Homework 5
CSCE-465-500
November 21, 2018
Joseph Martinsen

Paper-and-Pencil Problems

• 5.3. (8 pts)

3. In §5.1 *Introduction* we discuss the devious secretary Bob having an automatic means of generating many messages that Alice would sign, and many messages that Bob would like to send. By the birthday problem, by the time Bob has tried a total of 2^{32} messages, he will probably have found two with the same message digest. The problem is, both may be of the same type, which would not do him any good. How many messages must Bob try before it is probable that he'll have messages with matching digests, and that the messages will be of opposite types?

$2^{32} \cdot 2 = 2^{33}$

Bob will have to try 2^{33} messages

• 5.4. (8 pts)

4. In §5.2.4.2 *Hashing Large Messages*, we described a hash algorithm in which a constant was successively encrypted with blocks of the message. We showed that you could find two messages with the same hash value in about 2^{32} operations. So we suggested doubling the hash size by using the message twice, first in forward order to make up the first half of the hash, and then in reverse order for the second half of the hash. Assuming a 64-bit encryption block, how could you find two messages with the same hash value in about 2^{32} iterations? Hint: consider blockwise palindromic messages.

An attack can be conducted by targeting the hash in the same iteration. Compute a 32 bit hash of the forward key, flip this key for the 2^{nd} part of the hash and compare

• 5.14. (8 pts)

14. 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 instance, y can always be 51, and yet the output will be random if x is random. For the following functions, find sufficient conditions for x, y, and z under which the output will be random:

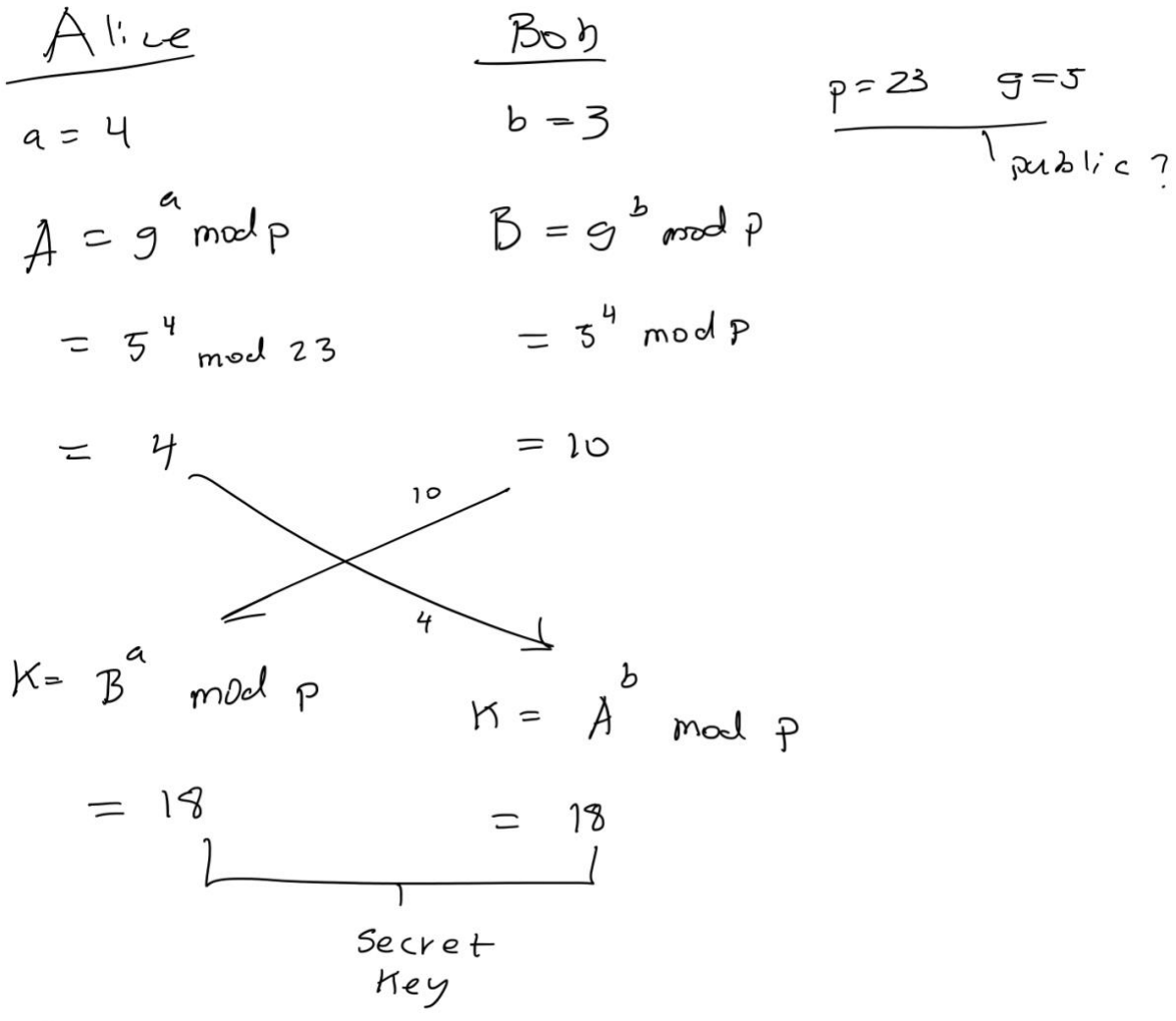
$\sim x$
$x \oplus y$
$x \vee y$
$x \wedge y$
$(x \wedge y) \vee (\sim x \wedge z)$ [the selection function]
$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ [the majority function]
$x \oplus y \oplus z$
$y \oplus (x \vee -z)$

