Andrew Audrain, Joseph Buchschacher, Thomas Mintun, and Samuel Scalise

Task 1

We decided to use python and the cryptographic library cryptography.io . In task one we did simple tests to test if SHA, AES, and random numbers were easy to do. For AES, we encrypted and decrypted a set message following the structure of the AES example in the cryptography.io websites notes. For the random numbers we used python's operating system method to generate random bytes. This generates random bytes of a selected byte length that can be converted into numbers. For SHA, we followed the structure given in cryptography.io's notes to create a digest for two different byte messages.

Code:

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
language = "Python"
security_suite = "cryptography and openssl"
task = "task 1"

print("\nLanguage:\t", language, "\nCryptographic Library:\t", security_suite,
"\nTask:\t", task)
print("Resources:\tClass Notes and Cryptography.io")

print("\nRandom Number Test")
rbytes1 = os.urandom(128)
rbytes2 = os.urandom(256)
print(rbytes1)
print(rbytes2)

print("\nAES Test")
backend = default_backend()
key = os.urandom(32)
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
encryptor = cipher.encryptor()
pt = b'Hello This is the secret message'
print("Plain Text:\t", pt)
ct = encryptor.update(pt) + encryptor.finalize()
decryptor = cipher.decryptor()
ptNew = decryptor.update(ct) + decryptor.finalize()
print("Cipher Text:\t", ct)
print("Decrypted Plain Text:\t", ptNew)

print("\nSHA 256 Test")
print("input 1:\t", b'abc')
print("input 1 size:\t", b'abc'.__sizeof__())
print("input 2:\t", b'123')
print("input 2 size:\t", b'123'.__sizeof__())
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
```

