Design

We write our verifier and prover down below. We use the hashlib library for sha256 and time to get the total time for attestion. I created a read_memory fucntion to read the "memory" from the txt files provided to generate the hash

In [ ]:
```
import random
import hashlib
import time
import numpy as np
```

In [ ]:
```
### memory ready, just makes things easier so i dont have to copy and paste
### reads memory file content
def read_memory(file):
    with open(file, 'r') as file:
        memory = file.readlines()
    return memory
```

Implementation of the verifier and prover

In [ ]:
```
def verifier():
    ### nonce generation
    nonce = random.randint(0, 256)
    start_time = time.time() ## start of creation of nonce
    final_hash = prover(nonce)
    end_time = time.time() ## prover function returns final hash
    #### compute local hash
    memory = read_memory("random_numbers_16.txt")
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
    if final_hash.hexdigest() == verifier_hash.hexdigest():
        print("Test Passed")
    else:
        print("Test Failed")
    return (end_time - start_time)


def prover(nonce):
    memory = read_memory("random_numbers_16.txt")
    hash = hashlib.sha256()
    hash.update(str(nonce).encode())
    for line in memory:
        hash.update(str(line).encode())
    return hash
```

Step 1

The time needed for an attestation. We do 10 runs and calculate the average time

In [ ]:
```
totaltime = 0
for i in range(10):
    t = verifier()
    totaltime += t
avg = totaltime/10
print("The average of time to complete an attestation is", avg)
```

```
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
The average of time to complete an attestation is 0.00041582584381103513
```

Step 2a

This is the precompute function that combines the zeros_8 and attack_8 files and copying the non-zero parts of the former to the non-zero parts of the latter

In [ ]:
```
def precompute():
    zeros = read_memory("zeros_8.txt")
    attack = read_memory("attack_8.txt")
    for i in range(len(attack)):
        if i > (len(attack) // 2) - 1:
            attack[i] = zeros[i - (len(attack) // 2) ]
    with open("precompute_file.txt", 'w') as file3:
        file3.writelines(attack)
precompute()
```

STEP 2B

Here is the malicous prover that uses the precompute file generated along with the attack_8 file. We basically compare the zero parts of the attack file with the precompute file to get the correct part of memory needed for the hash to not get rid of or change the attack file.

In [ ]:
```
def malicous_prover(nonce):
    bad_mem = read_memory("attack_8.txt")
    memory = read_memory("precompute_file.txt")
    hash = hashlib.sha256()
    hash.update(str(nonce).encode())
    for i in range(len(memory)):
        if(i > len(memory) // 2 - 1):
            hash.update(str(memory[i]).encode())
    return hash
```

STEP 3

Here we do the attack and confirms that it passes the verification. We do the 10 runs and report the average

In [ ]:
```
def verifier_attack():
    ### nonce generation
    nonce = random.randint(0, 256)
    start_time = time.time() ## start of creation of nonce
    final_hash = malicous_prover(nonce)
    end_time = time.time() ## prover function returns final hash
    #### compute local hash
    memory_sub = read_memory("zeros_8.txt")
    memory = memory_sub[:(len(memory_sub) // 2)]
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
    if final_hash.hexdigest() == verifier_hash.hexdigest():
        print("Test Passed")
    else:
        print("Test Failed")
    return (end_time - start_time)
```

In [ ]:
```
totaltime_p2 = 0
for i in range(10):
    t_p2 = verifier_attack()
    totaltime_p2 += t_p2
avg_p2 = totaltime_p2/10
print("The average of time to complete an attestation is", avg_p2)
```

```
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
The average of time to complete an attestation is 0.000911092758178711
```

STEP 4

Here we modify our verifier with the new threshold. The way that I generate the threshold is that I do 30 runs with the verifier and prover and get the time that it takes for an attestation. Then, I obtain an array containing these times and I get stats about this array and create the threshold off of it. I make the threshold to be equal to the median + the standard deviation times some constant. I chose a random constant and can be changed.

