Quantum Bitcoin and Cheque

Ankan Mukherjee[1,*]

[1] *Indian Institute of Technology Bombay, Mumbai, India*

ABSTRACT

The goal of the project is to explore the vulnerabilities of the current classical algorithm for encryption and hashing of crypto-currencies when subjected to quantum computing and to design more secure quantum algorithms.

I. INTRODUCTION

We shall start with a background of the classical computations and then develop quantum methods for the same.

A. Background

Blockchain is the method used at present for secure peer to peer transactions through decentralized currencies[1]. It comprises peers broadcasting transactions on a decentralized ledger, secured by a digital signature. The signature can be generated uniquely by the use of a private key held by the node generating the signature only. The other peers can verify the authenticity of the signature using a widely available public key. Since copies of ledgers are possessed by all the peers on the network, it is imperative to develop a form of authenticity check on the ledgers. This is doing by using a special string called the nonce, which is in turn dependent on the contents of the ledger, to generate a hash satisfying some conditions. Mathematics show that brute force search is the only method for generating such a nonce. The probability of success is of the order of $\frac{1}{2^N}$, where N is the number of bits used for hashing. For a standard 256-bit hash, the probability of finding a special nonce is too low for all practical purposes. Thus, "guessing" a nonce is nearly next to impossible at an individual level. To increase the security even further, blocks of transactions with a valid nonce are linked together in the form of a chain to prevent any fraudulent manipulation of previous transactions. This is called a block chain. Quantum computing is a form of computing

* 190260008@iitb.ac.in

that uses the quantum mechanical states of particles, called qubits, to store information, instead of storing them on classical switches. Two special properties of qubits are that they can undergo superposition and entanglement. These properties allow special quantum algorithms to be developed, which speed an otherwise slow classical process. Quantum computers are speculated to posses serious threats to the classical computing algorithms[2], including those used in the security of bitcoins. Quantum algorithms like Grover's algorithm and a more generalized version of it can search for the nonce much faster than a classical computer.

B. Objectives

The aim of the project is to find loopholes in the present system of cryptocurrency[1] when subjected to quantum computation and to find methods to improve the security. The problem at hand involves understanding the current methods of encryption and hashing, following by designing a quantum algorithm to break the same. At the last leg of the project, we shall solve these problems by using quantum algorithms for hashing.

C. Applications

Our work may be used to develop and enhance secure quantum algorithms for crypto-currencies. The algorithms presented in our work can also be used to design a new crypto-currency system on a real quantum computer back-end.

D. System Requirements

The code for our work uses Qiskit on a Jupyter Notebook on Python. The following are the minimum system requirements to run our code.

Processors: Intel Atom processor or Intel Core™ i3 processor or higher

Disk space: 1 GB or higher

Operating systems: Windows 7 or later, macOS 10.12.6 or later, and Ubuntu 16.04 or later

Python: 3.6 or higher

Numpy: 1.20.0 or higher

Jupyter Lab/Notebook: 6.3.0 or higher

Pip or Anaconda to install the above packages.

II. BLOCKCHAIN

In this section we shall discuss the main ideas of the blockchain system[1]. The basic unit of currency is defined as a block containing information about a transaction. The transaction is signed using an encrypted signature. The signature requires a private key to generate and a public key for verification. Details of some encryption methods are discussed later. Once a block signature has been verified, it is considered as a "legal block". To prevent peers from double spending, it is imperative to store all records of previous transactions in the block. This is made possible by storing some information of the previous block in the next block. Further, to ensure the correct chronology of these blocks, a timestamp (epoch) is used in the block, which marks the time of creation of the block. Since each block contains information about the previous block, the timestamp will be linear in the chain. Any discrepancies regarding this is indicative of a fraudulent block. Finally, we come to the main point that is relevant to our work. How does one ensure, in such a decentralized system, that there is no manipulation of information? This is achieved by hashing. Hashing the information in the block produces a 256-bit long hash. It is easy to verify a hash from a block but very difficult to generate a block that produces a given hash[3]. Also, changing even a single character can cause a huge change in the hash. Details of hashing are discussed in more detail later Whether a block should be trusted or not is governed by the concept of "Proof of Work". This involves finding an arbitrary string, called the nonce, by brute-force, to be appended at the end of the block so that its hash starts with a certain minimum number of zeroes. To prevent manipulation of previous transactions, each block starts with the hash of the previous block. Resources in terms of CPU and electricity are used in huge quantities to generate these nonce. Any fraudulent party working individually to manipulate information in the blockchain will be unable to compete in terms of resources to find a proof of work for successive chains. This will in turn ensure that the longest chain is the most trusted one in case of a fork and the other short chains are automatically rejected. Figure II.1 shows the contents of a block. Figure II.2 shows a pictorial representation of a blockchain.

