

RSA Encryption and Signature Lab

Jatin Kesnani
21K-3204 Student
FAST University

Abstract

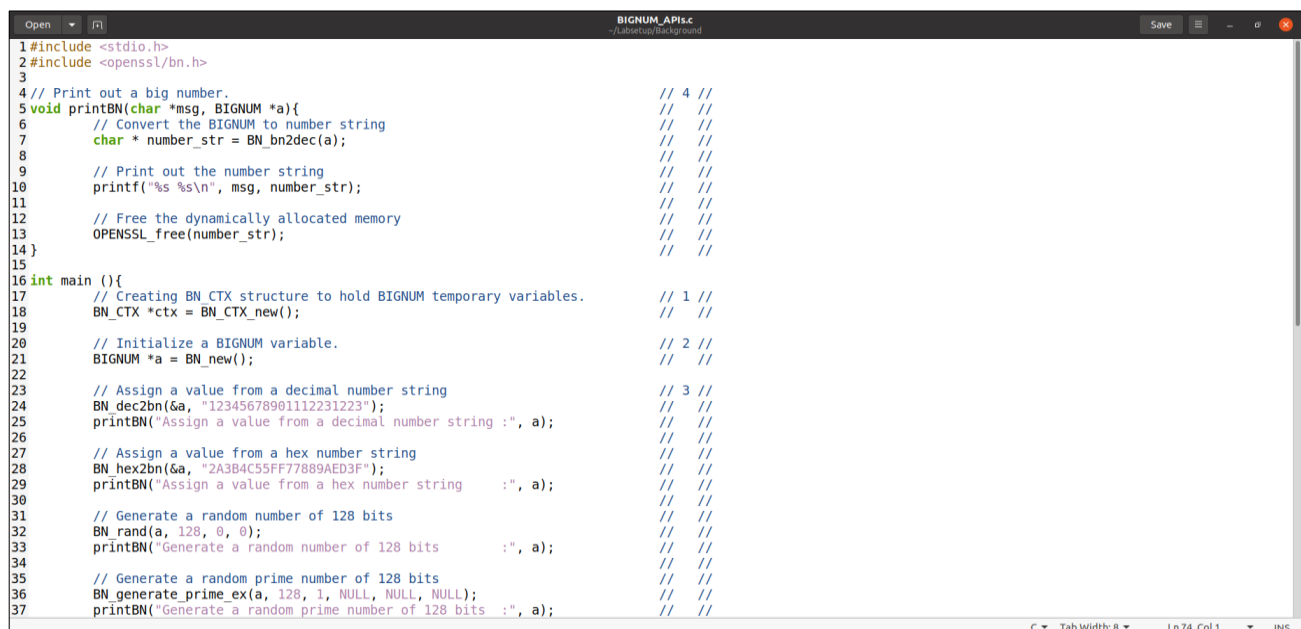
This report explores the practical implementation of the RSA cryptosystem, focusing on key generation, encryption, decryption, and digital signatures. The lab provides hands-on experience with RSA operations, applying theoretical concepts from lectures to real-world calculations. By performing these 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



```

1 #include <stdio.h>
2 #include <openssl/bn.h>
3
4 // Print out a big number. // 4 //
5 void printBN(char *msg, BIGNUM *a){ // //
6     // Convert the BIGNUM to number string // //
7     char * number_str = BN_bn2dec(a); // //
8     // Print out the number string // //
9     printf("%s %s\n", msg, number_str); // //
10    // Free the dynamically allocated memory // //
11    OPENSSL_free(number_str); // //
12}
13
14
15
16 int main (){
17     // Creating BN_CTX structure to hold BIGNUM temporary variables. // 1 //
18     BN_CTX *ctx = BN_CTX_new(); // //
19
20     // Initialize a BIGNUM variable. // 2 //
21     BIGNUM *a = BN_new(); // //
22
23     // Assign a value from a decimal number string // 3 //
24     BN_dec2bn(&a, "12345678901112231223"); // //
25     printBN("Assign a value from a decimal number string :", a); // //
26
27     // Assign a value from a hex number string // //
28     BN_hex2bn(&a, "2A3B4C5FF77889AED3F"); // //
29     printBN("Assign a value from a hex number string :", a); // //
30
31     // Generate a random number of 128 bits // //
32     BN_rand(a, 128, 0, 0); // //
33     printBN("Generate a random number of 128 bits :", a); // //
34
35     // Generate a random prime number of 128 bits // //
36     BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL); // //
37     printBN("Generate a random prime number of 128 bits :", a); // //

