Homework 5 Written Problems

- 5.2. (8 pts): Message digests are reasonably fast, but here's a much faster function to compute. Take your message, divide it into 128-bit chunks, and xor all the chunks together to get a 128-bit result. Do the standard message digest on the result. Is this a good message digest function?

In short, the answer is no. The proposed function will generate several collisions. Additionally, of the 6 conditions that must be met for a good hash function, it does not meet the following 3 conditions...

1. Given h, is it infeasible to find x H(y) = h → Does not meet
2. Given x, is it infeasible to find y H(y) = H(x) → Does not meet
3. is it infeasible to find any x, y H(y) = H(x) → Does not meet

- 5.14. (12 pts): For purposes of this exercise, we will define random as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function "+" and we have two inputs, x and y, then the output will be random if at least one of x and y are random. For the following functions, find sufficient conditions for x, y and z under which the output will be random:

$\sim x$

$x \oplus y$

$x \vee y$

$x \wedge y$

$(x \wedge y) \vee (\sim x \wedge z)$ [the selection function]

$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ [the majority function]

$x \oplus y \oplus z$

$y \oplus (x \vee \sim z)$

For each of the functions listed find sufficient conditions for the variables where produced output will be random.

1. $\sim X$: X is the random variable, other variables will be independent
2. $X \oplus Y$: If X or Y is random and have different binary, the output will be random. Z is allowed to be independent
3. $X \vee Y$: Same conditions above (X & Y random with different binary then output will be random)
4. $X \wedge Y$ Same conditions above (X & Y random with different binary then output will be random)
5. $(X \wedge Y) \vee (\sim X \wedge Z)$ [Selection function]: X & Y are different & independent values. The function will output a random result if the $(X \wedge Y)$ part of the function produces a non-zero number.
6. $(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$: The function will produce a random result if X, Y, or Z differ by a single bit.
7. $X \oplus Y \oplus Z$: Same as above: the function will produce a random result if X Y, or Z differ by a single bit.
8. $Y \oplus (X \vee \sim Z)$: Output will be random if X or Z differs by at least a single bit.

- 6.2. (8 pts): In [KPS] textbook, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the man-in-the-middle attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

Because the values produced by Diffie-Hellman cannot be sniffed and encrypted by a malicious attacker nor can the attacker successfully guess the shared secrets in the communication between the two parties.

- 6.8. (12 pts): Suppose Fred sees your RSA signature on m_1 and on m_2 (i.e., he sees $m_1^d \bmod n$ and $m_2^d \bmod n$). How does he compute the signature on each of $m_1^j \bmod n$ (for positive integer j), $m_1^{-1} \bmod n$, $m_1 m_2$, and in general $m_1^j m_2^k \bmod n$ (for arbitrary integers j and k)?

*Refered to a solution from Stackoverflow.com for help

1. $M_1^i \pmod{n}$ can be calculated using $(m_1^d)^i \pmod{n} = (m_1^i)^d$
2. The signature of the inverse m_1^{-1} is equal to

$(m_1^{-1})^d \bmod n = (m_1^{-d}) \bmod n = (m_1^d)^{-1} \bmod n.$

3. Compute using the multiplicative inverse of $m_1^d \bmod n$ using the Euclidian Algorithm

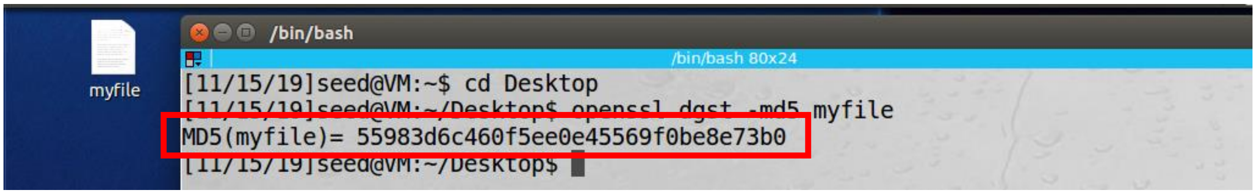
4. If the signature of $m_1 * m_2$ is desired, then calculate ...

