n);
17
18 uint64_t gcd(uint64_t a, uint64_t b);
19 uint64_t mul_inv(uint64_t a, uint64_t m);
20 uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m);
21 uint64_t mod_sub(uint64_t a, uint64_t b, uint64_t m);
22 uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m);
23 uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m);
24 int miller_rabin(uint64_t n);
25
26 #endif

```

mRSA.h

② mRSA_generate_key

```
16 void mRSA_generate_key(uint64_t *e, uint64_t *d, uint64_t *n)
17 {
18     uint64_t p, q;
19     uint64_t lambda_n;
20
21     uint64_t x = arc4random_uniform(0x7fffffff) + 0x80000000;
22     int num = 0;
23
24     while(1){
25         if (miller_rabin(x) && !num){
26             p = x;
27             x = arc4random_uniform(0x7fffffff) + 0x80000000;
28             num++;
29         }
30
31         else if (miller_rabin(x) && num){
32             q = x;
33             if (p * q > MINIMUM_N){
34                 *n = p * q;
35                 break;
36             }
37             else {
38                 num = 0;
39                 x = arc4random_uniform(0x7fffffff) + 0x80000000;
40             }
41         }
42         x++;
43     }
```

mRSA_generate_key 1

```
45     lambda_n = (p-1) * (q-1) / gcd(p-1, q-1);
46
47     // e, d
48
49     uint64_t random1, random2, i;
50
51     while (1){
52         random1 = arc4random_uniform(p-1);
53         random2 = arc4random_uniform(q-1);
54
55         i = random1 * random2 / gcd(p-1, q-1);
56
57         if (gcd(i, lambda_n) == 1){
58             *d = mul_inv(i, lambda_n);
59             if (*d == 0) continue;
60             else {
61                 *e = i;
62                 break;
63             }
64         }
65     }
66 }
```

mRSA_generate_key 2

p, q를 구하기 위해 첫 번째 while loop를 수행한다. 이때 랜덤한 32bit 정수 x를 arc4random_uniform을 통해 선언한다. arc4random_uniform은 upper bound 미만의 난수를 return하므로 arc4random_uniform(0x7fffffff)은 $0 \sim 2^{31} - 1$ 사이의 난수를 return한다. 이때 맨 앞 bit가 0이 아님을 보장하기 위해 0x80000000을 더해 $2^{31} \sim 2^{32} - 1$ 사이의 난수를 return한다. 이후 x를 1씩 증가시키며 miller_rabin 알고리즘을 통해 소수 여부를 확인하고 소수가 발견되면 p에 대입한 뒤 다시 x를 랜덤하게 재설정하고 동일한 loop를 반복하여 소수가 발견되면 q에 대입한다. 최종적으로 p와 q는 $2^{31} \sim 2^{32} - 1$ 사이의 소수로 설정된다.

이후 $\phi(n)$ 대신 $\lambda(n)$ 을 사용하기 위해 lambda_n를 선언한다. 이후 e, d를 구하기 위해 두 번째 while loop를 수행한다. 이때 랜덤한 64bit 정수 i를 구하기 위해 랜덤한 32bit 정수 random1, random2를 서로 곱해주어 64bit 정수를 만들고, 이때 혹시라도 값이 lambda_n보다 커지지 않게 하기 위해 gcd(p-1, q-1)을 나누어 최종적으로 i를 선언한다. 이때 i와 lambda_n이 서로소이고 그때 e의 inverse인 d가 존재한다면, 해당 i로 e, d 값을 구한다. i와 lambda_n이 서로소가 아니거나 e의 inverse인 d가 존재하지 않는다면 다시 랜덤한 64bit 정수 i를 구하여 loop를 반복한다.

③ mRSA_cipher

```
72 int mRSA_cipher(uint64_t *m, uint64_t k, uint64_t n)
73 {
74     *m = mod_pow(*m, k, n);
75
76     if (*m >= n) return 1;
77     else return 0;
78 }
```

지역 함수로 선언한 mod_pow를 이용해 $M^k \bmod n$ 을 계산한다. 이때 $m \geq n$ 이면 m의 값이 범위를 넘었으므로 오류로 처리해서 1을 return, 그 외의 경우에는 0을 return한다.

2. compile

```
xion@xion-VirtualBox:~/바탕화면/project/project4$ make
gcc -Wall -c test.c
test.c: In function 'main':
test.c:21:58: warning: format '%llx' expects argument of type 'long long int', but no argument was given [-Wformat]
    21 |         printf("Error: RSA key is not 64 bits: %llx\n", k);
        |                                     ^
test.c:21:58: note: (This warning is ignored on clang targets since they can't diagnose this)
test.c:21:58: warning: format '%llx' expects argument of type 'long long int', but no argument was given [-Wformat]
    21 |         printf("Error: RSA key is not 64 bits: %llx\n", k);
        |                                     ^
test.c:21:58: note: (This warning is ignored on clang targets since they can't diagnose this)
gcc -Wall -c mRSA.c
gcc -Wall -o test test.o mRSA.o -lbsd
```

test.c를 수정하지 않고 그대로 사용해 생기는 format warning을 제외한다면 모두 정상적으로 compile 되었다.

3. result

