Name: Amit Banerjee
UB-IT:50287084

Computer Security

Homework 3

All measurements for the assignment have been done on local machine with following specifications: -
Operating System: Mac Os (High Sierra)
Processor: 2.5 Ghz Intel Core i5
Memory:16 GB
Graphics: Intel HD Graphics 4000

Q.1.a.)
Measurements for **AES algorithm** in **CBC** mode: -

The larger file consists of 1Mb of data and smaller file consists of 1Kb data.

- Time taken to generate a new key: - 9.78 microseconds

- Time taken for encryption of larger file: - 3461.1225 microseconds
- Time taken for decryption of larger file: - 4467.01 microseconds

- Time taken for encryption of small file: - 17.8 microseconds
- Time taken for decryption of small file: - 22.17 microseconds

- Speed per byte for encryption: - 0.017 microseconds
- Speed per byte for decryption: - 0.021 microseconds

Source Code: -

```
from Crypto.Cipher import AES
from Crypto import Random
import time
start_time = time.time()

import os
#generate 16 byte key
start_time = time.time()
key=os.urandom(16)
print("key generation time :- %s seconds " % (time.time() - start_time))

#blocksize of AES as required
blocksize=AES.block_size

def encryptm(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()
```

```
    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    in_vector = Random.new().read(blocksize)
    cipher=AES.new(key, AES.MODE_CBC, in_vector)[2]
    start_time = time.time()
    encrypted=in_vector + cipher.encrypt(plaintext)
    print("encrypt mb time :-%s seconds " % (time.time() - start_time))
    f=open("encryptedmb.txt","wb")
    f.write(encrypted)

def decryptm(input_file,key):
    f=open("encryptedmb.txt","rb")
    ciphertext=f.read()
    in_vector=ciphertext[: blocksize]
    cipher = AES.new (key, AES.MODE_CBC, in_vector)
    start_time = time.time()
    decrypted=cipher.decrypt(ciphertext[blocksize:])
    print ("decrypt mb time :-%s seconds " % (time.time() - start_time))
    dec=decrypted.rstrip(b"\0")
    f=open("decryptmb.txt","wb")
    f.write(dec)

encryptm("mb file.txt",key)
decryptm("encryptedmb.txt",key)

#for small size
def encrypt(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()

    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    in_vector = Random.new().read(blocksize)
    cipher=AES.new(key, AES.MODE_CBC, in_vector)
    start_time = time.time()
    encrypted=in_vector + cipher.encrypt(plaintext)
    print("encrypt kb time :-%s seconds " % (time.time() - start_time))
    f=open("encryptedkb.txt","wb")
    f.write(encrypted)

def decrypt(input_file,key):
    f=open("encryptedkb.txt","rb")
    ciphertext=f.read()
    in_vector=ciphertext[:blocksize]
    cipher = AES.new(key, AES.MODE_CBC, in_vector)
    start_time = time.time()
    decrypted=cipher.decrypt(ciphertext[blocksize:])
    print("decrypt kb time :- %s seconds " % (time.time() - start_time))
```

```
    dec=decrypted.rstrip(b"\0")
    f=open("decryptkb.txt","wb")
    f.write(dec)

encrypt("kb file.txt",key)
decrypt("encryptedkb.txt",key)
```

Q1. b.) Measurements for **AES algorithm** in **CTR** mode with 128 bit key: -

- Time taken to generate a new key: - 10.75 microseconds
- Time taken for encryption of larger file: - 2243.99 microseconds
- Time taken for decryption of larger file: - 3067.01 microseconds

- Time taken for encryption of small file: - 27.8 microseconds
- Time taken for decryption of small file: - 28.8 microseconds

- Speed per byte for encryption: - 0.027 microseconds
- Speed per byte for decryption: -0.028 microseconds