In [ ]:
```
#### threashold creater
### run 30 times and get the average
def prover_correct(nonce):
    mmemory_sub = read_memory("zeros_8.txt")
    memory = mmemory_sub[:(len(mmemory_sub) // 2)]
    hash = hashlib.sha256()
```

```
    hash.update(str(nonce).encode())
    for line in memory:
        hash.update(str(line).encode())
    return hash

def verifier_thresh():
    ### nonce generation
    nonce = random.randint(0, 256)
    start_time = time.time() ## start of creation of nonce
    final_hash = prover_correct(nonce)
    end_time = time.time() ## prover function returns final hash
    #### compute local hash
    memory_sub = read_memory("zeros_8.txt")
    memory = memory_sub[:(len(memory_sub) // 2)]
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
    if final_hash.hexdigest() == verifier_hash.hexdigest():
        print("Test Passed")
    else:
        print("Test Failed")
    return (end_time - start_time)

threshold_array = []
for i in range(30):
    bruhtime = verifier_thresh()
    threshold_array.append(bruhtime)
threshold = np.median(threshold_array) + 1.3*np.std(threshold_array)
print("The threshold is ", threshold)
```

```
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
Test Passed
The threshold is  0.0004901355611038399
```

In [ ]:

```python
def verifier_mod():
    ### nonce generation
    passed = True
    nonce = random.randint(0, 256)
    start_time = time.time() ## start of creation of nonce
    final_hash = prover_correct(nonce)
    end_time = time.time() ## prover function returns final hash
    #### compute local hash
    memory_sub = read_memory("zeros_8.txt")
    memory = memory_sub[:(len(memory_sub) // 2)]
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
    if final_hash.hexdigest() == verifier_hash.hexdigest() and (end_time - sta
        passed = True
    else:
        passed = False
    return passed

def verifier_attack_mod():
    passed = True
    ### nonce generation
    nonce = random.randint(0, 256)
    start_time = time.time() ## start of creation of nonce
    final_hash = malicous_prover(nonce)
    end_time = time.time() ## prover function returns final hash
    #### compute local hash
```

```
    memory_sub = read_memory("zeros_8.txt")
    memory = memory_sub[:(len(memory_sub) // 2)]
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
    if final_hash.hexdigest() == verifier_hash.hexdigest() and (end_time - sta
        passed = True
    else:
        passed = False
    return passed
```

Step 5

Here we run the new modified verifier and prover. We get a true positive rate of 80% and a false positive rate of 20%.

In [ ]:
```
tp = 0
fp = 0
for i in range(5):
    bruh = verifier_mod()
    if bruh == True:
        tp += 1
print("True positive rate: ", (tp * 20), "%")
for i in range(5):
    bruh = verifier_attack_mod()
    if bruh == True:
        fp += 1
print("False positive rate: ", (fp * 20), "%")
```

```
True positive rate:  80 %
False positive rate:  20 %
```

Step 6

Here we calculate the noise. I used a random nromal distrubtion center around the time (either start or end time) and ill multiply it by 3 and add that to the start time. I found on average increase the noise by a constant factor while using a normal distrubtion centered around that time helped to reduce the true positive rate

In [ ]:
```
def verifier_mod_noise():
    ### nonce generation
    passed = True
    nonce = random.randint(0, 256)
    start_time = time.time() + 3*np.random.normal(loc=time.time())## start of 
    final_hash = prover_correct(nonce)
    end_time = time.time()  + 3*np.random.normal(loc=time.time())## prover fun
    #### compute local hash
    memory_sub = read_memory("zeros_8.txt")
    memory = memory_sub[:(len(memory_sub) // 2)]
    verifier_hash = hashlib.sha256()
    verifier_hash.update(str(nonce).encode())
    for line in memory:
        verifier_hash.update(str(line).encode())
```

```
    if final_hash.hexdigest() == verifier_hash.hexdigest() and (end_time - sta
        passed = True
    else:
        passed = False
    return passed

tp = 0
fp = 0
for i in range(10):
    bruh = verifier_mod_noise()
    if bruh == True:
        tp += 1
print("True positive rate: ", (tp * 10), "%")
```

```
True positive rate:  50 %
```

STEP 7

We cna improve the true positive rate by either increasing the sample size to normalize the threshold for the correct verifier function. We could also change our normal distrubution to a statistical model that is more representative of the time is takes for an attestation. One major thing we can do with this is to train and generate our threshold while using noise in our calculations. This would help smooth and normalize our threshold for the correct verifier to increase the true positive rate. Overall this methods should help increase the the true positive rate.