References
https://stackoverflow.com/questions/40769915/generate-random-letters-in-c - Task 4
https://www.openssl.org/docs/manmaster/man3/EVP_DigestInit.html - Task 4
https://rietta.com/blog/openssl-generating-rsa-key-from-command/ - Task 5
https://jumpnowtek.com/security/Code-signing-with-openssl.html - Task 6

HW4

Paper and Pencil

- Q1:
    - This is not a good function because there would be many collisions. Since this function is just the XOR of all the chunks that result in a 128-bit output, it doesn't have strong collision resistance. There would only need to be computations for 2^128 different possibilities to get the resultant 128-bit output which is pretty easy to calculate if you are using a program.
- Q2
    - ~x
        - x should be random because this function just takes the bitwise complement of the original input. So if x is random the output is random
    - x⊕y
        - For this condition, only one variable needs to be random for the output to be random. So it could be either x or y. This is true because XOR is dependent on both variables, no outputs can be independently decided.
    - x∨y
        - Both variables need to be random for X OR Y because if one of the variables is always set to one, the output will always be one. You can manipulate one variable to make the output not random.
    - x∧y
        - Both variables need to be random for X OR Y because if one of the variables is always set to 0, the output will always be 0. You can manipulate one variable to make the output not random.
    - (x∧y) ∨(~x∧z)
        - Any two variables can be random for the output to be random because neither of these should have an output of 1. So there must be two random variables.
    - (x∧y) ∨(x∧z) ∨(y∧z)
        - Since each variable is OR'd together at least 2 variables must be random because if this is the case there will be at least one random variable in each set of parenthesis.
    - x⊕y⊕z

    - For this condition, only one variable needs to be random for the output to be random. So it could be either x or y or z. This is true because XOR is dependent on all variables, no outputs can be independently decided.
  - y⊕(~x ∧ z)
    - Either y has to be random, or both x and z have to be random for there to be a random output. This uses the same logic from examples 2 and 4, it just combines them.
- Q3
  - This prevents the man in the middle attack because the attacker should be able the change the intercepted message without anybody noticing the message tampered with. Since the Diffie-Hellman value was encrypted, the attacker will not be able to decrypt the value since they only have access to the public key. Without the Diffie-hellman values, they cannot compute the shared secrets between the party.
- Q4
  - Fred sees $m_1^d mod\ n$and $m_2^d mod\ n$. So we know that $(m_1^d)^i mod\ n\ =\ (m_1^i)^d mod\ n$ where i is an integer.
  - To calculate m_1^j
    - $(m_1^d)^j mod\ n\ =\ (m_1^j)^d mod\ n$ where j is an arbitrary number
  - To calculate m1^(-1)
    - $(m_1^d)^{-1} mod\ n\ =\ (m_1^{-1})^d mod\ n$
  - To calculate m1m2
    - $(m_1 m_2)^d mod\ n\ =\ (m_1)^d mod\ n\ *\ (m_2)^d mod\ n$
  - To calculate $(m_1^j m_2^k)\ mod\ n$
    - $(m_1^j m_2^k)^d mod\ n\ =\ (m_1^j)^d mod\ n\ *\ (m_2^k)^d mod\ n$

Task 1:

- ```
  plaintext.txt
  [11/05/21]seed@VM:.../sf_shared_folder4$ man openssl
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 plainte
  xt.txt
  MD5(plaintext.txt)= 27a511252f0b31a5c94ebfb1c4403d95
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha1 plaint
  ext.txt
  SHA1(plaintext.txt)= 8d464b6469b36a0557790bc7c373bd533f7a3d95
  [11/05/21]seed@VM:.../sf_shared_folder4$ man openssl
  [11/05/21]seed@VM:.../sf_shared_folder4$
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -rmd160 plai
  ntext.txt
  RIPEMD160(plaintext.txt)= 567f77c1540618f740ff05112b5272b6e13db84b
  [11/05/21]seed@VM:.../sf_shared_folder4$
  ```

The purpose of this task was to play around with different message digest types. The outputs for each of the different message digest types were different lengths. MD5 was the shortest, sha1 was in the middle, and rmd160 was the longest.