Source Code:-

```
from Crypto.Cipher import AES
from Crypto.Util import Counter
import time
import os
#generate 16 byte key
start_time = time.clock()
key=os.urandom(16)
print("key generation time :- %s seconds " % (time.clock() - start_time))

#blocksize of AES as required
blocksize=AES.block_size

def encryptm(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()

    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    #in_vector = Random.new().read(blocksize)
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr)[3]
    start_time = time.time()
    encrypted=cipher.encrypt(plaintext)
    print("encryption mb time :- %s seconds " % (time.time() - start_time))
    f=open("encryptedmb.txt","wb")
    f.write(encrypted)
```

```
encryptm("mb file.txt",key)

def decryptm(input_file,key):
    f=open("encryptedmb.txt","rb")
    ciphertext=f.read()
    #in_vector=ciphertext[:blocksize]
    ctr=Counter.new(128)
    cipher = AES.new(key, AES.MODE_CTR, counter=ctr)[3]
    start_time = time.time()
    decrypted=cipher.decrypt(ciphertext)
    print("deryption mb time :- %s seconds " % (time.time() - start_time))
    dec=decrypted.rstrip(b"\0")
    f=open("decryptmb.txt","wb")
    f.write(dec)

decryptm("encryptedmb.txt",key)

#for small size
def encrypt(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()

    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr) [3]
    start_time = time.time()
    encrypted=cipher.encrypt(plaintext)
    print("encrypt kb time :-%s seconds " % (time.time() - start_time))
    f=open("encryptedkb.txt","wb")
    f.write(encrypted)

def decrypt(input_file,key):
    f=open("encryptedkb.txt","rb")
    ciphertext=f.read()
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr) [3]
    start_time=time.time()
    decrypted=cipher.decrypt(ciphertext)
    print("decrypt kb time :- %s seconds " % (time.time() - start_time))
    dec=decrypted.rstrip(b"\0")
    f=open("decryptkb.txt","wb")
    f.write(dec)

encrypt("kb file.txt",key)

decrypt("encryptedkb.txt",key)
```

Q1. c.) Measurements for **AES algorithm** with 256 bit key in **CTR** mode: -

- Time taken to generate a new key: - 12.8 microseconds
- Time taken for encryption for larger file: - 1879.9 microseconds
- Time taken for decryption for larger file: - 2034.18 microseconds

- Time taken for encryption for small file: - 28.07 microseconds
- Time taken for decryption for small file: - 39.98 microseconds

- Speed per byte for encryption: - 0.028 microseconds
- Speed per byte for decryption: - 0.039 microseconds

Source Code : -

```
from Crypto.Cipher import AES
from Crypto.Util import Counter
import time
import os
start_time = time.time()
#generate 16 byte key
key=os.urandom(32)
print("key generation time :- %s seconds " % (time.time() - start_time))

#blocksize of AES as required
blocksize=AES.block_size

#for bigger file sizes
def encryptm(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()

    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    #in_vector = Random.new().read(blocksize)
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr)

    start_time=time.time()
    encrypted=cipher.encrypt(plaintext)
    print("encryption mb time :- %s seconds " % (time.time() - start_time))

    f=open("encryptedmb.txt","wb")
    f.write(encrypted)

encryptm("mb file.txt",key)

def decryptm(input_file,key):
    f=open("encryptedmb.txt","rb")
    ciphertext=f.read()
    #in_vector=ciphertext[:blocksize]
```

```
    ctr=Counter.new(128)
    cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
    start_time=time.time()
    decrypted=cipher.decrypt(ciphertext)
    print("decryption  mb time :- %s seconds " % (time.time() - start_time))
    dec=decrypted.rstrip(b"\0")
    f=open("decryptmb.txt","wb")
    f.write(dec)


decryptm("encryptedmb.txt",key)

#for small file size
def encrypt(input_file,key):
    f=open(input_file,"rb")
    plaintext=f.read()

    filesize=len(plaintext)
    padding_reqd=(blocksize -filesize) % blocksize
    plaintext=plaintext + b"\0" * padding_reqd
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr)
    start_time=time.time()
    encrypted=cipher.encrypt(plaintext)
    print("encrypt kb time :- %s seconds " % (time.time() - start_time))
    f=open("encryptedkb.txt","wb")
    f.write(encrypted)

def decrypt(input_file,key):
    f=open("encryptedkb.txt","rb")
    ciphertext=f.read()
    ctr = Counter.new(128)
    cipher=AES.new(key, AES.MODE_CTR, counter=ctr)
    start_time=time.time()
    decrypted=cipher.decrypt(ciphertext)
    print("decrypt kb time :- %s seconds " % (time.time() - start_time))
    dec=decrypted.rstrip(b"\0")
    f=open("decryptkb.txt","wb")
    f.write(dec)

encrypt("kb file.txt",key)

decrypt("encryptedkb.txt",key)
```

Q1. d.) Measurements for various hashing algorithms: -

Sha 256 Algorithm: -

- Total time for smaller file: -32.3 microseconds
- Total time for larger file: - 3520 microseconds
- Speed/byte for hashing: - 0.0315 microseconds

