**CSC 302 Computer Security**

**RSA Public-Key Encryption and Signature Lab**

**Tyler Prehl**

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 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. 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. Background

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in program, because those operators can only operate on primitive 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 anymore; instead, we need to use an algorithm (i.e., a function) to computer their products.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this lab, we will use the Big Numbers 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.

2.1 BIGNUM APIs

All the big number APIs can be found from <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 several ways to assign a value to a BIGNUM variable.

//assign a value from a decimal number string

BN\_dec2bn(&a, “12345678901234567890”);

//assign a value from a hex number string

BN\_hex2bn(&a, “2A3B4C55FF77889ABE3F”);

//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 a big number
* 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)

* Computer res = a\*b mod c:

BN\_mod\_mul(res, a, b, n, ctx)

* Computer res = ac mod n:

BN\_mod\_exp(res, a, c, n, ctx)

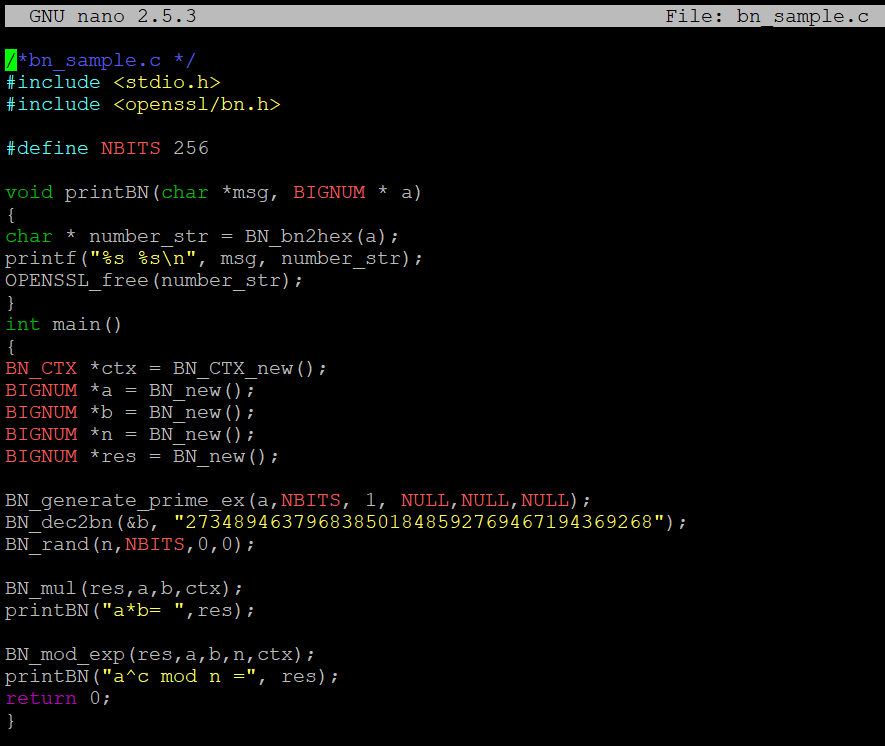
* Compute modular inverse, i.e., given a, find b, such that a\*b mod n = 1. The value b is called the inverse of a, with respect to modular n.

BN\_mod\_inverse(b, a, n, ctx)

2.2 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 (ab mod n). This is a c program and we use nano to create the file. Please the following command to create the program

nano bn\_sample.c



2.3 Compilation and run

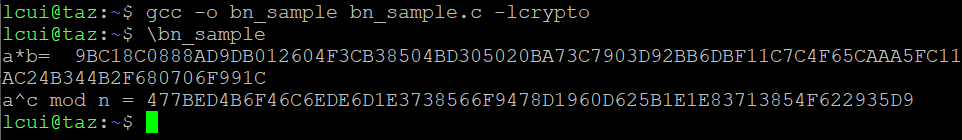
We use the following command to compile program

gcc -o bn\_sample bn\_sample.c -lcrypto

After compile the program, please use the following command to run the program

\bn\_sample

Here is the result I got:



3. Lab Tasks

3.1 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, 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

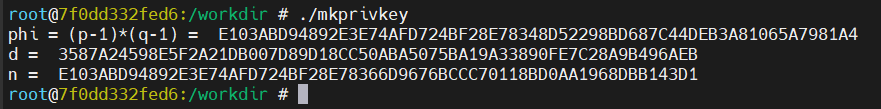
Please attach a screenshot of your private key (d, n).

Note to self – public key = (0D88C3, p\*q)

d = (1/e) mod phi

phi = (p-1)\*(q-1)

Private key = (d, n)



The private key is (d, n), where d and n are noted in the output. Unfortunately, based upon my attempt at decryption of the encrypted message in 3.3, something has been computed incorrectly. Below is a screenshot of the functions I used for computation of the various variables:

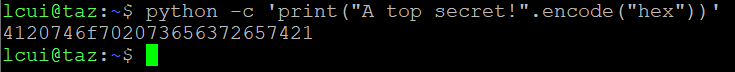
Text

Description automatically generated

However, for the sake of 3.4 and 3.5, my incorrect variables should serve okay because the encryption/decryption process works just fine using the incorrect keys I created.

3.2 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 ACII 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.



Please attach a screenshot of the cipher text for “A top secret!” that encrypted by the private key (d, n) you got in the previous task.

Text

Description automatically generated

And here is the function I used to create this cipher text:

A screenshot of a computer

Description automatically generated with medium confidence

3.3 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.



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



3.4 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 (please directly sign this message, instead of signing its has value):

M = I own you $2000.

Please make a slight change to the message M, such as changing $2000 to $3000, and sign the modified message. Attach the screenshots of both signatures, compare both signatures and describe what you observe.

Despite the fact that the hex-decimal encoded versions of the two different messages were identical from python:

Text

Description automatically generated

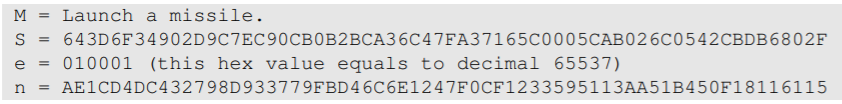
I was shocked to see how different the signatures are between the original “I own you $2000” message and “I own you $3000” message.



Side note – I think “I own you $2000” is probably supposed to be “I OWE you $2000”

3.5 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:

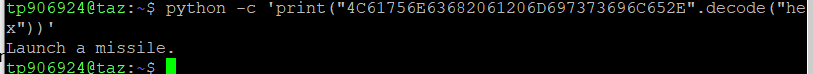


YES, this is Alice’s signature! Verified by getting the hex message, and then using python hex decoder to get the plaintext message:

A screenshot of a computer

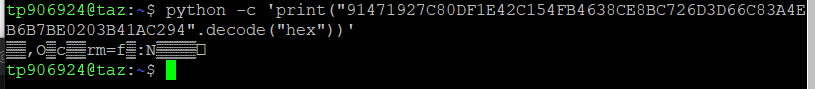
Description automatically generated with medium confidence





Suppose that the signature above is corrupted, such that the last bye of the signature changes from 2F to 3F, i.e, there is only one bit of change. Please repeat this task and describe what will happen to the verification process.

With just one bit of change, we get a completely non-sense message:



This shows the power of RSA encryption! Every piece of the encrypted message matters to a single bit!