

ECE 471 Lab 4

RSA Public-Key Encryption and Signature Lab

Mitchell Dzurick

3/23/2020

Github with all documentation - <https://www.github.com/mitchdz/ECE471>

Contents

| | | |
|----------|---|----------|
| 1 | Overview | 2 |
| 2 | lab flow | 3 |
| 3 | Tasks | 8 |
| 3.1 | Task 1: Deriving the Private Key | 8 |
| 3.1.1 | Task 1: Solution | 8 |
| 3.2 | Task 2: Encrypting a Message | 9 |
| 3.2.1 | Task 2: Solution | 9 |
| 3.3 | Task 3: Decrypting a Message | 9 |
| 3.3.1 | Task 3: Solution | 9 |
| 3.4 | Task 4: Signing a Message | 9 |
| 3.4.1 | Task 4: Solution | 9 |
| 3.5 | Task 5: Verifying a Signature | 9 |
| 3.5.1 | Task 5: Solution | 9 |
| 3.6 | Task 6: Manually Verifying an X.509 Certificate | 9 |
| 3.6.1 | Task 6: Solution | 9 |

RSA Public-Key Encryption and Signature Lab

Copyright © 2018 Wenliang Du, Syracuse University. The development of this document was partially funded by the National Science Foundation under Award No. 1303306 and 1718086. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.

1 Overview

RSA (RivestShamirAdleman) is one of the first public-key cryptosystems and is widely used for secure communication. The RSA algorithm first generates two large random prime numbers, and then use them to generate public and private key pairs, which can be used to do encryption, decryption, digital signature generation, and digital signature verification. The RSA algorithm is built upon number theories, and it can

be quite easily implemented with the support of libraries. The learning objective of this lab is for students to gain hands-on experiences on the RSA algorithm. From lectures, students should have learned the theoretic part of the RSA algorithm, so they know mathematically how to generate public/private keys and how to perform encryption/decryption and signature generation/verification. This lab enhances student's understanding of RSA by requiring them to go through every essential step of the RSA algorithm on actual numbers, so they can apply the theories learned from the class. Essentially, students will be implementing the RSA algorithm using the C program language. The lab covers the following security-related topics:

- Public-key cryptography
- The RSA algorithm and key generation
- Big number calculation
- Encryption and Decryption using RSA
- Digital signature
- X.509 certificate

Lab Environment. This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website.

2 lab flow

This entire lab is done with a single program titled *flow.c*. The program can be found here <https://github.com/mitchdz/ECE471/tree/master/lab4> with instructions on how to run. *flow.c* also utilizes a file titled *l4_util.h* which is a file with a few helper functions designed to make the flow more simple.