Sha512 Algorithm: -

- Total time for smaller file: - 30.9 microseconds
- Total time for larger file: - 2640 microseconds
- Speed/byte for hashing: - 0.030 microseconds

Sha3-256 Algorithm: -

- Total time for smaller file: -143 microseconds
- Total time for larger file: - 4329.5 microseconds
- Speed/byte for hashing: - 0.139 microseconds

Source Code: -

```
import time
import hashlib
filename ="kb file.txt"
f=open(filename,"rb")
bytes = f.read()
start_time=time.time()
hashop = hashlib.sha256(bytes).hexdigest();
print("\ntime reqd for sha256 for kb file:- %s seconds" % (time.time()- start_time))
print("SHA 256 hash output for kb file :-\n\n",hashop)

import hashlib
filename ="mb file.txt"
f=open(filename,"rb")
bytes = f.read()
start_time=time.time()
hashop = hashlib.sha256(bytes).hexdigest();[5]
print("\ntime reqd for sha256 for mb file:- %s seconds" % (time.time()- start_time))
print("SHA 256 hash output for mb file :-\n\n",hashop)

import hashlib
filename ="kb file.txt"
f=open(filename,"rb")
bytes = f.read()
start_time=time.time()
hashop = hashlib.sha512(bytes).hexdigest();[5]
print("\ntime reqd for sha512 for kb file:- %s seconds" % (time.time()- start_time))
print("SHA 512 hash output for kb file :-\n\n",hashop)

import hashlib
filename ="mb file.txt"
f=open(filename,"rb")
bytes = f.read()
start_time=time.time()
hashop = hashlib.sha512(bytes).hexdigest();
print("\ntime reqd for sha512 for mb file:- %s seconds" % (time.time()- start_time))
print("SHA 512 hash output for mb file :-\n\n",hashop)
```

```
#SHA3-256
#https://pycryptodome.readthedocs.io/en/latest/src/hash/sha3_256.html
from Crypto.Hash import SHA3_256
hashobj = SHA3_256.new()
filename ="kb file.txt"
f=open(filename,"rb")
bytes = f.read()
hashobj.update(bytes)
start_time=time.time()
print (hashobj.hexdigest())
print("\ntime reqd for sha3-256 for kb file:- %s seconds" % (time.time()- start_time))

from Crypto.Hash import SHA3_256
hashobj = SHA3_256.new()
filename ="mb file.txt"
f=open(filename,"rb")
bytes = f.read()
hashobj.update(bytes)
start_time=time.time()
print (hashobj.hexdigest())
print("\ntime reqd for sha3-256 for mb file:- %s seconds" % (time.time()- start_time))
```

Q1. e) Measurements for **RSA algorithm** with 2048 bit key

- Time taken for key generation: -170.45 milliseconds
- Time taken for encryption for larger file per chunk: - 2964.71 microseconds
- Time taken for decryption for larger file per chunk: - 13953 microseconds

- Time taken for encryption for small file per chunk: - 2204microseconds
- Time taken for decryption for small file chunk: - 8650microseconds

Speed per byte for encryption: - 13.75 microseconds
Speed per byte for decryption: - 33.78 microseconds

Source Code: -

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import time
start=time.time()
key = RSA.generate(2048)
print(time.time()-start)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)

pubkey = key.publickey().export_key()
public= open("public.pem","wb")
public.write(key.publickey().export_key() )
```

```
encryptdata = b''
blocksize = 16

with open("kb file.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        publickey = RSA.importKey(open("public.pem").read())
        paddedpub = PKCS1_OAEP.new(publickey)
        start=time.time()
        encryptdata =encryptdata+ paddedpub.encrypt(block)
        print(time.time()-start)
        block = infile.read(blocksize)

infile.close()

with open("encryptedkb.txt",'wb') as outfile:
    outfile.write(encryptdata)

decryptdata = b''
blocksize = 256
with open("encryptedkb.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        privatekey = RSA.importKey(open('private.pem').read())
        paddedpriv = PKCS1_OAEP.new(privatekey)
        start=time.time()
        decryptdata+=paddedpriv.decrypt(block)
        print(time.time()-start)
        block = infile.read(blocksize)

infile.close()
with open("decryptedkb.txt",'wb') as outfile:
    outfile.write(decryptdata)

encryptdata = b''
blocksize = 16

with open("mb file.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        publickey = RSA.importKey(open("public.pem").read())
        paddedpub = PKCS1_OAEP.new(publickey)
        encryptdata =encryptdata+ paddedpub.encrypt(block)
        block = infile.read(blocksize)

