Homework 5

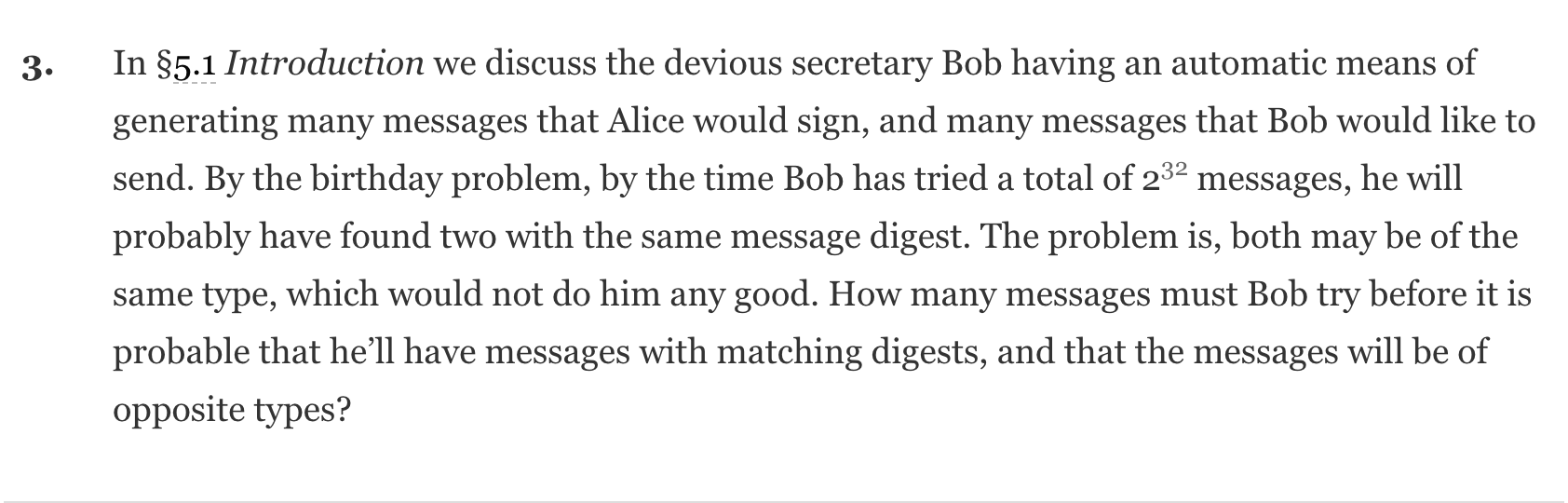
CSCE-465-500

November 21, 2018

Joseph Martinsen

# Paper-and-Pencil Problems

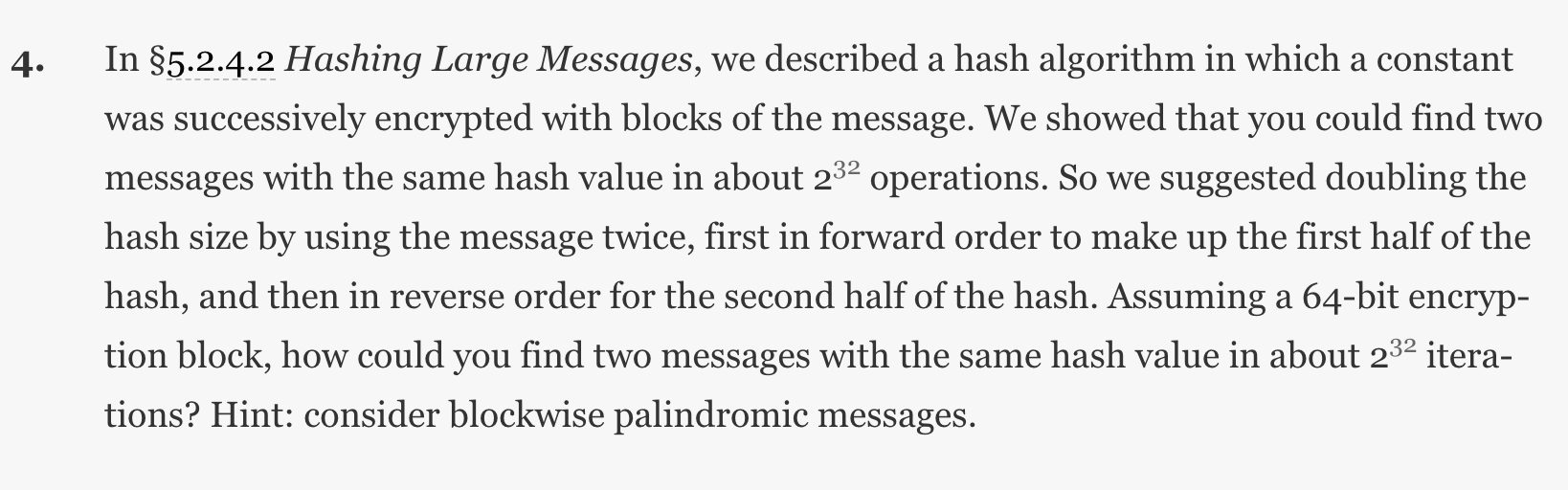
• 5.3. (8 pts)



