RSA Encryption and Signature Lab

Jatin Kesnani
21K-3204 Student
FAST University

Abstract

This report explores the practical implementation of the RSA cryptosystem, focusing on key generation, encryption, decryption, and digital signatures. The lab provides hands-on experience with RSA operations, applying theoretical concepts from lectures to real-world calculations. By performing these tasks, the lab enhances understanding of how RSA encryption and digital signatures function, reinforcing key cryptographic principles essential for secure communication.

1. Overview

The RSA Encryption and Signature Lab focuses on providing hands-on experience with one of the foundational public-key cryptosystems: RSA (Rivest-Shamir-Adleman). RSA relies on generating large prime numbers to create public and private key pairs, which are then used for secure encryption, decryption, and digital signatures. The primary goal of this lab is to apply theoretical knowledge of RSA from lectures to real-world numerical calculations. By performing operations such as key generation, message encryption, decryption, signature creation, and verification, this lab helps students deepen their understanding of RSA's underlying mathematical principles and its application in securing digital communication. The lab exercises aim to strengthen practical skills and highlight key steps in RSA's cryptographic process.

2. Background

2.1 BIGNUM APIs

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

// Print out a big number.                                                  // 4 //
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);                                           //   //
}                                                                           //   //

int main (){
        // Creating BN_CTX structure to hold BIGNUM temporary variables.    // 1 //
        BN_CTX *ctx = BN_CTX_new();                                         //   //

        // Initialize a BIGNUM variable.                                    // 2 //
        BIGNUM *a = BN_new();                                               //   //

        // Assign a value from a decimal number string                      // 3 //
        BN_dec2bn(&a, "12345678901112231223");                              //   //
        printBN("Assign a value from a decimal number string :", a);        //   //
                                                                            //   //
        // Assign a value from a hex number string                          //   //
        BN_hex2bn(&a, "2A3B4C55FF77889AED3F");                              //   //
        printBN("Assign a value from a hex number string     :", a);        //   //
                                                                            //   //
        // Generate a random number of 128 bits                             //   //
        BN_rand(a, 128, 0, 0);                                              //   //
        printBN("Generate a random number of 128 bits        :", a);        //   //
                                                                            //   //
        // Generate a random prime number of 128 bits                       //   //
        BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);                  //   //
        printBN("Generate a random prime number of 128 bits  :", a);        //   //
```

```c
        // Compute res = a − b and res = a + b:                            // 5 //
        BIGNUM *b = BN_new();                                               //   //
        BIGNUM *res = BN_new();                                             //   //
        BN_dec2bn(&a, "98765432101234567890");                              //   //
        BN_dec2bn(&b, "11223344556677889900");                              //   //
                                                                            //   //
        BN_sub(res, a, b);                                                  //   //
        printBN("res = a - b =", res);                                      //   //
                                                                            //   //
        BN_add(res, a, b);                                                  //   //
        printBN("res = a + b =", res);                                      //   //

        // Compute res = a ∗ b.                                             // 6 //
        BN_mul(res, a, b, ctx);                                             //   //
        printBN("res = a * b =", res);                                      //   //

        // Compute res = a ∗ b mod n:                                       // 7 //
        BIGNUM *n = BN_new();                                               //   //
        BN_rand(n, 128, 0, 0);                                              //   //
        BN_mod_mul(res, a, b, n, ctx);                                      //   //
        printBN("res = a ∗ b mod n =", res);                                //   //

        // Compute res = a^c mod n:                                         // 8 //
        BIGNUM *c = BN_new();                                               //   //
        BN_dec2bn(&c, "12345678901234567890");                              //   //
        BN_mod_exp(res, a, c, n, ctx);                                      //   //
        printBN("res = a^c mod n =", res);                                  //   //

        // Compute modular inverse                                          // 9 //
        BN_mod_inverse(b, a, n, ctx);                                       //   //
        printBN("b = a^−1 mod n =", res);                                   //   //

        return 0;
}
```

```
[10/06/24]seed@VM:~/.../Background$ gcc BIGNUM_APIs.c -o BIGNUM_APIs -lcrypto
[10/06/24]seed@VM:~/.../Background$ ./BIGNUM_APIs
Assign a value from a decimal number string : 12345678901112231223
Assign a value from a hex number string     : 199433250764282723495231
Generate a random number of 128 bits        : 263819849830412475905633197228605396991
Generate a random prime number of 128 bits  : 335100762933392706783275811709868934467
res = a - b = 87542087544556677990
res = a + b = 109988776657912457790
res = a * b = 1108478474761330717297806597943495311000
res = a ∗ b mod n = 117784160510676538044286565913169866150
res = a^c mod n = 134376229984719574797899837389732361070
b = a^−1 mod n = 134376229984719574797899837389732361070
[10/06/24]seed@VM:~/.../Background$ 
```

This program demonstrates the use of OpenSSL's BIGNUM API for performing arithmetic and cryptographic operations on large numbers, which are essential for algorithms like RSA. The code initializes large integers using BIGNUM structures and performs operations such as addition, subtraction, and multiplication. It also computes modular arithmetic, including modular multiplication, modular exponentiation, and modular inverses. The BN_CTX context is used to manage temporary variables for efficient memory usage. The program prints the results of these operations using a custom printBN() function, which converts the BIGNUM values to readable strings. Through this exercise, fundamental operations on large integers required for cryptographic computations are explored.

2.2 A Complete Example