Task 2:

- ```
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 -hmac "
  abcdefg" plaintext.txt
  HMAC-MD5(plaintext.txt)= 3bd994a301e7a28066157599b9b407f4
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 -hmac "
  abcdefg123456" plaintext.txt
  HMAC-MD5(plaintext.txt)= 7f928f24cee91a34aeafc574254f97c9
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 -hmac "
  abc12" plaintext.txt
  HMAC-MD5(plaintext.txt)= d1f9a63e3bdb3afce8e9880d11f11c45
  [11/05/21]seed@VM:.../sf_shared_folder4$
  ```
- ```
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha1 -hmac
  "abcdefg" plaintext.txt
  HMAC-SHA1(plaintext.txt)= 70613619688f04c3bbac260279bd36970da72d42
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha1 -hmac
  "abcdefg123456" plaintext.txt
  HMAC-SHA1(plaintext.txt)= f8cf9c448a11d125604012e9235f22488f791381
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha1 -hmac
  "abc12" plaintext.txt
  HMAC-SHA1(plaintext.txt)= 738219d2e8504a0f189b6e3ac5fb2bdc49e3476b
  [11/05/21]seed@VM:.../sf_shared_folder4$
  ```
  ```

- ```
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -hma
  c "abcdefg" plaintext.txt
  HMAC-SHA256(plaintext.txt)= c0c32980bf0489dc066b64b3b96abf12fc38d3
  518493bfb23135d6c126220fb8
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -hma
  c "abcdefg123456" plaintext.txt
  HMAC-SHA256(plaintext.txt)= 894b2ccff86a8afd817bb2bbc3b54a26d82a43
  6c88a5d091e36f052f80c99adf
  [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -hma
  c "abc12" plaintext.txt
  HMAC-SHA256(plaintext.txt)= ca66d7bf6faf496c6d7ec7d512b166e47d4d81
  011a8879a1482d24a7f320eadc
  [11/05/21]seed@VM:.../sf_shared_folder4$
  ```

For task two I used keys of different lengths to test the 3 different message digest types (md5, sha1, sha256). From my observations, it is evident that changing the key lengths did not affect the length keyed hash. This occurs because if the key size is shorter or longer than the block size, it will shift the size of the key to match that of the block size.

Task 3:

- MD5 - 32
  - ```
    [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 task3.t
    xt
    MD5(task3.txt)= 5100195a424ff69391c077a7b3d6c4d3
    [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -md5 task3.t
    xt
    MD5(task3.txt)= a621e0f6ce07412d79cf7b436ce719d9
    [11/05/21]seed@VM:.../sf_shared_folder4$
    ```
- SHA256
  - ```
    [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 task
    3.txt
    SHA256(task3.txt)= 1caf185782ac2e54e1323685bafadc3b3872e176a2d3a6f
    0fa9b00b019ab2d54
    [11/05/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 task
    3.txt
    SHA256(task3.txt)= 528c9dd0cc597cd174a89b5baf35593a91d58f7a33e2842
    e07b1c586d80bba3e
    [11/05/21]seed@VM:.../sf_shared_folder4$
    ```
- Code Output for Calculating Similar bits
    ```

```
[11/05/21]seed@VM:.../sf_shared_folder4$ gcc -o task3 task3.c
[11/05/21]seed@VM:.../sf_shared_folder4$ ./task3
Number of similar bits for md5 : 60
Number of similar bits for sha256 : 132
[11/05/21]seed@VM:.../sf_shared_folder4$
```

```
char* convertHexToBinary(char h){
    char* binary = "";

    if(h =='0'){
        binary = "0000";
    }else if ((h =='1')){
        binary = "0001";
    }else if ((h =='2')){
        binary = "0010";
    }else if ((h =='3')){
        binary = "0011";
    }else if ((h =='4')){
        binary = "0100";
    }else if ((h =='5')){
        binary = "0101";
    }else if ((h =='6')){
        binary = "0110";
    }else if ((h =='7')){
        binary = "0111";
    }else if ((h =='8')){
        binary = "1000";
    }else if ((h =='9')){
        binary = "1001";
    }else if ((h =='a')){
        binary = "1010";
    }else if ((h =='b')){
        binary = "1011";
    }else if ((h =='c')){
        binary = "1100";
    }else if ((h =='d')){
        binary = "1101";
    }else if ((h =='e')){
        binary = "1110";
    }else if ((h =='f')){
        binary = "1111";
    }
    return binary;
}
```