Bob will have to try messages

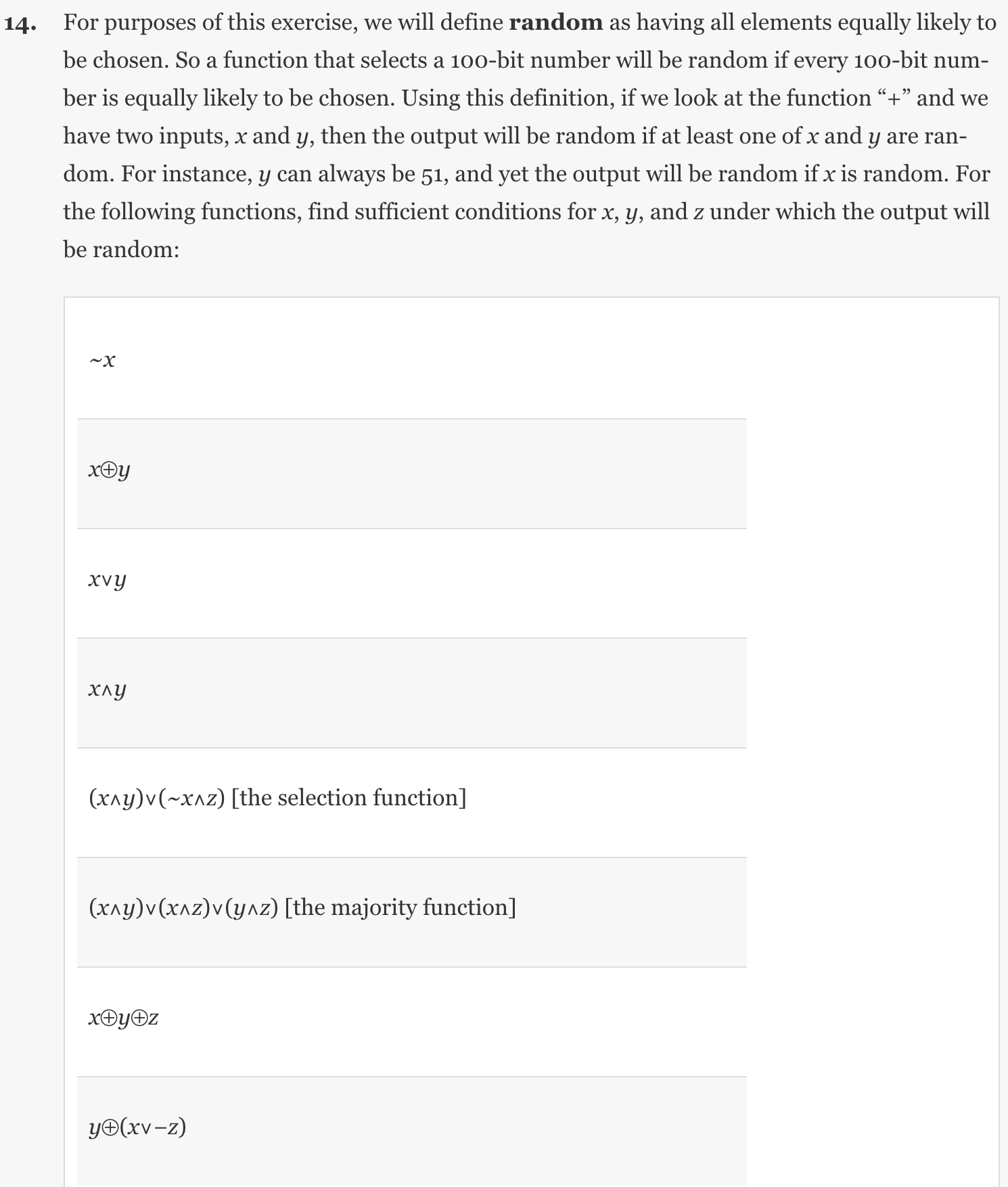
­­­

• 5.4. (8 pts)