$(m_1 * m_2)^d \bmod n = ((m_1)^d \bmod n)*(m_2^d \bmod n) \bmod n$

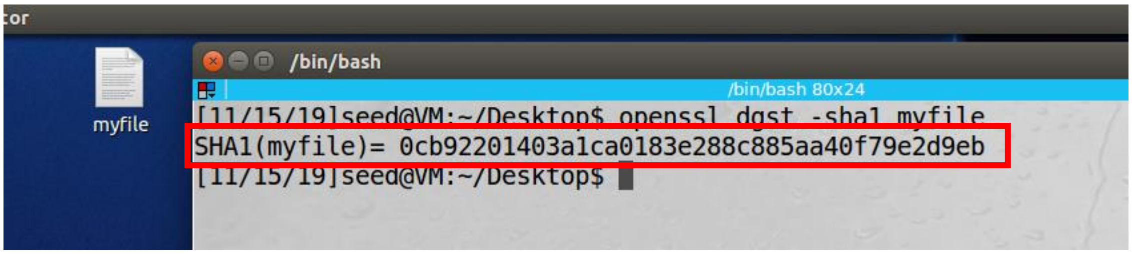
Homework 5 Lab Tasks

3.1 Task 1: Generating Message Digest and MAC

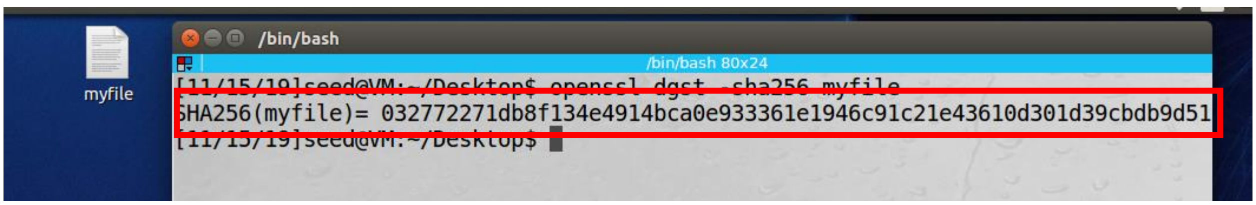
Using md5



Using sha1

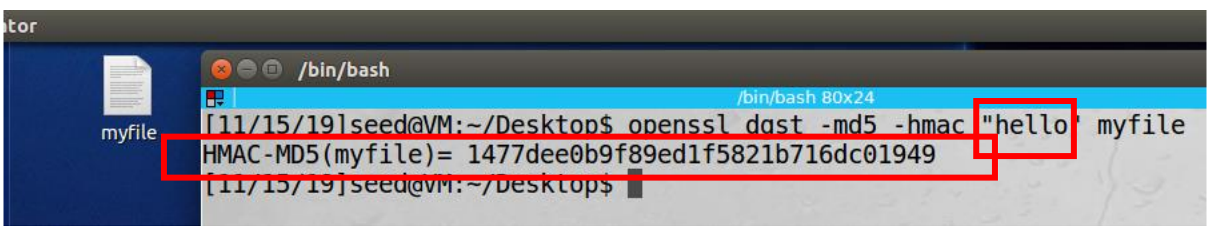


Using sha256

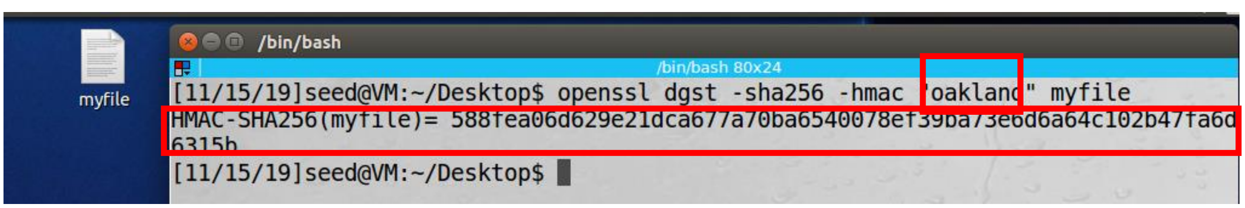


3.2 Task 2: Keyed Hash and HMAC

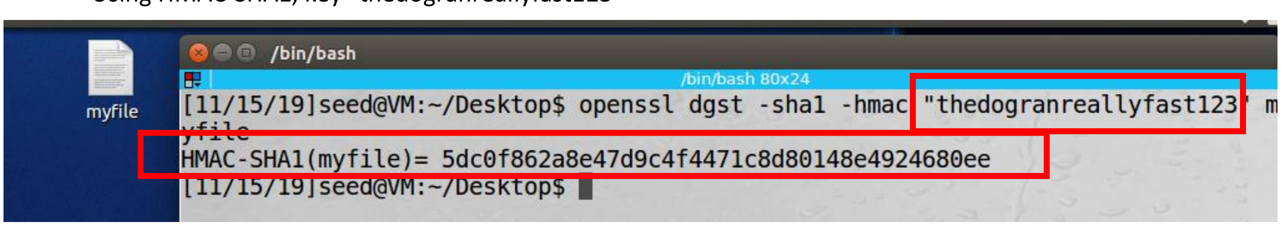
Using HMAC-MD5, key “hello”



Using HMAC-SHA256, key “oakland”



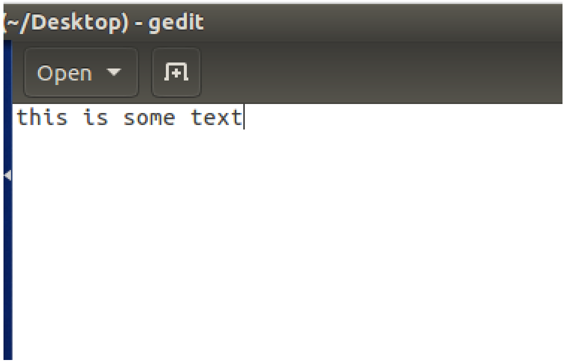
Using HMAC-SHA1, key “thedogranreallyfast123”