```
int findSimilarBits(char h1[], char h2[], int size){
    char* b1;
    char* b2;
    int count = 0;

    for(int i=0; i<size; ++i){
        b1 = convertHexToBinary(h1[i]);
        b2 = convertHexToBinary(h2[i]);
        for(int j=0; j<4; ++j){
            if(b1[j] == b2[j]){
                count++;
            }
        }
    }

    return count;
}
```

```
int main(int argc, char **argv){

    char h1[] = "5100195a424ff69391c077a7b3d6c4d3";
    char h2[] = "a621e0f6ce07412d79cf7b436ce719d9";

    char h3[] = "1caf185782ac2e54e1323685bafadc3b3872e176a2d3a6f0fa9b00b019ab2d54";
    char h4[] = "528c9dd0cc597cd174a89b5baf35593a91d58f7a33e2842e07b1c586d80bba3e";

    printf("Number of similar bits for md5 : %i\n", findSimilarBits(h1, h2, sizeof(h1)-1));
    printf("Number of similar bits for sha256 : %i\n", findSimilarBits(h3, h4, sizeof(h3)-1));

    return 0;

}
```

To complete this task I create a hex to binary function to convert a single hex character to its binary equivalent. This function was used in my other function called FindSimilarBits(). That was used to compare the bits of the two hash keys. If the bits were similar I would increase the count by 1. I simply printed the count of similar bits in the main function.

Overall H1 and H2 were not that similar after 1 bit was flipped in the file. There were a few characters that matched but most of them were different.

Task 4:
The goal of task 4 was to use the brute force method to break collision (free) properties.
Code:

```
int isDgstEqual(unsigned char dgst1[EVP_MAX_MD_SIZE], unsigned char dgst2[EVP_MAX_MD_SIZE]){
    for(int i=0; i<3; ++i){
        if(dgst1[i] != dgst2[i]){
            return 0;
        }
    }
    return 1;
}

int isMsgEqual(char* msg1, char* msg2){
    for(int i=0; i<sizeof(msg1); ++i){
        if(msg1[i] != msg2[i]){
            return 0;
        }
    }
    return 1;
}

/*Referenced from stack overflow*/
char* getRandomMsg(char msg[20]){
    char * str = "abcdefghijklmnopqrstuvwxyz";
    for(int i=0; i<19; ++i){
        msg[i] = str[rand()%26];
    }

    msg[19] = '\0';
    return msg;
}
```

The functions above are very important. The getRandomMsg generates a random msg for me. The isDgstEqual equal function just checks whether the message digests for both msg1 and msg2 are equal, but only the first 3 characters. If it is it returns a 1 if not it returns a 0. isMsgEqual checks to see if the messages themselves are equal.

```
void getHash(char* hash, char* msg,  unsigned char md_value[EVP_MAX_MD_SIZE]){

    EVP_MD_CTX *mdctx;
    const EVP_MD *md;
    int md_len;

    OpenSSL_add_all_digests();

    md = EVP_get_digestbyname(hash);
    if (md == NULL) {
        printf("Unknown message digest %s\n", hash);
        exit(1);
    }

     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_cleanup(mdctx);
}
```

This getHash function was created mostly using the reference given from the homework document. The only things I changed were what parameters were being passed through. I also deleted one of the EVP_DigestUpdate commands because I wanted each msg to have its own digest.