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 2nd part of the hash and compare

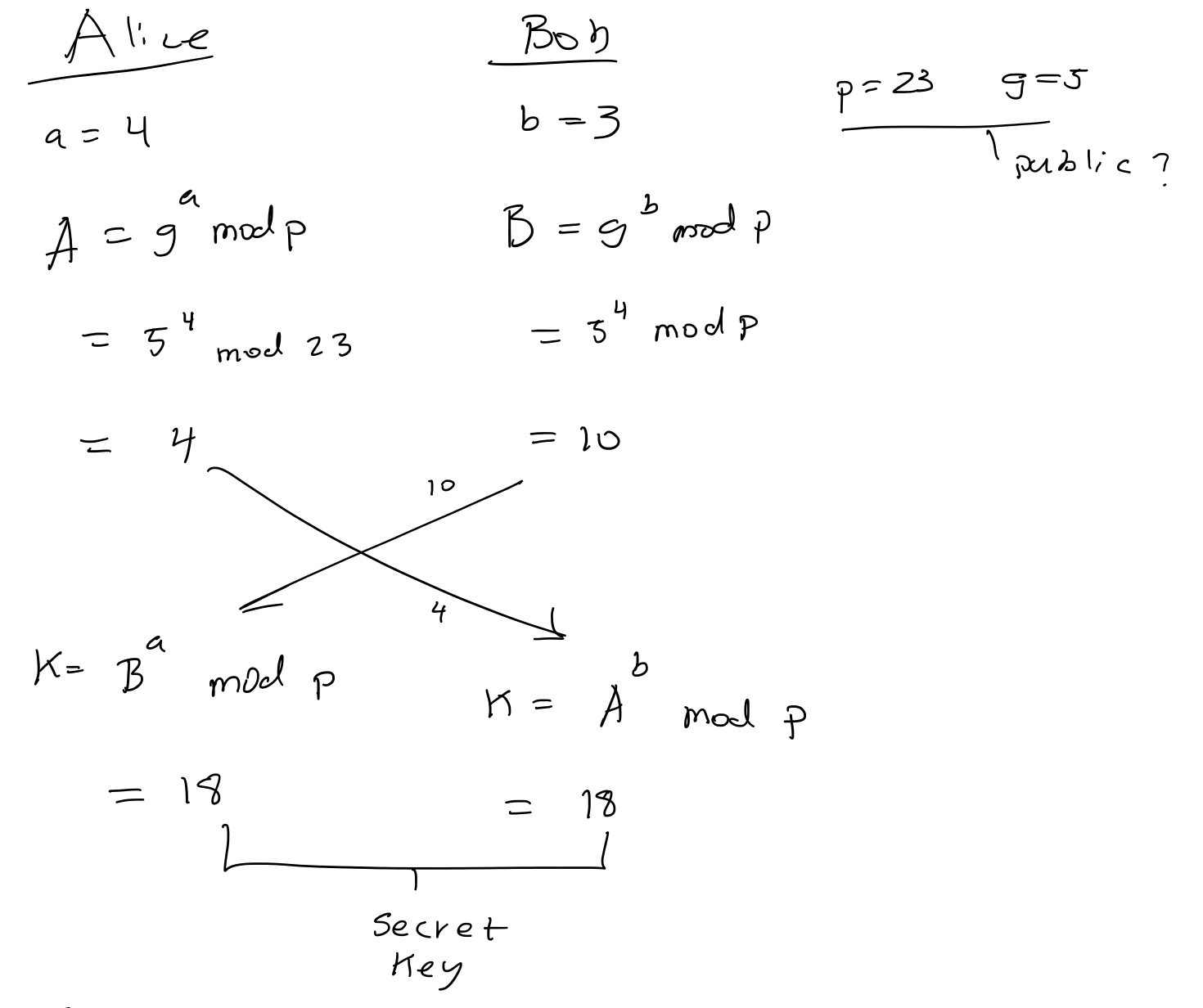
• 5.14. (8 pts)



• 6.2. (8 pts)