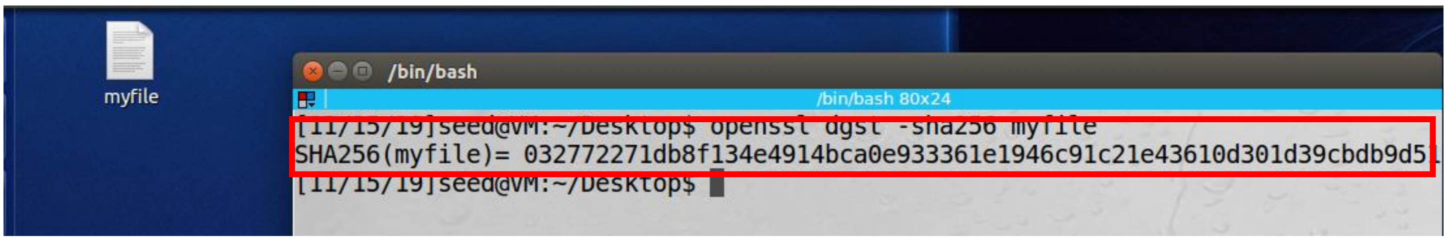
When using the HMAC function, a key of any size can be used because HMAC because it is a cryptographic hash function and allows for the mapping of data **arbitrary** in size to a bit string that is fixed in size.

3.3 Task 3: The Randomness of One-way Hash [9 pts]

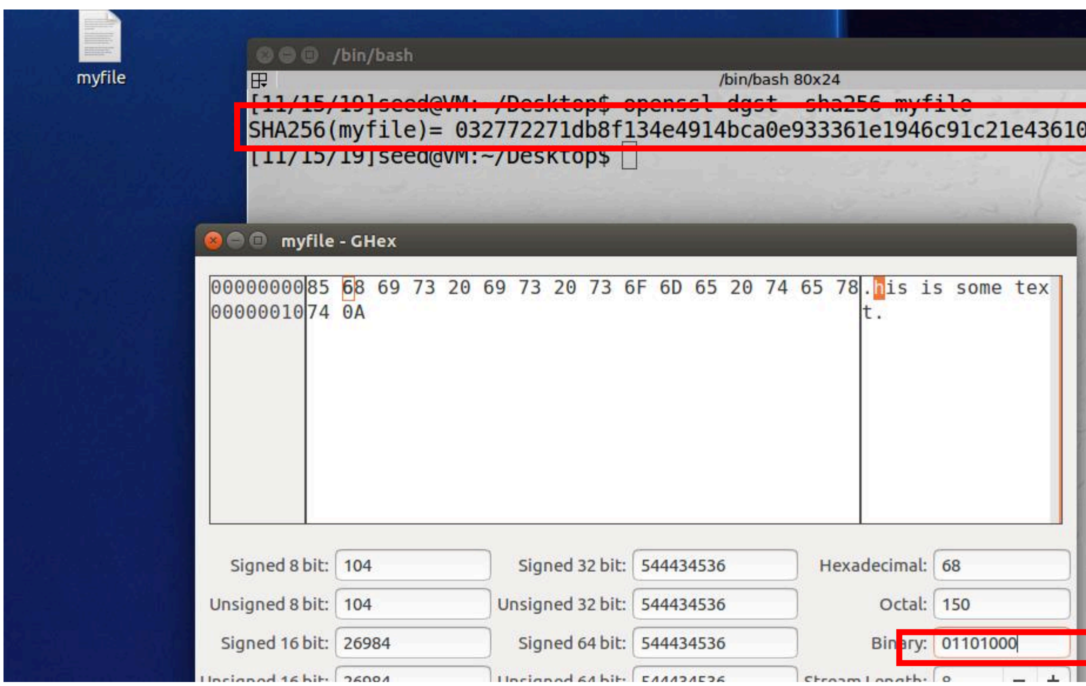
Create a text file of any length



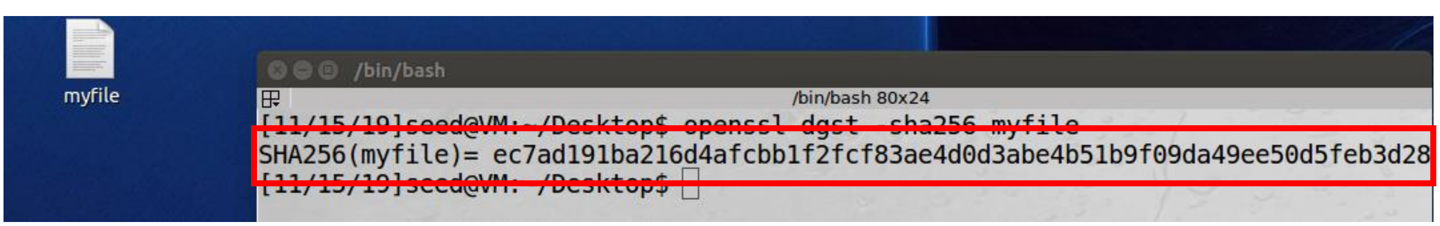
Generate a hash h1 for the file using a specific algorithm, sha-256 used



