- Q1 (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?

  The four requirements for a good digest function are as follows:

  - Easy to compute H(m)
  - Given H(m) but not m, it's computationally infeasible to fine m
  - Given H(m), its computationally infeasible to find m' such that H(m') = H(m)
  - Computationally infeasible to fine m1, m2, such that H(m1) = H(m2)

  The presented message digest function is not good because it runs int collisions. For example lets us say there is a an m1 that is 0011 1010 0000 1111 and an m2 that is 1111 1010 0000 0011. Because m2 is just a rearrangement of m1, both of them will have the same xor resultant of 0110. Thus, both m's will produce the same hash as well.

- Q2 (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

  - Independent: y and z
  - Random: z

  x $\oplus$ y

  - Independent: z
  - Random: x or y or both. x and y need to have at least a 1-bit difference.

  x $\vee$ y

  - Same conditions as x $\oplus$ y

  x $\wedge$ y

  - Same conditions as x $\oplus$ y

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

  - Two possible conditions
    - x and y are different

- ~x and z are different

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

- We have x, y, and z with at least one of them having at least a 1-bit difference

$x \oplus y \oplus z$

- Two possible conditions
  - Two of them are different with having at least a 1-bit difference
  - All three are different are different with having at least a 1-bit difference

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

- x is different to z (vice versa)

- Q3 (8 pts): In KPS textbook [1] , 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?

  It is because of the structure of the algorithm itself is asymmetric. The public keys are used for encryption while the private keys are used for decryption. Thus, since the attacker does not have the private keys, he/she is not able to decrypt the messages.

• Q4 (12 pts): Suppose Fred sees your RSA signature on m1 and on m2 (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)?Lab and Programming Tasks [60 + 10 bonus pts]

- Based on the given information, I will give the following assumptions and facts
  - Assumption: Fred saw $m_1^d \bmod n$ and $m_2^d \bmod n$
  - Fact: $m_1^{di} \bmod n = m_1^{(i)\wedge d} \bmod n$
  - Statement: m1m2 = m3 since m1 is being multiplied to m2.
- Using the fact, for every $m_1^j \bmod n$, the signature will be $m_1^{dj} \bmod n$ for every positive integer j.
- Using the fact, the signature of $m_1^{-1} \bmod n$ is $m_1^{-d} \bmod n$ where $m_1^{-d} \bmod n$ is the **multiplicative inverse** of $m_1^d \bmod n$ and thus can be calculated using the **extended Euclidean algorithm.**
  - **Note: I learned this from MATH 470**
- Using the statement, the signature of m1m2 is $m_3^d \bmod n$
- Using the same logic as the above bullet, multiply m1jm2k together to get a new variable in which we will call M and calculate Md mod n

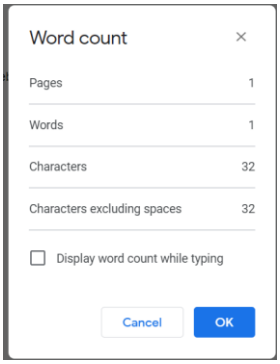
Task 1: Generating Message Digest and MAC

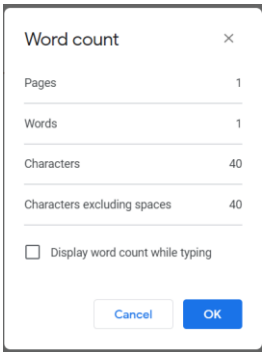
```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha1 random_words.txt
SHA1(random_words.txt)= 78c91c4aebd86af4181f3cc79a7be3c6708978ef
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 random_words.txt
MD5(random_words.txt)= 89fbdf2bc0762db53421f8b21eb628be
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -rmd160 random_words.txt
RIPEMD160(random_words.txt)= 43edd4800d82bb2aa0a2d3ecee3167c3158a483b
[11/01/21]seed@VM:~/.../HW_3$
```

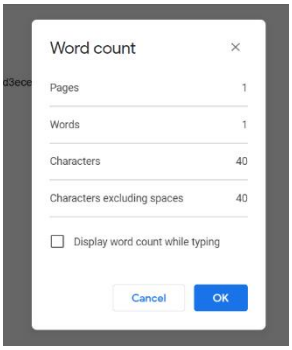
- I have generated a sha1, md5, and rmd160 message digests for my random_words.txt file.

```
Hello! This is a file with a bunch of random text. I will continue to write things that don't make sense, so I can test them in task 1. There is an omnipresent being that observes all over the universe. The world turtle is where our universe resides. It's just so big that we can not see it as of yet. This is some crazy text that does not make sense at all. Like this is just crazy. Hopefully this file is big enough for part one of this assignment.
```

- As stated in the name itself, the random_words.txt file is just a txt file full of random texts.
- When observing the resultant digests, all three hash functions produced different message digests for the same exact file. This is obvious due to the fact that they are 3 different hash functions.