```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;
}
```

```
[10/06/24]seed@VM:~/.../Background$ gcc bn_sample.c -o bn_sample -lcrypto
[10/06/24]seed@VM:~/.../Background$ ./bn_sample
a * b =  B48B7DE2F90B5A88F152C0E3AAEF0495F99A3AD5FF56CA4CAEFBB728EC82A1C1F0C401E9FFEDCD34962BB817194B5A4C
a^c mod n =  7B1E2BC97ED60B893A3CC96E01136B031A43124DCEA93F272341301828B9361C
[10/06/24]seed@VM:~/.../Background$
```

This example demonstrates how to use OpenSSL's BIGNUM API to perform cryptographic operations. The program initializes three BIGNUM variables: a, b, and n. The prime number a is generated, b is assigned a large decimal value, and n is randomly initialized. The program then computes two operations: the multiplication of a and b, and modular exponentiation, which calculates a^b mod n. The results are printed in hexadecimal format using the printBN() function. This example highlights basic large-number operations in cryptography, like modular exponentiation and multiplication.

3. Lab Tasks

3.1 Task 1: Deriving the Private Key

Task1.c
~/Labsetup/Task1

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

void printBN(char *msg, BIGNUM *a) {
    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 *p = BN_new(), *q = BN_new(), *e = BN_new(), *n = BN_new(), *d = BN_new(), *one = BN_new(), *phi = BN_new(), *result1 = BN_new(), *result2 = BN_new();

    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");
    BN_hex2bn(&one, "1");

    BN_mul(n, p, q, ctx);
    printBN("n = p * q =", n);

    BN_sub(result1, p, one);
    BN_sub(result2, q, one);
    BN_mul(phi, result1, result2, ctx);
    printBN("ϕ(n) =", phi);

    BN_mod_inverse(d, e, phi, ctx);
    printBN("Private key d =", d);

    BN_free(p);
    BN_free(q);
    BN_free(n);
    BN_free(phi);
    BN_free(e);
    BN_free(d);
    BN_free(one);
    BN_CTX_free(ctx);
    return 0;
}
```

seed@VM: ~/.../Task1

```
[10/06/24]seed@VM:~/.../Task1$ gcc Task1.c -o Task1 -lcrypto
[10/06/24]seed@VM:~/.../Task1$ ./Task1
n = p * q = E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
ϕ(n) = E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
Private key d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[10/06/24]seed@VM:~/.../Task1$
```

This code calculates the private key d in the RSA algorithm using the given values of p, q, and e. First, it initializes necessary BIGNUM variables using OpenSSL. Then, the product $n = p \times q$ is computed, where n is part of the public key. The Euler's totient function $\phi(n) = (p-1) \times (q-1)$ is calculated next. Finally, the private key d is derived by finding the modular inverse of e with respect to $\phi(n)$. The result is printed in hexadecimal format. The code efficiently handles large numbers, thanks to OpenSSL's BIGNUM library.

3.2 Task 2: Encrypting a Message

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
void printBN(char *msg, BIGNUM *a) {
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
        OPENSSL_free(number_str);
}
void hex_to_ascii(char *hex_string) {
        char ascii_output[1024] = {0};
        int length = strlen(hex_string);
        for (int i = 0; i < length; i += 2) {
                char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
                int ascii_char;
                sscanf(hex_pair, "%x", &ascii_char);
                ascii_output[i / 2] = (char)ascii_char;
        }
        printf("Decrypted ASCII String: %s\n", ascii_output);
}
int main() {
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *n = BN_new(), *e = BN_new(), *M = BN_new(), *d = BN_new(), *enc_result = BN_new(), *dec_result = BN_new();
        BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&e, "010001");
        BN_hex2bn(&M, "4120746f702073656372657421");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
        BN_mod_exp(enc_result, M, e, n, ctx);
        printBN("Encrypted Message =", enc_result);

        printf("--> Verifying Encryption Result <--\n");
        BN_mod_exp(dec_result, enc_result, d, n, ctx);
        printBN("Decrypted Message =", dec_result);

        char *decrypted_hex = BN_bn2hex(dec_result);
        hex_to_ascii(decrypted_hex);
        OPENSSL_free(decrypted_hex);
        BN_free(n);
        BN_free(e);
        BN_free(M);
        BN_free(enc_result);
        BN_free(dec_result);
        BN_CTX_free(ctx);
        return 0;
}
```

```
[10/07/24]seed@VM:~/.../Task2$ python3 -c 'print("A top secret!".encode("utf-8").hex())'
4120746f702073656372657421
[10/07/24]seed@VM:~/.../Task2$ gcc Task2.c -o Task2 -lcrypto
[10/07/24]seed@VM:~/.../Task2$ ./Task2
Encrypted Message = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
--> Verifying Encryption Result <--
Decrypted Message = 4120746F702073656372657421
Decrypted ASCII String: A top secret!
[10/07/24]seed@VM:~/.../Task2$
```

The message "A top secret!" is encrypted using RSA. The message is first converted to its hexadecimal form (M) and then encrypted with the public key (e, n) using the BN_mod_exp function. The encrypted result is printed, followed by decryption using the private key d to recover the original message. The decrypted hexadecimal value is then converted back to ASCII to verify the encryption and decryption process.

3.3 Task 3: Decrypting a Message

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

void printBN(char *msg, BIGNUM *a) {
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
        OPENSSL_free(number_str);
}

void hex_to_ascii(char *hex_string) {
        char ascii_output[1024] = {0};
        int length = strlen(hex_string);
        for (int i = 0; i < length; i += 2) {
                char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
                int ascii_char;
                sscanf(hex_pair, "%x", &ascii_char);
                ascii_output[i / 2] = (char)ascii_char;
        }
        printf("Decrypted ASCII String: %s\n", ascii_output);
}
int main() {
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *n = BN_new(), *d = BN_new(), *enc_result = BN_new(), *dec_result = BN_new();

        BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
        BN_hex2bn(&enc_result, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

        BN_mod_exp(dec_result, enc_result, d, n, ctx);
        printBN("Decrypted Message =", dec_result);

        char *decrypted_hex = BN_bn2hex(dec_result);
        hex_to_ascii(decrypted_hex);

        OPENSSL_free(decrypted_hex);
        BN_free(n);
        BN_free(d);
        BN_free(enc_result);
        BN_free(dec_result);
        BN_CTX_free(ctx);
        return 0;
}
```

```
[10/07/24]seed@VM:~/.../Task3$ gcc Task3.c -o Task3 -lcrypto
[10/07/24]seed@VM:~/.../Task3$ ./Task3
Decrypted Message = 50617373776F72642069732064656573
Decrypted ASCII String: Password is dees
[10/07/24]seed@VM:~/.../Task3$ python3 -c 'print(bytes.fromhex("50617373776F72642069732064656573").decode("utf-8"))'
Password is dees
[10/07/24]seed@VM:~/.../Task3$
```