Flip one bit of the input file using ghex



Generate the hash value for the modified file



The 2 generated hashes are incredibly different. Here is a short program to count the number of same bits between h1 and h2
The program found that there were only 3 corresponding values in the 2 hashes.

```
Desktop — -bash — 80×24
mooninites-438:Desktop codywilliams$ python script.py
3
mooninites-438:Desktop codywilliams$
```

```python
h1 = "032772271db8f134e4914bca0e933361e1946c91c21e43610d301d39cbdb9d51"
h2 = "ec7ad191ba216d4afcbb1f2fcf83ae4d0d3abe4b51b9f09da49ee50d5feb3d28"

count = 0

for x in range(len(h1)):
    if h1[x] == h2[x]:
        count = count + 1

print count
```

3.4 Task 4: Hash Collision-Free Property [20+10 bonus pts]

Components of program

1. randomMessage
   a. uses a random seed to create a randomly generated message for hash collision checking

```c
//this function is used to create random strings for purposes
//hash collision detection
void randomMessage(char *msg) {
    int i;
    for (i=0;i<11;i++)
        msg[i] = rand()%256-128;
}
```

2. getHash function
   a. given a message and a digest name, this function produces the hash for the given message according to the given digest

```c
void getHash(char * hashname, char *msg, unsigned char *md_value) {

    //Initialize digest parameters
    EVP_MD_CTX *mdctx;
    const EVP_MD *md;
    int md_len, i;

    //Add all digests to the program
    // credit to John Dorman for this suggestion
    OpenSSL_add_all_digests();

    //Throw an error if we are given a bad hash
    //taken from example program
    md = EVP_get_digestbyname(hashname);
    if(!md) {
        printf("Unknown message digest %s\n", hashname);
        exit(1);
    }

    //generate and return hash
    //hash generation taken from sample program
    mdctx = EVP_MD_CTX_create();
    EVP_DigestInit_ex(mdctx, md, NULL);
    EVP_DigestUpdate(mdctx, msg, strlen(msg));
    EVP_DigestFinal_ex(mdctx, md_value, &md_len);
    EVP_MD_CTX_destroy(mdctx);
}
```

3. crackHash
    a. given a specified hash, this function will generate random messages and their corresponding hash values until the produced hash matches the given hash

```
int crackHash(char * hashname) {
    //Initialize message parameters
    char msg1[11], msg2[11];
    unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
    int count=0, i;

    //Get the hash that we will try to be cracking
    //This is the hash that will be cmpared in every iteraton of tl
    //test
    randomMessage(msg1);
    getHash(hashname, msg1, digt1);
    // run the crack
    do {
        //generate random message and hash
        randomMessage(msg2);
        getHash(hashname, msg2, digt2);
        count++;
        //compare the 2 hashes
    } while (strncmp(digt1, digt2, 3)!=0);
    printf("hash cracked: %d tries, digest =", count, msg1, msg2);
    for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
    printf("\n");
    return count;
}
```

4. crackCollision
    a. this function generates two random messages and their corresponding hash values and checks to see if the produced hash values are equivalent.

```
int crackCollision(char * hashname) {
    //Initilize our message inputs
    char msg1[11], msg2[11];
    unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
    int count=0, i;
    //generate random hashes
    //check if the hashes are equal until there are 2 equal values
    do {
        //Genreate random message and has1
        randomMessage(msg1);
        getHash(hashname, msg1, digt1);
        //Generate random message and hash2
        randomMessage(msg2);
        getHash(hashname, msg2, digt2);
        count++;

        //Compare the 2 hashes
    } while (strncmp(digt1, digt2, 3)!=0);
    //printf("\n cracked after %d tries! %s and %s has same digest ",
    printf("hash cracked: %d tries, digest = ", count);
    for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
    printf("\n");
    return count;
}
```

5. Main
    a. This function calls crackCollision 10 times and outputs the average number of tries the function utilizes to find the matching hashes
    b. This function also calls crackHash 5 times and outputs the average number of tries the function utilizes to find the matching corresponding hash for the given hash value

```
main(int argc, char *argv[])
{
    //will be testing using md5 for simplicity sake
    char *hashname;
    hashname = "md5";

    //create a random seed
    srand((int)time(0));

    //initialize counter variables
    int i;
    int count;

    //Run through the collision detection checker 15 times and
    //output the average of each of these times
    for (i=0,count=0;i<10;i++){
        count+=crackCollision(hashname);
    }
    printf("collision-free cracking average: %d \n", count/10);

    //Run through the one-way hash collision detection checker 5 times
    //output the average of each of these times
    for (i=0,count=0;i<5;i++){
        count+=crackHash(hashname);
    }
    printf("one-way cracking average: %d \n", count/5);
}
```