```
xion@xion-VirtualBox:~/바탕화면/project/project#4$ ./test
e = 11cfedc9fe5e9815
d = 39c126a76512347d
n = a5d313bc0a3ab3bf
m = 0, c = 0, v = 0
m = 1, c = 1, v = 1
m = 2, c = 9016738912167985583, v = 2
m = 3, c = 11157507707754085312, v = 3
m = 4, c = 678647153808447941, v = 4
m = 5, c = 10953787637928883474, v = 5
m = 6, c = 11076907099530740615, v = 6
m = 7, c = 54652651993780470363, v = 7
m = 8, c = 4315251267791780429, v = 8
m = 9, c = 11467969563167523336, v = 9
m = 10, c = 10049380521149048720, v = 10
m = 11, c = 8533507700776340255, v = 11
m = 12, c = 4986792351270700093, v = 12
m = 13, c = 1238773555678971227, v = 13
m = 14, c = 3488665735365000193, v = 14
m = 15, c = 2642459375204830572, v = 15
m = 16, c = 4076264733307883849, v = 16
m = 17, c = 101911319888084417, v = 17
m = 18, c = 7179005755334467379, v = 18
m = 19, c = 843686427776272727, v = 19
e = 02587406b8937ca5
d = 0a1e074c25420a89
n = cbf7af33bd0961ff
m = e604b0c7a82b3416, c = 138696474a06d92b, v = 1a0d0193eb21d217
m = 1e13eaeabac4c8e7, c = 940f07477065ebbc, v = 1e13eaeabac4c8e7
m = a0cb8b6407fb96b0, c = 18a506e9ad9665c9, v = a0cb8b6407fb96b0
m = 150e8db61ff97b90, c = a70d2e72d86e5dd0, v = 150e8db61ff97b90
m = 8613576c80740c50, c = 6749ae94a8d4b51f, v = 8613576c80740c50
m = 76697a4e59a4591b, c = 9ccd3075feb066f6, v = 76697a4e59a4591b
m = bb162b51c3d9e978, c = 224155e493883380, v = bb162b51c3d9e978
m = 3679c2080fdb3e6c, c = 904e97559198b118, v = 3679c2080fdb3e6c
m = ff980a0a6e0a40b2, c = 743b10bd07f77f54, v = 33a05ad6b100deb3
m = b9e0797c86d5328a, c = 46bb55ea01f4a82f, v = b9e0797c86d5328a
m = c31364531602843b, c = 1bc3ffab0f92665f, v = c31364531602843b
m = 3ac9e23893f87d41, c = c1d74c68d53f6b8d, v = 3ac9e23893f87d41
m = 2bca6749cb1377fb, c = 292cec6427ded801, v = 2bca6749cb1377fb
m = bdbbd17ee6cfc8e, c = 281b293916ada288, v = bdbbd17ee6cfc8e
m = 64e2987906d02e71, c = 8c4dda24669e8085, v = 64e2987906d02e71
m = e4b142e2aceaa98a, c = 8ed17b9fa47a1f43, v = 18b993aeefe1478b
m = 05bcb9ffed663fd3, c = a55735fab1b6d9ba, v = 05bcb9ffed663fd3
m = 4e57bba40f93d8f1, c = 84c88f70d03deac9, v = 4e57bba40f93d8f1
m = 0ecc8c27a5ffcf72, c = 82dd6fdbbce30c, v = 0ecc8c27a5ffcf72
m = 7e74fa3f4d8aa8ce, c = a97667d0519b3f8e, v = 7e74fa3f4d8aa8ce
Random testing.....No error found!
```

p, q, e, d이 모두 랜덤하게 설정되기 때문에 예상출력과는 값이 다르지만 최종적으로 Random testing에서 No error found!가 출력되는 것을 확인할 수 있다. ‘>’ 터미널 명령어를 통해 결과를 mRSA.txt에 저장하여 별도로 제출한다.

4. feedback

맨 처음 구현한 초기 버전의 경우 약 10번의 실행 중 1~2번 꼴로 Random testing 중 generate_key에 error가 발생했고, error가 발생하지 않더라도 running time이 평균 1분 이상으로 매우 비효율적이었다. 이후 최적화를 위해 코드를 둘러보다가 miller_rabin 알고리즘에서 for loop를 제거하고 매우 효율적인 비트연산으로 대체하는 등 miller_rabin 알고리즘을 최적화하였고, 최종적으로 약 100번의 실행 중에도 한 번의 error도 발생하지 않았으며, running time도 평균 5~6초 내외로 파격적으로 감소했다.