The ciphertext was decrypted using RSA with the provided private key d and modulus n. After the decryption process using OpenSSL's BIGNUM library, the result was a hexadecimal string **50617373776F72642069732064656573**. This hex string was then converted back to its ASCII representation, revealing the original message: **"Password is dees"**. The implementation verifies the decryption process by applying the private key and converting the hex result back to plain text.

3.4 Task 4: Signing a Message

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/rsa.h>
#include <string.h>

void printBN(char *msg, BIGNUM *a) {
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
        OPENSSL_free(number_str);
}

void signMessage(BIGNUM *d, BIGNUM *n, char *message, BIGNUM *signature, BN_CTX *ctx) {
        BIGNUM *m = BN_new();
        BN_bin2bn((unsigned char*)message, strlen(message), m);
        BN_mod_exp(signature, m, d, n, ctx);
        BN_free(m);
}

int main() {
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *n = BN_new(), *d = BN_new(), *signature = BN_new();
        BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

        printf("Original Message: I owe you $2000.\n");
        signMessage(d, n, "I owe you $2000.", signature, ctx);
        printBN("Signature for Original Message:", signature);

        printf("Modified Message: I owe you $3000.\n");
        signMessage(d, n, "I owe you $3000.", signature, ctx);
        printBN("Signature for Modified Message:", signature);

        BN_free(n);
        BN_free(d);
        BN_free(signature);
        BN_CTX_free(ctx);
        return 0;
}
```

```
[10/07/24]seed@VM:~/.../Task4$ gcc Task4.c -o Task4 -lcrypto
[10/07/24]seed@VM:~/.../Task4$ ./Task4
Original Message: I owe you $2000.
Signature for Original Message: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Modified Message: I owe you $3000.
Signature for Modified Message: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
[10/07/24]seed@VM:~/.../Task4$
```

The results show that the signatures for the two messages—"I owe you $2000." and the modified message "I owe you $3000."—are completely different. The signature for the original message is 55A4E7F1... while the signature for the modified message is BCC20FB7.... This illustrates the sensitivity of RSA signatures to any changes in the message. Even a small modification (changing "$2000" to "$3000") results in a drastically different signature, ensuring that any alteration in the message can be easily detected. This property provides message integrity and non-repudiation in digital signatures.

3.5 Task 5: Verifying a Signature

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

char *hex_to_ascii(char *hex_string) {
    static char ascii_output[1024] = {0};
    int length = strlen(hex_string);
    for (int i = 0; i < length; i += 2) {
        char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
        int ascii_char;
        sscanf(hex_pair, "%x", &ascii_char);
        ascii_output[i / 2] = (char)ascii_char;
    }
    return ascii_output;
}

int main(){
        BN_CTX *ctx = BN_CTX_new();
        BIGNUM *M = BN_new(), *S1 = BN_new(), *S2 = BN_new(), *e = BN_new(), *n = BN_new();

        BN_hex2bn(&S1,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
        BN_hex2bn(&S2,"643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");
        BN_hex2bn(&e,"010001");
        BN_hex2bn(&n,"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

        BN_mod_exp(M, S1, e, n, ctx);
        printf("Message from Original Signature: %s", BN_bn2hex(M));
        printf("\nASCII Message: %s\n", hex_to_ascii(BN_bn2hex(M)));

        BN_mod_exp(M, S2, e, n, ctx);
        printf("\nMessage from Corrupted Signature: %s\n", BN_bn2hex(M));
        printf("ASCII Message:\n");
        printf("%s\n", hex_to_ascii(BN_bn2hex(M)));

        BN_free(M);
        BN_free(S1);
        BN_free(S2);
        BN_free(e);
        BN_free(n);
        BN_CTX_free(ctx);
        return 0;
}
```