```
digest.update(b'abc')
digest.update(b'123')
digest.finalize()
print("size of digest:\t", digest.__sizeof__())
```

Task 2

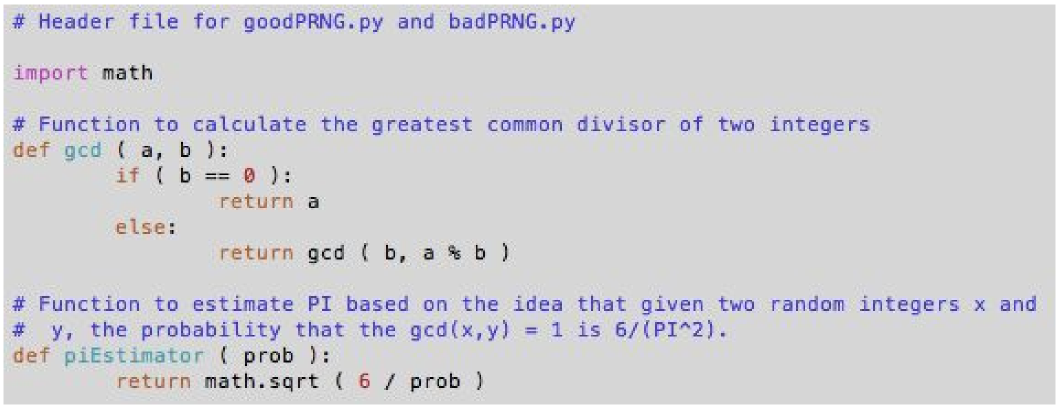
Task 2 Objective:
The objective of task 2 was to test some PRNGs. Given two random numbers *x* and *y*, the probability that gcd(x,y) = 1 is $6/\pi^2$. In other words, when generating a pair of random numbers, the probability that those two numbers are coprimes should be $6/\pi^2$. Knowing this, it is then possible to estimate the value of π by calculating the before-mentioned probability *P*, treating π like a variable, and solving for π via the formula $P = 6/\pi^2$ --> $\pi = \sqrt{(6 / P)}$. Once this formula is deduced, the probability just needs to be calculated and plugged in.

Procedure and Source Code:
I used the following two different PRNGs to estimate the value of π:

- "Bad" PRNG: a linear congruential generator with the values a=13, c=0, m=31, and X_0=1. Method is not cryptographically secure.
- "Good" PRNG: Python's SystemRandom() function to generate cryptographically secure random integers within a certain range. Recommended both by Python and our cryptography environment, cryptography.io, as a cryptographically secure way to generate pseudorandom numbers. It has the same API as random(), but uses os.urandom() under the hood to give the generator access to more sources of entropy.

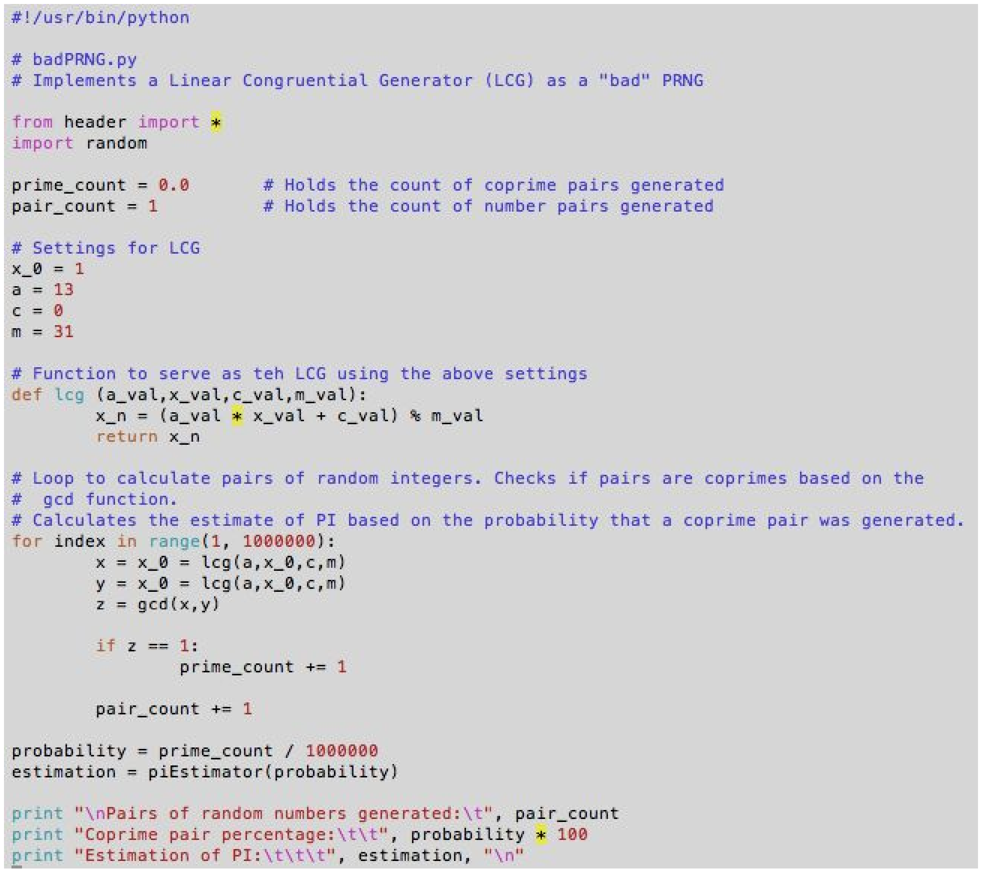
The following screenshots are of the source code used to test the PRNGs' effectiveness to estimate π:

- Header.py: Contains functions to solve for π and calculate the gcd of two integers.



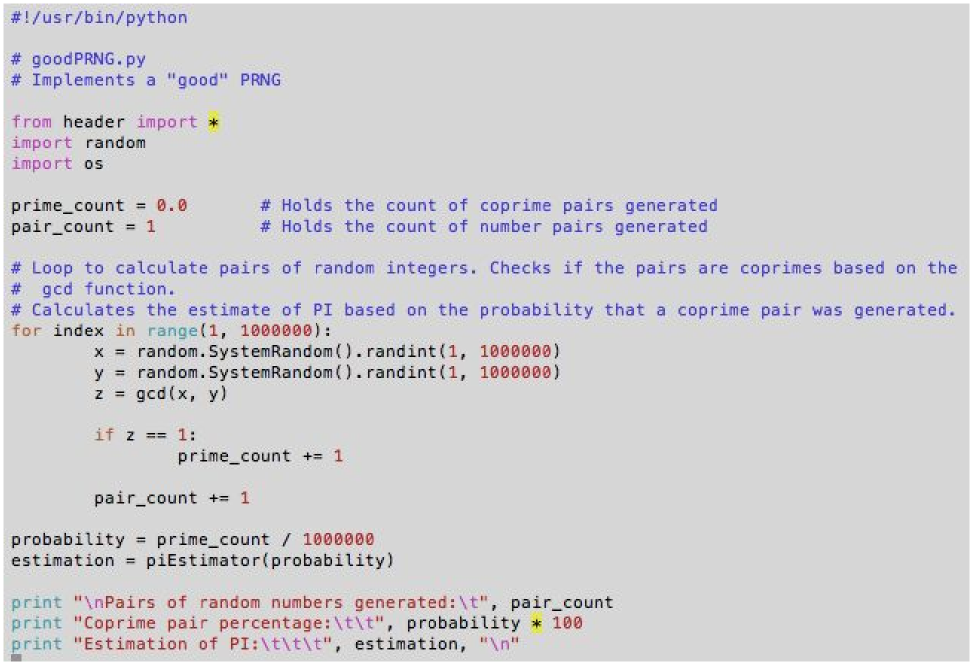
```
# Header file for goodPRNG.py and badPRNG.py