```c
int main(int argc, char **argv){

    char* hash = "md5";
    srand((int)time(0));

    //********************************Case 1**************************************;
    /* Both messages are random */

    int totalTries = 0;
    int flag = 1;

    char msg1[20], msg2[20];

    unsigned char dgst1[EVP_MAX_MD_SIZE], dgst2[EVP_MAX_MD_SIZE];

    getHash(hash, getRandomMsg(msg1), dgst1);
    getHash(hash, getRandomMsg(msg2), dgst2);

    while(flag==1){

        getHash(hash, getRandomMsg(msg1), dgst1);
        getHash(hash, getRandomMsg(msg2), dgst2);

        totalTries++;

        if(isDgstEqual(dgst1, dgst2)==1 && isMsgEqual(msg1, msg2)==0){
            flag = 0;
        }

    }

    printf("Same Hash was found! It took %d tries \n", totalTries);
```

```c
    //********************************Case 2**************************************
    /* One message is random, and one hash is known*/

    int totalTriesPart2 = 0;
    int flagPart2 = 1;

    char msg3[4], msg4[4];

    unsigned char dgst3[EVP_MAX_MD_SIZE], dgst4[EVP_MAX_MD_SIZE];

    getHash(hash, getRandomMsg(msg3), dgst3);
    getHash(hash, getRandomMsg(msg4), dgst4);

    while(flagPart2==1){

        getHash(hash, getRandomMsg(msg4), dgst4);
        totalTriesPart2++;

        if((isDgstEqual(dgst3, dgst4)==1) && isMsgEqual(msg3, msg4)==0 ){
            flagPart2 = 0;
            printf("M3: %s, M4: %s \n", msg3, msg4);
        }
    }

    printf("Message with same hash was found! It took %d tries \n", totalTriesPart2);

    exit(0);

}
```

For the first case, I basically created the msg[]'s and dgst[]'s to compare. I called the getHash function for both msg1 and msg2 and passed in the randomized messages and the digests. After this, it was pretty simple to figure out if the hashes were the same, all I had to do was call my isDgstEqual function to check if they were equal, if they were I leave the while loop and print the number of tries it took to get there.

For the second case, I generate msg1 once and called getHash() once for that msg. This signified the given/known message and hash. After that, I pretty much did the same thing as for case one, except I only randomized msg2 every time instead of randomizing both messages.

```
[11/07/21]seed@VM:.../sf_shared_folder4$ make
gcc -I/usr/local/ssl/include/ -L/usr/local/ssl/lib/ -o task4
task4.c -lcrypto
[11/07/21]seed@VM:.../sf_shared_folder4$ ./task4
Same Hash was found! It took 45350 tries
Message with same hash was found! It took 6784754 tries
[11/07/21]seed@VM:.../sf_shared_folder4$ ./task4
Same Hash was found! It took 65078 tries
Message with same hash was found! It took 19866479 tries
[11/07/21]seed@VM:.../sf_shared_folder4$ ./task4
Same Hash was found! It took 115937 tries
Message with same hash was found! It took 22312910 tries
[11/07/21]seed@VM:.../sf_shared_folder4$ ./task4
Same Hash was found! It took 153905 tries
Message with same hash was found! It took 26030862 tries
[11/07/21]seed@VM:.../sf_shared_folder4$ ./task4
Same Hash was found! It took 110836 tries
Message with same hash was found! It took 12670098 tries
[11/07/21]seed@VM:.../sf_shared_folder4$
```

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
Case 1	45350	65078	115937	153905	110836	98221.2
Case 2	6784754	19866479	22312910	26030862	12679908	17533020.6

From observing the outputs above it is evident that case 1 of finding the number of tries it takes to get two random msgs with the same hash is easier to find than case 2. If you look at the average number of tries from the table above, case 1 is much smaller than case 2. Therefore case 1 is easier to break than case 2 using brute force.

Task 5:

- Creating public and private keys
  - ```
    [11/06/21]seed@VM:.../sf_shared_folder4$ openssl genrsa -des3 -out
     private.pem 1024
    Generating RSA private key, 1024 bit long modulus
    ..........++++++
    ..........++++++
    e is 65537 (0x10001)
    Enter pass phrase for private.pem:
    Verifying - Enter pass phrase for private.pem:
    [11/06/21]seed@VM:.../sf_shared_folder4$ openssl rsa -in private.p
    em -outform PEM -pubout -out public.pem
    Enter pass phrase for private.pem:
    writing RSA key
    ```
    ```