infile.close()

with open("encryptedmb.txt",'wb') as outfile:
    outfile.write(encryptdata)
```

```
decryptdata = b''
blocksize = 256
with open("encryptedmb.txt","rb") as infile:
   block = infile.read(blocksize)
   while block:
      privatekey = RSA.importKey(open('private.pem').read())
      paddedpriv = PKCS1_OAEP.new(privatekey)
      decryptdata+=paddedpriv.decrypt(block)
      block = infile.read(blocksize)

infile.close()
with open("decryptedmb.txt",'wb') as outfile:
   outfile.write(decryptdata)
```

Q1. f) Measurements for **RSA algorithm** with 3072-bit key

- Time taken for key generation: - 259.75 milliseconds
- Time taken for encryption for larger file: - 4065 microseconds
- Time taken for decryption for larger file: - 36460 microseconds

- Time taken for encryption for smaller file: - 3720 microseconds
- Time taken for decryption for smaller file: - 28075 microseconds

- Speed per byte for encryption: - 23.2 microseconds
- Speed per byte for decryption: - 73.11 microseconds

Source Code: -

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import time
start=time.time()
key = RSA.generate(3072)
print(time.time()-start)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)[1]

pubkey = key.publickey().export_key()
public= open("public.pem","wb")
public.write(key.publickey().export_key() [1])

encryptdata = b''
blocksize = 16

with open("kb file.txt","rb") as infile:
   block = infile.read(blocksize)
   while block:
      publickey = RSA.importKey(open("public.pem"). read())
```

```
        paddedpub = PKCS1_OAEP.new(publickey)[1]
        encryptdata =encryptdata+ paddedpub.encrypt(block)
        block = infile.read(blocksize)

infile.close()

with open("encryptedkb.txt",'wb') as outfile:
    outfile.write(encryptdata)

decryptdata = b''
blocksize = 384
with open("encryptedkb.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        privatekey = RSA.importKey(open('private.pem'). read())
        paddedpriv = PKCS1_OAEP.new(privatekey)
        decryptdata+=paddedpriv.decrypt(block)
        block = infile.read(blocksize)

infile.close()
with open("decryptedkb.txt",'wb') as outfile:
    outfile.write(decryptdata)

encryptdata = b''
blocksize = 16

encryptdata = b''
blocksize = 16

with open("mb file.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        publickey = RSA.importKey(open("public.pem"). read())
        paddedpub = PKCS1_OAEP.new(publickey) [1]
        encryptdata =encryptdata+ paddedpub.encrypt(block)
        block = infile.read(blocksize)

infile.close()

with open("encryptedmb.txt",'wb') as outfile:
    outfile.write(encryptdata)

decryptdata = b''
blocksize = 384
with open("encryptedmb.txt","rb") as infile:
    block = infile.read(blocksize)
    while block:
        privatekey = RSA.importKey(open('private.pem').read())
        paddedpriv = PKCS1_OAEP.new(privatekey)[1]
        decryptdata+=paddedpriv.decrypt(block)
        block = infile.read(blocksize)
```

```
infile.close()
with open("decryptedmb.txt",'wb') as outfile:
    outfile.write(decryptdata)
```

Q1. g) Measurements for **DSA Algorithm** with 2048 bit key: -

- Time taken to generate key: -2824.3 microseconds

- Time taken to generate signature for larger file: - 4020.9 microseconds
- Time taken for verification of larger file: - 4534.9 microseconds

- Time taken to generate signature for small file: - 1268.5 microseconds
- Time taken for verification of smaller file: -1391.05 microseconds

- Speed per byte for generation: -1.23 microseconds
- Speed per byte for verification: -1.35 microseconds

Source Code: -

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa
import time
start=time.time()
private_key = dsa.generate_private_key(key_size=2048,backend=default_backend())
print(time.time()-start)

f=open("kb file.txt","rb")
data=f.read()

start=time.time()
signature = private_key.sign(data,hashes.SHA256())
print(time.time()-start)

public_key = private_key.public_key()
start=time.time()
public_key.verify(signature,data,hashes.SHA256())
print(time.time()-start)