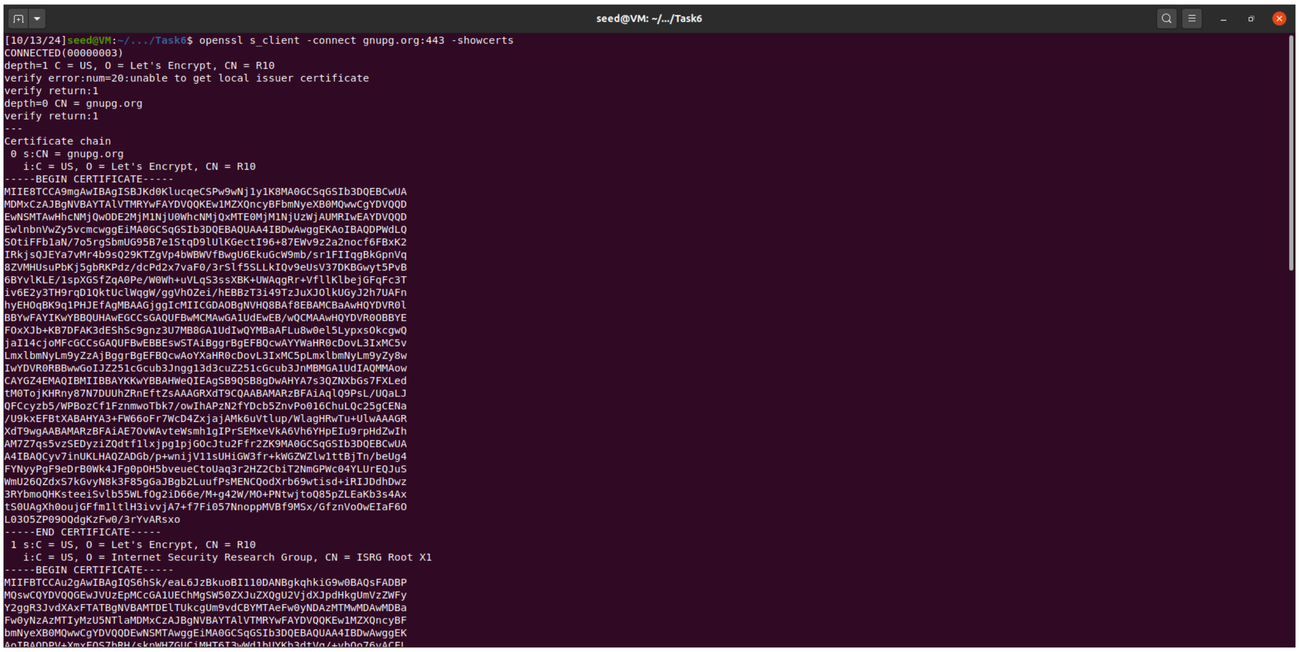
```
[10/07/24]seed@VM:~/.../Task5$ python3 -c 'print("Launch a missile.".encode("utf-8").hex())'
4c61756e63682061206d697373696c652e
[10/07/24]seed@VM:~/.../Task5$ gcc Task5.c -o Task5 -lcrypto
[10/07/24]seed@VM:~/.../Task5$ ./Task5
Message from Original Signature: 4C61756E63682061206D697373696C652E
ASCII Message: Launch a missile.

Message from Corrupted Signature: 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
ASCII Message:
��,O�c��rm=f�:N�����
[10/07/24]seed@VM:~/.../Task5$
```

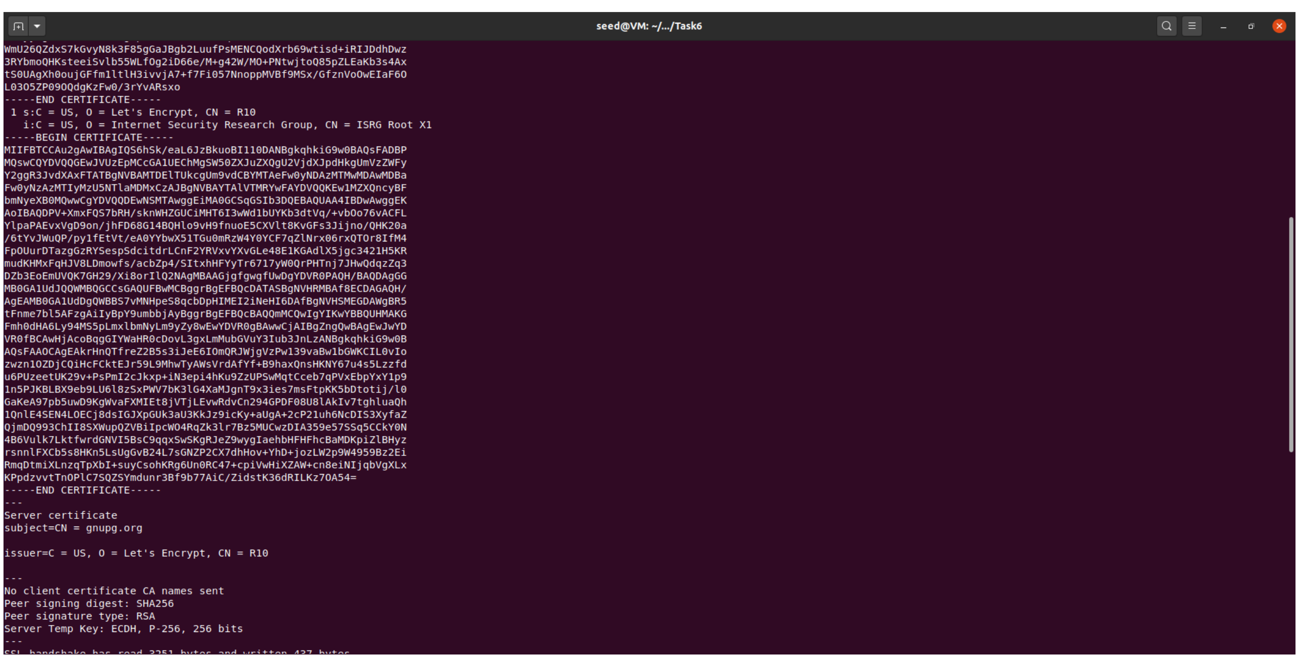
The program successfully verified Alice's digital signature for the message "Launch a missile," producing the expected ASCII output. The original signature generated the hexadecimal value 4C61756E63682061206D697373696C652E, which decoded to the correct message.