```

```
Open  BIGNUM_APIs.c  Save  -  x
38
39 // Compute res = a - b and res = a + b: // 5 //
40 BIGNUM *b = BN_new(); // //
41 BIGNUM *res = BN_new(); // //
42 BN_dec2bn(&a, "98765432101234567890"); // //
43 BN_dec2bn(&b, "11223344556677889900"); // //
44 // //
45 BN_sub(res, a, b); // //
46 printBN("res = a - b =", res); // //
47 // //
48 BN_add(res, a, b); // //
49 printBN("res = a + b =", res); // //
50 // //
51 // Compute res = a * b. // 6 //
52 BN_mul(res, a, b, ctx); // //
53 printBN("res = a * b =", res); // //
54 // //
55 // Compute res = a + b mod n: // 7 //
56 BIGNUM *n = BN_new(); // //
57 BN_rand(n, 128, 0, 0); // //
58 BN_mod_mul(res, a, b, n, ctx); // //
59 printBN("res = a * b mod n =", res); // //
60 // //
61 // Compute res = a^c mod n: // 8 //
62 BIGNUM *c = BN_new(); // //
63 BN_dec2bn(&c, "12345678901234567890"); // //
64 BN_mod_exp(res, a, c, n, ctx); // //
65 printBN("res = a^c mod n =", res); // //
66 // //
67 // Compute modular inverse // 9 //
68 BN_mod_inverse(b, a, n, ctx); // //
69 printBN("b = a^-1 mod n =", res); // //
70 // //
71 return 0;
72 }
73
74
```

```
seed@VM: ~/Background
[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

```
bn_sample.c
~/labsetup/Background

1 #include <stdio.h>
2 #include <openssl/bn.h>
3
4 #define NBITS 256
5
6 void printBN(char *msg, BIGNUM *a){
7     // Use BN_bn2hex(a) for hex string
8     // Use BN_bn2dec(a) for decimal string
9     char *number_str = BN_bn2hex(a);
10    printf("%s %s\n", msg, number_str);
11    OPENSSL_free(number_str);
12 }
13
14 int main(){
15     BN_CTX *ctx = BN_CTX_new();
16
17     BIGNUM *a = BN_new();
18     BIGNUM *b = BN_new();
19     BIGNUM *n = BN_new();
20     BIGNUM *res = BN_new();
21
22     // Initialize a, b, n
23     BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
24     BN_dec2bn(&b, "273489463796838501848592769467194369268");
25     BN_rand(n, NBITS, 0, 0);
26
27     // res = a*b
28     BN_mul(res, a, b, ctx);
29     printBN("a * b = ", res);
30
31     // res = a^b mod n
32     BN_mod_exp(res, a, b, n, ctx);
33     printBN("a^c mod n = ", res);
34
35     return 0;
36 }
37
```

```
seed@VM: ~/Background
[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 $a^b \bmod n$. 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

```
Open Task1.c - /Lab04/Task1 Save
1 #include <stdio.h>
2 #include <openssl/bn.h>
3
4 void printBN(char *msg, BIGNUM *a) {
5     char *number_str = BN_bn2hex(a);
6     printf("%s %s\n", msg, number_str);
7     OPENSSL_free(number_str);
8 }
9
10 int main() {
11     BN_CTX *ctx = BN_CTX_new();
12     BIGNUM *p = BN_new(), *q = BN_new(), *e = BN_new(), *n = BN_new(), *one = BN_new(), *phi = BN_new(), *result1 = BN_new(), *result2 = BN_new();
13     BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
14     BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
15     BN_hex2bn(&e, "0088C3");
16     BN_hex2bn(&one, "1");
17
18     BN_mul(n, p, q, ctx);
19     printBN("n = p * q =", n);
20
21     BN_sub(result1, p, one);
22     BN_sub(result2, q, one);
23     BN_mul(phi, result1, result2, ctx);
24     printBN("phi(n) =", phi);
25
26     BN_mod_inverse(d, e, phi, ctx);
27     printBN("Private key d =", d);
28
29     BN_free(p);
30     BN_free(q);
31     BN_free(n);
32     BN_free(phi);
33     BN_free(e);
34     BN_free(d);
35     BN_free(one);
36     BN_CTX_free(ctx);
37     return 0;
38 }
39
```