- 6.2. (8 pts)

a. x can be random. y and z can be anything
b. x/y can be random if they differ one bit. Z can be anything
c. X/y can be random and differ by at least 1 bit. Z can be anything
d. x/y can be random and must have at least one 1. Z can be anything.

e. Either x/y are different or ~x/z are different
f. Much like (b), at least 2 out of x,y,z have to differ by at least 1 bit
g. X or ~z should be different and the result should differ by at least 1 bit with y.

2. In section §6.4.2 *Defenses Against Man-in-the-Middle Attack*, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

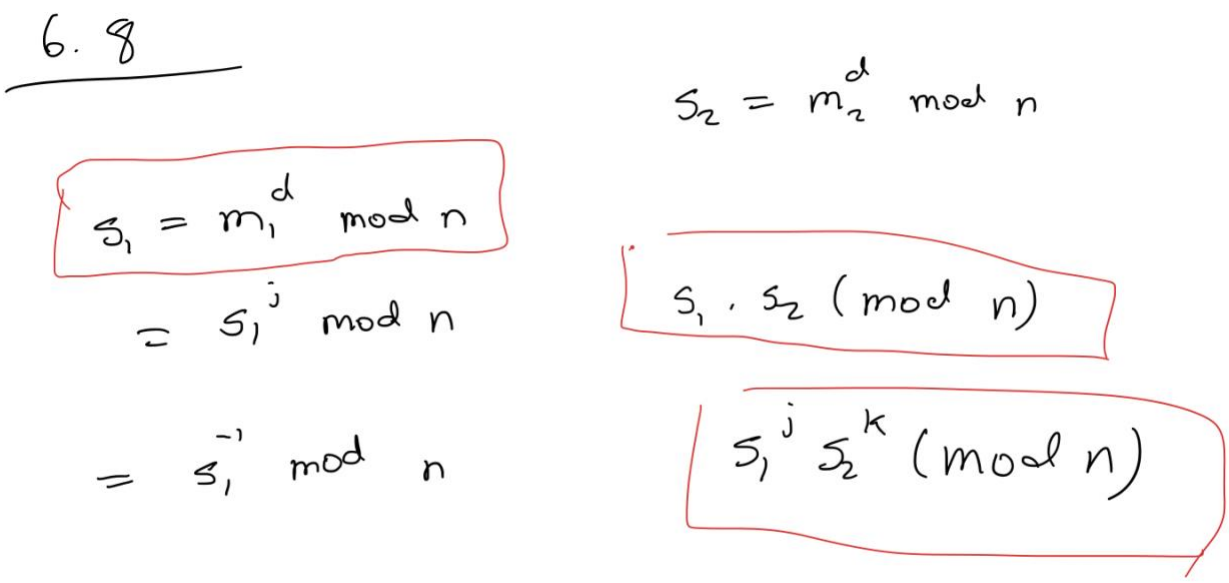


Regular Diffie-Hellman is a symmetric key exchange protocol that is vulnerable to man in the middle. Encrypting the value means that only the private keys can decrypt the key. In a man in the middle attack, the attacker Eve will not have the ability decrypt this value and thus will not be able to decrypt the secret.

• 6.8. (8 pts)

8.

