RSA Public-Key Encryption and Signature Lab
ISA (Information Security Assurance) 562
Project Report

Team Members:

Mihul Singh (G01373835) – Initial Setup, Task-4, and Task-5
Ramaswamy Iyappan (G01348097) – Task-1, Task-2, and Task-3

Project Overview:

This project focuses on implementing the RSA algorithm to generate private/public keys and perform encryption/decryption and signature generation/verification using the C programming language. Concepts covered in this project include Public-key cryptography, RSA algorithm and key generation, Big number calculation, Encryption and Decryption using RSA, and Digital Signature.

Task-1: Deriving the Private Key

Initially we choose some p, q and e as three big prime numbers and let $\boldsymbol{n = p * q}$. We will be using $(\boldsymbol{e, n})$ as the public key. For the purpose of this project, numbers used here are 128 bits whereas 512 bit numbers are used in general. We use the following hexadecimal values for p, q, and e.

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

Now, we need to calculate the **private key d**.

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

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

// p-1, q-1
BN_sub(p_1,p,one);
BN_sub(q_1,q,one);

// phi(n) = (p-1)(q-1)
BN_mul(phi_n, p_1, q_1, ctx);
printBN("phi(n) = (p-1)(q-1) = ", phi_n);

// d = e^-1 mod phi_n
BN_mod_inverse(d, e, phi_n, ctx);
printBN("d = e^-1 mod phi(n) = ", d);
```

Figure: task1.c program file

Using the base code provided in the lab manual *Crypto_RSA*, we initialize the required variables for BIGNUM and the prime numbers. We know that, $d = e^{-1} mod\ \varphi(n)$ and $\varphi(n) = (p-1)(q-1)$. So we compute the values for these using BIGNUM's predefined API functions like BN_mul() and BN_mod_inverse().

Command to run this file:

```
~ gcc -o task1.out task1.c -lcrypto
~ ./task1.out
```

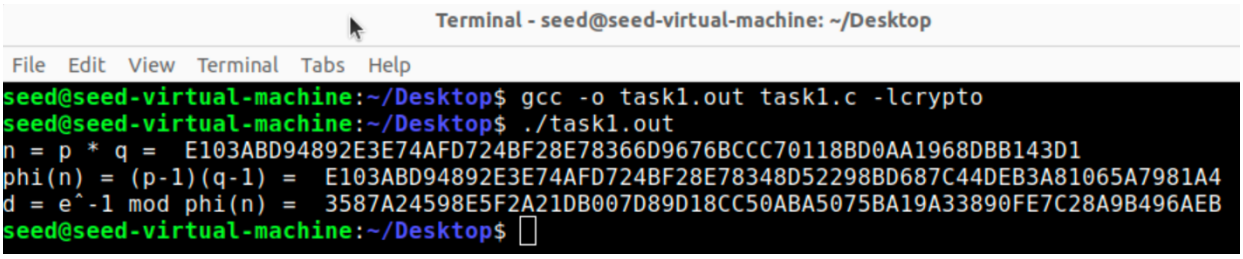


Figure: Running task1.c to compute d

We can clearly see the computed values by running the file `task1.c` and get the value of **d** as **`3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB`**

Therefore, the **private key** (d, n) will be
(`3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB`,
`E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1`)

Task-2: Encrypting a Message

Here, we use the public-key (e, n) to encrypt the message "`A top secret!`". Since a string can be represented using ASCII characters in a computer program, we can convert these characters into Hexadecimal and then again convert the hex string to a BIGNUM using the $BN\_hex2bn()$ API function. We use the following python command to convert a given ASCII string to Hexadecimal.

```
$ python3 -c 'print("A top secret!".encode("utf-8").hex())'
```

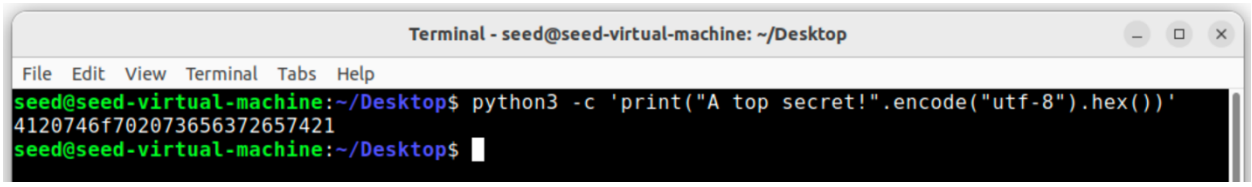


Figure: Convert plaintext to hex string using python

Hence, the hexadecimal equivalent of "`A top secret!`" is **`4120746f702073656372657421`**

We use the following Hexadecimal values as the Public and Private key for this task.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

M = 4120746f702073656372657421

Where, $n = p * q$, $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$, d is the private-key and M is the message to be encrypted, all represented in Hexadecimal notation.

Now, we need to compute the cipher text C which is the encryption of the plain text M. The cipher text C can be computed using $C = M^e mod\ n$.