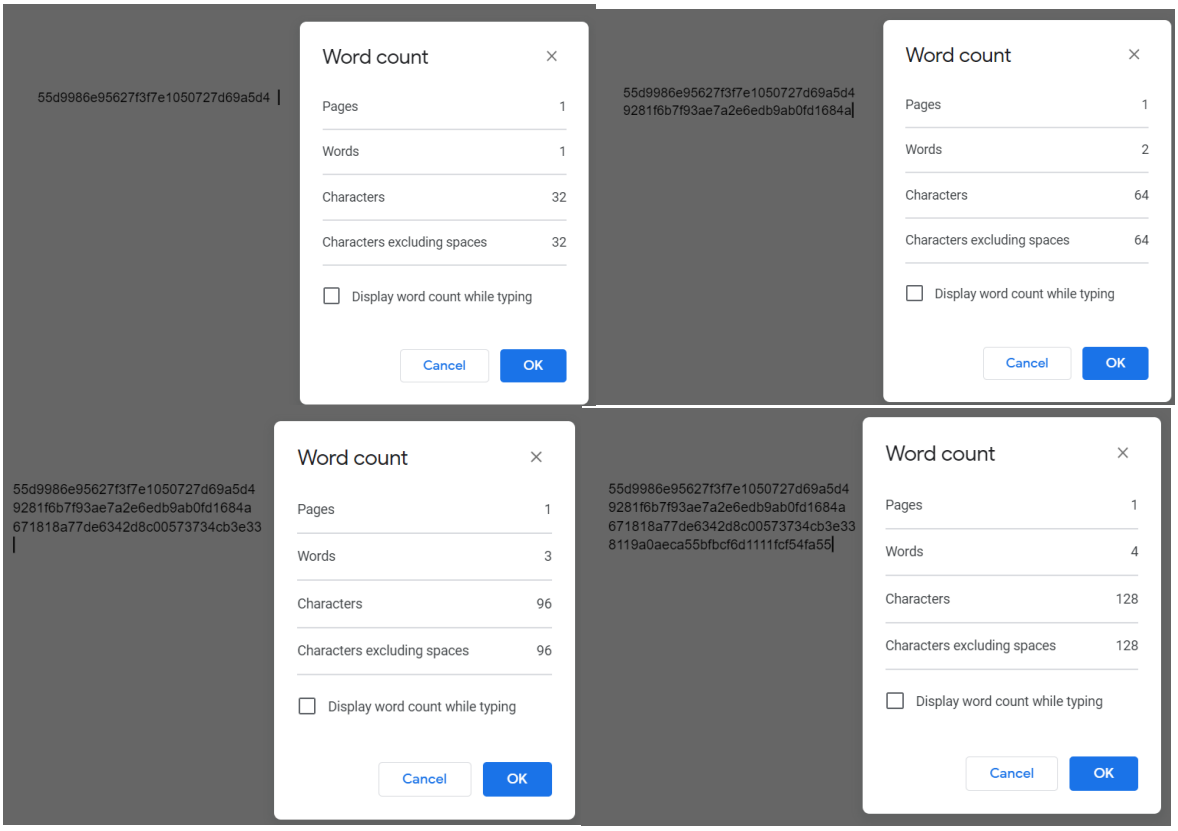
- A key observation I have made is that md5 is the algorithm that produces the smallest size message digest of 16 bytes (128 bits) while the other two hash functions produce message digests of 20 bytes (160 bits).
    - **NOTE:** The order of the pictures from left to right are md5, sha1, and ripemd160
    - **NOTE:** Since each output is in hex and is concatenated in one line, divide the number of characters by 2 for each output.

Task 2: Keyed Hash and HMAC [9 pts]