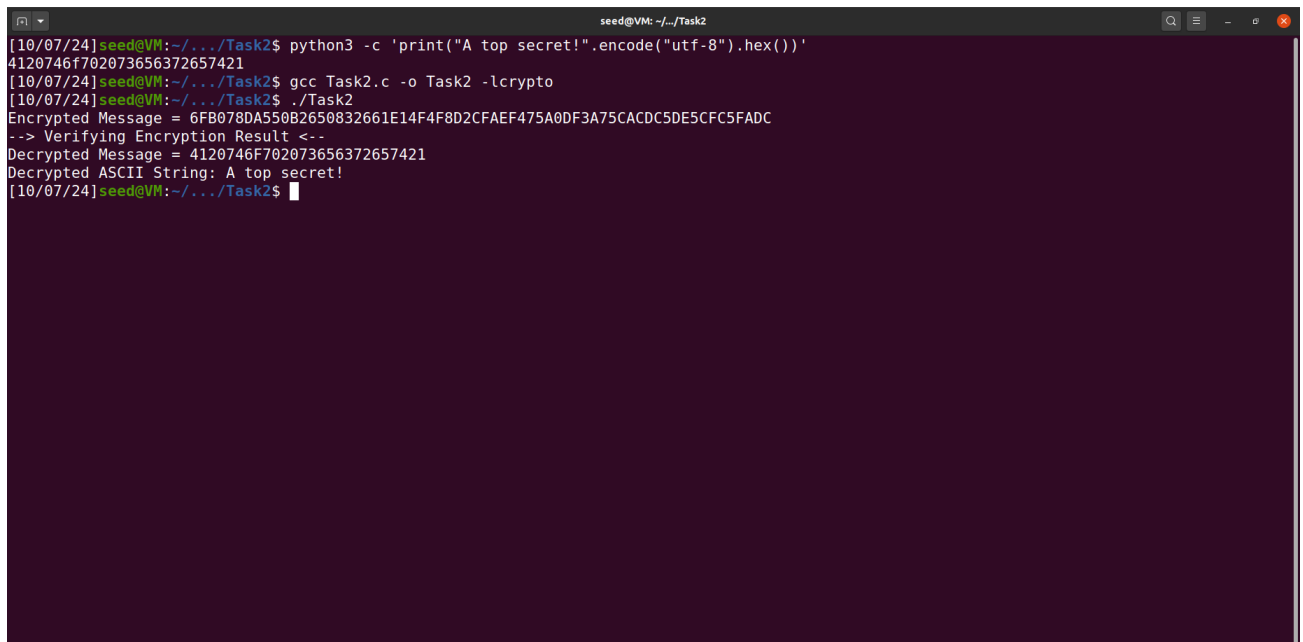
```
seed@VM: ~/Task1
[10/06/24]seed@VM:~/Task1$ gcc Task1.c -o Task1 -lcrypto
[10/06/24]seed@VM:~/Task1$ ./Task1
n = p * q = E103ABD094892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
phi(n) = E103ABD094892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
Private key d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[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



```
1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <string.h>
4 void printBN(char *msg, BIGNUM *a) {
5     char *number_str = BN_bn2hex(a);
6     printf("%s %s\n", msg, number_str);
7     OPENSSL_free(number_str);
8 }
9 void hex_to_ascii(char *hex_string) {
10    char ascii_output[1024] = {0};
11    int length = strlen(hex_string);
12    for (int i = 0; i < length; i += 2) {
13        char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
14        int ascii_char;
15        sscanf(hex_pair, "%x", &ascii_char);
16        ascii_output[i / 2] = (char)ascii_char;
17    }
18    printf("Decrypted ASCII String: %s\n", ascii_output);
19 }
20 int main() {
21     BN_CTX *ctx = BN_CTX_new();
22     BIGNUM *n = BN_new(), *e = BN_new(), *M = BN_new(), *d = BN_new(), *enc_result = BN_new(), *dec_result = BN_new();
23     BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDD3A4D0CB81629242FB1A5");
24     BN_hex2bn(&e, "010001");
25     BN_hex2bn(&M, "4120746f702073656372657421");
26     BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
27     BN_mod_exp(enc_result, M, e, n, ctx);
28     printBN("Encrypted Message =", enc_result);
29
30     printf("-> Verifying Encryption Result <--\n");
31     BN_mod_exp(dec_result, enc_result, d, n, ctx);
32     printBN("Decrypted Message =", dec_result);
33
34     char *decrypted_hex = BN_bn2hex(dec_result);
35     hex_to_ascii(decrypted_hex);
36     OPENSSL_free(decrypted_hex);
37     BN_free(n);
38     BN_free(e);
39     BN_free(M);
40     BN_free(enc_result);
41     BN_free(dec_result);
42     BN_CTX_free(ctx);
43     return 0;
44 }
45 }
```