- Timings for Encrypting message with the public key
    - ```
      [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl rsautl -encr
      ypt -in message.txt -inkey public.pem -pubin -out message_enc.txt

      real    0m0.102s
      user    0m0.008s
      sys     0m0.016s
      [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl rsautl -encr
      ypt -in message.txt -inkey public.pem -pubin -out message_enc.txt

      real    0m0.094s
      user    0m0.004s
      sys     0m0.028s
      [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl rsautl -encr
      ypt -in message.txt -inkey public.pem -pubin -out message_enc.txt

      real    0m0.117s
      user    0m0.004s
      sys     0m0.028s
      [11/06/21]seed@VM:.../sf_shared_folder4$
      ```
- Timing for Decrypting message with the private key
    - ```
      ypt -in message_enc.txt -inkey private.pem  -out message_enc_decry
      pted.txt
      Enter pass phrase for private.pem:

      real    0m2.661s
      user    0m0.008s
      sys     0m0.032s
      [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl rsautl -decr
      ypt -in message_enc.txt -inkey private.pem  -out message_enc_decry
      pted.txt
      Enter pass phrase for private.pem:

      real    0m2.095s
      user    0m0.008s
      sys     0m0.024s
      [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl rsautl -decr
      ypt -in message_enc.txt -inkey private.pem  -out message_enc_decry
      pted.txt
      Enter pass phrase for private.pem:

      real    0m2.343s
      user    0m0.008s
      sys     0m0.024s
      [11/06/21]seed@VM:.../sf_shared_folder4$
      ```
- Timing for Encrypting message with aes

- ```
  -in message.txt -out message_enc_aes.txt
  enter aes-128-cbc encryption password:
  Verifying - enter aes-128-cbc encryption password:

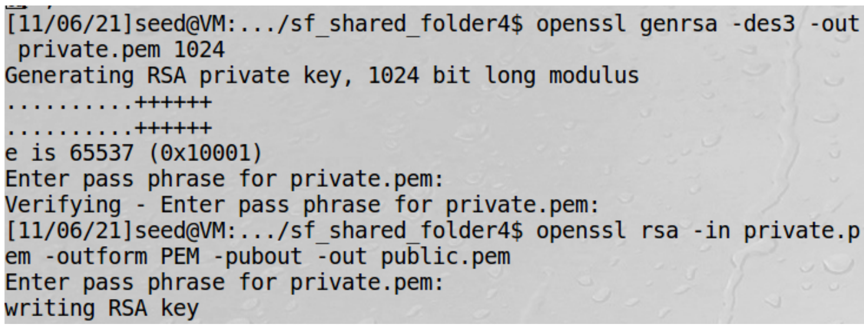
  real    0m4.892s
  user    0m0.008s
  sys     0m0.016s
  [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl aes-128-cbc
  -in message.txt -out message_enc_aes.txt
  enter aes-128-cbc encryption password:
  Verifying - enter aes-128-cbc encryption password:

  real    0m4.488s
  user    0m0.004s
  sys     0m0.024s
  [11/06/21]seed@VM:.../sf_shared_folder4$ time openssl aes-128-cbc
  -in message.txt -out message_enc_aes.txt
  enter aes-128-cbc encryption password:
  Verifying - enter aes-128-cbc encryption password:

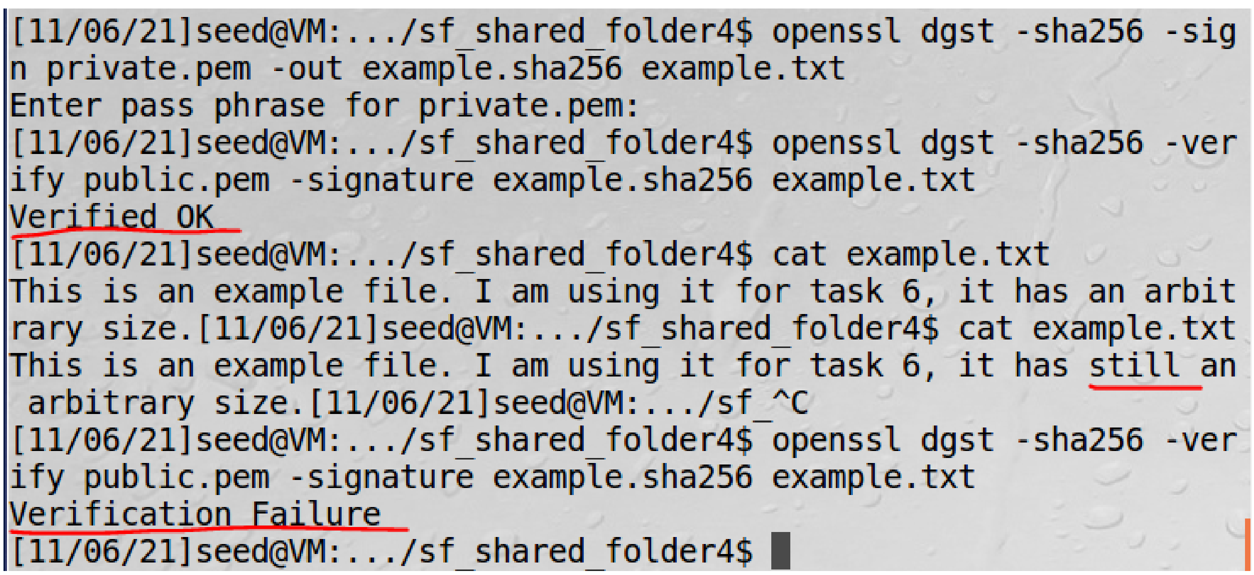
  real    0m3.660s
  user    0m0.008s
  sys     0m0.020s
  [11/06/21]seed@VM:.../sf_shared_folder4$
  ```