1. How many trials it will take you to find two messages with the same hash values using the brute-force method? You should repeat your experiment for multiple times, and report your average number of trials.
    a. After 10 trials of hash collision detecting, the average number of tries was about 30000

```
[11/18/19]seed@VM:~/.../program$ ./hash
hash cracked: 9424 tries, digest = d41d8c
hash cracked: 20147 tries, digest = 00b474
hash cracked: 1394 tries, digest = 00fd86
hash cracked: 45855 tries, digest = 001b75
hash cracked: 2177 tries, digest = d41d8c
hash cracked: 138741 tries, digest = d41d8c
hash cracked: 1568 tries, digest = 00de1f
hash cracked: 55163 tries, digest = 005780
hash cracked: 26855 tries, digest = 00488b
hash cracked: 14983 tries, digest = 0089ed
collision-free cracking average: 31630
```

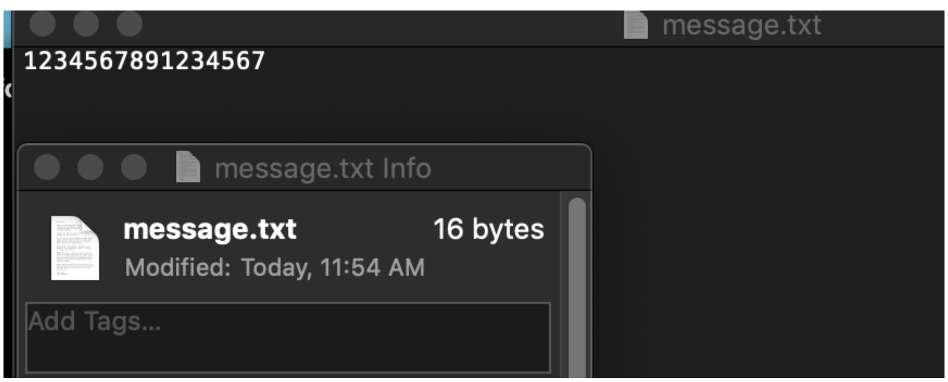
2. How many trials will it take you to find a message that has the same hash value as a given/known message's hash value using the brute-force method? Similarly, you should report the average.
    a. After 5 trials of one-way hash cracking , the average number of trails was about 11500000

```
hash cracked: 11965079 tries, digest =255966
hash cracked: 3318563 tries, digest =d8ee06
hash cracked: 7939398 tries, digest =01ee63
hash cracked: 164016 tries, digest =15c40a
hash cracked: 34042755 tries, digest =17c268
one-way cracking average: 11485962
[11/18/19]seed@VM:~/.../program$
```

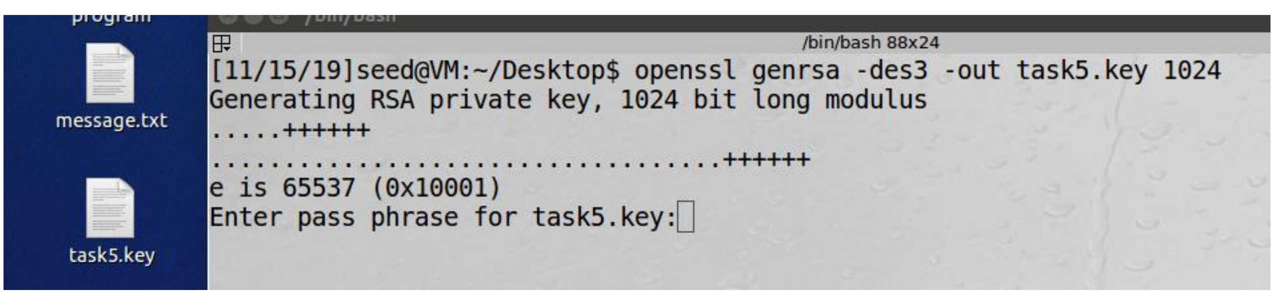
3. Based on your observation, which case is easier to break using the brute-force method?
    a. It is clearly easier to break the collision-free property using the brute force method