```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *M = BN_new();
BIGNUM *decrypted_msg = BN_new();
BIGNUM *d = BN_new();
BIGNUM *C = BN_new();

// Initialize n, M, e and d
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&M, "4120746f702073656372657421");
BN_hex2bn(&e, "010001");
BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

printBN("Initial message   : ",M);

// Encryption: C = M^e mod n
BN_mod_exp(C, M, e, n, ctx);
printBN("C  :  M^e mod n   = ", C);

// Decryption: M = C^d mod n
BN_mod_exp(decrypted_msg, C, d, n, ctx);
printBN("Decrypted message : ", decrypted_msg);
```

Figure: task2.c program file

We initialize the required BIGNUM and Hexadecimal values the same way as before. Then compute C using $BN\_mod\_exp()$.

Command to run this file:

```
~ gcc -o task2.out task2.c -lcrypto
~ ./task2.out
```

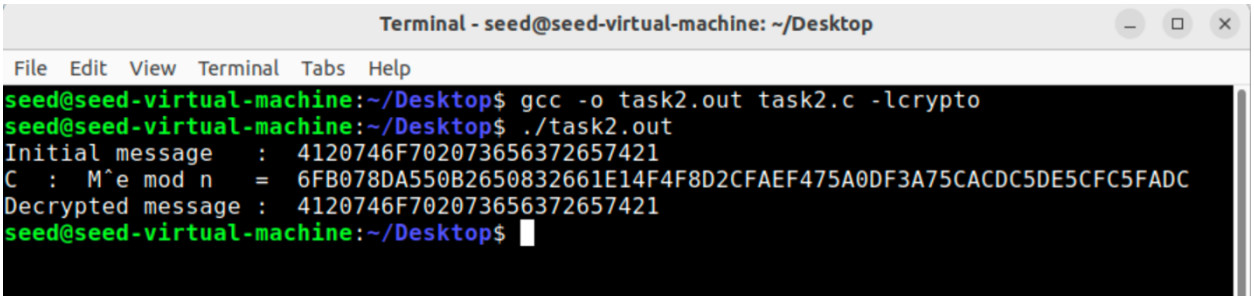


Figure: Running task2.c to encrypt message M

By running the file task2.c, we get the required cipher text C. After this, we have again decrypted it to get the plain text back so that we can verify whether our calculations were correct. Here in the above figure, we can clearly see that the decrypted message *(Hexadecimal)* obtained using the cipher text C, clearly matches with the initial message *(Hexadecimal)*. Hence, our computations are correct and message M is encrypted successfully.

Task-3: Decrypting a Message

Here, we use the same public/private keys from **task-2** to decrypt a different message using a given cipher text `C`. This is how we normally communicate over the network using RSA encryption where we automatically generate very big prime numbers p, and q to initially establish a connection, and then perform encryption/decryption over different messages using the same set of keys. The given cipher text `C` is as follows.

`C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F`

We can decrypt a cipher text `C` using the private key (d, n) as $M = C^d mod\ n$, where `M` is the decrypted plaintext.

```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *M = BN_new();
BIGNUM *d = BN_new();
BIGNUM *C = BN_new();

// Initialize n, e, d and C
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&e, "010001");
BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

// Decryption: M = C^d mod n
BN_mod_exp(M, C, d, n, ctx);
printBN("Decrypted message : ", M);
```

Figure: task3.c program file

In the file `task3.c`, we initialize required variables and values for n, e, d and C. Compute plaintext `M` using $BN\_mod\_exp()$.

Command to run this file:

```
~ gcc -o task3.out task3.c -lcrypto
~ ./task3.out
```

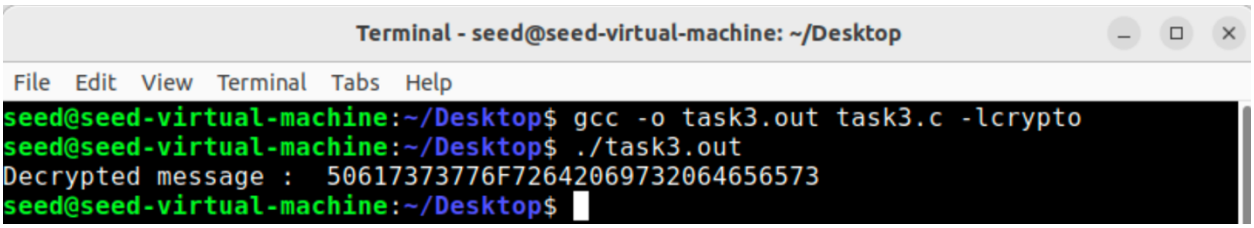


Figure: Running task3.c to decrypt cipher text C

We get a hex string as the output for the decryption of the cipher text. We can then convert this hex string to ASCII characters using the following Python command to get back the encrypted message.