```
[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 = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FAD5C
--> 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

```
Open Task3.c Save
1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <string.h>
4
5 void printBN(char *msg, BIGNUM *a) {
6     char *number_str = BN_bn2hex(a);
7     printf("%s %s\n", msg, number_str);
8     OPENSSL_free(number_str);
9 }
10
11 void hex_to_ascii(char *hex_string) {
12     char ascii_output[1024] = {0};
13     int length = strlen(hex_string);
14     for (int i = 0; i < length; i += 2) {
15         char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
16         int ascii_char;
17         sscanf(hex_pair, "%x", &ascii_char);
18         ascii_output[i / 2] = (char)ascii_char;
19     }
20     printf("Decrypted ASCII String: %s\n", ascii_output);
21 }
22
23 int main() {
24     BN_CTX *ctx = BN_CTX_new();
25     BIGNUM *n = BN_new(), *d = BN_new(), *enc_result = BN_new(), *dec_result = BN_new();
26     BN_hex2bn(&n, "DCBFEE3E51F62E09CE7032E2677A78946A849DC4CDE3A4D0CB81629242FB1A5");
27     BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
28     BN_hex2bn(&enc_result, "8C0F9710F2F3672B28811407E2DABBE1DA0FEBB8DFC7DCB67396567EA1E2493F");
29
30     BN_mod_exp(dec_result, enc_result, d, n, ctx);
31     printBN("Decrypted Message = ", dec_result);
32
33     char *decrypted_hex = BN_bn2hex(dec_result);
34     hex_to_ascii(decrypted_hex);
35
36     OPENSSL_free(decrypted_hex);
37     BN_free(n);
38     BN_free(d);
39     BN_free(enc_result);
40     BN_free(dec_result);
41     BN_CTX_free(ctx);
42     return 0;
43 }
44
```

```
seed@VM: ~/Task3
[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

```
Open Task4.c Save
1 #include <stdio.h>
2 #include <openssl/bn.h>
3 #include <openssl/rsa.h>
4 #include <string.h>
5
6 void printBN(char *msg, BIGNUM *a) {
7     char *number_str = BN_bn2hex(a);
8     printf("%s %s\n", msg, number_str);
9     OPENSSL_free(number_str);
10 }
11
12 void signMessage(BIGNUM *d, BIGNUM *n, char *message, BIGNUM *signature, BN_CTX *ctx) {
13     BIGNUM *m = BN_new();
14     BN_bin2bn((unsigned char *)message, strlen(message), m);
15     BN_mod_exp(signature, m, d, n, ctx);
16     BN_free(m);
17 }
18
19 int main() {
20     BN_CTX *ctx = BN_CTX_new();
21     BIGNUM *n = BN_new(), *d = BN_new(), *signature = BN_new();
22     BN_hex2bn(&n, "DCBFEE3E51F62E09CE7032E2677A78946A849DC4CDE3A4D0CB81629242FB1A5");
23     BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
24
25     printf("Original Message: I owe you $2000.\n");
26     signMessage(d, n, "I owe you $2000.", signature, ctx);
27     printBN("Signature for Original Message:", signature);
28
29     printf("Modified Message: I owe you $3000.\n");
30     signMessage(d, n, "I owe you $3000.", signature, ctx);
31     printBN("Signature for Modified Message:", signature);
32
33     BN_free(n);
34     BN_free(d);
35     BN_free(signature);
36     BN_CTX_free(ctx);
37     return 0;
38 }
39
```

