RSA Cryptosystem Lab

1 Overview

RSA (Rivest-Shamir-Adleman) 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 uses 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

2 Lab Environment

This lab has been tested on the Ubuntu 16.04 VM that you should have installed on your computer during the first lab of this course.

3 Submission

Submit a PDF document with your answers to the tasks and questions in this lab. Include the task numbers and the questions. Place your answers immediately after the task number or question. Note that some answers require you to include a screen shot. Paste any code that you write or use in the corresponding answer. Some answers ask you to attach the source code to the Blackboard assignment. Use meaningful and easy to read names for your source code files.

4 Background

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only operate on simple data types, such as 32-bit integer and 64-bit long integer types. The numbers involved in the RSA algorithms are typically more than 512 bits long. For example, to multiple two 32-bit integer numbers `a` and `b`, we just need to use `a*b` in our program. However, if they are big numbers, we cannot do that any more; instead, we need to use an algorithm (i.e., a function) to compute their products. There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this lab, we will use the Big Number library provided by `openssl`. To use this library, we will define each big number as a `BIGNUM` type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

4.1 BIGNUM APIs

All the big number APIs (application programming interfaces) can be found in `https://linux.die.net/man/3/bn`. In the following, we describe some of the APIs that are needed for this lab.

- Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a `BN_CTX` structure is created to hold BIGNUM temporary variables used by library functions. We need to create such a structure, and pass it to the functions that requires it.

  ```
  BN_CTX *ctx = BN_CTX_new()
  ```

- Initialize a BIGNUM variable

  ```
  BIGNUM *a = BN_new()
  ```

- There are a number of ways to assign a value to a BIGNUM variable

  ```
  // Assign a value from a decimal number string
  BN_dec2bn(&a, "12345678901112231223");
  // Assign a value from a hex number string
  BN_hex2bn(&a, "2A3B4C55FF77889AED3F");
  // Generate a random number of 128 bits
  BN_rand(a, 128, 0, 0);
  // Generate a random prime number of 128 bits
  BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
  ```

- Print out big number

  ```
  void printBN(char *msg, BIGNUM * a)
  {
  // Convert the BIGNUM to number string
  char * number_str = BN_bn2dec(a);
  // Print out the number string
  ```

```
printf("%s %s\n", msg, number_str);
// Free the dynamically allocated memory
OPENSSL_free(number_str);
}
```

- Computer `res=a-b` and `res=a+b`

```
BN_sub(res, a, b);
BN_add(res, a, b);
```

- Compute `res = a * b`. It should be noted that a `BN_CTX` structure is need in this API.

```
BN_mul(res, a, b, ctx)
```

- Compute `res=a*b mod n`.

```
BN_mod_mul(res, a, b, n, ctx)
```

- Compute modular inverse, i.e., given `a`, find `b`, such that `a * b mod n = 1`. The value `b` is called the inverse of `a`, with respect to modular `n`.

```
BN_mod_inverse(b, a, n, ctx);
```

4.2 A Complete Example

We show a complete example in the following. In this example, we initialize three BIGNUM variables, `a`, `b`, and `n`; we then compute `a * b` and `(a* b mod n)`.

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

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);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *a = BN_new();
BIGNUM *b = BN_new();
```

```
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();


// Initialize a, b, n
BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
BN_dec2bn(&b, "273489463796838501848592769467194369268");
BN_rand(n, NBITS, 0, 0);

// res = a*b
BN_mul(res, a, b, ctx);
printBN("a * b = ", res);

// res = a^b mod n
BN_mod_exp(res, a, b, n, ctx);
printBN("a^c mod n = ", res);
return 0;
}
```

Compilation. We can use the following command to compile `bn_sample.c` (the `-l` option tells the compiler to use the crypto library).

```
$ gcc bn_sample.c -lcrypto -o example
```

5 Lab Tasks

To avoid mistakes, please avoid manually typing the numbers used the lab tasks. Instead, copy and paste the numbers from this PDF file.

Deliverables. Write your code from labs 1-5 in a single file called `rsa-lab.c`. Use comments to clearly label the parts corresponding to each of the 5 tasks. I should be able to run your C file on my SEED VM. Note that there are additional details on the deliverables that are described in each task.

5.1 Task 1: Deriving the Private Key

Let `p, q`, and `e` be three prime numbers. Let `n = p*q`. We will use `(e, n)` as the public key. Please calculate the private key `d`. The hexadecimal values of `p, q`, and `e` are listed in the following. It should be noted that although `p` and `q` used in this task are quite large numbers, they are not large enough to be secure. We intentionally make them small for the sake of simplicity. In practice, these numbers should be at least 512 bits long (the one used here are only 128 bits).

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

Deliverable. Your code should print out "The private key is d=...", where the dots should be replaced with the actual value of `d` that you calculated.

5.2 Task 2: Encrypting a Message

Let `(e, n)` be the public key. Please encrypt the message "`A top secret!`" (the quotations are not included). We need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API BN hex2bn(). The following python command can be used to convert a plain ASCII string to a hex string.

```
$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
```

The public keys are listed in the followings (hexadecimal). We also provide the private key d to help you verify your encryption result.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30
```

Deliverable. Your code should print out "The hex plaintext is ... and the hex ciphertext is ...", where the dots should be replaced with the the actual hex values.

5.3 Task 3: Decrypting a Message

The public/private keys used in this task are the same as the ones used in Task 2. Please decrypt the following ciphertext C, and convert it back to a plain ASCII string.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

You can use the following python command to convert a hex string back to to a plain ASCII string.

```
$ python -c 'print("4120746f702073656372657421".decode("hex"))'
A top secret!
```

Deliverable. Your code should print out "The plaintext of ciphertext ... is ..." where the dots should be replaced with the actual values.

5.3.1 Task 4: Signing a Message

The public/private keys used in this task are the same as the ones used in Task 2. Please generate a signature for the following message. We mentioned in class that encrypting the plaintext with the private key was not useful to provide confidentiality since anybody with that public key can decrypt it. But it is useful to provide authenticity and accountability as only the person with the private key can generate a ciphertext that can be decrypted with the corresponding public key. We call the ciphertext created with the private key the signature. Hence, for this task , you need to encrypt the following message using the private key.

```
M = I owe you $2000.
```

Deliverable. Your code should print out "The signature for the message in hex is ...", where the dots should be replaced with the actual values.

5.4 Task 5: Verifying a Signature

Bob receives a message `M = "Launch a missile."` from Alice, with her signature `S`. We know that Alice's public key is `(e, n)`. Please verify whether the signature is indeed Alice's or not. The public key and signature (hexadecimal) are listed in the following:

```
M = Launch a missle.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

Deliverable. Your code should output the message "The provided signature is ... and the computed signature is...". Replace the dots with the actual values. If S and the computed signature are equal, output " The signature matches!", Otherwise, print "Verification failed".

Then, add another section to your code where you repeat this task but for a corrupted signature such that the last byte of the provided signature S changes from 2F to 3F, i.e, there is only one bit of change.