- Running openssl speed commands for rsa and aes
  - Rsa
    - ```
      [11/06/21]seed@VM:.../sf_shared_folder4$ openssl speed rsa
      Doing 512 bit private rsa's for 10s: 25642 512 bit private RSA's i
      n 3.48s
      Doing 512 bit public rsa's for 10s: 295070 512 bit public RSA's in
       3.52s
      Doing 1024 bit private rsa's for 10s: 4373 1024 bit private RSA's
      in 3.49s
      Doing 1024 bit public rsa's for 10s: 90977 1024 bit public RSA's i
      n 3.51s
      Doing 2048 bit private rsa's for 10s: 647 2048 bit private RSA's i
      n 3.49s
      Doing 2048 bit public rsa's for 10s: 23977 2048 bit public RSA's i
      n 3.39s
      Doing 4096 bit private rsa's for 10s: 93 4096 bit private RSA's in
       3.51s
      Doing 4096 bit public rsa's for 10s: 6302 4096 bit public RSA's in
       3.54s
      OpenSSL 1.0.2g  1 Mar 2016
      built on: reproducible build, date unspecified
      options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial)
      blowfish(idx)
      compiler: cc -I. -I.. -I../include  -fPIC -DOPENSSL_PIC -DOPENSSL_
      THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -g -O2
      -fstack-protector-strong -Wformat -Werror=format-security -Wdate-t
      ```

- ```
  ime -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,
  --noexecstack -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE
  2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_
  ASM -DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVPAES_ASM -DWH
  IRLPOOL_ASM -DGHASH_ASM
                    sign    verify    sign/s verify/s
  rsa  512 bits 0.000136s 0.000012s   7368.4  83826.7
  rsa 1024 bits 0.000798s 0.000039s   1253.0  25919.4
  rsa 2048 bits 0.005394s 0.000141s    185.4   7072.9
  rsa 4096 bits 0.037742s 0.000562s     26.5   1780.2
  ```
- Aes
  - ```
    [11/06/21]seed@VM:.../sf_shared_folder4$ openssl speed aes
    Doing aes-128 cbc for 3s on 16 size blocks: 6760342 aes-128 cbc's
    in 1.08s
    Doing aes-128 cbc for 3s on 64 size blocks: 1834339 aes-128 cbc's
    in 1.04s
    Doing aes-128 cbc for 3s on 256 size blocks: 478473 aes-128 cbc's
    in 1.10s
    Doing aes-128 cbc for 3s on 1024 size blocks: 297560 aes-128 cbc's
     in 1.08s
    Doing aes-128 cbc for 3s on 8192 size blocks: 34393 aes-128 cbc's
    in 1.03s
    Doing aes-192 cbc for 3s on 16 size blocks: 5416659 aes-192 cbc's
    in 1.05s
    Doing aes-192 cbc for 3s on 64 size blocks: 1463240 aes-192 cbc's
    in 1.06s
    Doing aes-192 cbc for 3s on 256 size blocks: 395357 aes-192 cbc's
    in 1.02s
    Doing aes-192 cbc for 3s on 1024 size blocks: 230561 aes-192 cbc's
     in 1.02s
    Doing aes-192 cbc for 3s on 8192 size blocks: 31386 aes-192 cbc's
    in 1.06s
    Doing aes-256 cbc for 3s on 16 size blocks: 4635922 aes-256 cbc's
    in 1.05s
    ```
  - ```
    OpenSSL 1.0.2g  1 Mar 2016
    built on: reproducible build, date unspecified
    options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial)
    blowfish(idx)
    compiler: cc -I. -I.. -I../include  -fPIC -DOPENSSL_PIC -DOPENSSL_
    THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -g -O2
    -fstack-protector-strong -Wformat -Werror=format-security -Wdate-t
    ime -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,
    --noexecstack -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE
    2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_
    ASM -DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVPAES_ASM -DWH
    IRLPOOL_ASM -DGHASH_ASM
    The 'numbers' are in 1000s of bytes per second processed.
    type             16 bytes     64 bytes    256 bytes   1024 bytes
     8192 bytes
    aes-128 cbc     100153.21k   112882.40k   111353.72k   282130.96k
      273541.22k
    aes-192 cbc      82539.57k    88346.57k    99226.85k   231465.16k
      242560.48k
    aes-256 cbc      70642.62k    77926.09k    80842.79k   212424.15k
      203823.26k
    ```
    ```