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 -hmac "abcdefg" random_words.txt
HMAC-MD5(random_words.txt)= 55d9986e95627f3f7e1050727d69a5d4
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 -hmac "abcdefghijklmn" random_words.txt
HMAC-MD5(random_words.txt)= 9281f6b7f93ae7a2e6edb9ab0fd1684a
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 -hmac "a" random_words.txt
HMAC-MD5(random_words.txt)= 671818a77de6342d8c00573734cb3e33
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 -hmac "aaabbbccc" random_words.txt
HMAC-MD5(random_words.txt)= 8119a0aeca55bfbcf6d1111fcf54fa55
```

- I produced message digests for the randmon_words.txt file by using md5 and hmac.
- When producing these message digests, I used keys of many different sizes as shown in the picture.

55d9986e95627f3f7e1050727d69a5d4
9281f6b7f93ae7a2e6edb9ab0fd1684a
671818a77de6342d8c00573734cb3e33
8119a0aeca55bfbcf6d1111fcf54fa55

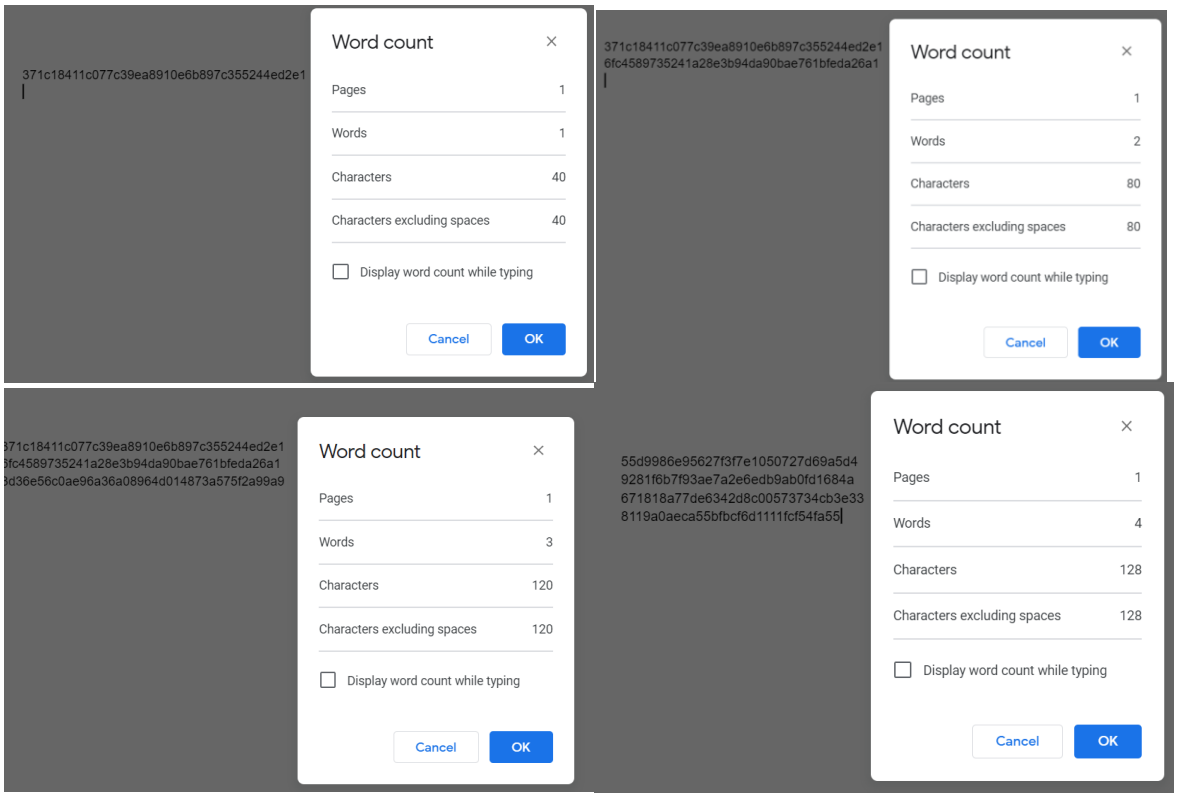


- When observing all four message digests that were produced using different key lengths, they visually appear to be of different lengths
- However, this is not the case. As shown in the 4 pictures, all the message digests are of length 32.
- Thus, the length of the key does not affect the size of the message digests for md5 hmac

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha1 -hmac "abcdefg" random_words.txt
HMAC-SHA1(random_words.txt)= 371c18411c077c39ea8910e6b897c355244ed2e1
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha1 -hmac "abcdefghijklmn" random_words.txt
HMAC-SHA1(random_words.txt)= 6fc4589735241a28e3b94da90bae761bfeda26a1
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha1 -hmac "a" random_words.txt
HMAC-SHA1(random_words.txt)= 3d36e56c0ae96a36a08964d014873a575f2a99a9
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha1 -hmac "aaabbbccc" random_words.txt
HMAC-SHA1(random_words.txt)= e17f069516abe10cd3d3540a82aebdc87b6cb6b8
```

- I produced message digests for the randmon_words.txt file by using sha1 and hmac.
- When producing these message digests, I used the same keys as last time

371c18411c077c39ea8910e6b897c355244ed2e1
6fc4589735241a28e3b94da90bae761bfeda26a1
3d36e56c0ae96a36a08964d014873a575f2a99a9
e17f069516abe10cd3d3540a82aebdc87b6cb6b8

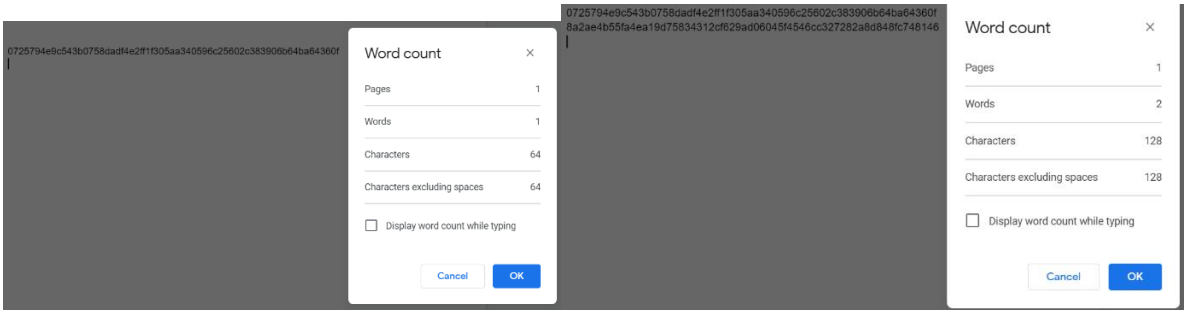


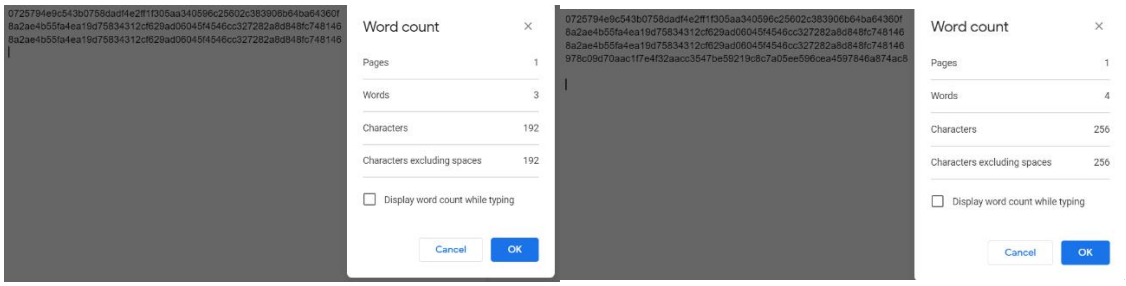
- Again, the message digests visually appear to be of different lengths.
- However, they are all of the same length as displayed by the four images above
- This time, since I am using sha1 hmac, the length of the message digests are of length 40
- For sha1 hmac, the length of the keys do not affect the length of the digests as well