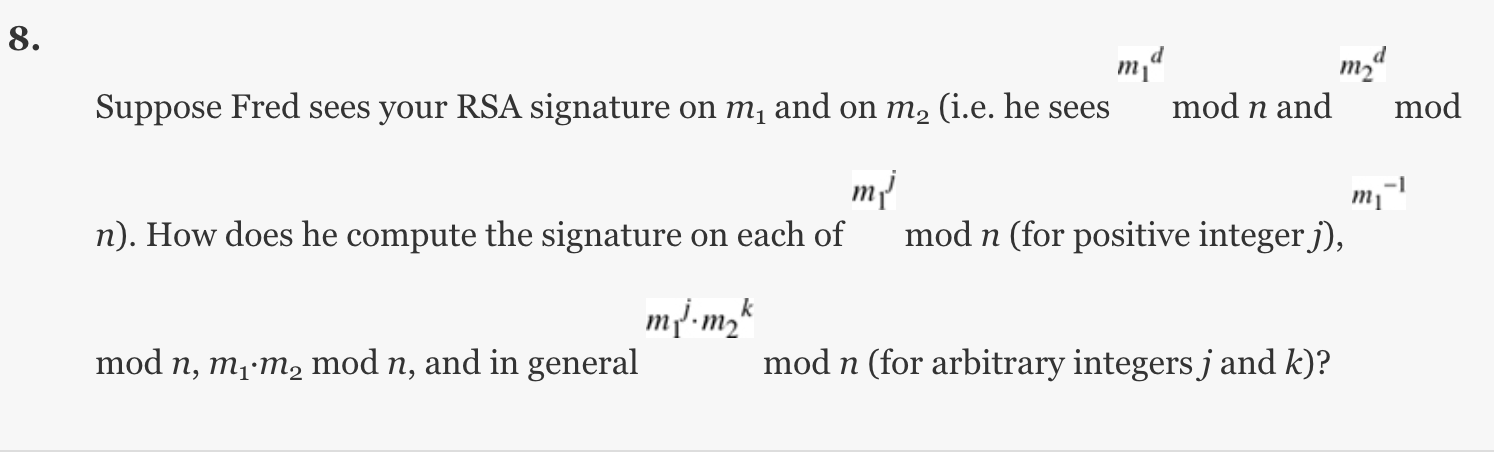
1. x can be random. y and z can be anything
2. x/y can be random if they differ one bit. Z can be anything
3. X/y can be random and differ by at least 1 bit. Z can be anything
4. x/y can be random and must have at least one 1. Z can be anything.
5. Either x/y are different or ~x/z are different
6. Much like (b), at least 2 out of x,y,z have to differ by at least 1 bit
7. X or ~z should be different and the result should differ by at least 1 bit with y.





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)



# 

# 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*

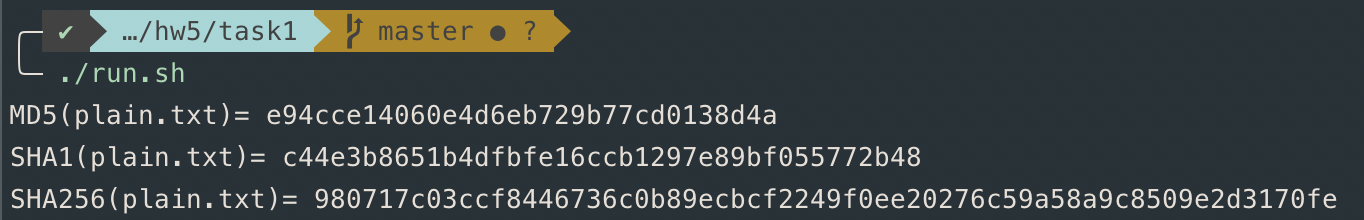
filename="plain.txt"

*for* dgsttype *in* -md5 -sha1 -sha256; *do*

openssl dgst $dgsttype $filename

*done*

Below are the results of running this script.