```
$ python3 -c 'print(bytes.fromhex("50617373776F726420
69732064656573").decode("utf-8"))'
```

Figure: Convert hex-string to plaintext using python

Hence, the decryption of the cipher text,
`C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F`
gives us **"Password is dees".**

Task-4: Signing a Message

We need to sign a given message "`I owe you $2000.`". Generally in RSA encryption, we sign a message with the private key $(\boldsymbol{d}, \boldsymbol{n})$ using the RSA function. That is, generate a Signature `S` by encrypting a message `M` with the private key $(\boldsymbol{d}, \boldsymbol{n})$, such that,

$$S = E(PR_a, H(M))$$

$$\Rightarrow S = M^d mod\ n$$

Where $H(M)$ is the hash of message `M`. For the purpose of this task, we will sign directly on the Message `M` instead of $H(M)$ and use the same public and private keys from **task 2**. First we convert the given string to hexadecimal format, using the same python command as before.

```
$ python3 -c 'print("I owe you $2000.".encode("utf-8").hex())'
```

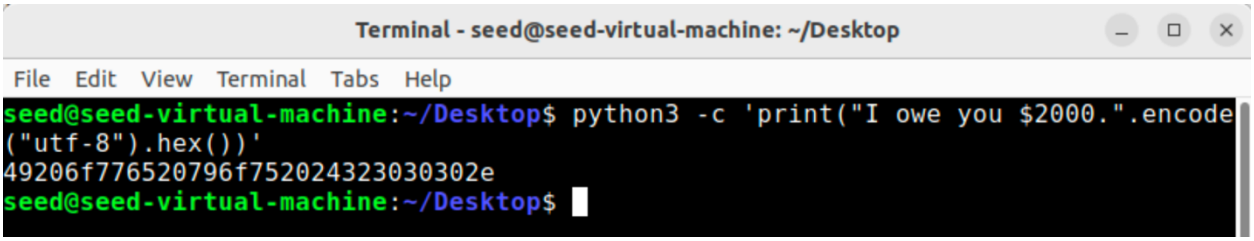


Figure: Convert plaintext to hex string using python

So, now that we have the hex **`49206f776520796f752024323030302e`** of "`I owe you $2000.`", we can use it in our program and generate the signature `S`.

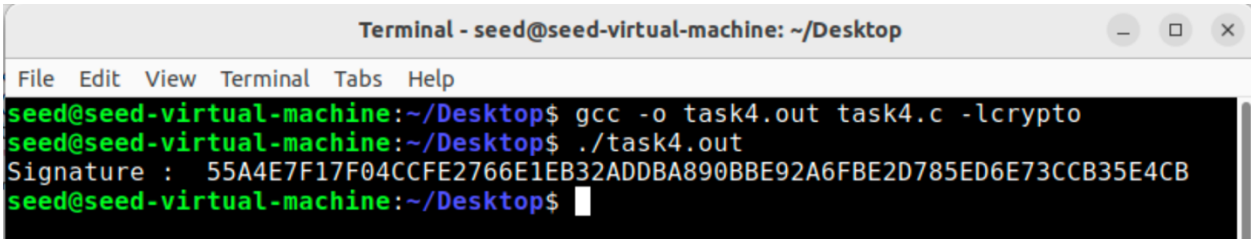


Figure: Running task4.c to generate Signature S

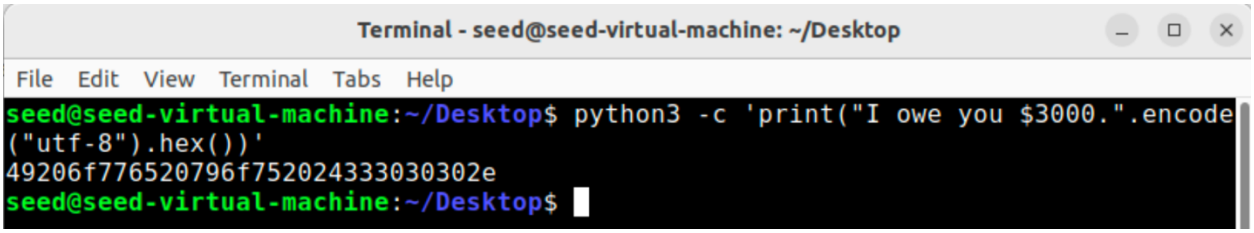
After running `task4.c` program file, we get the `signature S` as
55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB