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 -hmac "abcdefg" random_words.txt
HMAC-SHA256(random_words.txt)= 0725794e9c543b0758dadf4e2ff1f305aa340596c25602c383906b64ba64360f
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 -hmac "abcdefghijklmn" random_words.txt
HMAC-SHA256(random_words.txt)= 8a2ae4b55fa4ea19d75834312cf629ad06045f4546cc327282a8d848fc748146
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 -hmac "a" random_words.txt
HMAC-SHA256(random_words.txt)= f4f928e503da4cccd83545a10efe466118b5e898c0fbdd2b8a2c83ddb433bf8c
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 -hmac "aaabbbccc" random_words.txt
HMAC-SHA256(random_words.txt)= 978c09d70aac1f7e4f32aacc3547be59219c8c7a05ee596cea4597846a874ac8
```

- Finally, I produced the sha256 hmac message digests
- Again, I used the same keys

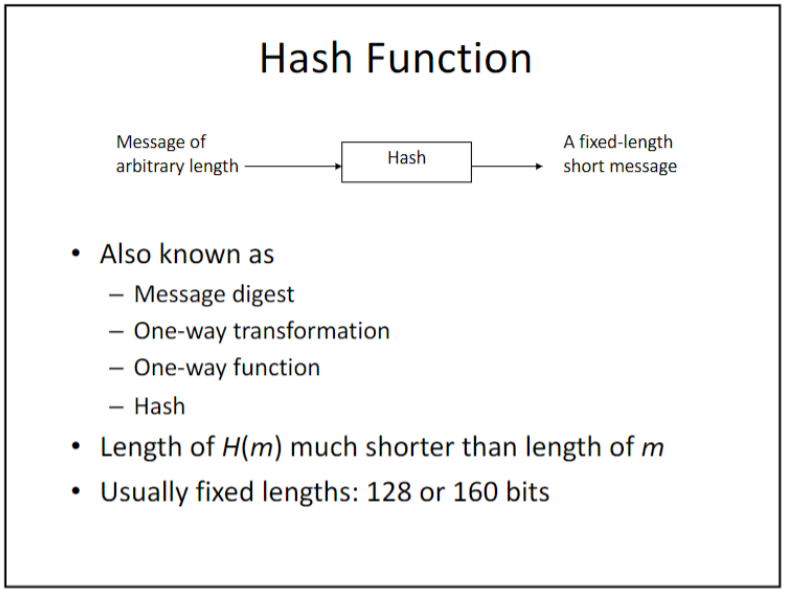
0725794e9c543b0758dadf4e2ff1f305aa340596c25602c383906b64ba64360f
8a2ae4b55fa4ea19d75834312cf629ad06045f4546cc327282a8d848fc748146
8a2ae4b55fa4ea19d75834312cf629ad06045f4546cc327282a8d848fc748146
978c09d70aac1f7e4f32aacc3547be59219c8c7a05ee596cea4597846a874ac8





- The message digests visually appear to be of slightly different lengths.
- However, they are all of the same length as displayed by the four images above
- This time, since I am using sha256 hmac, the length of the message digests are of length 64
- For sha256 hmac, the length of the keys do not affect the length of the digests as well

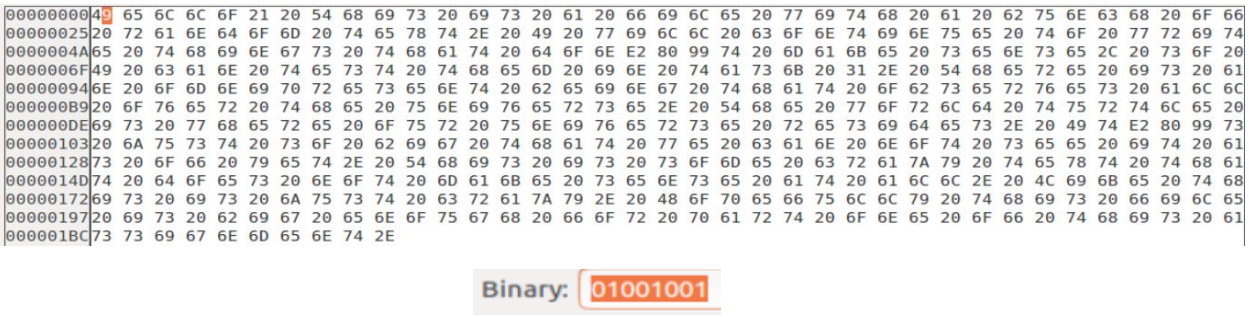
Notice how the length of a key does not affect the size of the message digests produced by all three types. The main reason for this is the design of the hash functions themselves.



The primary function of a hash function is to take any message of any length and always produce a message digest of a **fixed length**. Since the hash function will always produce a message digest of a fixed length (usually 128 or 160 bits), the length of a key does not affect the length of a produced message digest.

Task 3: The Randomness of One-way Hash

- This is the hex representation of my random_words.txt file. The first byte has a hex value of 49 and thus its binary representation is 01001001



- In order to flip one bit of the file, I changed the first byte of the hex from 48 to 49 and created a new file called random_mod.txt to store said change.
- Thus, the binary has gone from 01001000 to 01001001

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 random_words.txt
MD5(random_words.txt)= 89fbdf2bc0762db53421f8b21eb628be
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 random_mod.txt
MD5(random_mod.txt)= c3d97bc9addff51b079014ea6d329c05
```