import math

# Function to calculate the greatest common divisor of two integers
def gcd ( a, b ):
        if ( b == 0 ):
                return a
        else:
                return gcd ( b, a % b )

# Function to estimate PI based on the idea that given two random integers x and
#  y, the probability that the gcd(x,y) = 1 is 6/(PI^2).
def piEstimator ( prob ):
        return math.sqrt ( 6 / prob )
```

- badPRNG.py



```
#!/usr/bin/python

# badPRNG.py
# Implements a Linear Congruential Generator (LCG) as a "bad" PRNG

from header import *
import random

prime_count = 0.0       # Holds the count of coprime pairs generated
pair_count = 1          # Holds the count of number pairs generated

# Settings for LCG
x_0 = 1
a = 13
c = 0
m = 31

# Function to serve as teh LCG using the above settings
def lcg (a_val,x_val,c_val,m_val):
        x_n = (a_val * x_val + c_val) % m_val
        return x_n

# Loop to calculate pairs of random integers. Checks if pairs are coprimes based on the
#  gcd function.
# Calculates the estimate of PI based on the probability that a coprime pair was generated.
for index in range(1, 1000000):
        x = x_0 = lcg(a,x_0,c,m)
        y = x_0 = lcg(a,x_0,c,m)
        z = gcd(x,y)

        if z == 1:
                prime_count += 1

        pair_count += 1

probability = prime_count / 1000000
estimation = piEstimator(probability)

print "\nPairs of random numbers generated:\t", pair_count
print "Coprime pair percentage:\t\t", probability * 100
print "Estimation of PI:\t\t\t", estimation, "\n"
```

- goodPRNG.py: Program to find *P* using a cryptographically secure PRNG.



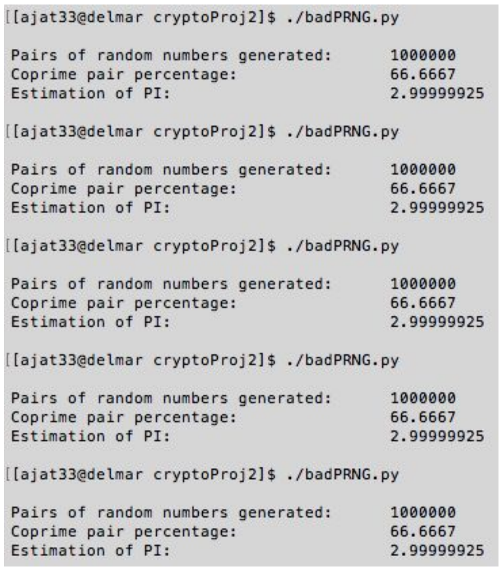
```
#!/usr/bin/python

# goodPRNG.py
# Implements a "good" PRNG

from header import *
import random
import os

prime_count = 0.0       # Holds the count of coprime pairs generated
pair_count = 1          # Holds the count of number pairs generated