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), $m_1^{-1} \bmod n$, $m_1 \cdot m_2 \bmod n$, and in general $m_1^j \cdot m_2^k \bmod n$ (for arbitrary integers j and k)?



Task 1: Generating Message Digest and MAC

I created the script below to run three different algorithms of a plain.txt file that contained some song lyrics. The script run md5, sha1 and sha256 dgst on the file.

```
#!/usr/bin/env bash

filename="plain.txt"

for dgsttype in -md5 -sha1 -sha256; do
   openssl dgst $dgsttype $filename
done
```

Below are the results of running this script.

```
✔ …/hw5/task1  master ● ?
└─ ./run.sh
MD5(plain.txt)= e94cce14060e4d6eb729b77cd0138d4a
SHA1(plain.txt)= c44e3b8651b4dfbfe16ccb1297e89bf055772b48
SHA256(plain.txt)= 980717c03ccf8446736c0b89ecbcf2249f0ee20276c59a58a9c8509e2d3170fe
```

Observations

From running these three different algorithms, it can be seen that the hashes are of different lengths. MD5 consisted of 32 characters, SHA1 was 40 characters and SHA256 had 64 characters.

Task 2: Keyed Hash and HMAC

I created the script below to generate keys of different lengths using HMAC-MD5, HMC-SHA256 and HMAC-SHA1 for the same plain.txt file I used in task 1.

```
#!/usr/bin/env bash

filename="plain.txt"

for keyLen in 3 15 21 32; do
   key=$(openssl rand -base64 $keyLen)
   echo "key: $key"
   for dgsttype in -md5 -sha1 -sha256; do
      openssl dgst $dgsttype -hmac $key $filename
   done
   echo ''
done
```

Below are the results.

```
✔ …/hw5/task2  master ● ?
./run.sh
key: zfl9
HMAC-MD5(plain.txt)= 84e4f873911c78c6d69d5108d13f43cb
HMAC-SHA1(plain.txt)= dd251f5cc8dcee81ca227153162ef883bd39fd84
HMAC-SHA256(plain.txt)= c91f1d9c678c1df88707552af5f148ff14cb3a6bb2b3e2d04f4fda7f2f1be9a1

key: 6HRfvq0LpwBli2KFvPDx
HMAC-MD5(plain.txt)= 01edee207b4742083820bdb8d53a1541
HMAC-SHA1(plain.txt)= aa65d0b3295337b79ff9463a4bfd3845e5ffd611
HMAC-SHA256(plain.txt)= 60e4d1af686af8fabaffdf76d9c2d7140d272cccabdaae26a6e4461efdddc287

key: zYQoor3c1jtgr2NqFgGIhgv549x4
HMAC-MD5(plain.txt)= e79caf1855e42688bfe92591d7729f95
HMAC-SHA1(plain.txt)= 44fb30bd0131fe0ed3a1d864dcd667e5e4ebae3d
HMAC-SHA256(plain.txt)= bdfae0ef3ea660875b4b16e8fcbe0fc50b48ed2070b2024dd66c2229701ca476

key: 4cBV51g4yk2DkDRk2cX0pN19bQiRN2M8Ur3WBy71P8o=
HMAC-MD5(plain.txt)= 6d392afffa7c9d87faf10b2484b68eea
HMAC-SHA1(plain.txt)= 7d27513371b3d8cd08c4cda16e28225524a5cefb
HMAC-SHA256(plain.txt)= 4103bcb27f6edcc4481d8600e4669af145fd547623c2ed1d30031b089f710a4c
```

Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

For best results, it seems that the key size should be similar in length to the hash although it does not seem required. Zeros are padded if the key is smaller. They should result in being the same size in order to perform a proper XOR run.

Task 3: The Randomness of One-way Hash

Basic Script to run md5 and sha256 on the file

```
filename="plain.txt"

for dgsttype in -md5 -sha256; do
   openssl dgst $dgsttype $filename
done
```

Before and After flipping 1 bit

```
✔ …/hw5/task3  master ● ?
└─ ./run.sh # Orignal plain.txt
MD5(plain.txt)= e94cce14060e4d6eb729b77cd0138d4a
SHA256(plain.txt)= 980717c03ccf8446736c0b89ecbcf2249f0ee20276c59a58a9c8509e2d3170fe
✔ …/hw5/task3  master ● ?
└─ ./run.sh # After flipping 1 bit
MD5(plain.txt)= 1d0ffcb4e2bc7f932f9db5553bc0cb54
SHA256(plain.txt)= 151b2fd33149c7c19f3e1c81c654ea899f7873d6315b4bed53bef3de22e60f33
```