- After the modification, I proceeded to created an md5 hash for both random_words.txt and random_mod.txt
- **Note:** H1 represents the hash for random_words.txt and H2 represents the hash for random_mod.txt
- When observing the two hashes, it appears they are completely different.

```
#include <string>
#include <iostream>
using namespace std;

void hex_to_binary(string &hex){
    string converted_hex="";
    int hex_length=hex.length();

    for(int i=0; i<hex_length; i++){
        if(hex[i] == '0'){
            converted_hex = converted_hex + "0000";
            continue;
        }
        else if(hex[i] == '1'){
            converted_hex = converted_hex + "0001";
            continue;
        }
        else if(hex[i] == '2'){
            converted_hex = converted_hex + "0010";
            continue;
        }
        else if(hex[i] == '3'){
            converted_hex = converted_hex + "0011";
            continue;
        }
        else if(hex[i] == '4'){
            converted_hex = converted_hex + "0100";
            continue;
        }
        else if(hex[i] == '5'){
            converted_hex = converted_hex + "0101";
            continue;
        }
        else if(hex[i] == '6'){
            converted_hex = converted_hex + "0110";
            continue;
        }
        else if(hex[i] == '7'){
            converted_hex = converted_hex + "0111";
            continue;
        }
        else if(hex[i] == '8'){
            converted_hex = converted_hex + "1000";
            continue;
        }
        else if(hex[i] == '9'){
            converted_hex = converted_hex + "1001";
            continue;
        }
        else if(hex[i] == 'a'){
            converted_hex = converted_hex + "1010";
            continue;
        }
        else if(hex[i] == 'b'){
            converted_hex = converted_hex + "1011";
            continue;
        }
        else if(hex[i] == 'c'){
            converted_hex = converted_hex + "1100";
            continue;
        }
        else if(hex[i] == 'd'){
            converted_hex = converted_hex + "1101";
            continue;
        }
        else if(hex[i] == 'e'){
            converted_hex = converted_hex + "1110";
            continue;
        }
        else if(hex[i] == 'f'){
            converted_hex = converted_hex + "1111";
            continue;
        }
        else{
            cout << "There is an invalid hex value at index " << i << endl << "the value is " << hex[i] << "endl";
            return;
        }
    }

    hex = converted_hex;
}
```

- Before making anymore observations, I decided to create a program that would compare the two hashes together through their hex and binary bits.

- For this part of the code, it converts the hex bits to binary bits for a given hash.
- It does this by looping through each hex bit and converting it to its respective binary value.

```
int amount_of_same_bits(string &hash1, string &hash2){
    int length = hash1.length();
    int amnt = 0;
    for(int i=0; i<length; i++){
        if(hash1[i] == hash2[i]){
            amnt++;
        }
    }
    return amnt;
}
```

- This part of the code does the comparison between the two hashes by looping through the indices of both hashes and comparing the values.

```
int main(){
    string h1;
    string h2;
    cout << "Enter H1: ";
    cin >> h1;
    cout << "Enter H2: ";
    cin >> h2;


    cout << "The number of hex bits that are the same between H1 and H2 is: " << amount_of_same_bits(h1, h2) << endl;
    hex_to_binary(h1);
    hex_to_binary(h2);
    cout << "The number of binary bits that are the same between H1 and H2 is: " << amount_of_same_bits(h1, h2) << endl;
}
```

- Finally, this part of the code accepts the hashes from user input and compared both hashes in their hex and binary forms.

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 random_words.txt
MD5(random_words.txt)= 89fbdf2bc0762db53421f8b21eb628be
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -md5 random_mod.txt
MD5(random_mod.txt)= c3d97bc9addff51b079014ea6d329c05
[11/01/21]seed@VM:~/.../HW_3$ g++ Task_3.cpp -o Task_3 -g
[11/01/21]seed@VM:~/.../HW_3$ ./Task_3
Enter H1: 89fbdf2bc0762db53421f8b21eb628be
Enter H2: c3d97bc9addff51b079014ea6d329c05
The number of hex bits that are the same between H1 and H2 is: 0
The number of binary bits that are the same between H1 and H2 is: 65
[11/01/21]seed@VM:~/.../HW_3$
```

- I then proceeded to compile the run the code while inputting both H1 and H2 to be compared.
- As shown in the picture, there are zero similarities between the hashes in their hex form.
- On the other hand, there are 65 similarities in their binary form.

```
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 random_words.txt
SHA256(random_words.txt)= efd760e932df4cf24b2f2447be77c0b3061fb4e4cc6be98ceaa2d2678b9dc27f
[11/01/21]seed@VM:~/.../HW_3$ openssl dgst -sha256 random_mod.txt
SHA256(random_mod.txt)= 383ca2840b0b1ea78fdd5f59b84b2caff44f28f2bd673b1fdc8573a562dfdfd9
[11/01/21]seed@VM:~/.../HW_3$ ./Task_3
Enter H1: efd760e932df4cf24b2f2447be77c0b3061fb4e4cc6be98ceaa2d2678b9dc27f
Enter H2: 383ca2840b0b1ea78fdd5f59b84b2caff44f28f2bd673b1fdc8573a562dfdfd9
The number of hex bits that are the same between H1 and H2 is: 3
The number of binary bits that are the same between H1 and H2 is: 132
[11/01/21]seed@VM:~/.../HW_3$
```

- I then proceeded to create the sha256 hashes for the same files and compare them using my produced code.
- In their hex form, H1 and H2 has only 3 similarities while there are 132 similarities in their binary form.
- By my observations, changing only one bit changes the entire has drastically. There are no obvious similarities whatsoever.
- For md5 and sha256, the hex similarities are only 0 and 3 respectively. With such small numbers, it is proven that hash functions do an excellent job of scrambling without similarities.
- Even though md5 and sha256 has 65 and 132 similarities respectively, it needs to be known that 4 binary bits makes 1 hex bit. Thus, taking the 4:1 ratio int account, 65 and 132 aren't such big numbers after all.

Task 4: Hash Collision-Free Property