# Loop to calculate pairs of random integers. Checks if the pairs are coprimes based on the
#  gcd function.
# Calculates the estimate of PI based on the probability that a coprime pair was generated.
for index in range(1, 1000000):
        x = random.SystemRandom().randint(1, 1000000)
        y = random.SystemRandom().randint(1, 1000000)
        z = gcd(x, y)

        if z == 1:
                prime_count += 1

        pair_count += 1

probability = prime_count / 1000000
estimation = piEstimator(probability)

print "\nPairs of random numbers generated:\t", pair_count
print "Coprime pair percentage:\t\t", probability * 100
print "Estimation of PI:\t\t\t", estimation, "\n"
```

In each program, a loop is run 1,000,000 times because I am a fan of overkill. Each run through the loop, a pair of random integers is generated, and the gcd of those two integers is calculated.

If the gcd == 1, then a the coprime counter variable, coprime_count, gets incremented by one. Then *P* is calculated by *P* = coprime_count / 1,000,000 after the loop has reached its end. This *P* value is then plugged into the piEstimator function to estimate the value of **π**.

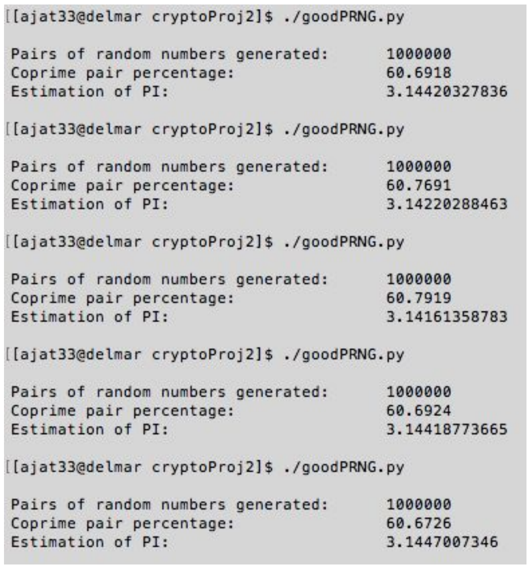
Results:
The following screenshots are examples of the code being run five times each:

badPRNG.py



```
[[ajat33@delmar cryptoProj2]$ ./badPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                66.6667
 Estimation of PI:                       2.99999925

[[ajat33@delmar cryptoProj2]$ ./badPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                66.6667
 Estimation of PI:                       2.99999925

[[ajat33@delmar cryptoProj2]$ ./badPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                66.6667
 Estimation of PI:                       2.99999925

[[ajat33@delmar cryptoProj2]$ ./badPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                66.6667
 Estimation of PI:                       2.99999925

[[ajat33@delmar cryptoProj2]$ ./badPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                66.6667
 Estimation of PI:                       2.99999925
```

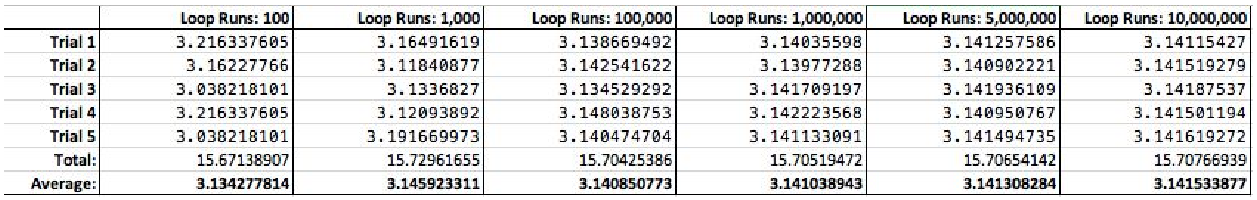
goodPRNG.py



```
[[ajat33@delmar cryptoProj2]$ ./goodPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                60.6918
 Estimation of PI:                       3.14420327836

[[ajat33@delmar cryptoProj2]$ ./goodPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                60.7691
 Estimation of PI:                       3.14220288463

[[ajat33@delmar cryptoProj2]$ ./goodPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                60.7919
 Estimation of PI:                       3.14161358783

[[ajat33@delmar cryptoProj2]$ ./goodPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                60.6924
 Estimation of PI:                       3.14418773665