3.5 Task 5: Performance Comparison: RSA versus AES [8 pts]

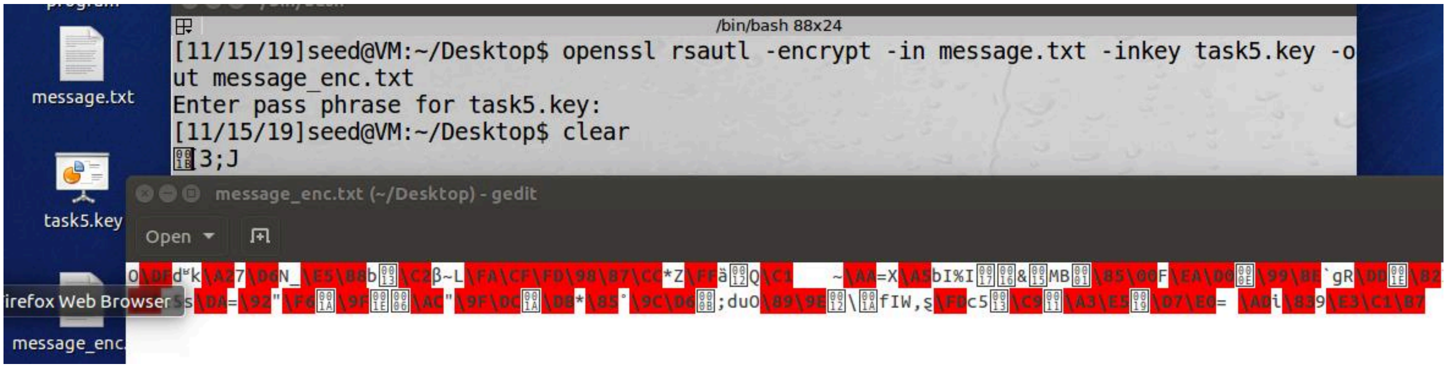
Prepare a 16 byte-message called message.txt



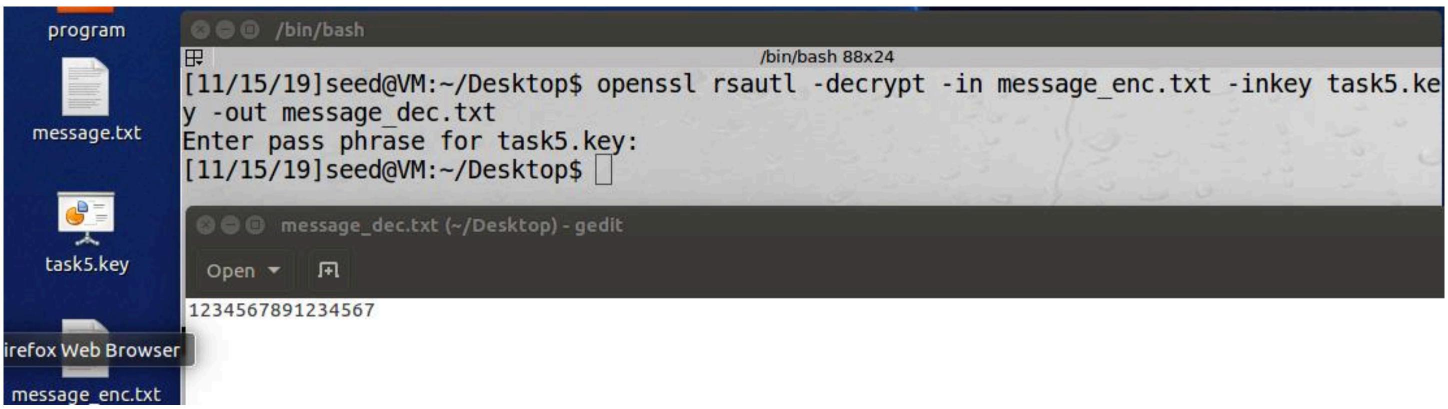
generate a 1024-bit RSA public/private key pair



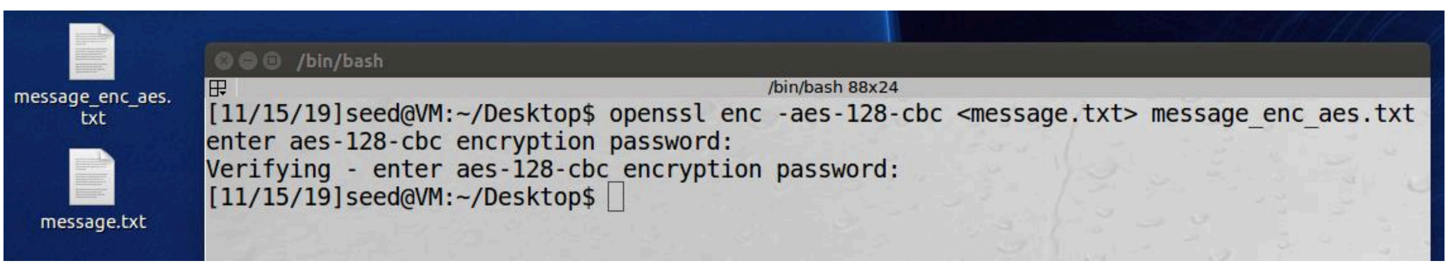
Encrypt message.txt using the public key; save the output in message enc.txt.



Decrypt message enc.txt using the private key.