```cpp
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

string random_message(int n){
    char alphabet[52] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g',
                          'h', 'i', 'j', 'k', 'l', 'm', 'n',
                          'o', 'p', 'q', 'r', 's', 't', 'u',
                          'v', 'w', 'x', 'y', 'z',
                          'A', 'B', 'C', 'D', 'E', 'F', 'G',
                          'H', 'I', 'J', 'K', 'L', 'M', 'N',
                          'O', 'P', 'Q', 'R', 'S', 'T', 'U',
                          'V', 'W', 'X', 'Y', 'Z'};

    string message = "";
    for (int i = 0; i < n; i++){
        message = message + alphabet[rand() % 52];
    }

    return message;
}
```

- I created a random string generator in order to attempt part 1 of task 4.
- All it does is just randomly choose the chars to create a string of length n.

```
void print_digest(unsigned char md_value[], unsigned int digest_length){
    for (int i = 0; i < digest_length; i++){
        printf("%02x", md_value[i]);
    }
    printf("\n");
}
```

- I then created a print function in order to print the message digests for the “matching” messages.

```
int main(int argc, char *argv[]){
    srand(time(NULL));
    int z = 5;
    int y = 1;
    bool looping = true;
    while(looping){
        EVP_MD_CTX *mdctx;
        const EVP_MD *md = EVP_md5();
        string mess1 = random_message(10);
        string mess2 = random_message(10);
        //char mess1[message1.length() + 1] = message1.c_str();
        //char mess2[] = "Hello Wo rld\n";
        unsigned char md_value_1[EVP_MAX_MD_SIZE];
        unsigned char md_value_2[EVP_MAX_MD_SIZE];
        unsigned int md_len;

        //Digest for message 1
        mdctx = EVP_MD_CTX_create();
        EVP_DigestInit_ex(mdctx, md, NULL);
        EVP_DigestUpdate(mdctx, mess1.c_str(), strlen(mess1.c_str()));
        EVP_DigestFinal_ex(mdctx, md_value_1, &md_len);
        EVP_MD_CTX_destroy(mdctx);

        //Digest for message 2
        mdctx = EVP_MD_CTX_create();
        EVP_DigestInit_ex(mdctx, md, NULL);
        EVP_DigestUpdate(mdctx, mess2.c_str(), strlen(mess2.c_str()));
        EVP_DigestFinal_ex(mdctx, md_value_2, &md_len);
        EVP_MD_CTX_destroy(mdctx);


        //compare the first 24 bits of h1 and h2
        if(memcmp ( md_value_1, md_value_2, 3 ) == 0){
            cout << "Found the two messages!" << endl;
            printf("M1: %s \n", mess1.c_str());
            printf("M2: %s \n", mess2.c_str());
            printf("Digest 1: ");
            print_digest(md_value_1, md_len);
            printf("Digest 2: ");
            print_digest(md_value_2, md_len);
            cout << "Trials: " << y << endl;
            looping = false;
        }
        y++;
    }
}
```

- I then created my main function.
- I have two messages known as mess1 and mess2 which will be produced by the random message generator.
- I also have md_value_1 and md_value_2 which will store the message digests of mess1 and mess2.
- The hash function used is md5.
- The program then uses the memcmp function to compare the first 24 bits between the two message digests.
- If they match, then the code will exit the loop.
- If they do not match, then the code will continue looping.

```
[11/05/21]seed@VM:~/.../HW_4$ g++ Task_4.cpp -lcrypto -o digest
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: OglJEjcxbB
M2: LCxxgTCBXJ
Digest 1: ad64a8d08d431a8e5e7590e0e861c37e
Digest 2: ad64a80c82dd1c46d890b40e01945ef6
Trials: 8092488
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: apuTFDlZOL
M2: mRsEEmHhnV
Digest 1: 3518a049fc004f86f1c780674112a710
Digest 2: 3518a0da715590d5daf4eeff29a76b17
Trials: 60124963
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: hcuQDOWpCj
M2: VhwumVxRfR
Digest 1: 2557daeacd9846ec3b85f946a6bac889
Digest 2: 2557da589d1b0b3cb248c5ef8f36b121
Trials: 72746963
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: eMVFrVRdaM
M2: secgQgvlpA
Digest 1: 22e64634e3dffd318f0a47c0e60aeede
Digest 2: 22e6461e7afbf0b8e9a93114ac5011b5
Trials: 32444946
[11/05/21]seed@VM:~/.../HW_4$
```

- I then proceeded to run the code to see how long it will take to find two messages with the "same" message digests.
- After running the code multiple times, the average calculate was 43352340 trials.

```
string mess2 = "Helloworld";
```

- I then modified my code to replace mess2 with "Helloworld" in order to attempt part 2 of task 4.