[[ajat33@delmar cryptoProj2]$ ./goodPRNG.py

 Pairs of random numbers generated:      1000000
 Coprime pair percentage:                60.6726
 Estimation of PI:                       3.1447007346
```

The bad PRNG.py file did not need to be run 5 times nor did its loop need to be run 1,000,000 times. I just did it to be consistent with the goodPRNG.py testing, though the good PRNG's loop did not need to be done 1,000,000 times either. The numbers I got with a smaller sample size remained consistent with those I'm showing here (see below table). This principle regarding the relationship of **π** with *P* holds regardless of the amount of primes generated. The bad PRNG was clearly not a ideal tool to estimate the value of **π**.

Since the results for the bad PRNG are deterministic, they don't change from run to run. The good PRNG provides a much closer estimate to the value of **π**. The more runs through the loop one does, the closer the estimate is to **π**. The table below breaks down different runs using the good PRNG with different sized loops and shows the average value of **π** based 5 runs. Regardless of the size of the loop, the estimate of **π** remains in the ballpark for **π**'s actual value and is consistently closer than the estimate provided by the bad PRNG.



	Loop Runs: 100	Loop Runs: 1,000	Loop Runs: 100,000	Loop Runs: 1,000,000	Loop Runs: 5,000,000	Loop Runs: 10,000,000
Trial 1	3.216337605	3.16491619	3.138669492	3.14035598	3.141257586	3.14115427
Trial 2	3.16227766	3.11840877	3.142541622	3.13977288	3.140902221	3.141519279
Trial 3	3.038218101	3.1336827	3.134529292	3.141709197	3.141936109	3.14187537
Trial 4	3.216337605	3.12093892	3.148038753	3.142223568	3.140950767	3.141501194
Trial 5	3.038218101	3.191669973	3.140474704	3.141133091	3.141494735	3.141619272
Total:	15.67138907	15.72961655	15.70425386	15.70519472	15.70654142	15.70766939
Average:	**3.134277814**	**3.145923311**	**3.140850773**	**3.141038943**	**3.141308284**	**3.141533877**

Task 3

The objective of task 3 was to generate data files of various size filled with random data, apply SHA-256 hash function to this data, and apply another hashing function from the library to the same data for comparison. Our code applies the hash functions to the random data files several times many times and uses the average of those executions to function call times to approximate how many hash functions could be done in one second. The other hash function the program uses cryptography.io's implementation of the MD5 hash algorithm.

Code:

```
import os
import time
import sys
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes

def bytes(x):
    return (sys.getsizeof(x) - 33)

def file_gen(file_name, length):
    # generate a file of random data
    f = open(file_name, "wb")
    pt = os.urandom(length)
    print(str(bytes(pt)) + " byte file generated")
    f.write(pt)
    f.close()
    return

def sha_hash(file_name):
    # read data file and create a hash
    shaHash = hashes.Hash(hashes.SHA256(), backend=default_backend())
    f = open(file_name, "rb")
    data = f.read()
    f.close()

    # Record time while generating hash based on the data
    start_time = time.perf_counter_ns()
    shaHash.update(data)
    shaHV = shaHash.finalize()
    elapse = time.perf_counter_ns() - start_time

    # save hash as a file
    f = open(file_name + "sha", "wb")
    f.write(shaHV)
    f.close()

    # return time taken to create hash from data
    #print(str(bytes(shaHV)) + " byte SHA-256 hash created in " + str(elapse) + " ns")
    return elapse
```

```
def md5_hash(file_name):
    # read data file and create a hash
    md5Hash = hashes.Hash(hashes.MD5(), backend=default_backend())
    f = open(file_name, "rb")
    data = f.read()
    f.close()

    # Record time while generating hash based on the data
    start_time = time.perf_counter_ns()
    md5Hash.update(data)
    md5HV = md5Hash.finalize()
    elapse = time.perf_counter_ns() - start_time

    # save hash as a file
    f = open(file_name + "md5", "wb")
    f.write(md5HV)
    f.close()

    # return time taken to create hash from data
    return elapse

runs = 100
filesize = [8, 16, 32, 500, 1000, 2000] #length in bytes of the data files to be
created

# Generate several files of different lengths
for i in range(6):
    file_gen("file" + str(i), filesize[i])
print()

# Hash the file with SHA256 multiple times and calculate hashes/sec from the average
for i in range(6):
    shaSum = 0
    for j in range(len(filesize)):
        shaSum = shaSum + sha_hash("file" + str(i))
    shaAvg = shaSum / runs
    print(str(filesize[i]) + " byte file hashed with SHA256 " + str(runs) + " times")
    print("\t" + '{0:.3f}'.format(shaAvg) + " ns/hash")
    shaPerSec = 1 / (shaAvg * 0.000000001)  # Calculate hash/sec from average ns
    print("\t" + '{0:.3f}'.format(shaPerSec) + " hash/s")
print()