The working of a blockchain system is summarized below.

1. New transactions are broadcast to all nodes.

2. Each node collects new transactions into a block.

3. Each node works on finding a difficult proof-of-work for its block.

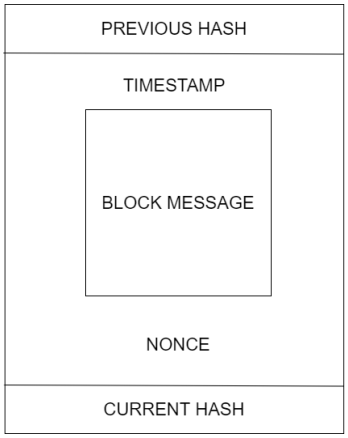

FIG. II.1. A typical block

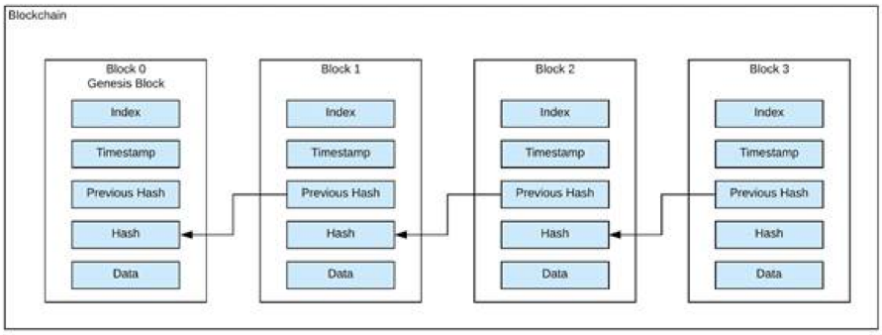

FIG. II.2. The structure of a block chain[4]

4. When a node finds a proof-of-work, it broadcasts the block to all nodes.

5. Nodes accept the block only if all transactions in it are valid and not already spent.

6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

III. CLASSICAL ALGORITHMS

In this section, we will take a look at each of the components of the blockchain implemented classically.

A. Encryption

Encryption is used to digitally sign transactions in a blockchain. Encryption is the process by which data is converted into a secure string of bits that can be uniquely decrypted. Encryption requires the use of a private key to decrypt an encrypted message and a public key to verify the encryption. Several encryption algorithms exists in the market, the most popular ones being the 3DES, AES and RSA[5].

B. Hashing

Hashing is the process by which a message of arbitrary lengths is converted to a bit string of fixed length. Unlike encryption, hashes are not unique. Several messages can have the same hash. Hashes must be easy to verify given a message. The reverse process, however, should be a hard one, i.e., given a hash, it should be practically impossible to find a message generating the hash. Hash collisions should also be rare, i.e., given a message and its hash, it should be practically impossible to find another message with the same hash. Finally, hashes must be distinctly different even for change of a single character in the message. Some commonly used hashing algorithms include SHA-1, Pearson Hash and SHA-256. In this paper, we will demonstrate the SHA-256. We will then design a surrogate hashing purpose that satisfies the aforementioned properties of hashes but is easy to simulate on a quantum computer.