Python3 Script to determine the differences

```python
#!/usr/bin/env python3

MD5_H1 = "700a2be0783cbaebbc42fd95a1ab0b93daafbce6"
MD5_H2 = "1d0ffcb4e2bc7f932f9db5553bc0cb54"

SHA256_H1 = "980717c03ccf8446736c0b89ecbcf2249f0ee20276c59a58a9c8509e2d3170fe"
SHA256_H2 = "151b2fd33149c7c19f3e1c81c654ea899f7873d6315b4bed53bef3de22e60f33"


def main():
    print(
        f"For MD5 {bit_string_diff(MD5_H1, MD5_H2)} bits are the same\n"
        f"For SHA256 {bit_string_diff(SHA256_H1, SHA256_H2)} bits are the same"
    )


def string2bits(s=''):
    return "".join([bin(ord(x))[2:].zfill(8) for x in s])


def bit_string_diff(h1: str, h2: str) -> str:
    count = i = 0
    h1_bits = string2bits(h1)
    h2_bits = string2bits(h2)

    while i < len(h1_bits) and i < len(h2_bits):
        if h1_bits[i] == h2_bits[i]:
            count += 1
        i += 1
    return f"{count} of H1-{len(h1_bits)}/H2-{len(h2_bits)}"
```

```
if __name__ == "__main__":
    main()
```

This script converts H1 and H2 to binary and finds the similar bits between the two. The basis of the string2bits functions was found here(https://stackoverflow.com/a/40949538/7249729) and modified/adapted by me. Below is the result of running this python script.

```
✔ …/hw5/task3  master ● ?
└─ ./diff.py
For MD5 181 of H1-320/H2-256 bits are the same
For SHA256 335 of H1-512/H2-512 bits are the same
```

These are the results of running the python script.

Observations

For MD5, H1 contained 320 bits and H2 had 256 bits. Between the two, 181 bits were the same. For SHA256, both H1 and H2 consisted of 512 bits with 335 being shared between the two.

It can be seen that even though, only 1 bit was flipped, the results were not entirely different but there was a significant change between the two files.

Task 4: Hash Collision-Free Property

Below is a helper class I made to help for 4.1 and 4.2. Its purpose is to store data and also to hash, generate a random message, return a modified hash (24 bits) and print the message and digest.

hash_gen.hpp

```
#ifndef _utils_H_
#define _utils_H_

#include <array>
#include <string>
#include <stdio.h>
#include <openssl/evp.h>
#include <iostream>

#define MSG_LEN 30
#define REDUCE_HASH_LEN 3
```

```
class HashGen
{
private:
  const EVP_MD *md;

public:
  std::string msg;
  std::array<unsigned char, EVP_MAX_MD_SIZE> md_value;
  unsigned int md_len;

  HashGen(const EVP_MD *md);
  void gen_hash();
  std::string gen_msg();
  void gen_all();
  void print_msg();
  void print_digest();
  std::array<unsigned char, REDUCE_HASH_LEN> get_short_digest();
};

#endif
```

hash_gen.cpp

```
#include "hash_gen.hpp"

HashGen::HashGen(const EVP_MD *md) : md(md){};

void HashGen::gen_hash()
{
   if (this->msg.empty())
      this->msg = this->gen_msg();

   EVP_MD_CTX *mdctx;
   mdctx = EVP_MD_CTX_create();

   EVP_DigestInit_ex(mdctx, this->md, NULL);
   EVP_DigestUpdate(mdctx, this->msg.c_str(), this->msg.length());
   EVP_DigestFinal_ex(mdctx, this->md_value.data(), &this->md_len);
   EVP_MD_CTX_destroy(mdctx);
}

std::string HashGen::gen_msg()
{
   auto randchar = []() -> char {
```

```
        const char charset[] =
          "0123456789"
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
          "abcdefghijklmnopqrstuvwxyz";
        const size_t max_index = (sizeof(charset) - 1);
        return charset[rand() % max_index];
    };
    std::string str(MSG_LEN, 0);
    std::generate_n(str.begin(), MSG_LEN, randchar);
    return str;
}

void HashGen::gen_all()
{
    this->msg = this->gen_msg();
    this->gen_hash();
}

void HashGen::print_msg()
{
    for (size_t i = 0; i < MSG_LEN; i++)
        std::cout << this->msg[i];
    std::cout << '\n';
}

void HashGen::print_digest()
{
    for (auto i : this->md_value)
        printf("%02x", i);
    std::cout << '\n';
}

std::array<unsigned char, REDUCE_HASH_LEN> HashGen::get_short_digest()
{
    std::array<unsigned char, REDUCE_HASH_LEN> t;

    for (size_t i = 0; i < REDUCE_HASH_LEN; i++)
        t[i] = this->md_value[i];
    return t;
}
```