```
[11/05/21]seed@VM:~/.../HW_4$ g++ Task_4.cpp -lcrypto -o digest
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: AOGqkBGgcD
M2: Helloworld
Digest 1: a165967931ddc0af2793ba3e0f0d83f9
Digest 2: a165968b0a8084a041aed89bf40d581f
Trials: 65113041
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: RjEAlqqJkR
M2: Helloworld
Digest 1: a165962da8482544487e6de25112fc56
Digest 2: a165968b0a8084a041aed89bf40d581f
Trials: 35372651
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: RrQQOjQWej
M2: Helloworld
Digest 1: a16596086af89bdae015d57d118a4111
Digest 2: a165968b0a8084a041aed89bf40d581f
Trials: 37296262
[11/05/21]seed@VM:~/.../HW_4$ ./digest
Found the two messages!
M1: ubMQIHeayH
M2: Helloworld
Digest 1: a16596726452b1340d021f9ddf1c5814
Digest 2: a165968b0a8084a041aed89bf40d581f
Trials: 16824257
[11/05/21]seed@VM:~/.../HW_4$
```

- I then proceeded to run the code multiple times and the calculated average was 38651552.75 trials.
- When comparing the two averages, it appears that finding a message has the same hash value as a given/known message's hash value using the brute-force method is easier than finding two messages with the same hash values using the brute-force method.

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

```
[11/04/21]seed@VM:~/.../HW_4$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
.....................................++++++
.....................................................................................++++++
e is 65537 (0x10001)
[11/04/21]seed@VM:~/.../HW_4$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCp7haPhATY94iWs4l7ak37UJXdyr69xwt6BwK36JFqEJkl0amW
Ep8u+32bIkExcl2DqKZMggbMiKrTQECSODQvmbN2w7DisXwNZsEcqiGg74qGNPw+
njWZreb50PtIrBS8UG9Jt2xuPqMwSWNvhkWuNzEdORQUjptGU8cxWQbHdQIDAQAB
AoGBAJJzkZoKq6raWyuWfCyTR7YKk77c11rzyt5WYG9jzr9cosDHW6LtVafGM1Ca
Sxl0eUfY98oEXhu0xZTfUUczjTT4wqrV0wucnZTJxMUbH/gTo7RFm9MF4EW95UaL
q1vI+lyWtDKaYzToLTJabnLzkSsqdfLsj8W10noXKKrqh2nBAkEA3kZsjHoIPMFT
rypplIdykAfVBYDj920ADA3lygz+VlpRZEP6a2njopna/oLcIbOOYrGqTSLZaea6
rGUF+rbA7QJBAMO2fCrgqIvpbiDg9GfzkGSWeycyN1YVIuHSrSxVMBKQ8F7FtteE
wtB1OsCZP1NUNUwacTcT24LesojZhEVYt6kCQAKa0KrUw0PzAAgNdy9GZxFJBjYK
TKGZpWZ+wzKa9GAOruV95nFbzE8bfnV9ExdSA1kku9orjmCvU4CxrmBBxAECQQCr
iez3IAZa63TNeJ5/5mmu4H588nSyDMCQMjn624fRHhyw1JbX+9NNJxMh8LAid3Jn
i92jqtIBuPO/IvfeWE5RAkA9bfliIAjyZv+qW4mgbObLqAw/RgfYg7orBtEnidwK
zNgzAIwxybQpBscfc0C7h9yfR72N3rpPFKdNSbMbzx3h
-----END RSA PRIVATE KEY-----
[11/04/21]seed@VM:~/.../HW_4$
[11/04/21]seed@VM:~/.../HW_4$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCp7haPhATY94iWs4l7ak37UJXd
yr69xwt6BwK36JFqEJkl0amWEp8u+32bIkExcl2DqKZMggbMiKrTQECSODQvmbN2
w7DisXwNZsEcqiGg74qGNPw+njWZreb50PtIrBS8UG9Jt2xuPqMwSWNvhkWuNzEd
ORQUjptGU8cxWQbHdQIDAQAB
-----END PUBLIC KEY-----
[11/04/21]seed@VM:~/.../HW_4$
```

- I created a simple 16 byte text file with using the word “four” four times since a char is exactly 1 byte.
- I then proceeded to create the RSA public and private key pairs on the terminal using the openssl commands.

```
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.008s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.011s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.009s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.010s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.009s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.009s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -encrypt -in 16_bytes.txt -pubin -inkey public.pem -out 16_encrypt.txt

real    0m0.009s
user    0m0.008s
sys     0m0.000s
```

- I then proceeded to encrypt the 16 byte message with rsa encryption and timed it as well.
- Since the operation was fast, I decided to perform the RSA encryption multiple times in order to get the average.

- **Note:** "real" is the total amount of time elapsed and we will be using that.
- The average time RSA encryption takes for a 16 byte message is about 0.009 seconds.

```
[11/04/21]seed@VM:~/.../HW_4$ xxd 16_encrypt.txt
00000000: 2b73 bbd4 7f94 de50 e669 7281 a7f7 fcd1  +s.....P.ir.....
00000010: cf9e 2f98 84d5 4fad ec14 cc05 3908 72a5  ../...O.....9.r.
00000020: 56f5 3798 202f b590 7401 2403 5aa0 cd8c  V.7. /..t.$.Z...
00000030: 8313 f5b1 3637 1be6 84df bf3b 68b4 0deb  ....67.....;h...
00000040: 1708 49d2 668b acd1 dfb2 89b6 b6f6 ac92  ..I.f...........
00000050: f0d1 bb74 8194 b080 8a06 b0fd cba2 7674  ...t..........vt
00000060: 5a9c e32d da6e d09a 2aab 1e86 2326 43be  Z..-.n..*...#&C.
00000070: e8e1 fb6c 6483 8b31 8598 a402 939e 5513  ...ld..1......U.
[11/04/21]seed@VM:~/.../HW_4$
```

- In order to ensure the 16 byte message was encrypted, I decided to read the file on the terminal.