```
seed@VM: ~/Task4
[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

```
Open Task5.c Save
1 #include<stdio.h>
2 #include<openssl/bn.h>
3 #include <string.h>
4
5 char *hex_to_ascii(char *hex_string) {
6     static char ascii_output[1024] = {0};
7     int length = strlen(hex_string);
8     for (int i = 0; i < length; i += 2) {
9         char hex_pair[3] = {hex_string[i], hex_string[i + 1], '\0'};
10        int ascii_char;
11        sscanf(hex_pair, "%x", &ascii_char);
12        ascii_output[i / 2] = (char)ascii_char;
13    }
14    return ascii_output;
15 }
16
17 int main(){
18     BN_CTX *ctx = BN_CTX_new();
19     BIGNUM *M = BN_new(), *S1 = BN_new(), *S2 = BN_new(), *e = BN_new(), *n = BN_new();
20
21     BN_hex2bn(&S1, "643D6F34902D9C7EC90CB082BCA36C47FA37165C0005CAB026C0542CBDB6802F");
22     BN_hex2bn(&S2, "643D6F34902D9C7EC90CB082BCA36C47FA37165C0005CAB026C0542CBDB6803F");
23     BN_hex2bn(&e, "010001");
24     BN_hex2bn(&n, "AE1CD4DC4327980933779FB046C6E1247F0CF1233595113AA51B450F18116115");
25
26     BN_mod_exp(M, S1, e, n, ctx);
27     printf("Message from Original Signature: %s", BN_bn2hex(M));
28     printf("\nASCII Message: %s\n", hex_to_ascii(BN_bn2hex(M)));
29
30     BN_mod_exp(M, S2, e, n, ctx);
31     printf("\nMessage from Corrupted Signature: %s\n", BN_bn2hex(M));
32     printf("ASCII Message:\n");
33     printf("%s\n", hex_to_ascii(BN_bn2hex(M)));
34
35     BN_free(M);
36     BN_free(S1);
37     BN_free(S2);
38     BN_free(e);
39     BN_free(n);
40     BN_CTX_free(ctx);
41     return 0;
42 }
43
```

```
seed@VM: ~/Task5
[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:
00,00c00rm=f0:N00000
[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.

3.6.1 Download a certificate from a real web server.

[illegible]

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

```
[10/13/24]seed@VM: ~/Task6$ openssl x509 -in c1.pem -noout -modulus
Modulus=CF57E5E6C45412EDB447FEC92758764650288C1D3E88DF059DD58518298000B55ABFFAF6CE
A3BEAF002148625A5A3C012FC55803F689FF8E1143EBC185E01407968F6F1FD7E78A8139097565B7C2
AF185B372628E7A3F4072B6D1AFFA858BC95AE40FFE9CB57C4B58B7F780D18618C17E754C68B4991C0
6E18D18085EEA665368C74EABC50ACEAF2F1F381693948AB0036B3806CD16127ACAS275C8AD76B2C2
9C5D98455C6F6178C62DEE3C135286010957E6381CDF8D851F92919AE74A1CCC45A87255F0B0E6A307
ECFDA718669E3F488B71847158C93AFAEF5EF25B44283C74E78FB247C1076ACD9A870D96F712812651
540AEC61F6F7F5E2F28AC89508D0
[10/13/24]seed@VM: ~/Task6$

[10/13/24]seed@VM: ~/Task6$ openssl x509 -in c1.pem -text -noout | grep Exponent
Exponent: 65537 (0x10001)
[10/13/24]seed@VM: ~/Task6$
```

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.

```
Log ID : EE:CD:D0:64:D5:D8:1A:CE:C5:5C:B7:9D:B4:CD:13:A2:
32:87:46:7C:BC:EC:DE:C3:51:48:59:46:71:1F:B5:98
Timestamp : Aug 17 00:55:25.193 2024 GMT
Extensions: none
Signature : ecdsa-with-SHA256
30:45:02:20:2A:95:0F:4F:B0:BF:D4:41:A2:C9:40:50:
9C:CB:36:F9:FD:63:C1:A3:30:9F:D4:5C:E7:9B:0A:13:
6E:4E:FF:A3:02:21:00:FC:CD:D9:F6:03:71:BE:59:9E:
F3:E8:D3:5E:82:86:E2:D0:73:6E:60:08:43:5A:FD:4F:
64:C4:41:41:B5:70:01

Signed Certificate Timestamp:
Version : v1 (0x0)
Log ID : DF:E1:56:EB:AA:05:AF:B5:9C:0F:86:71:8D:A8:C0:32:
4E:AE:56:D9:6E:A7:F5:A5:6A:01:D1:C1:3B:BE:52:5C
Timestamp : Aug 17 00:55:25.378 2024 GMT
Extensions: none
Signature : ecdsa-with-SHA256
30:45:02:20:04:EC:EB:D6:02:FB:5E:5A:C9:A1:D6:02:
0F:AD:21:0C:C5:E5:64:03:A5:61:E9:81:E9:10:8B:BD:
AE:91:DD:67:02:21:00:CE:D9:EE:AB:39:BF:34:84:0F:
2C:E2:65:07:6D:7F:59:71:8E:98:35:A6:31:8E:70:9B:
6E:D8:57:EB:D9:92:BD