## 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*

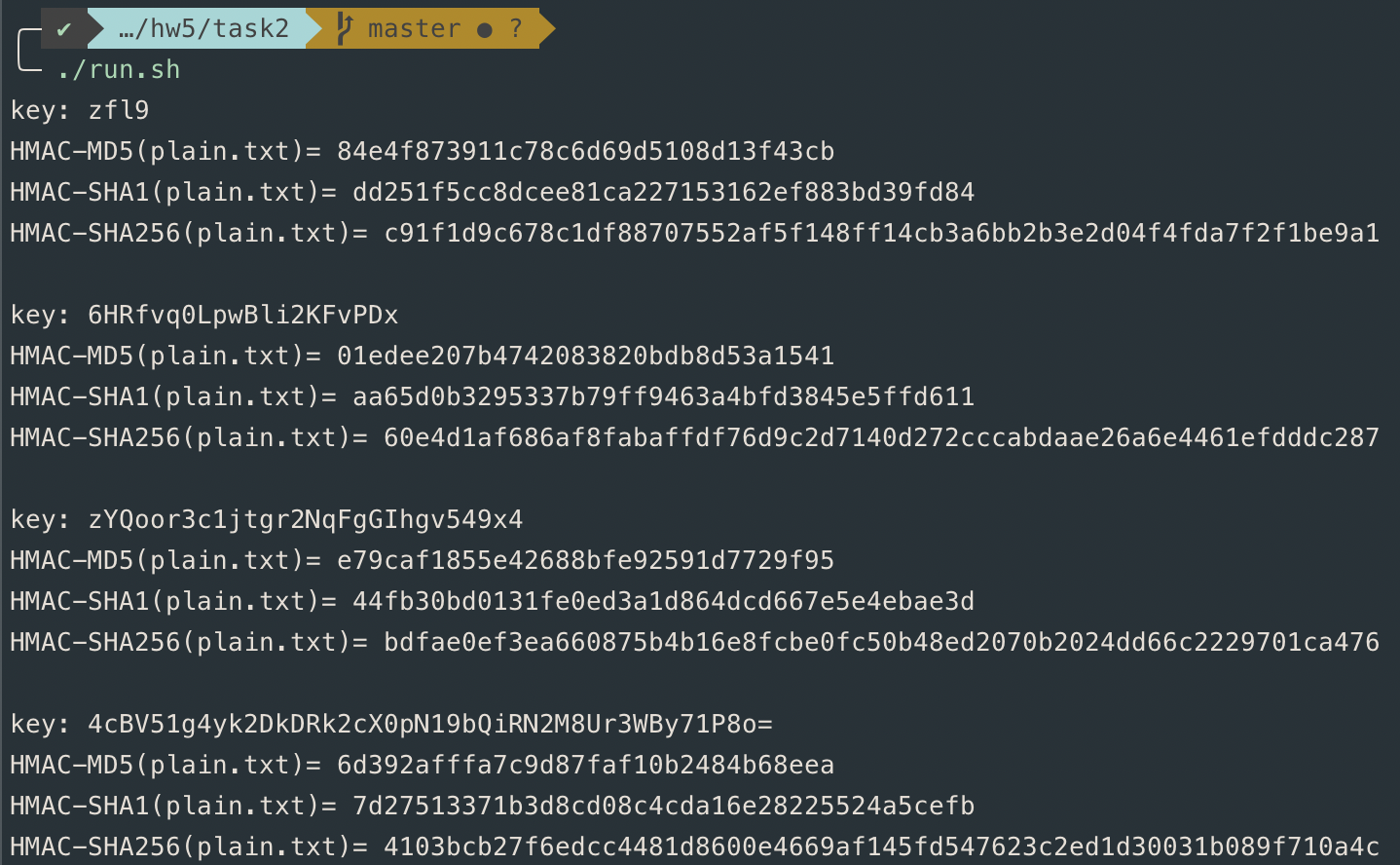
openssl dgst $dgsttype -hmac $key $filename

*done*

echo ''

*done*

Below are the results.



*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

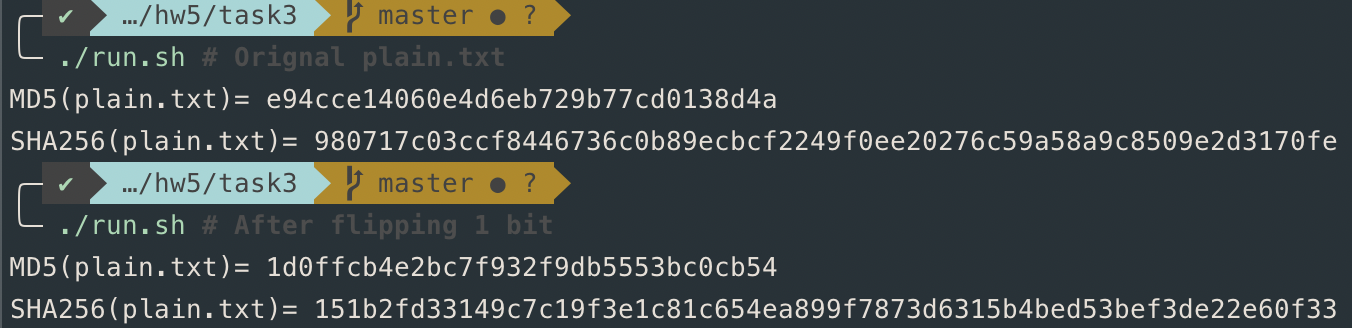
filename="plain.txt"

*for* dgsttype *in* -md5 -sha256; *do*

openssl dgst $dgsttype $filename

*done*

### Before and After flipping 1 bit



### Python3 Script to determine the differences

*#!/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

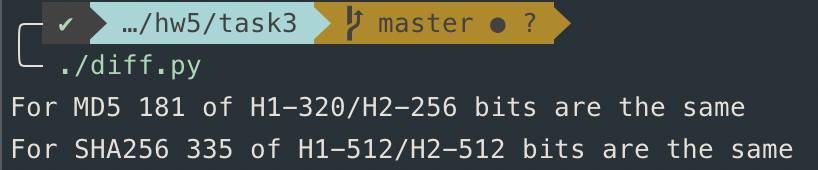
i += 1

*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.