The purpose of task 5 was to study the performance of public-key algorithms. For each operation, I ran it 3 times to make sure I was getting accurate results back. For encrypting the message with a public key, that looked be a fast command with an average of 0.104 s. Decrypting with a private key took an average of 2.366 seconds. Encrypting using aes-128-cbc took an average of 4.35 seconds. So it looks like using the public/private key method is faster than using aes. After running the speed commands the RSA's averaged around 3.5 seconds for a block of aunty size and for AES 1.03 seconds. So they were not particularly similar for me.

Task 6:

- 

```
[11/06/21]seed@VM:.../sf_shared_folder4$ openssl genrsa -des3 -out
 private.pem 1024
Generating RSA private key, 1024 bit long modulus
.........++++++
.........++++++
e is 65537 (0x10001)
Enter pass phrase for private.pem:
Verifying - Enter pass phrase for private.pem:
[11/06/21]seed@VM:.../sf_shared_folder4$ openssl rsa -in private.p
em -outform PEM -pubout -out public.pem
Enter pass phrase for private.pem:
writing RSA key
```

- 

```
[11/06/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -sig
n private.pem -out example.sha256 example.txt
Enter pass phrase for private.pem:
[11/06/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -ver
ify public.pem -signature example.sha256 example.txt
Verified OK
[11/06/21]seed@VM:.../sf_shared_folder4$ cat example.txt
This is an example file. I am using it for task 6, it has an arbit
rary size.[11/06/21]seed@VM:.../sf_shared_folder4$ cat example.txt
This is an example file. I am using it for task 6, it has still an
 arbitrary size.[11/06/21]seed@VM:.../sf_^C
[11/06/21]seed@VM:.../sf_shared_folder4$ openssl dgst -sha256 -ver
ify public.pem -signature example.sha256 example.txt
Verification Failure
[11/06/21]seed@VM:.../sf_shared_folder4$
```

From the picture about you can see that the first command I ran, signed the sha256 hash and saved the output in a file called example.sha256. The next command I ran verified the digital signature for the example.sha256 file.

After running the command with the verify tag. I noticed the result of changing the file. As you can see from the picture above running the verify command the first time resulted in the "Verification OK" output, but after modifying the file by adding the word "still", when I ran the verify command, it output "Verification Failure." This shows that signing and verifying files is good because it can potentially prevent man-in-the-middle attacks since it alerts you of the file has been changed because the hashes don’t match up. The reason it does this is that the modified file does not match with the public key and signature.