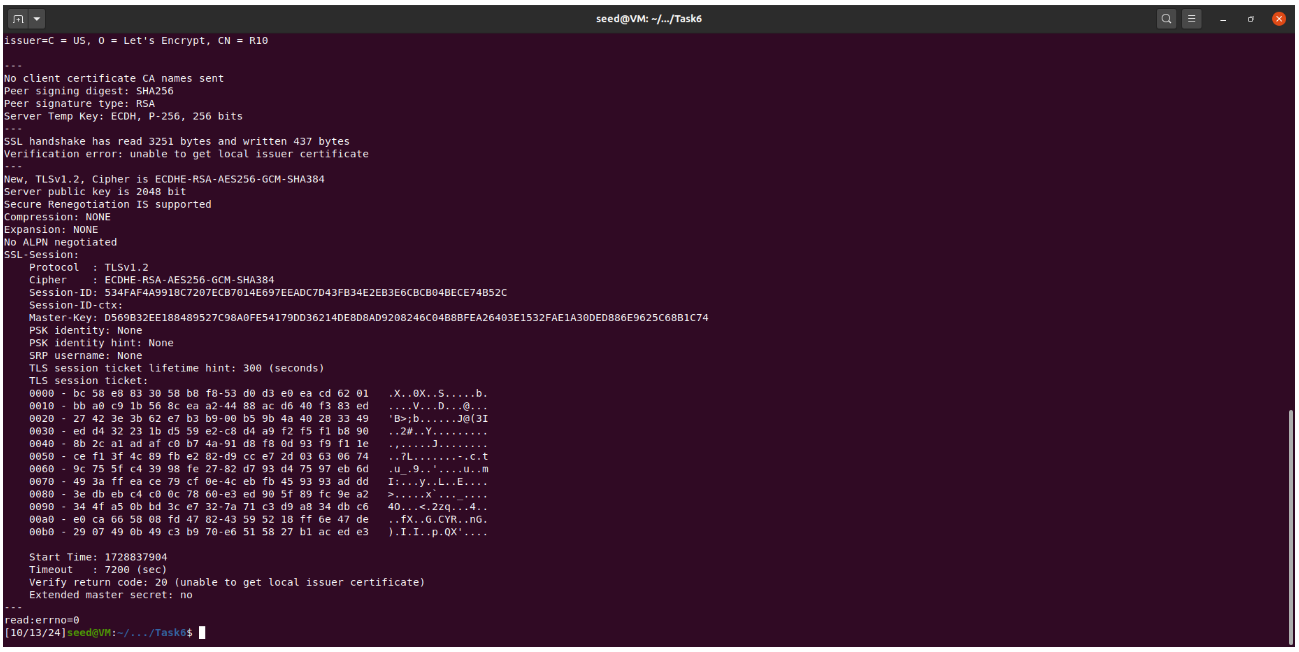
In contrast, the corrupted signature, altered by changing the last byte from 2F to 3F, resulted in an invalid output. The hexadecimal value derived from the corrupted signature did not correspond to a meaningful ASCII message. This highlights the sensitivity of RSA signatures to any changes; even minor alterations lead to failure in producing the expected original message, confirming the integrity and authenticity of the signature.

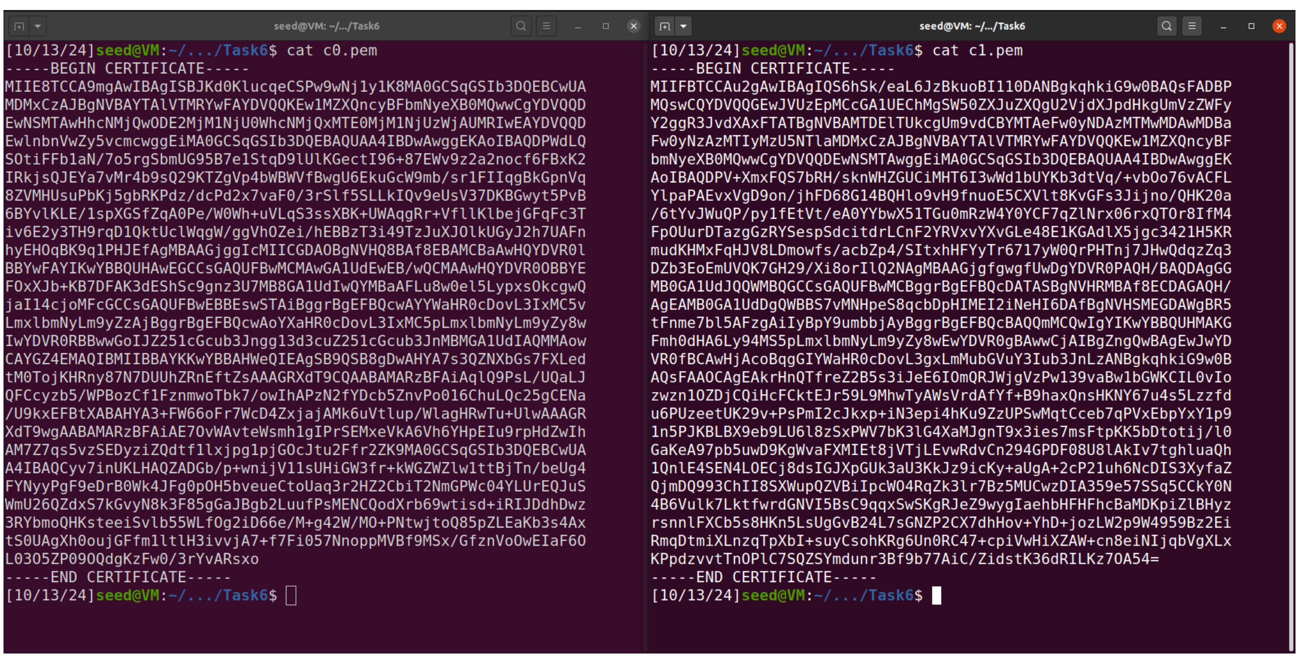
3.6 Task 6: Manually Verifying an X.509 Certificate