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';

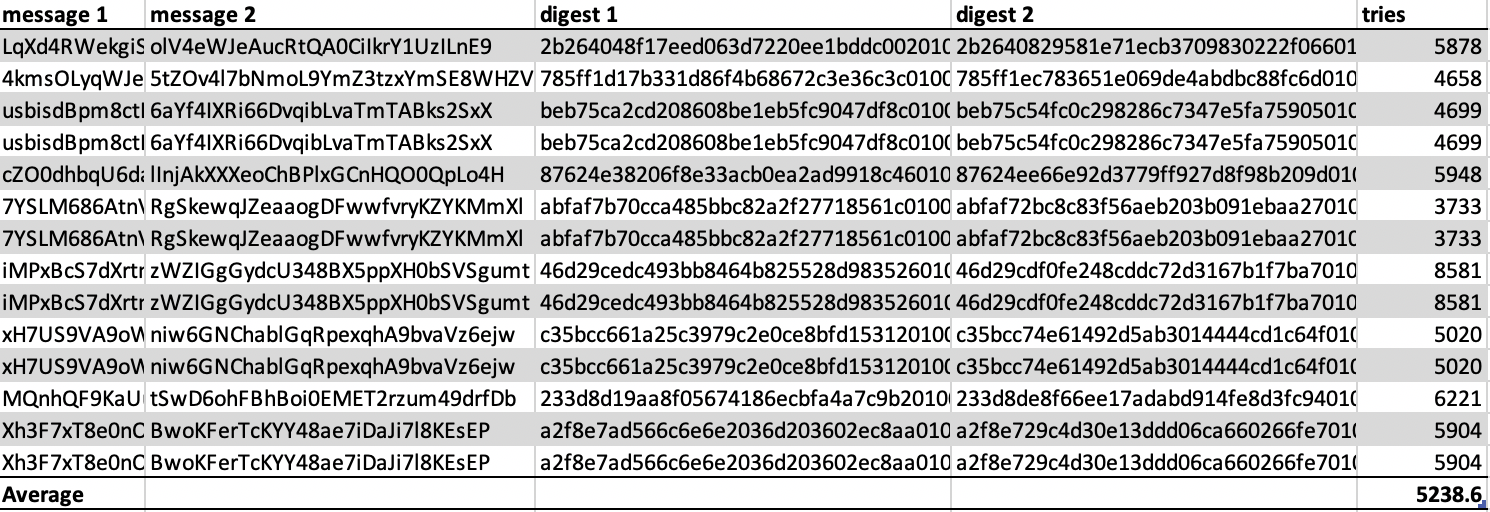
EVP\_cleanup();

*return* 0;