#MB file

f=open("mb file.txt","rb")
datamb=f.read()

start=time.time()
signature = private_key.sign(datamb,hashes.SHA256())
print(time.time()-start)
public_key = private_key.public_key()
```

start=time.time()
public_key.verify(signature,datamb,hashes.SHA256())
print(time.time()-start)

#https://pycryptodome.readthedocs.io/en/latest/src/public_key/dsa.html

Q1. h) Measurements for **DSA Algorithm** with 3072 bit key: -

- Time taken to generate key: - 1.328 seconds

- Time taken to generate Signature for larger file: - 5163.9 microseconds
- Time taken for verification of larger file: - 5742.9 microseconds

- Time taken to generate signature for smaller file: - 1933.8 microseconds
- Time taken for verification of smaller file: - 2275.66 microseconds

- Speed per byte for generation: -1.887 microseconds
- Speed per byte for verification: -2.22 microseconds

Source Code: -

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa
import time
start=time.time()
private_key = dsa.generate_private_key(key_size=3072,backend=default_backend())
print(time.time()-start)

f=open ("kb file.txt","rb")
data=f.read()

start=time.time()
signature = private_key.sign(data,hashes.SHA256())
print(time.time()-start)

public_key = private_key.public_key()
start=time.time()
public_key.verify(signature,data,hashes.SHA256()) [6]
print(time.time()-start)

#MB file

f=open("mb file.txt","rb")
datamb=f.read()

start=time.time()
signature = private_key.sign(datamb,hashes.SHA256())
print(time.time()-start)
public_key = private_key.public_key()

start=time.time()
public_key.verify(signature,datamb,hashes.SHA256()) [6]
print(time.time()-start)

Observations: -

Please find below observations which can be made based on the performance of various algorithms.

- how encryption and decryption times differ for a given encryption algorithm;

Explanation: -
The time required for encryption is faster than decryption. In case of AES, during encryption the mix columns process uses much smaller coefficients than inverse mix columns. The subkeys which are generated from the 128-bit key are used in the order in which they are generated i.e. K [0], K [1] …K [9]. In case of decryption there is an overhead time which is required to reverse the order of keys which are imputed to the algorithm.
It can also be observed that encryption and decryption times for block cipher mode CTR is approximately the same. This is because the block cipher is used in the forward direction for both operations of encryption and decryption. For operations in AES CBC mode, the block cipher is used in inverse mode. Hence the time required for decryption is greater than the time required for encryption.
The slowest times are recorded for RSA algorithm as it involves the overhead calculations of using very large prime numbers. The algorithm for asymmetric and symmetric encryptions is very different. In asymmetric algorithms the public key is available to everyone, which gives the adversary a starting point to execute brute force attack. To prevent any adversary from being able to guess the private key, very large prime numbers along with modulo operations in the order of 2^{2048}. Decryption times are greater than encryption time for RSA algorithm because of the modular exponentiation which is performed. The decryption exponent of RSA is almost as long as modulus. Thus, doubling the modular size makes encryption take twice as long, but makes decryption take four times as long.
Such concerns are not considered in case of algorithms like AES as the adversary does not have any hint where to start in case of a brute force attack.

- how per byte speed changes for different algorithms between small and large files;

Explanation: -
The observation which can be made is that speed per byte of the larger file is always greater than the smaller file used as an input for the algorithm. This happens because the larger file contains an order of 2^{20} bytes while the smaller file contains 2^{10} bytes. In case of AES, the algorithm performs encryption and decryption on blocks of size 128 bytes. Hence the number of blocks which will be required for encryption and decryption operations will be more in case of larger file.
The speed per byte operation for AES algorithm is the fastest as it follows a symmetric key mechanism. AES operation in CTR mode has the fastest times as the counter mode uses the same block cipher in encryption and decryption without performing any inversions. Hash calculations are faster than asymmetric modes such as RSA because they produce a fixed length output irrespective of the input size. The time can vary only in cases where key size is varied for different types of hashing algorithms such as SHA256, SHA 512 etc. DSA algorithm speeds are slower than hashing algorithm as DSA algorithm includes hashing of plaintext as part of the process.

- how key generation, encryption, and decryption times differ with the increase in the key size;

Explanation: -
According to observations it can be concluded that an increase in key size increases the amount of time it takes for encryption and decryption operations. The amount of time which gets increased depends of the algorithm being used. Taking the case of AES algorithm in CTR mode, it can be seen that an increase in