3.6.1 Download a certificate from a real web server.



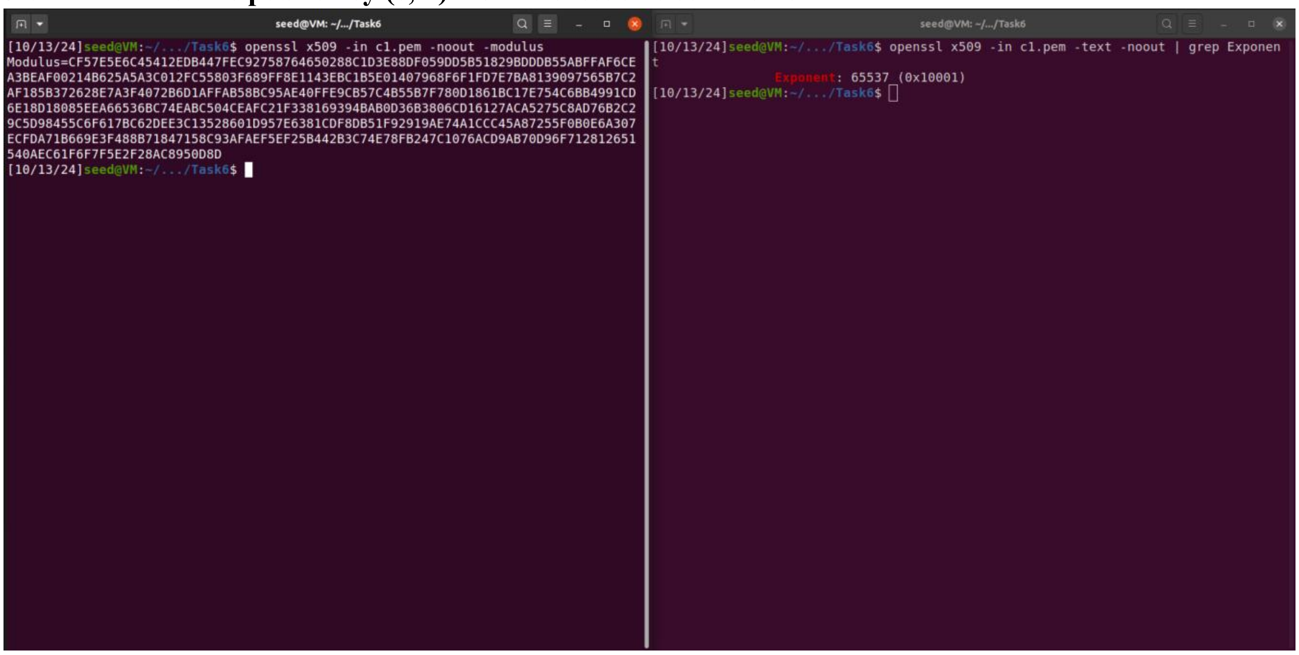






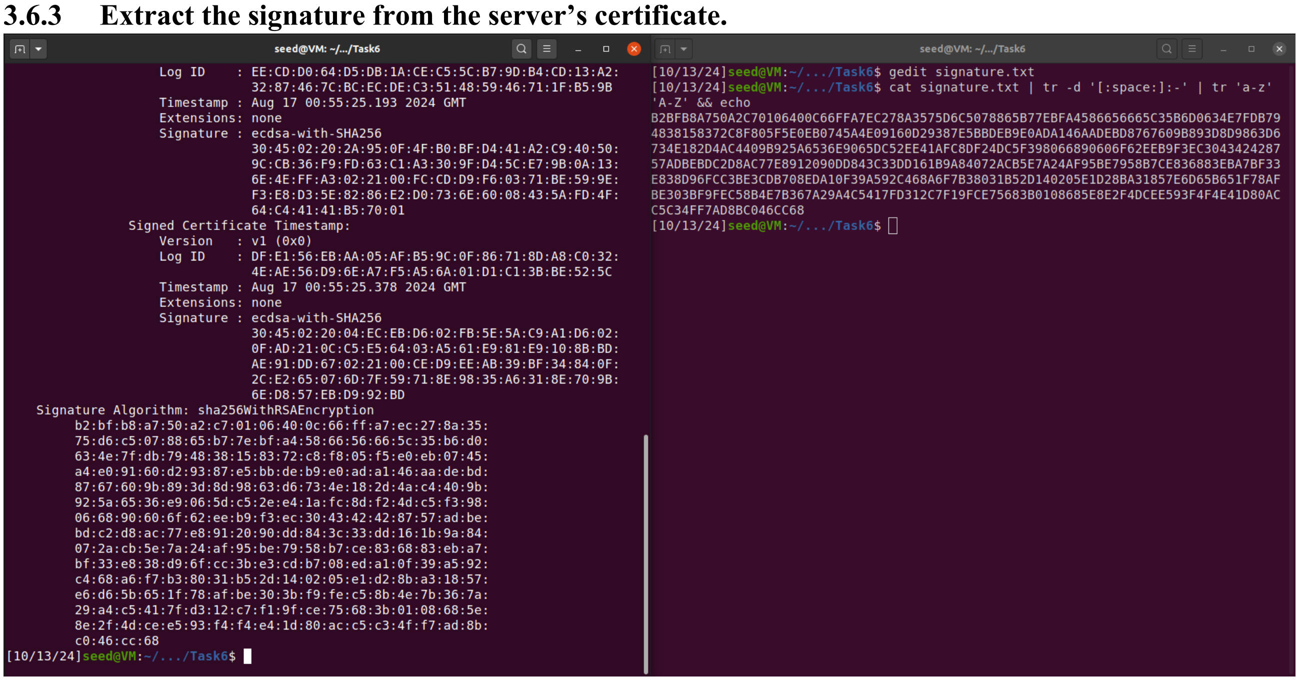
The X.509 certificate for **gnupg.org** was retrieved using the openssl s_client command. The server certificate was saved as c0.pem, and the intermediate CA's certificate was saved as c1.pem. These certificates will be used to manually verify the server's certificate by checking its issuer's signature.