The code is pasted below:

```
//flow.c
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>
#include "l4_util.h"

int main ()
{
    printf("Lab 4 RSA Public-Key Encryption and Signature Lab\n");
    // part 3.1 Deriving a private key
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();

    // Initialize p, q, e
    BN_dec2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_dec2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_dec2bn(&e, "0D88C3"); //modulus

    BIGNUM *part1PrivatKey = RSA_get_priv(p, q, e);
    printf("(3.1)\n");
    printBN("Task 1 private key ", part1PrivatKey);
    printf("\n");

    // Part 3.2
    printf("(3.2)\n");
    BIGNUM* enc = BN_new();
    BIGNUM* dec = BN_new();
    // Init private key d
    BIGNUM* privateKey3_2 = BN_new();
    BN_hex2bn(&privateKey3_2, DHEX3_2);

    BIGNUM* publicKey = BN_new();
    BN_hex2bn(&publicKey, NHEX3_2);
    printBN("Public key: ", publicKey);
    printf("\n");
```

```

BIGNUM* mod = BN_new();
BN_hex2bn(&mod, EHEX3_2);

BIGNUM* message = BN_new();
BN_hex2bn(&message, MHEX3_2);

printBN("Message Hex:", message);
enc = RSA_ENC(message, mod, publicKey);
printBN("Encrypted message:", enc);
dec = RSA_DEC(enc, privateKey3_2, publicKey);
printf("Decrypted message: ");
printHEX(BN_bn2hex(dec));
printf("\n");

/* Part 3.3 */
printf("(3.3)\n");
BIGNUM* task3_C = BN_new();
BN_hex2bn(&task3_C, CHEX3_3);

dec = RSA_DEC(task3_C, privateKey3_2, publicKey);
printf("Decrypted message: "); printHEX(BN_bn2hex(dec)); printf("\n");

/* part 3.4 Signing a Message */
printf("(3.4)\n");
BIGNUM* BN_t4 = BN_new();

// python -c 'print("I owe you $2000".encode("hex"))'
BN_hex2bn(&BN_t4, "49206f776520796f75202432303030");
enc = RSA_ENC(BN_t4, privateKey3_2, publicKey);
printBN("Signature: ", enc);
dec = RSA_DEC(enc, mod, publicKey);
printf("message: "); printHEX(BN_bn2hex(dec));

// python -c 'print("I owe you $3000".encode("hex"))'
BN_hex2bn(&BN_t4, "49206f776520796f75202433303030");
enc = RSA_ENC(BN_t4, privateKey3_2, publicKey);
printBN("Signature: ", enc);
dec = RSA_DEC(enc, mod, publicKey);
printf("message: "); printHEX(BN_bn2hex(dec)); printf("\n");

/* Part 3.5 Verifying a signature */
printf("(3.5)\n");
BIGNUM *S = BN_new();

```

```

BN_hex2bn(&publicKey, NHEX3_5);
BN_hex2bn(&S, SHEX3_5);

// correct signature
dec = RSA_DEC(S, mod, publicKey);
printf("message (regular) : "); printHEX(BN_bn2hex(dec)); printf("\n");

// corrupted signature
BN_hex2bn(&S, SHEX3_5p2);
dec = RSA_DEC(S, mod, publicKey);
printHEX(BN_bn2hex(dec)); printf("\n");

/* Part 3.6 Manually Verifying an X.509 Certificate */
printf("(3.6)\n");

return 0;
}

//4_util.h
#include <stdio.h>
#include <string.h>
#include <string.h>

#define PHEX3_1 "F7E75FDC469067FFDC4E847C51F452DF"
#define QHEX3_1 "E85CED54AF57E53E092113E62F436F4F"
#define EHEX3_1 "0D88C3"

#define DHEX3_2 "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D"
#define NHEX3_2 "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4DOCB81629242FB1A5"
#define MHEX3_2 "4120746f702073656372657421"
#define EHEX3_2 "010001"

#define CHEX3_3 "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBD7C7DCB67396567EA1E2493F"

#define SIGHEX3_4 "239a09ea0d5fdaea94ec97130b1c74c89764226065bbe614da7fb9f851be7beabd5f8"
#define MHEX3_4p1 "I owe you $2000"
#define MHEX3_4p2 "I owe you $3000"

#define MHEX3_5 "Launch a missile."
#define SHEX3_5 "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F"
#define SHEX3_5p2 "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F"
#define EHEX3_5 "010001"
#define NHEX3_5 "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115"

```

```

#define NBITS 256

BIGNUM *RSA_get_priv(BIGNUM* p, BIGNUM* q, BIGNUM* e){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM* p_minus_one = BN_new();
    BIGNUM* q_minus_one = BN_new();
    BIGNUM* one = BN_new();
    BIGNUM* ret = BN_new();

    BN_dec2bn(&one, "1");
    BN_sub(p_minus_one, p, one);
    BN_sub(q_minus_one, q, one);
    BN_mul(ret, p_minus_one, q_minus_one, ctx);

    BIGNUM* res = BN_new();
    BN_mod_inverse(res, e, ret, ctx);
    BN_CTX_free(ctx);
    return res;
}

void printBN(char *msg, BIGNUM * a)
{
    /* Use BN_bn2hex(a) for hex string
    *
    * * Use BN_bn2dec(a) for decimal string */
    char * number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

BIGNUM* RSA_ENC(BIGNUM* message, BIGNUM* mod, BIGNUM* pub_key){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM* enc = BN_new();
    BN_mod_exp(enc, message, mod, pub_key, ctx);
    BN_CTX_free(ctx);
    return enc;
}

BIGNUM* RSA_DEC(BIGNUM* enc, BIGNUM* priv_key, BIGNUM* pub_key){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM* dec = BN_new();
    BN_mod_exp(dec, enc, priv_key, pub_key, ctx);
    BN_CTX_free(ctx);
    return dec;
}