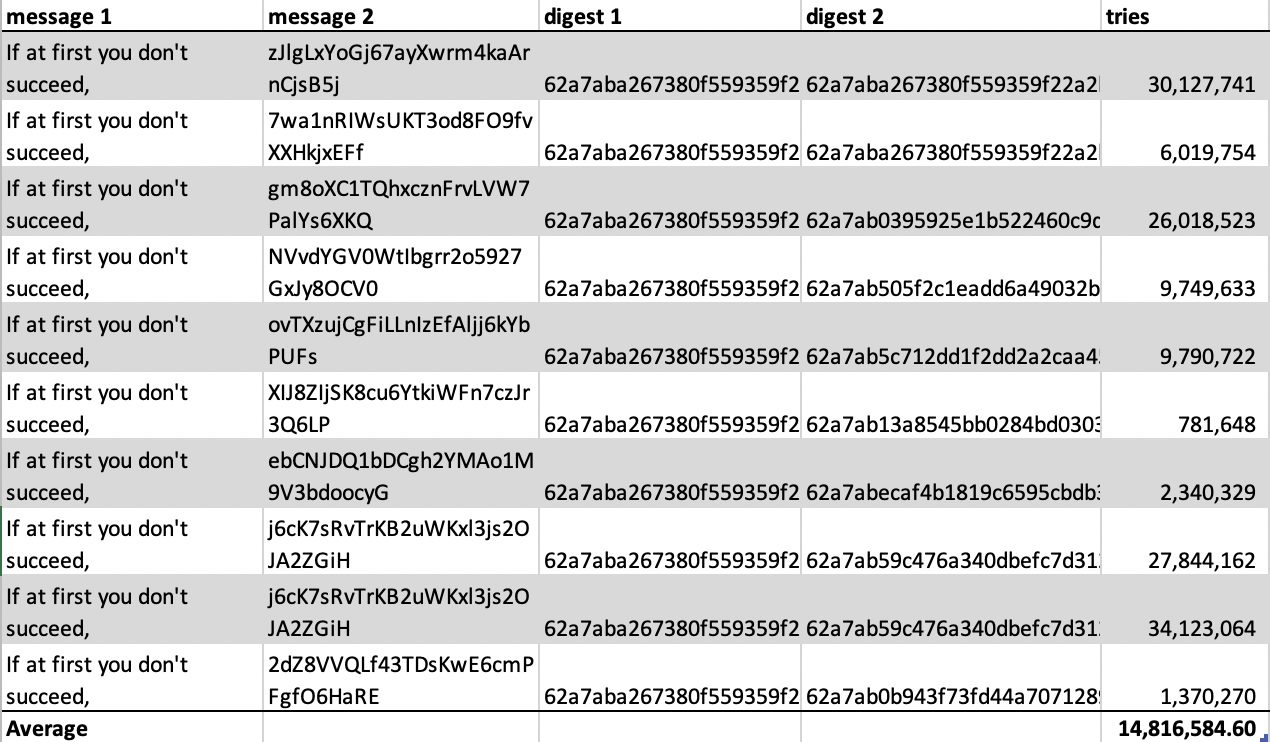
}

## Results

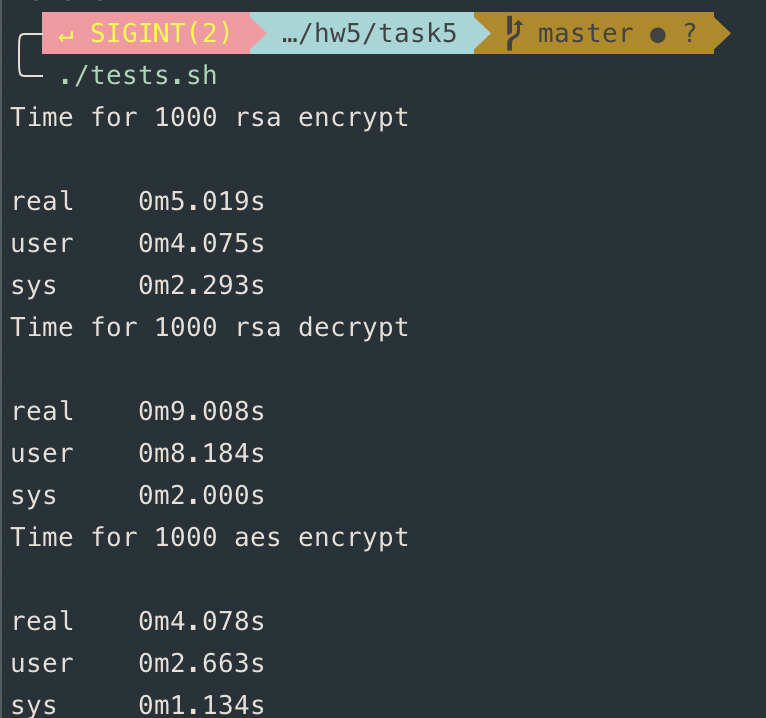
### Subtask 1



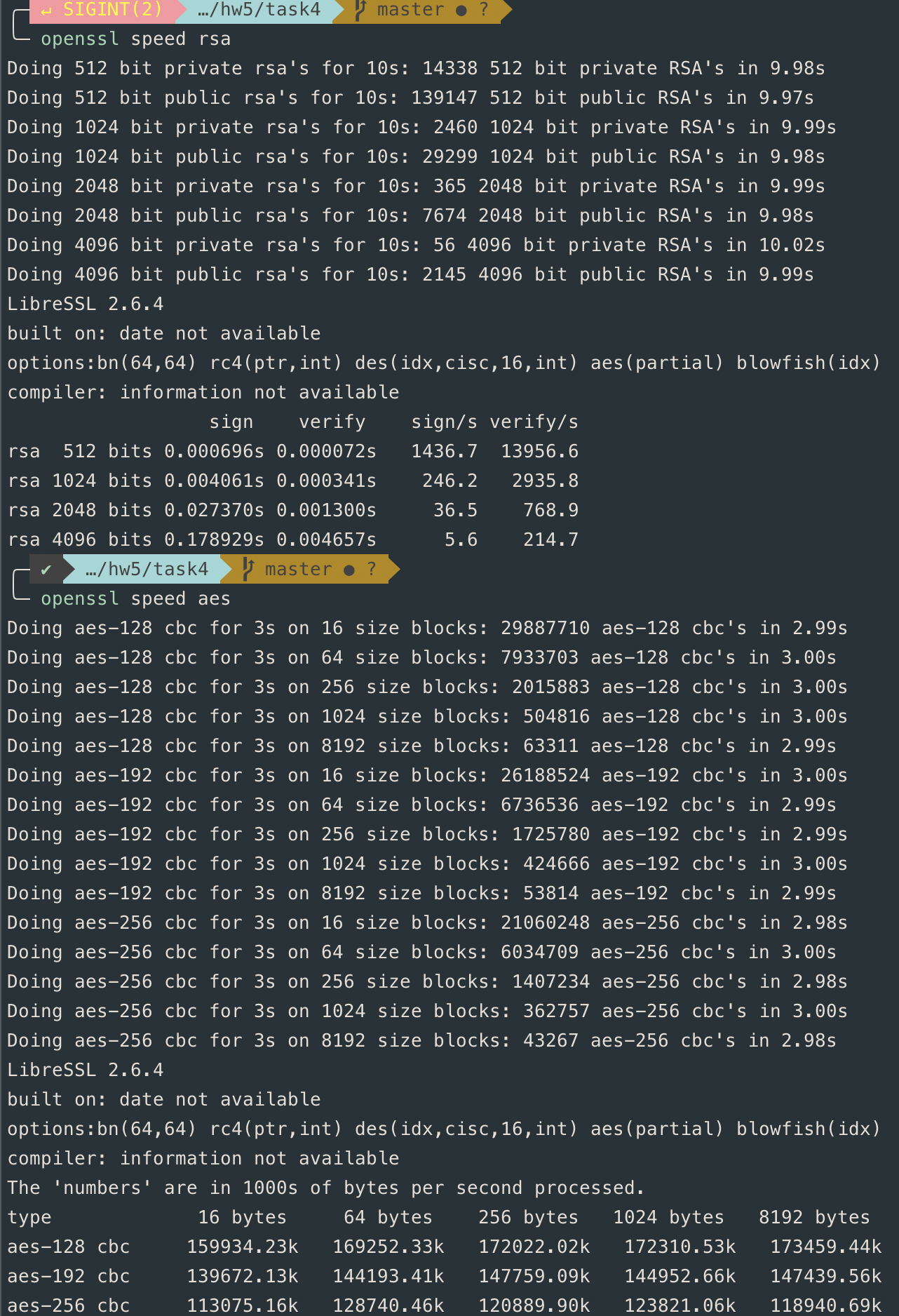
### Subtask 2



# Task 5: Performance Comparison: RSA versus AES