Subtask 1

Below is the driver for performing the dictionary attack. A random message is generated and then hash. The hash (24bits) are looked up in the map to see if it has been created before, if it has it compares to see if the message is different, if it is, we have found our desired collision. If the hash is not found in the map, the hash, message pair is added to the map and the process is repeated.

```
#include <string.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <map>
#include <array>

#include "hash_gen.hpp"

int main(int argc __attribute__((unused)), char *argv[] __attribute__((unused)))
{
   srand(time(NULL));

   OpenSSL_add_all_digests();

   const EVP_MD *md = EVP_get_digestbyname("md5");

   // run the loop
   HashGen hg(md);
   std::map<std::array<unsigned char, REDUCE_HASH_LEN>, std::string> msg_digest_map; // key: digest, val: msg
   bool collision = false;
   std::map<std::array<unsigned char, REDUCE_HASH_LEN>, std::string>::iterator it;
   int count = 0;

   do
   {
      hg.gen_all();
      count += 1;
      it = msg_digest_map.find(std::array<unsigned char, REDUCE_HASH_LEN>(hg.get_short_digest()));

      if (it != msg_digest_map.end() && it->second != hg.msg)
         collision = true;
      else
         msg_digest_map[hg.get_short_digest()] = hg.msg;

   } while (!collision);
```

```
    std::cout << "Message 1: " << it->second << '\n';
    std::cout << "Message 2: ";
    hg.print_msg();
    std::cout << '\n';

    std::cout << "Digest 1: ";
    hg.print_digest();

    hg.msg = it->second;
    hg.gen_hash();
    std::cout << "Digest 2: ";
    hg.print_digest();
    std::cout << '\n';

    std::cout << "Tries: " << count << '\n';

    EVP_cleanup();
    return 0;
}
```

Subtask 2

This is the driver program for subtask 2. The desired message, “If at first you don't succeed”, is first hashed and stored. Next, a random message is generated and hashed. If this hash is the same as our desired message’s hash (24 bits), we compare the messages for a difference. If they are different, we have found a collision. If a collision was not found, we start back again generating another random message.

```
#include <string.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <map>
#include <array>

#include "hash_gen.hpp"

int main(int argc __attribute__((unused)), char *argv[] __attribute__((unused)))
{
    srand(time(NULL));

    OpenSSL_add_all_digests();
```

```
  const EVP_MD *md = EVP_get_digestbyname("md5");

  // run the loop
  HashGen h1(md);
  HashGen h2(md);
  int count = 0;

  h1.msg = "If at first you don't succeed,";
  h1.gen_hash();

  do
  {
     h2.gen_all();
     count += 1;
  } while (!(h1.get_short_digest() == h2.get_short_digest() && h1.msg != h2.msg));

  std::cout << "Message 1: ";
  h1.print_msg();
  std::cout << "Message 2: ";
  h2.print_msg();
  std::cout << '\n';

  std::cout << "Digest 1: ";
  h1.print_digest();
  std::cout << "Digest 2: ";
  h1.print_digest();
  std::cout << '\n';

  std::cout << "Tries: " << count << '\n';

  EVP_cleanup();
  return 0;
}
```