```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *M = BN_new();
BIGNUM *d = BN_new();
BIGNUM *S = BN_new();

// Initialize n, e, d and C
BN_hex2bn(&n,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&e, "010001");
BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
BN_hex2bn(&M, "49206f776520796f752024323030302e"); //I owe you $2000.
//BN_hex2bn(&M, "49206f776520796f752024333030302e"); //I owe you $3000.

// Digital signature: S = E(PR_a, H(M)) = M^d mod n
// where d is the private-key and M is the message.
BN_mod_exp(S, M, d, n, ctx);
printBN("Signature : ", S);
```

Figure: task4.c program file

Now let us try to modify just a single bit of the message, say `$3000` instead of `$2000` and repeat the process of generating a Signature. Get the hex string of the modified message with,

```
$ python3 -c 'print("I owe you $3000.".encode("utf-8").hex())'
```



Replacing this hex string in `task4.c` and running it again gives us,

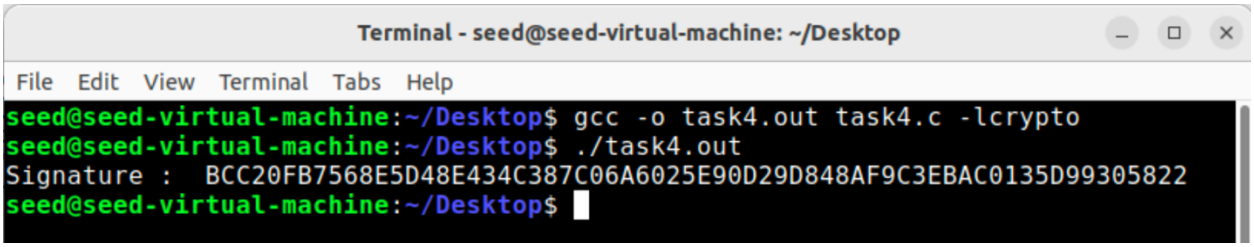


Figure: Running task4.c to generate Signature S

BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822 as the Signature `S`, which is completely different than the previous signature generated for `$2000`.

Hence, we can clearly observe that even for a single-bit modification in the message, the Signature generated is totally different than the one generated for the original message. Therefore, the signature does not leave out any patterns in processing a string and is totally generated at random, which proves to be highly secure since having collisions in this case is nearly impossible!

Task-5 Verifying a Signature

The most important part in receiving an encrypted message is to verify if it was really from the actual person. Let's consider a scenario where Bob receives a message `M = 'Launch a missile.'` from Alice, with her Signature `S`. We know Alice's public key is $(\boldsymbol{e}, \boldsymbol{n})$**.** We must verify if the signature is indeed **Alice's** or not. We have the public key, message and signature as follows.

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

We can use Alice's public key $(\boldsymbol{e}, \boldsymbol{n})$ to compute Verification Signature `V` to verify her Signature `S`.

$$V = E(PU_a, S)$$

$$\Rightarrow V = S^e mod\ n$$

Here, $S = M^d mod\ n$ and $V = S^e mod\ n$. Therefore, **V=M** for a valid signature.
First we must convert the message `M` to a hex-string using Python.

```
$ python3 -c 'print("Launch a missile.".encode("utf-8").hex())'
```

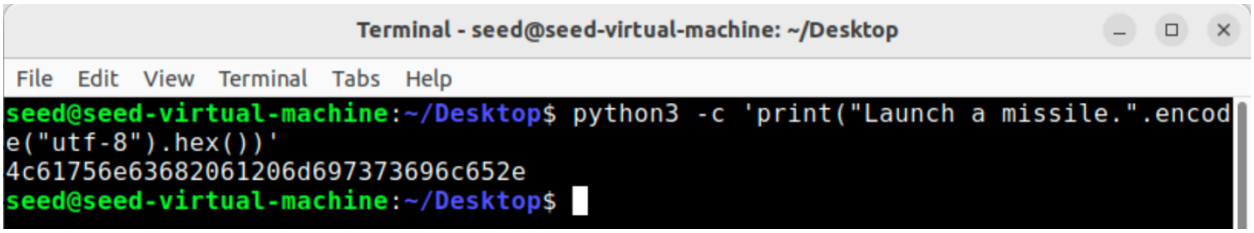


Figure: Convert plaintext to hex string using python

The generated hex-string is **`4c61756e63682061206d697373696c652e.`**

```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *M = BN_new();
BIGNUM *V = BN_new();
BIGNUM *S = BN_new();

// Initialize n, e, d and C
BN_hex2bn(&n,"AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
BN_hex2bn(&e, "010001");
BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
BN_hex2bn(&M, "4c61756e63682061206d697373696c652e"); //Launch a missile.