# Hash with the file with MD5 multiple times and calculate hashes/sec from the average
for i in range(6):
    md5Sum = 0
    for j in range(len(filesize)):
        md5Sum = md5Sum + md5_hash("file" + str(i))
    md5Avg = md5Sum / runs
    print(str(filesize[i]) + " byte file hashed with MD5 " + str(runs) + " times")
    print("\t" + '{0:.3f}'.format(md5Avg) + "\tns/hash")
    md5PerSec = 1 / (md5Avg * 0.000000001) # Calculate hash/sec from average ns
    print("\t" + '{0:.3f}'.format(md5PerSec) + "\thash/s")
print()
```

The file will output data, SHA-256, and MD5 files for each file size being tested and shell output like the sample below. The time taken by the program to hash files of different sizes did not consistently correspond to to the size of the files, but function calls for larger files tended to take longer than those for smaller files. Compared to SHA-256, MD5 hash function calls through the cryptography.io library tend to take longer than SHA-256 calls and will produce a 16 byte hash value half the size of the 32 byte hash value being produced by SHA-256. All of these numbers may vary greatly depending on various factors such as the platform the code is being executed on, background processes running, and how the system is storing the execution data.

Sample Output:

```
8 byte file generated
16 byte file generated
32 byte file generated
500 byte file generated
1000 byte file generated
2000 byte file generated

8 byte file hashed with SHA256 100 times
       1744.830 ns/hash
       573121.737 hash/s
16 byte file hashed with SHA256 100 times
       2204.620 ns/hash
       453592.909 hash/s
32 byte file hashed with SHA256 100 times
       2032.210 ns/hash
       492075.130 hash/s
500 byte file hashed with SHA256 100 times
       2081.460 ns/hash
       480432.004 hash/s
1000 byte file hashed with SHA256 100 times
       2167.680 ns/hash
       461322.704 hash/s
2000 byte file hashed with SHA256 100 times
       2311.400 ns/hash
       432638.228 hash/s

8 byte file hashed with MD5 100 times
       2015.790     ns/hash
       496083.421   hash/s
16 byte file hashed with MD5 100 times
       1946.010     ns/hash
       513871.974   hash/s
32 byte file hashed with MD5 100 times
```

```
       2397.590     ns/hash
       417085.490   hash/s
500 byte file hashed with MD5 100 times
       1933.670     ns/hash
       517151.324   hash/s
1000 byte file hashed with MD5 100 times
       3239.230     ns/hash
       308715.343   hash/s
2000 byte file hashed with MD5 100 times
       3058.590     ns/hash
       326948.038   hash/s
```

In general, to find a collision through brute-force to a particular value for SHA-256 it would take an undetermined amount of time because there are too many variables involved. We assumed that the hash contains random data and not dictionary words. And we were able to calculate the time based on the machine that we are using. There is specially designed hardware that can crack SHA-256 faster. The equation that we used to calculate the time is:

$$2^{256} * \text{(Seconds to perform one hash)} = \text{TIME (seconds)}$$

Time to brute force:
8 byte file hashed with SHA256: 2.019414e+71 sec
16 byte file hashed with SHA256: 2.5527756e+71 sec
32 byte file hashed with SHA256: 2.5527756e+71 sec
500 byte file hashed with SHA256: 2.410166e+71 sec
1000 byte file hashed with SHA256: 2.510002e+71 sec
2000 byte file hashed with SHA256: 2.6764184e+71 sec

8 byte file hashed with MD5: 2.3341254e+71 sec
16 byte file hashed with MD5: 2.2533256e+71 sec
32 byte file hashed with MD5: 2.7762196e+71 sec
500 byte file hashed with MD5: 2.2390369e+71 sec
1000 byte file hashed with MD5: 3.7507721e+71 sec
2000 byte file hashed with MD5: 3.5416053e+71 sec

Task 4

The results shown from the code below show that AES encrypts and decrypts much faster than RSA. The AES results could encrypt and decrypt about 100 times more bytes a second than RSA. The Results also showed that AES decrypts about twice as fast as it encrypts, while RSA decrypts around 3 to 5 times faster than it encrypts. In AES the different size of the files didn’t affect the speed of bytes per second being encrypted and decrypted, but in RSA the smaller file was faster than the lager file generally. In RSA, the time resolution makes the program have to be run multiple times to check the trends. In AES, the encryption and decryption speed in bytes per second stayed relatively the same for different sized files with the same key size. The different key-sizes did change the speed of AES though. The 256-bit key was slightly slower than the 128-bit key. The 128-bit key could encrypt and decrypt about one or two more files a second than the 256-bit key version.

Code:

```
import os
import time
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes


def file_generator(file_name, file_size):

   """Generates random binary files of a certain Byte size"""

   f = open(file_name, 'wb')
   f.write(os.urandom(file_size))
   f.close()
   return


def file_write(file_name, data):

   """ Writes and replaces data in a file"""

   f = open(file_name, 'wb')
   f.write(data)
   f.close()
   return


def file_time(cipher_time, size):

   """This function determines how many files can be ciphered in 1 second"""

   # starts a count and time sum
   count = 0
   time_total = 0
   # loops with the time sum is less than 1000000000 nanoseconds (1 second)
   while time_total < 1000000000:
       count = count + 1
       time_total = cipher_time + time_total
   # count needs to be subtracted by 1 because we need the number of files under 1
second
   count = count - 1
```

```
    print("\t\t\tThe file size in Bytes:\t", size)
    print("\t\t\tThe number of files ciphered in 1 second:\t", count)
    print("\t\t\tBytes per second:\t", (size / cipher_time * 1000000000))
    return


def aes_key_size(key_size, file_name):

    """This runs the AES cipher needed based on key size"""

    # reads in file and sets daa into a variable
    f = open(file_name, 'rb')
    pt_original = f.read()
    f.close()
    size = pt_original.__sizeof__()

    # sets up the encryptor / decryptor
    backend = default_backend()
    key = os.urandom(key_size)
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
    # the timing and encryption of the pt
    encryptor_size_one = cipher.encryptor()
    cipher_time = time.perf_counter_ns()
    ct = encryptor_size_one.update(pt_original) + encryptor_size_one.finalize()
    cipher_time = time.perf_counter_ns() - cipher_time
    file_write(file_name, ct)
    print("\t\t\tEncryption Time in Seconds:\t", cipher_time / 1000000000)
    file_time(cipher_time, size)
    # opens file and reads the data into a variable
    f = open(file_name, 'rb')
    file = f.read()
    f.close()
    size = file.__sizeof__()
    # the timing and decryption of the file
    decryptor_size_one = cipher.decryptor()
    cipher_time = time.perf_counter_ns()
    pt_new = decryptor_size_one.update(file) + decryptor_size_one.finalize()
    cipher_time = time.perf_counter_ns() - cipher_time
    file_write(file_name, pt_new)
    print("\n\t\t\tDecryption Time in Seconds:\t", cipher_time / 1000000000)
    file_time(cipher_time, size)
    # checks to see if the original message and the decrypted message are the same
    status = pt_new == pt_original
    print("\n\t\t\tMessage Match:\t", status)
    return


def rsa_private_key_size(file_name):
    """This Function Performs RSA"""

    # opens the file for the given file name and puts its data to a variable
    f = open(file_name, 'rb')
    pt_original = f.read()
    f.close()
    size = pt_original.__sizeof__()

    private_key_size = 2048
    # size 2048 because that is the current standard rsa key size
    # private_key is the generated private key
    # used settings suggested by the library's site
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=private_key_size,
        backend=default_backend()
    )
    # public_key is the generated public key based on the private key
    public_key = private_key.public_key()
    # get starting time and then encrypts the file
    cipher_time = time.perf_counter_ns()
    cipher_text = public_key.encrypt(
        pt_original,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    # gets the total time encrypting, overwrites encrypted message to the file, and
sends that to the file_time function
    cipher_time = time.perf_counter_ns() - cipher_time
    file_write(file_name, cipher_text)
    print("\t\t\tEncryption Time in Seconds:", cipher_time / 1000000000)
    file_time(cipher_time, size)
    # opens up the overwritten file and sends its data to a variable. Also gets the
size of the new data
    f = open(file_name, 'rb')
    file = f.read()
    f.close()
    size = file.__sizeof__()
    # starts the timer and decrypts the message
    cipher_time = time.perf_counter_ns()
    pt_new = private_key.decrypt(
        file,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    # gets the total time, overwrites the file with the decrypted message, and sends
the time to the file_time function
    cipher_time = time.perf_counter_ns() - cipher_time
    file_write(file_name, pt_new)
    print("\n\t\t\tDecryption Time in Seconds:", cipher_time / 1000000000)
    file_time(cipher_time, size)
    # checks to see if the original message and the decrypted message are the same
    status = pt_original == pt_new
    print("\n\t\t\tMessage Match:", status)
    return