### RSA and AES Speed benchmarks



*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"

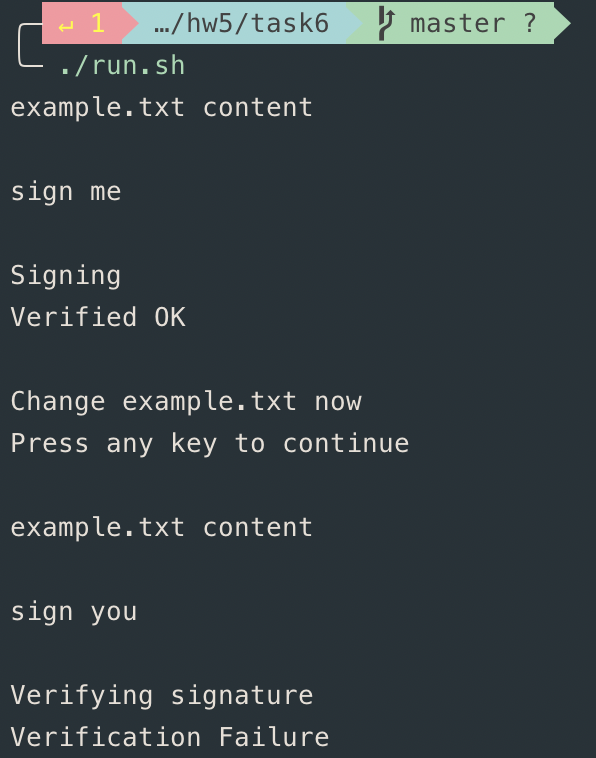
printf "\n\n$in content\n\n"

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.