Results

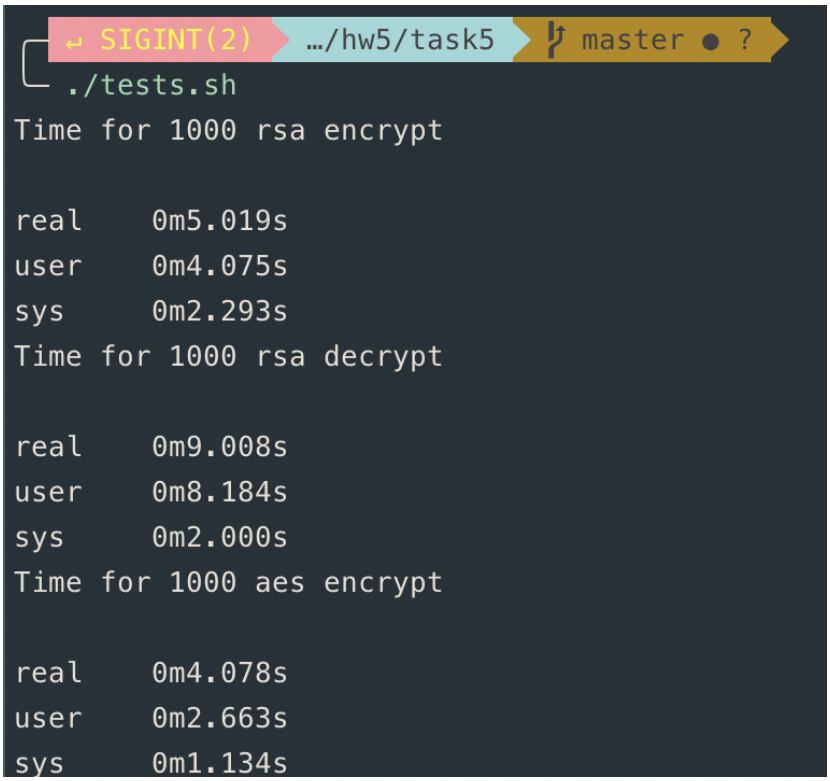
Subtask 1

message 1	message 2	digest 1	digest 2	tries
LqXd4RWekgiS	olV4eWJeAucRtQA0CilkrY1UzILnE9	2b264048f17eed063d7220ee1bddc00201(	2b2640829581e71ecb3709830222f06601	5878
4kmsOLyqWJe	5tZOv4l7bNmoL9YmZ3tzxYmSE8WHZV	785ff1d17b331d86f4b68672c3e36c3c010(	785ff1ec783651e069de4abdbc88fc6d010	4658
usbisdBpm8ctl	6aYf4lXRi66DvqibLvaTmTABks2SxX	beb75ca2cd208608be1eb5fc9047df8c010(	beb75c54fc0c298286c7347e5fa7590501(	4699
usbisdBpm8ctl	6aYf4lXRi66DvqibLvaTmTABks2SxX	beb75ca2cd208608be1eb5fc9047df8c010(	beb75c54fc0c298286c7347e5fa7590501(	4699
cZO0dhbqU6d:	lInjAkXXXeoChBPlxGCnHQO0QpLo4H	87624e38206f8e33acb0ea2ad9918c46010	87624ee66e92d3779ff927d8f98b209d01(	5948
7YSLM686Atn\	RgSkewqJZeaaogDFwwfvryKZYKMmXl	abfaf7b70cca485bbc82a2f27718561c0100	abfaf72bc8c83f56aeb203b091ebaa2701(	3733
7YSLM686Atn\	RgSkewqJZeaaogDFwwfvryKZYKMmXl	abfaf7b70cca485bbc82a2f27718561c0100	abfaf72bc8c83f56aeb203b091ebaa2701(	3733
iMPxBcS7dXrtr	zWZIGgGydcU348BX5ppXH0bSVSgumt	46d29cedc493bb8464b825528d98352601(	46d29cdf0fe248cddc72d3167b1f7ba7010	8581
iMPxBcS7dXrtr	zWZIGgGydcU348BX5ppXH0bSVSgumt	46d29cedc493bb8464b825528d98352601(	46d29cdf0fe248cddc72d3167b1f7ba7010	8581
xH7US9VA9oV	niw6GNChablGqRpexqhA9bvaVz6ejw	c35bcc661a25c3979c2e0ce8bfd153120100	c35bcc74e61492d5ab3014444cd1c64f01(	5020
xH7US9VA9oV	niw6GNChablGqRpexqhA9bvaVz6ejw	c35bcc661a25c3979c2e0ce8bfd153120100	c35bcc74e61492d5ab3014444cd1c64f01(	5020
MQnhQF9KaU	tSwD6ohFBhBoi0EMET2rzum49drfDb	233d8d19aa8f05674186ecbfa4a7c9b2010(	233d8de8f66ee17adabd914fe8d3fc9401(	6221
Xh3F7xT8e0nC	BwoKFerTcKYY48ae7iDaJi7l8KEsEP	a2f8e7ad566c6e6e2036d203602ec8aa010(	a2f8e729c4d30e13ddd06ca660266fe701(	5904
Xh3F7xT8e0nC	BwoKFerTcKYY48ae7iDaJi7l8KEsEP	a2f8e7ad566c6e6e2036d203602ec8aa010(	a2f8e729c4d30e13ddd06ca660266fe701(	5904
Average				**5238.6**