# file generation
file_one_name = 'fileone.txt'
file_two_name = 'filetwo.txt'
file_one_size = 12800000
file_two_size = 64000000

file_generator(file_one_name, file_one_size)
file_generator(file_two_name, file_two_size)

# AES
print("\nAES:")
print("\tAES 256 Bit Key:")
print("\t\tFile One:")
aes_key_size(32, file_one_name)

print("\n\t\tFile Two:")
aes_key_size(32, file_two_name)

print("\n\tAES 128 Bit Key:")
print("\t\tFile One:")
aes_key_size(16, file_one_name)

print("\n\t\tFile Two:")
aes_key_size(16, file_two_name)

# RSA
file_one_size = 50
file_two_size = 70
file_generator(file_one_name, file_one_size)
file_generator(file_two_name, file_two_size)

print("\nRSA:")
print("\t\tFile One:")
rsa_private_key_size(file_one_name)

print("\n\t\tFile Two:")
rsa_private_key_size(file_two_name)
```

Output:

```
AES:
        AES 256 Bit Key:
                File One:
                        Encryption Time in Seconds:  0.069163378
                        The file size in Bytes:  12800033
                        The number of files ciphered in 1 second:     14
                        Bytes per second:       185069517.57041132

                        Decryption Time in Seconds:  0.033331136
                        The file size in Bytes:  12800033
                        The number of files ciphered in 1 second:     30
                        Bytes per second:       384026305.0140265

                        Message Match:          True

                File Two:
                        Encryption Time in Seconds:  0.355677615
                        The file size in Bytes:  64000033
                        The number of files ciphered in 1 second:     2
                        Bytes per second:       179938321.39253408

                        Decryption Time in Seconds:  0.194350421
                        The file size in Bytes:  64000033
                        The number of files ciphered in 1 second:     5
                        Bytes per second:       329302260.6830371

                        Message Match:          True

        AES 128 Bit Key:
                File One:
                        Encryption Time in Seconds:  0.06380842
                        The file size in Bytes:  12800033
                        The number of files ciphered in 1 second:     15
                        Bytes per second:       200601002.1874856

                        Decryption Time in Seconds:  0.033712514
                        The file size in Bytes:  12800033
                        The number of files ciphered in 1 second:     29
                        Bytes per second:       379681948.3708631

                        Message Match:          True
                File Two:
                        Encryption Time in Seconds:  0.298115475
                        The file size in Bytes:  64000033
                        The number of files ciphered in 1 second:      3
                        Bytes per second:        214682022.12582222

                        Decryption Time in Seconds:  0.158808264
                        The file size in Bytes:  64000033
                        The number of files ciphered in 1 second:      6
                        Bytes per second:        403001905.49277717

                        Message Match:           True

RSA:
                File One:
                        Encryption Time in Seconds: 0.000320336
                        The file size in Bytes:  83
                        The number of files ciphered in 1 second:      3121
                        Bytes per second:        259102.94191099348

                        Decryption Time in Seconds: 0.00284953
                        The file size in Bytes:  289
                        The number of files ciphered in 1 second:      350
                        Bytes per second:        101420.23421406337

                        Message Match: True

                File Two:
                        Encryption Time in Seconds: 0.000182586
                        The file size in Bytes:  103
                        The number of files ciphered in 1 second:      5476
                        Bytes per second:        564117.7308227356

                        Decryption Time in Seconds: 0.002685852
                        The file size in Bytes:  289
                        The number of files ciphered in 1 second:      372
                        Bytes per second:        107600.86557263766

                        Message Match: True
```

Task 1 and 4 were completed by Samuel Scalise, task 2 was completed by Andrew Audrain, task 3 was coded by Joseph Buchschacher, and computations and analysis of task 3 was done by Thomas Mintun.