/* Verify a digital signature using public-key (e, n)
        E(PU_a, S): E(PU_a, E(PR_a, H(M))) = S^e mod n
        Here, S = M^d mod n and V = S^e mod n, so V=M */
BN_mod_exp(V, S, e, n, ctx);
printBN("Original Message : ", M);
printBN("Computed Message : ", V);

if (BN_cmp(M, V)==0){
        printf("Signature verified!\n");
}
else {
        printf("Verification failed!\n");
}
```

Figure: task5.c program file

The code in `task5.c` is similar to the previous programs. So we compute `V` using the Signature and public key of Alice. If it matches with the message `M`, then the Signature was indeed from Alice.

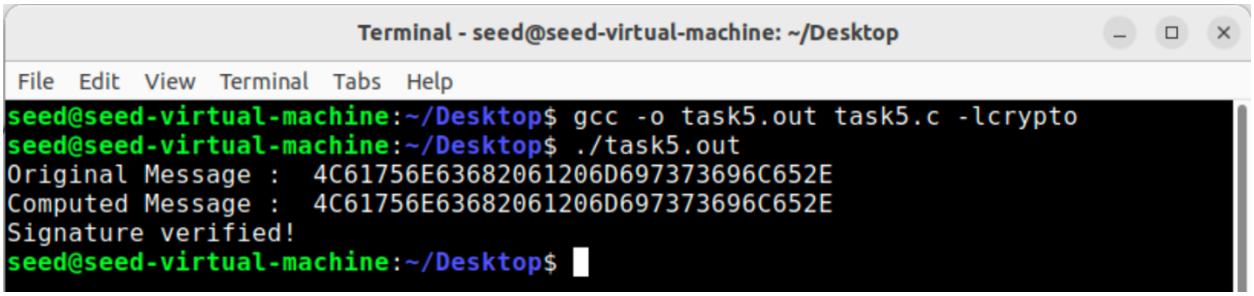


Figure: Running task5.c to compute Verification V

As you can see from the above terminal output, after running `task5.c`, we get `V` which is exactly the same as the message `M`. Therefore we can conclude that the **signature `S` was actually from Alice** and not anyone else.

Now, let's say the Signature gets corrupted in the middle while sending it through the network, and the last 2 bits are `3F` instead of `2F` (which means only **one** bit is changed).

So the signature S will be modified as
`S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F`

Let's repeat the process of verification by running `task5.c` again with the **modified Signature**.

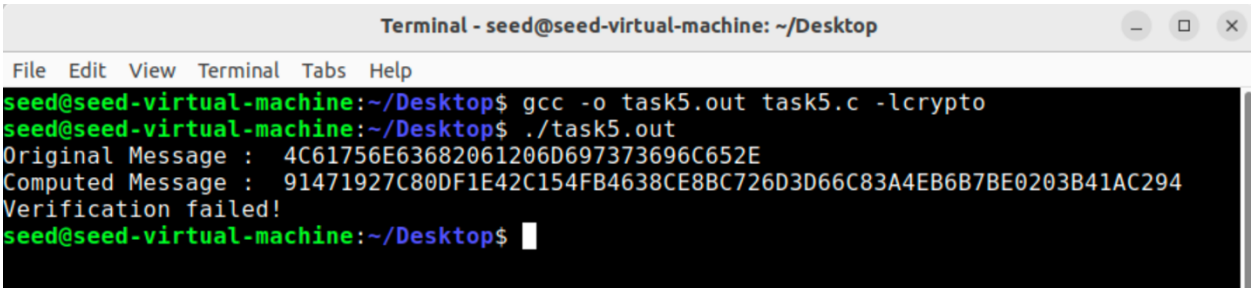


Figure: Running task5.c to compute Verification V

We can observe that the Computed signature `V` is very different from `M` and is totally generated at random. Therefore, even a single bit change in the Signature, would represent a completely different Signature and Message. Hence, if anyone modifies even a single bit of the message, it can be easily identified in the Signature verification process.