3.6.2 Extract the public key (e, n) from the issuer's certificate.



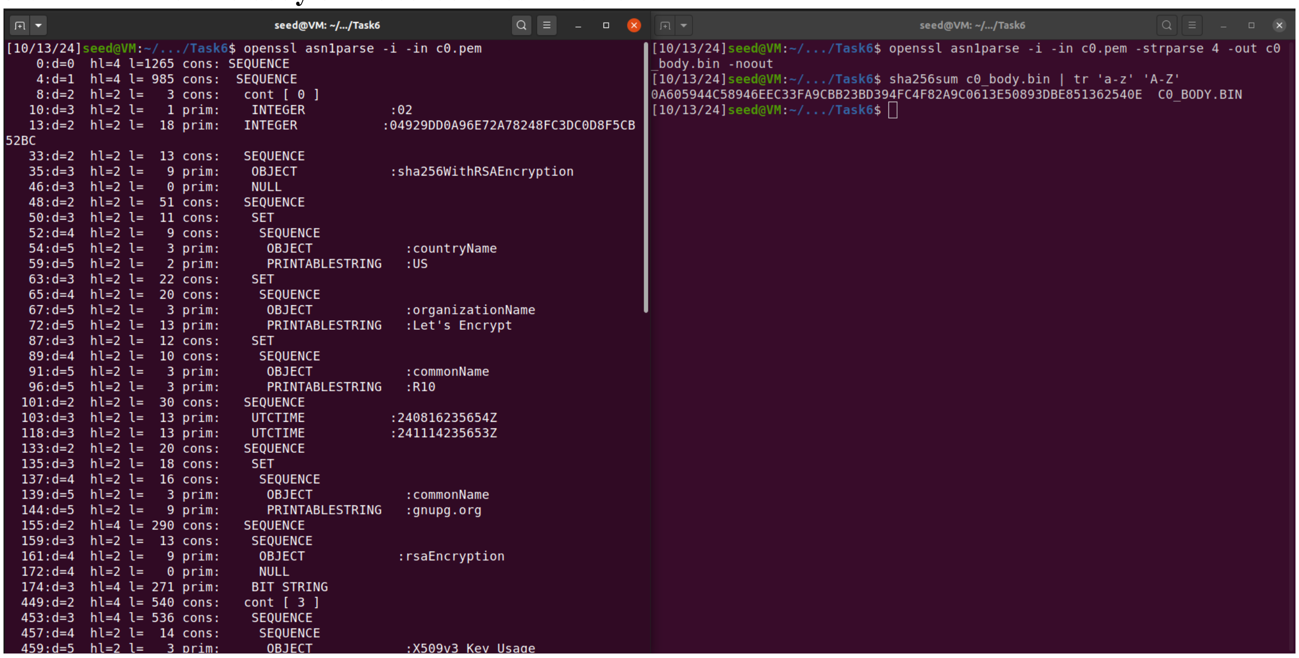
To verify the server certificate, the public key (e, n) from the issuer's certificate (c1.pem) was extracted. The modulus (n) was retrieved using the openssl x509 -modulus command, while the exponent (e) was identified by printing all fields with openssl x509 -text. These key values will be used to verify the issuer's signature on the server certificate.