Subtask 2

message 1	message 2	digest 1	digest 2	tries
If at first you don't succeed,	zJlgLxYoGj67ayXwrm4kaArnCjsB5j	62a7aba267380f559359f2	62a7aba267380f559359f22a2	30,127,741
If at first you don't succeed,	7wa1nRIWsUKT3od8FO9fvXXHkjxEFf	62a7aba267380f559359f2	62a7aba267380f559359f22a2	6,019,754
If at first you don't succeed,	gm8oXC1TQhxcznFrvLVW7PalYs6XKQ	62a7aba267380f559359f2	62a7ab0395925e1b522460c9c	26,018,523
If at first you don't succeed,	NVvdYGV0WtIbgrr2o5927GxJy8OCV0	62a7aba267380f559359f2	62a7ab505f2c1eadd6a49032b	9,749,633
If at first you don't succeed,	ovTXzujCgFiLLnIzEfAljj6kYbPUFs	62a7aba267380f559359f2	62a7ab5c712dd1f2dd2a2caa4	9,790,722
If at first you don't succeed,	XlJ8ZljSK8cu6YtkiWFn7czJr3Q6LP	62a7aba267380f559359f2	62a7ab13a8545bb0284bd0303	781,648
If at first you don't succeed,	ebCNJDQ1bDCgh2YMAo1M9V3bdoocyG	62a7aba267380f559359f2	62a7abecaf4b1819c6595cbdb3	2,340,329
If at first you don't succeed,	j6cK7sRvTrKB2uWKxl3js2OJA2ZGiH	62a7aba267380f559359f2	62a7ab59c476a340dbefc7d31	27,844,162
If at first you don't succeed,	j6cK7sRvTrKB2uWKxl3js2OJA2ZGiH	62a7aba267380f559359f2	62a7ab59c476a340dbefc7d31	34,123,064
If at first you don't succeed,	2dZ8VVQLf43TDsKwE6cmPFgfO6HaRE	62a7aba267380f559359f2	62a7ab0b943f73fd44a707128	1,370,270
Average				**14,816,584.60**

Task 5: Performance Comparison: RSA versus AES


↵ SIGINT(2)
…/hw5/task5
master ● ?
./tests.sh
Time for 1000 rsa encrypt
real 0m5.019s
user 0m4.075s
sys 0m2.293s
Time for 1000 rsa decrypt
real 0m9.008s
user 0m8.184s
sys 0m2.000s
Time for 1000 aes encrypt
real 0m4.078s
user 0m2.663s
sys 0m1.134s

RSA and AES Speed benchmarks

```
↵ SIGINT(2)  …/hw5/task4  master ● ?
└─ openssl speed rsa
Doing 512 bit private rsa's for 10s: 14338 512 bit private RSA's in 9.98s
Doing 512 bit public rsa's for 10s: 139147 512 bit public RSA's in 9.97s
Doing 1024 bit private rsa's for 10s: 2460 1024 bit private RSA's in 9.99s
Doing 1024 bit public rsa's for 10s: 29299 1024 bit public RSA's in 9.98s
Doing 2048 bit private rsa's for 10s: 365 2048 bit private RSA's in 9.99s
Doing 2048 bit public rsa's for 10s: 7674 2048 bit public RSA's in 9.98s
Doing 4096 bit private rsa's for 10s: 56 4096 bit private RSA's in 10.02s
Doing 4096 bit public rsa's for 10s: 2145 4096 bit public RSA's in 9.99s
LibreSSL 2.6.4
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000696s 0.000072s   1436.7  13956.6
rsa 1024 bits 0.004061s 0.000341s    246.2   2935.8
rsa 2048 bits 0.027370s 0.001300s     36.5    768.9
rsa 4096 bits 0.178929s 0.004657s      5.6    214.7
✔  …/hw5/task4  master ● ?
└─ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 29887710 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 64 size blocks: 7933703 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 2015883 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 504816 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 63311 aes-128 cbc's in 2.99s
Doing aes-192 cbc for 3s on 16 size blocks: 26188524 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 64 size blocks: 6736536 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 256 size blocks: 1725780 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 1024 size blocks: 424666 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 8192 size blocks: 53814 aes-192 cbc's in 2.99s
Doing aes-256 cbc for 3s on 16 size blocks: 21060248 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 64 size blocks: 6034709 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 256 size blocks: 1407234 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 1024 size blocks: 362757 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 8192 size blocks: 43267 aes-256 cbc's in 2.98s
LibreSSL 2.6.4
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc     159934.23k   169252.33k   172022.02k   172310.53k   173459.44k
aes-192 cbc     139672.13k   144193.41k   147759.09k   144952.66k   147439.56k
aes-256 cbc     113075.16k   128740.46k   120889.90k   123821.06k   118940.69k
```