```

```

int hex2int(char c){
// return (int)strtol(c, NULL, 16);
if (c >= 97)
c = c - 32;
int first = c / 16 - 3;
int second = c % 16;
int res = first * 10 + second;
if (res > 9) res--;
return res;
}

int hex2ascii(const char c, const char d){return (hex2int(c) * 16) + hex2int(d);}

void printHEX(const char* st){
int length = strlen(st);
if (length % 2 != 0) {
printf("%s\n", "hex length needs to be even.");
return;
}
int i;
char buf = 0;
for(i = 0; i < length; i++) {
if(i % 2 != 0)
printf("%c", hex2ascii(buf, st[i]));
else
buf = st[i];
}
printf("\n");
}

```

The commands used in the makefile in the git directory to compile are as follows:

```
$ gcc src/flow.c -o bin/flow -lcrypto && ./bin/flow
```

The output of running bin/flow are below:

Lab 4 RSA Public-Key Encryption and Signature Lab

(3.1)

Task 1 private key 0x3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

(3.2)

Public key: DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5

Message Hex: 4120746F702073656372657421

Encrypted message: 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

Decrypted message: A top secret!

(3.3)

Decrypted message: Password is dees

(3.4)

Signature: 80A55421D72345AC199836F60D51DC9594E2BDB4AE20C804823FB71660DE7B82

message: I owe you \$2000

Signature: 04FC9C53ED7BBE4ED4BE2C24B0BDF7184B96290B4ED4E3959F58E94B1ECEA2EB

message: I owe you \$3000

(3.5)

message (regular) : Launch a missile.

message (corrupted) : <corrupted output>

(3.6)

Figure 1: output of running the flow program

3 Tasks

3.1 Task 1: Deriving the Private Key

3.1.1 Task 1: Solution

The derived private key can be seen in Figure 1 where the output indicates the key from this section. The key is derived through a custom function named `BIGNUM *RSA_get_priv(BIGNUM* p, BIGNUM* q, BIGNUM* e)`. The function is shown below.

```
BIGNUM *RSA_get_priv(BIGNUM* p, BIGNUM* q, BIGNUM* e){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM* p_minus_one = BN_new();
    BIGNUM* q_minus_one = BN_new();
    BIGNUM* one = BN_new();
    BIGNUM* ret = BN_new();
```



```

BN_dec2bn(&one, "1");
BN_sub(p_minus_one, p, one);
BN_sub(q_minus_one, q, one);
// (p - 1)(q - 1) also called the totient
BN_mul(ret, p_minus_one, q_minus_one, ctx);

BIGNUM* res = BN_new();
BN_mod_inverse(res, e, ret, ctx);
BN_CTX_free(ctx);
return res;
}

```

The function above calculates $\Phi(N) = (p - 1)(q - 1)$ and then uses the totient to calculate d with a reverse modulus operation.

3.2 Task 2: Encrypting a Message

3.2.1 Task 2: Solution

3.3 Task 3: Decrypting a Message

3.3.1 Task 3: Solution

3.4 Task 4: Signing a Message

3.4.1 Task 4: Solution

3.5 Task 5: Verifying a Signature

3.5.1 Task 5: Solution

3.6 Task 6: Manually Verifying an X.509 Certificate

3.6.1 Task 6: Solution