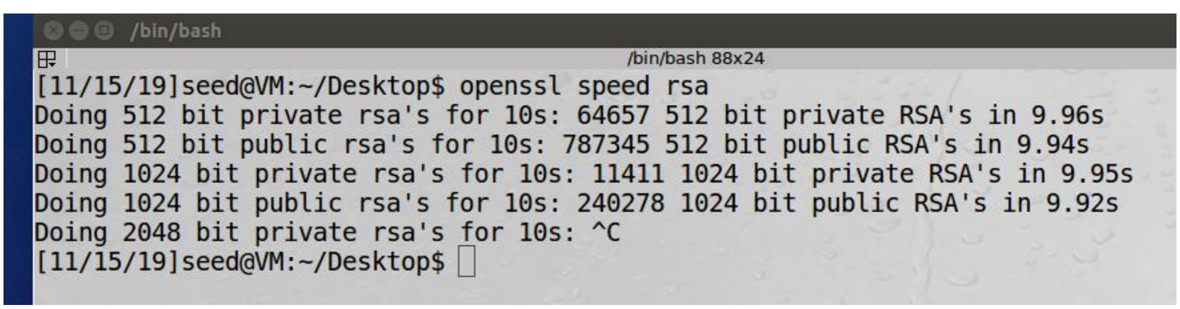


Encrypt message.txt using a 128-bit AES key.

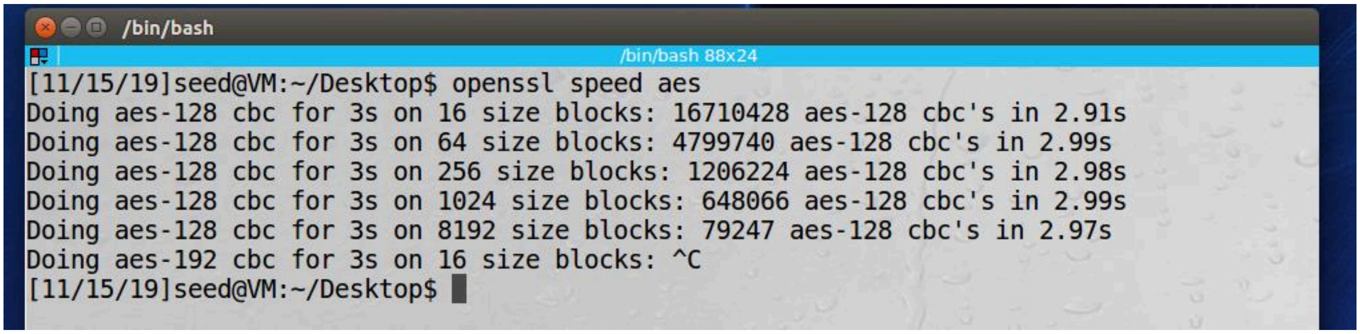


On my machine, the operations occurred too fast to notice a significant difference. The times appeared to be similar

Measuring speed using openssl speed rsa



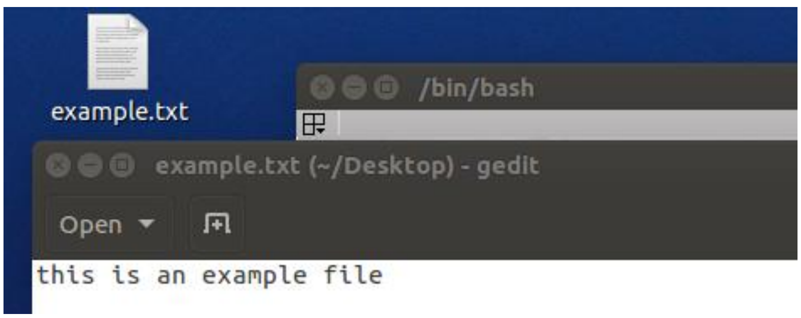
Measuring speed using openssl speed aes



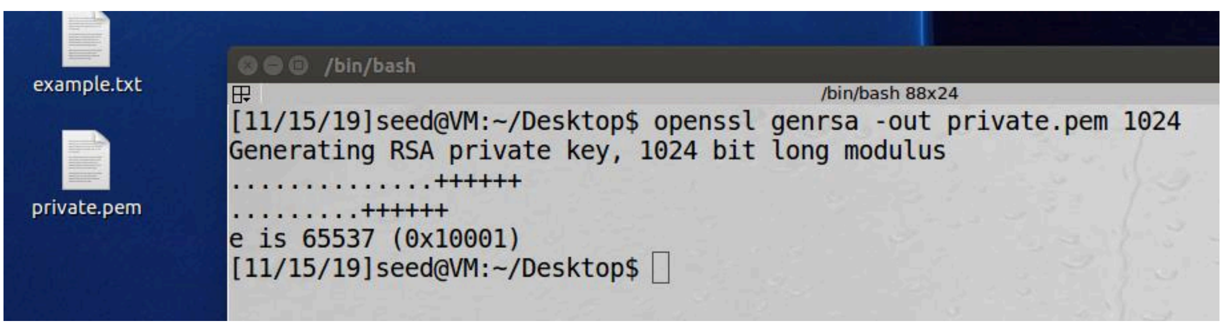
After these tests, it appears that the aes function runs much quicker than the rsa function