```
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.009s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.010s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.009s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.012s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.010s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.012s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl rsautl -decrypt -in 16_encrypt.txt -inkey private.pem -out 16_decrypt.txt

real    0m0.011s
user    0m0.008s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$
```

- Just like with the encryption, I decrypted multiple times.
- The average time it takes for RSA decryption on a 16 byte message is about 0.010 seconds.

```
[11/04/21]seed@VM:~/.../HW_4$ cat 16_decrypt.txt
fourfourfourfour
```

- In order to ensure that the 16 byte message was decrypted properly, I decided to read the file on the terminal.

```
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.007s
user    0m0.000s
sys     0m0.004s
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.008s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.008s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.008s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.009s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$ time openssl enc -aes-128-ecb -e  -in 16_bytes.txt -out 16_aes_dec.txt -K 0011223344556677888
9aabbccddeeff

real    0m0.008s
user    0m0.004s
sys     0m0.000s
[11/04/21]seed@VM:~/.../HW_4$
```

- I then proceeded to encrypt the 16 byte message using AES-128-ecb encryption.
- **Note:** The key used is the same key that was used for the previous homework assignment.
- The average time for AES-128-ecb encryption for a 16 byte message is about 0.008 seconds.

```
[11/04/21]seed@VM:~/.../HW_4$ xxd 16_aes_dec.txt
00000000: 840d d671 f83b e124 4b9e d199 5f31 d94a  ...q.;.$K..._1.J
00000010: 81dd a943 68eb 2f27 8e4b 1c86 1a87 dd09  ...Ch./'.K......
```

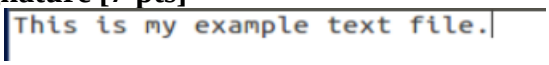
- In order to ensure that the 16 byte message was actually encrypted, I read the file on the terminal.

```
Doing 1024 bit private rsa's for 10s: 12513 1024 bit private RSA's in 9.95s
Doing 1024 bit public rsa's for 10s: 244743 1024 bit public RSA's in 9.92s
```

```
Doing aes-128 cbc for 3s on 16 size blocks: 16778400 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 64 size blocks: 4983974 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 256 size blocks: 1298296 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 803081 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 8192 size blocks: 102205 aes-128 cbc's in 2.99s
```

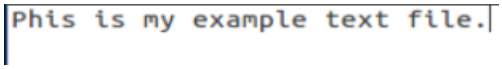
- I then proceeded to run the openssl speed command for both aes and rsa encryptions.
- When observing these results, there is a drastic difference between my results and the speed command results.
- If my assumption is correct, the reason for such a drastic difference the size of the text files being used.
- The text file used for my encryptions is only a mere 16 bytes which is not a big file at all.
- For the speed command, it could be working with a much larger text file; if this is the case, it would explain why the speed command has larger results.

Task 6: Create Digital Signature [7 pts]




```
[11/04/21]seed@VM:~/.../HW_4$ ls
16_aes_dec.txt  16_decrypt.txt  16_enc.txt   general_enc.cpp          public.pem        Task_3.cpp
16_aes_enc.txt  16_dec.txt      digest       peda-session-Task_3.txt  random_mod.txt    Task_4.cpp
16_bytes.txt    16_encrypt.txt  example.txt  private.pem              random_words.txt
[11/04/21]seed@VM:~/.../HW_4$ openssl dgst -sha256 -sign private.pem -out example.sha256 example.txt
[11/04/21]seed@VM:~/.../HW_4$ openssl dgst -sha256 -verify public.pem -out example.sha256 example.txt
No signature to verify: use the -signature option
[11/04/21]seed@VM:~/.../HW_4$ openssl dgst -sha256 -verify public.pem -out -signature example.sha256 example.txt
Terminator
ure to verify: use the -signature option
[11/04/21]seed@VM:~/.../HW_4$ openssl dgst -sha256 -verify public.pem  -signature example.sha256 example.txt
Verified OK
[11/04/21]seed@VM:~/.../HW_4$
```

- For this task, I created a simple example.txt file.
- **Note:** I am using the same private and public keys that were generated in task 5.
- I then ran openssl dgst command to create the sha256 message digest and signed the example.txt file using the private.pem file.
- I then proceeded to run openssl dgst command to verify the example.txt file using the public key.
- As a result, I got “Verified OK”.
- **Note:** I messed up a few times when trying to verify the file as I misunderstood the syntax a few times.



- I then modified the example.txt file by replacing the first character with a “P” instead of a “T”.

```
[11/04/21]seed@VM:~/.../HW_4$ openssl dgst -sha256 -verify public.pem  -signature example.sha256 example.txt
Verification Failure
```

- I then proceeded to run the openssl dgst command to verify the example.txt file again using the same public key.
- Since I modified the file, I got “Verification Failure”.

Digital signatures are useful because they help with the authentication process. When a message (with its signature) passes the verification process, then the receiver can be confident that the message is coming from the person they are expecting.