key size from 128 to 256 did not lead to a major increase in speed per byte times. This occurs because AES is a symmetric algorithm. For asymmetric algorithms like RSA and DSA, the difference in speed per byte operations is much more noticeable. Increasing the key size of RSA algorithm from 2048 to 3072 lead to an increase in speed per byte times by a factor of 1.5.
It can also be stated that key generation time increases when key size is increased. This happens because the system requires more time to create a key of larger byte size. Using a larger key lead to better security as it is harder to crack. The entire cycle of cryptography revolves around finding the key being used. If the key can be acquired for any algorithm, the adversary will be able to decrypt or encrypt any piece of text and hence there would not be any security left in the algorithm.

- how hashing time differs between the algorithms and with increase of the hash output size;

Explanation: -
The SHA algorithms which are being used in the problem are SHA-256, SHA-512 and SHA3-256. According to observations stated above, it can be concluded that SHA-512 is faster than SHA-256 algorithm for a larger input file. For smaller sized files, the time between two algorithms is comparable. This happens because SHA performs one round of compression function for a smaller sized file. The speed improvement can be noticed in larger files because multiple iterations of the SHA algorithm are being used. Each iteration performs 80 rounds of compression functions in SHA-512 as compared to SHA-256 algorithm where 64 rounds of compression functions are used.
SHA3-256 is an algorithm which has been designed for providing security to input files of the order 2^{2048} or higher. There are many additional steps which are executed inside the algorithm that are not required according to current standards. Hence the hashing time for this algorithm is much larger than the other two algorithms.

- how performance of symmetric key encryption (AES), hash functions, and public-key encryption (RSA) compare to each other.

Explanation: -
According to observations it can be said that the performance of symmetric key algorithm is fastest followed by hash functions and then in asymmetric algorithms like RSA and DSA. Symmetric key algorithms are designed to use a single key for both decryption and encryption purposes. Asymmetric key encryption uses a public key for encryption and private key for decryption. Since public key is made available to all users, it becomes an additional task to prevent the key from getting detected by adversaries. Hence public key algorithms like RSA use very large prime numbers and modulo operations for preventing brute force attacks. This leads to large overhead calculations. This leads to slower speed per bytes and hence RSA should not be used for encryptions and decryptions of large input files. It is generally used for encryption of symmetric keys as it increases the security of applications.
Hash functions like SHA-256, SHA-512 perform better than RSA while they are slower than symmetric key algorithms. Hash functions are slower than AES algorithm when the input size is less because SHA uses 256 or 512 bits as input. They are much more efficient in case of larger file sizes and take much more time in case of smaller inputs like 64 bytes as a lot of operations are needed to ignore rest of the bits of SHA.

Q2. a)

Preconditions: -

- $s_1, s_2 \in S$
- $o \in O$ [4]

Delete_all_rights (s_1, s_2, o)
{
If ($r^* \in A_i [s_1, o]$ AND if ($own \notin A_i [s_2, o]$)), then

$A_{i+1}[s_2, o] = A_{i+1}[s_2, o] - \{r\}$
$(\forall (x, y) \in S*O - \{s_2, o\})$
$[A_{i+1}[x, y] = A_i [x, y]]$
}

The above function states that check whether s_1 has modify rights over object o and s_2 does not have own access on object o. If the condition is satisfied, delete rights from s_1 in the next transition state. In the next step we remove the set $\{s_2, o\}$ from the universe which had access over object o. In the last step, all access rights of other subject object pairs are added to the next transition state.

Q2. b)

Preconditions: -

- $s_1, s_2 \in S$
- $o \in O$.[4]

copy_all_rights (s_1, s_2, o)
{
while ($\exists$ r* $\in A_i [s_1, o]$)
{
$A_{i+1}[s_2, o] = A_i [s_1, o] \cup \{r\}$
}
}

The above function checks whether subject s_1 has rights which have copy grant access. While there exists such rights, copy these rights to subject s_2.

Works Cited

[1]. *https://pycryptodome.readthedocs.io/en/latest/index.html.* Accessed 10 03, 2018. https://pycryptodome.readthedocs.io/en/latest/src/examples.html#.

[2]. "dlitz." Accessed 10 03, 2018. https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html.

[3]. "dlitz.net." Accessed 10 03, 2018. #https://www.dlitz.net/software/pycrypto/api/current/Crypto.Util.Counter-module.html.

[4]. "https://www.cs.purdue.edu/." Accessed 10 3, 2018. https://www.cs.purdue.edu/homes/clifton/cs526/HRU.pdf.

[5]. "Pycryptodome." Accessed 10 03, 2018. https://pycryptodome.readthedocs.io/en/latest/src/hash/sha3_256.html.

[6]. "Pycryptodome." Accessed 10 03, 2018. #https://pycryptodome.readthedocs.io/en/latest/src/public_key/dsa.html.