3.6 Task 6: Create Digital Signature

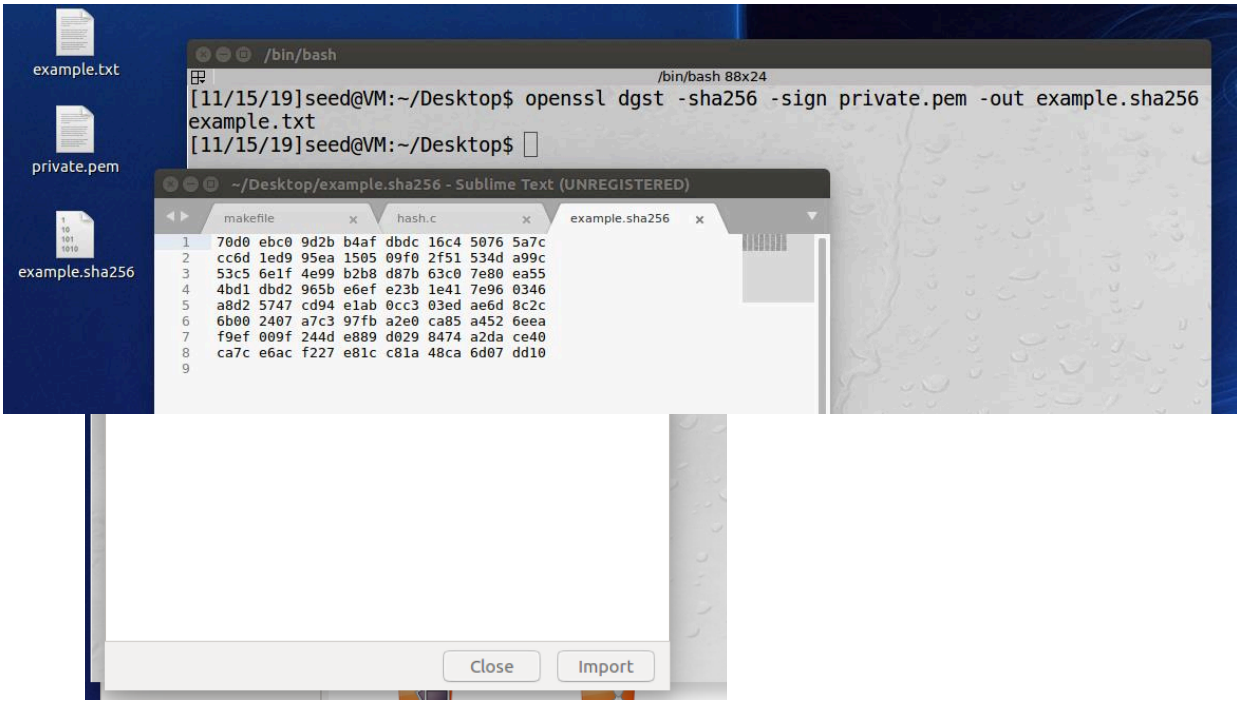
Prepare example.txt of any size



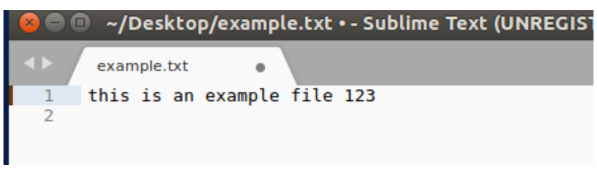
Also prepare an RSA public/private key pair



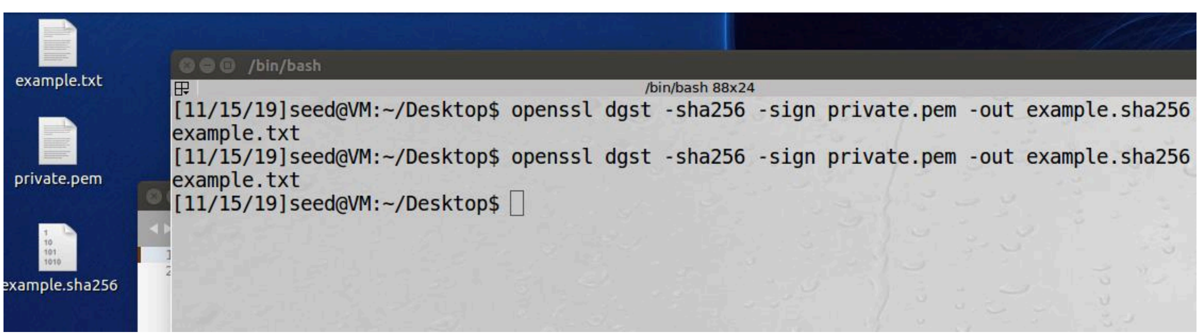
Creating a digital signature and with SHA256 hash of example.txt; save the output in example.sha256.
Verifying the digital signature (the below command is just a combination of steps 1&2 in the assignment guideline)



Modifying example.txt



reverifying signature



The commands I used for the previous actions
Preparing an rsa key: openssl genrsa -out private.pem 1024
Creating and verifying signature: openssl dgst -sha256 -sign private.pem -out example.sha256 example.txt

Digital signatures are useful because they act as a virtual fingerprint that is unique to a person or communicating entity. Additionally, digital signatures are used to identify users and protect the legitimacy of digital messages or documents.