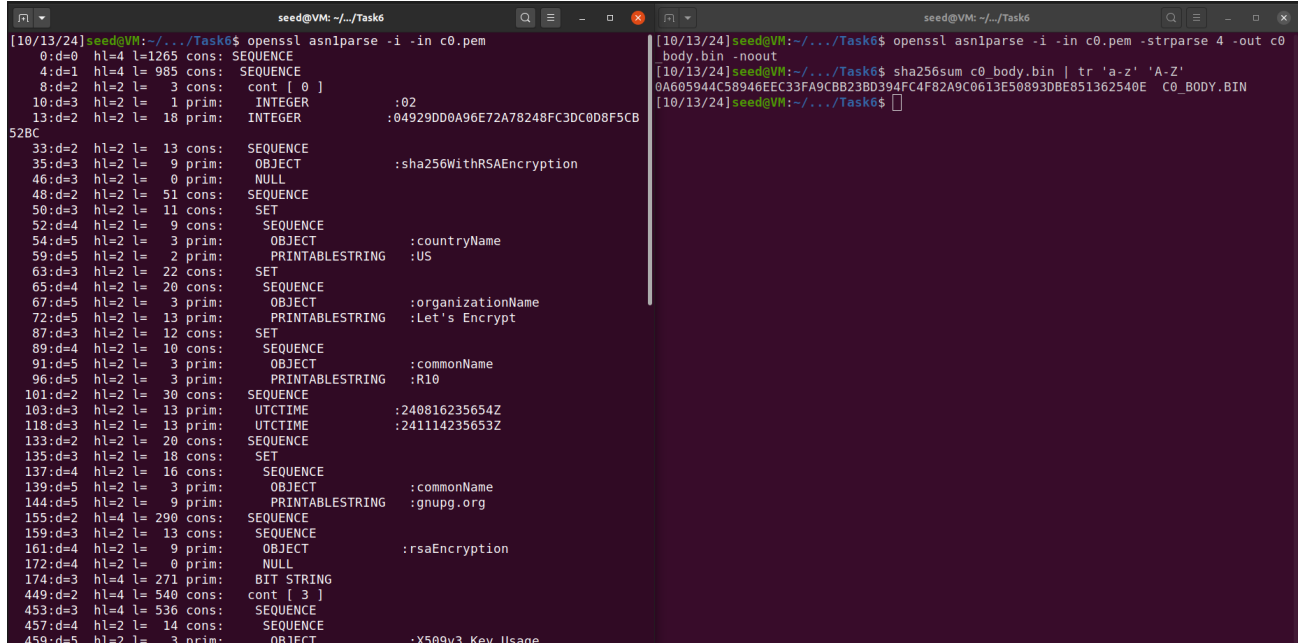
Signature Algorithm: sha256WithRSAEncryption
b2:bf:b8:a7:50:a2:c7:01:00:40:0c:66:ff:a7:ec:27:8a:35:
75:d6:c5:07:88:65:b7:7e:bf:a4:58:66:56:66:5c:35:b6:d0:
63:4e:7f:db:79:49:38:15:83:72:c8:f8:05:f5:e0:eb:07:45:
a4:e0:91:60:d2:03:07:e5:bb:de:b9:e0:ad:a1:46:aa:de:bd:
87:67:60:9b:80:3d:8d:98:63:d6:73:4e:18:2d:4a:cd:40:9b:
92:5a:65:36:e9:06:5d:c5:2e:e4:1a:fc:8d:f2:4d:c5:f3:98:
06:68:90:60:6f:62:ee:b9:f3:ac:30:43:42:42:87:57:ad:be:
bd:c2:d8:ac:77:e8:91:20:90:dd:84:3c:33:dd:16:1b:9a:84:
07:2a:cb:5e:7a:24:af:95:be:79:58:b7:ce:83:68:83:eb:a7:
bf:33:e8:38:d9:6f:cc:3b:e3:cd:b7:08:ed:a1:0f:39:a5:92:
c4:68:a6:f7:b3:80:31:b5:2d:14:02:05:e1:d2:8b:a3:18:57:
e6:d6:5b:65:1f:78:af:be:30:3b:f9:fe:c5:8b:4e:7b:36:7a:
29:a4:c5:41:7f:d3:12:c7:f1:9f:ce:75:68:3b:01:08:68:5e:
8e:2f:4d:ce:e5:93:f4:f4:e4:1d:80:ac:c5:c3:4f:f7:ad:8b:
c0:46:cc:68