3.6.3 Extract the signature from the server's certificate.



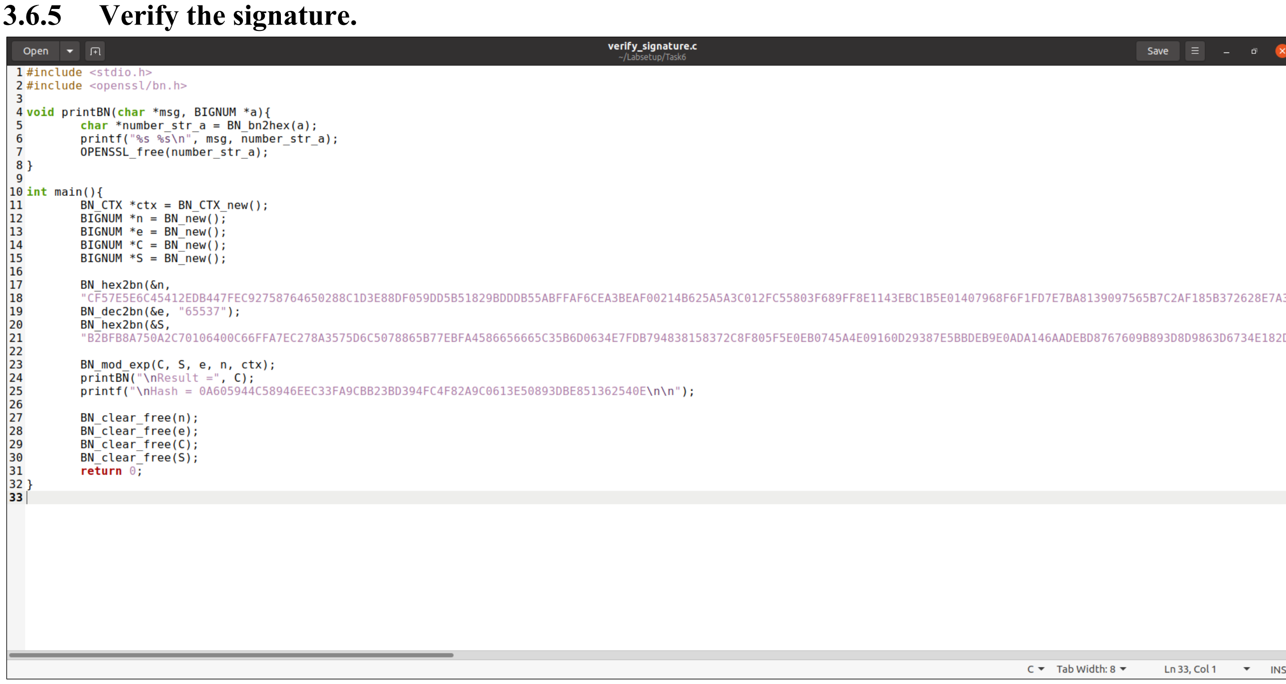
To extract the signature from the server's certificate (c0.pem), I used openssl x509 -text to print all fields and copied the signature block into a file named signature. I then used the tr command to remove spaces and colons, converting the signature into a clean hex string. The final command also ensured the hex string was in uppercase format for further processing.

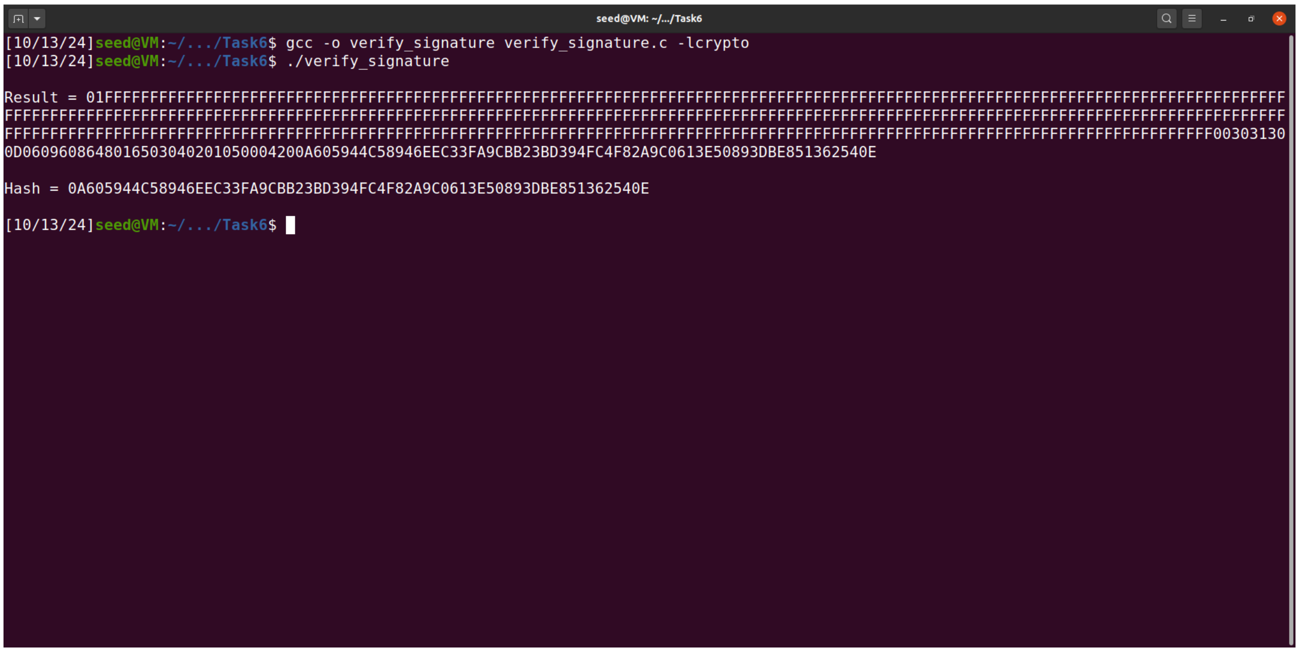
3.6.4 Extract the body of the server's certificate.



To extract the body of the server's certificate (c0.pem), I used openssl asn1parse to identify the offsets for the body and signature. The command openssl asn1parse -strparse 4 extracted the certificate body into c0_body.bin. Finally, I computed the SHA-256 hash of the certificate body using sha256sum and converted the output to uppercase.

3.6.5 Verify the signature.





The signature verification was successfully completed using a custom C program with OpenSSL. By extracting the public key, modulus, and exponent from the certificate, the signature was decrypted, and the calculated hash was compared with the expected hash. The result matched, confirming the integrity of the certificate's signature. This process was critical for understanding how certificate verification ensures trust in SSL/TLS connections without relying on built-in OpenSSL commands.