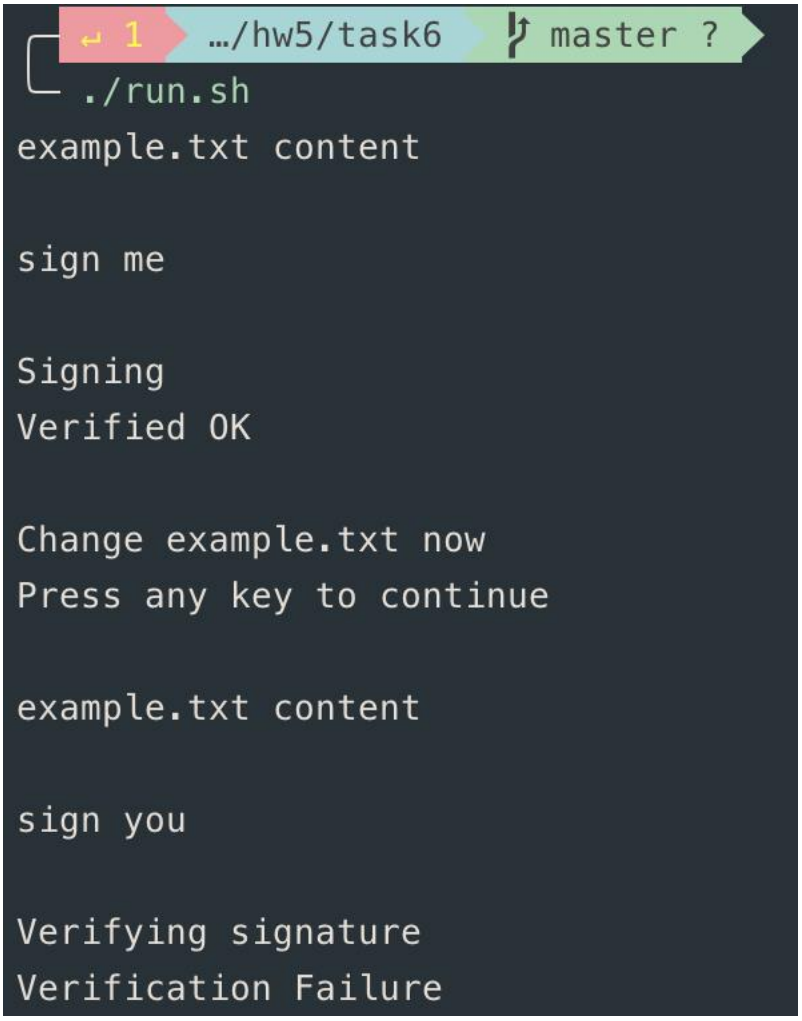
Please describe whether your observations are similar to those from the outputs of the speed command.

Task 6: Create Digital Signature

Below is a script I created to sign the file, verify the signature, wait for me to change the file and the verify the signature again.

```
#!/usr/bin/env bash

in=example.txt
signature=example.sha256
pub_key=id_rsa.pub.pem
priv_key=id_rsa


printf "$in content\n\n"
cat $in

printf "\n\nSigning\n"
openssl dgst -sha256 -sign $priv_key -out $signature $in
openssl dgst -sha256 -verify $pub_key -signature $signature $in

printf "\nChange $in now\n"
read -n 1 -s -r -p "Press any key to continue"

printf "\n\n$in content\n\n"
cat $in


printf "\n\nVerifying signature\n"
openssl dgst -sha256 -verify $pub_key -signature $signature $in
```

Below is the results of executing the script.



Please describe how you did the above operations (e.g., what commands do you use, etc.). Explain your observations. Please also explain why digital signatures are useful.

I made an example.txt file that contained the text “sign me”. I signed the file with the private key and verified the signature with the public key. The verification succeeded. I then changed the content of example.txt to contain the text “sign you”. I then verified the signature again with the public key but the verification failed.

Signatures are useful because a person A can validate or agree with the current state of a file and then person B can verify that they are in fact looking at the same version/variation of the file that person A was looking at.