[10/13/24]seed@VM: ~/Task6$

[10/13/24]seed@VM: ~/Task6$ gedit signature.txt
[10/13/24]seed@VM: ~/Task6$ cat signature.txt | tr -d '[:space:]' | tr 'a-z'
'A-Z' && echo
B2BFBA750A2C70106400C66FFA7EC278A3575D6C5078865B77EBFA4586656665C35B6D0634E7FD879
4838158372C8F805F5E0E80745A4E09160D29387E5B8DEB9E0ADA146A8DEBD8767609B8930809863D6
734E182D4AC4409B925A6536E90650C52EE41AFC8D72ADC5F398066890606F62EEB9F3EC3043424287
57ADBE8DC2D8AC77E8912090D0843C3DD161B9A84072ACB5E7A24AF95BE7958B7CE836883EBA7BF33
F83D096FC3BE3CDB78EDA10F39A592C468A6F7B38031B52D140205E1D28BA31857E6D658651F78AF
BE303BF9FEC58B4E7B367A29A4C5417FD312C7F19FCE75683B0108685E8E2F4DCEE593F4F4E41D88AC
C5C34FF7AD8BC046CC68
[10/13/24]seed@VM: ~/Task6$
```

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.

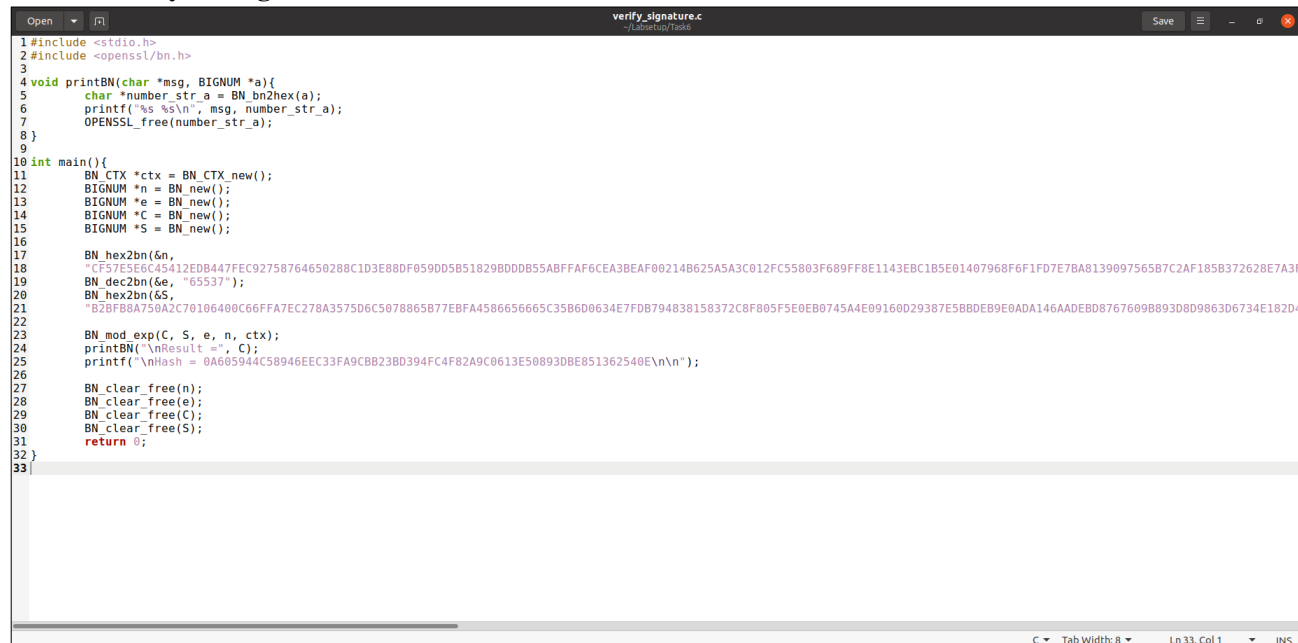


```
[10/13/24] seed@VM: ~/Task6$ openssl asn1parse -i -in c0.pem
0:d=0 hl=4 l=1265 cons: SEQUENCE
4:d=1 hl=4 l= 985 cons: SEQUENCE
8:d=2 hl=2 l= 3 cons: cont [ 0 ]
10:d=3 hl=2 l= 1 prim: INTEGER           :02
13:d=2 hl=2 l= 18 prim: INTEGER           :04929DD0A96E72A78248FC3DC08F5CB
52B:
33:d=2 hl=2 l= 13 cons: SEQUENCE
35:d=3 hl=2 l= 9 prim: OBJECT           :sha256WithRSAEncryption
46:d=3 hl=2 l= 0 prim: NULL
48:d=2 hl=2 l= 51 cons: SEQUENCE
50:d=3 hl=2 l= 11 cons: SET
52:d=4 hl=2 l= 9 cons: SEQUENCE
54:d=5 hl=2 l= 3 prim: OBJECT           :countryName
59:d=5 hl=2 l= 2 prim: PRINTABLESTRING :US
63:d=3 hl=2 l= 22 cons: SET
65:d=4 hl=2 l= 20 cons: SEQUENCE
67:d=5 hl=2 l= 3 prim: OBJECT           :organizationName
72:d=5 hl=2 l= 13 prim: PRINTABLESTRING :Let's Encrypt
87:d=3 hl=2 l= 12 cons: SET
89:d=4 hl=2 l= 10 cons: SEQUENCE
91:d=5 hl=2 l= 3 prim: OBJECT           :commonName
96:d=5 hl=2 l= 3 prim: PRINTABLESTRING :R10
101:d=2 hl=2 l= 30 cons: SEQUENCE
103:d=3 hl=2 l= 13 prim: UTCTIME          :240816235654Z
118:d=3 hl=2 l= 13 prim: UTCTIME          :241114235653Z
133:d=2 hl=2 l= 20 cons: SEQUENCE
135:d=3 hl=2 l= 18 cons: SET
137:d=4 hl=2 l= 16 cons: SEQUENCE
139:d=5 hl=2 l= 3 prim: OBJECT           :commonName
144:d=5 hl=2 l= 9 prim: PRINTABLESTRING :gnupg.org
155:d=2 hl=4 l= 290 cons: SEQUENCE
159:d=3 hl=2 l= 13 cons: SEQUENCE
161:d=4 hl=2 l= 9 prim: OBJECT           :rsaEncryption
172:d=4 hl=2 l= 0 prim: NULL
174:d=3 hl=4 l= 271 prim: BIT STRING
449:d=2 hl=4 l= 540 cons: cont [ 3 ]
453:d=3 hl=4 l= 536 cons: SEQUENCE
457:d=4 hl=2 l= 14 cons: SEQUENCE
459:d=5 hl=2 l= 3 prim: OBJECT           :X509v3_Ext_Usage

[10/13/24] seed@VM: ~/Task6$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
[10/13/24] seed@VM: ~/Task6$ sha256sum c0_body.bin | tr 'a-z' 'A-Z'
0A605944C58946EEC33FA9CBB23BD394FC4F82A9C0613E50893DBE851362540E  c0_BODY.BIN
[10/13/24] seed@VM: ~/Task6$
```

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.



```
1 #include <stdio.h>
2 #include <openssl/bn.h>
3
4 void printBN(char *msg, BIGNUM *a){
5     char *number_str_a = BN_bn2hex(a);
6     printf("%s %s\n", msg, number_str_a);
7     OPENSSL_free(number_str_a);
8 }
9
10 int main(){
11     BN_CTX *ctx = BN_CTX_new();
12     BIGNUM *n = BN_new();
13     BIGNUM *e = BN_new();
14     BIGNUM *C = BN_new();
15     BIGNUM *S = BN_new();
16
17     BN_hex2bn(&n,
18         "CF57E5E6C45412EDB447FEC92758764650288C1D3E88DF059DD5B518298DD0B55ABFFAF6CEA3BEAF00214B625A5A3C012FC55803F689FF8E1143EB01B5E01407968F6F1FD7E7BA8139097565B7C2AF185B372628E7A3F");
19     BN_dec2bn(&e, "65537");
20     BN_hex2bn(&S,
21         "62BF68A750A2C70106400C66FFA7EC278A3575D6C5078865B77EBFA4586656665C35B6D06347FDB79483B158372C8F805F5E0EB0745A4E09160D29387E58BDEB9E0ADA146AADEBD8767609B893D8D9863D6734E182D4");
22
23     BN_mod_exp(C, S, e, n, ctx);
24     printBN("\nResult =", C);
25     printf("\nHash = 0A605944C58946EEC33FA9CBB23BD394FC4F82A9C0613E50893DBE851362540E\n\n");
26
27     BN_clear_free(n);
28     BN_clear_free(e);
29     BN_clear_free(C);
30     BN_clear_free(S);
31     return 0;
32 }
33
```