SHA-256

This comprises using XOR and shift operators on registers pre-loaded with existing values. The code for the same is mentioned below. The code is written in python.

```
import math
import numpy as np
```

```
#The first 32 prime numbers are used to preset the hash registers by considering
    ↪ the fractional part of their square and cube roots
PRIMES=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
    ↪ 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
    ↪ 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
    ↪  233, 239, 241, 251, 257, 263, 269, 271,277,281,283,293,307,311]


#Right shift by n bits
def SHR_int(x,n):
    return x>>n
def SHR(s,n):
    x=int(s,2)
    return str(bin(x>>n))[2:].zfill(32)


#Right rotate by n bits
def ROTR_int(x,n):
    return (x>>n)|(x<<(32-n))&0xFFFFFFFF
def ROTR(s,n):
    x=int(s,2)
    return str(bin((x>>n)|(x<<(32-n))&0xFFFFFFFF))[2:].zfill(32)


#Linear combination in terms of XOR
def SIG0(x):
    return ROTR_int(x,2)^ROTR_int(x,13)^ROTR_int(x,22)
def SIG1(x):
    return ROTR_int(x,6)^ROTR_int(x,11)^ROTR_int(x,25)
    #return x
def sig0(s):
    x=int(s,2)
    return str(bin(ROTR_int(x,7)^ROTR_int(x,18)^SHR_int(x,3)))[2:].zfill(32)
def sig1(s):
    x=int(s,2)
```

```
    return str(bin(ROTR_int(x,17)^ROTR_int(x,19)^SHR_int(x,10)))[2:].zfill(32)
def che(x,y,z):
    return (x&y)|(~x&z)
def maj(x,y,z):
    return (x&y)^(y&z)^(z&x)

#Converts message to ASCII
def toAscii(message):
    return ''.join(str(bin(ord(c)))[2:].zfill(8) for c in message)

#Pads the message with zeroes till length is a multiple of 512 and then appends
    ↪ the length at the end
def padding(message_ascii):
    l=len(message_ascii)
    size=(l//512+1)*512
    pad=message_ascii+'1'
    for j in range(size-len(pad)-64):
        pad=pad+'0'
    pad=pad+str(bin(len(toAscii(message))))[2:].zfill(64)
    return pad,size

#Creates the message block
def message_block(message):
    padded_message=padding(toAscii(message))[0]
    nblocks=padding(toAscii(message))[1]//512
    w=[[None for _ in range(64)] for _ in range(nblocks)]
    for i in range(nblocks):
        for j in range(16):
            w[i][j]=padded_message[512*i+32*j:512*i+32*(j+1)]
        for j in range(16,64):
            w[i][j]=str(bin((int(sig1(w[i][j-2]),2)+int(w[i][j-7],2)+int(sig0(w[i
                ↪ ][j-15]),2)+int(w[i][j-16],2))%int(2**32)))[2:].zfill(32)
```

```
    return w

#Compression where registers are continuously updated with values of the linear
    ↪ functions above to generate the hash
def compress(w,H0):
    H=H0
    for j in range(len(w)):
        wj=int(w[j],2)
        T1=(K[j]+wj+SIG1(H[4])+che(H[4],H[5],H[6])+H[7])%int(2**32)
        T2=(SIG0(H[0])+maj(H[0],H[1],H[2]))%int(2**32)
        H=[(T1+T2)%int(2**32) ]+H[:-1]
        H[4]=(H[4]+T1)%int(2**32)
    return [(H0[i]+H[i])%int(2**32) for i in range(len(H))]

#Generates the hash in hexadecimal
def hashgen(message,H0):
    w=message_block(message)
    H=H0
    for i in range(len(w)):
        H=compress(w[i],H)
    return ''.join(str(hex(x))[2:].zfill(8) for x in H)

#Actually initializing the registers
K=[]
H=[]
for i in range(64):
    K.append(int(math.modf((PRIMES[i])**(1/3))[0]*(2**32)))
for i in range(8):
    H.append(int(math.modf((PRIMES[i])**(1/2))[0]*(2**32)))

#Inputiing the message from the user
message=input().rstrip()
```

```
#Printing the hash
print(hashgen(message,H))
```

Now we shall check the values of the hash for some sample strings. This can be done by running the code given above.

Message: `Hello World`

Hash: `a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e`

Making a slight change

Message: `Hello Workd`

Hash: `5ab45e33f10c8c9eb8005ba117fbd6e9d4ce3e61d5b199c08cbb7bbebf58cf68`

We see that the hash changes significantly.

Surrogate Hash Function: Pearson Hashes and LFSRs

The SHA-256 is a rather cumbersome method of hashing that requires significant amount of time and memory when simulating using qiskit. To overcome this difficulty, we will use a surrogate hash called the Pearson Hash[6]. The Pearson Hash makes use of a lookup table, which can be achieved using Linear Feedback Shift Registers (LFSR)[7]. These are easy to implement in qiskit[8], and the hash so generated satisfies the properties of a good hash. Since the purpose of our project is the search for nonce for a given hashing algorithm and not hashing, using a surrogate hashing algorithm will make no difference.

An LFSR comprises shift registers in which, at each iteration, the elements shift to the adjacent register (like in an ordinary shift register) and the first register takes the value of a linear function of some of the registers. Galois Theory of Fields dictates the working of these LFSRs. The values which are XORed are called taps. The LFSR structure is shown in Figure III.1 where $\oplus$ denotes XOR.

Pearson hashing uses two LFSRs. It works by taking the XOR of a character with the register and then performing two different sets of LFSRs. While LFSRs by themselves are reversible, it is impossible to extract the characters of a message from the hashing owing to the repeated XOR operations.

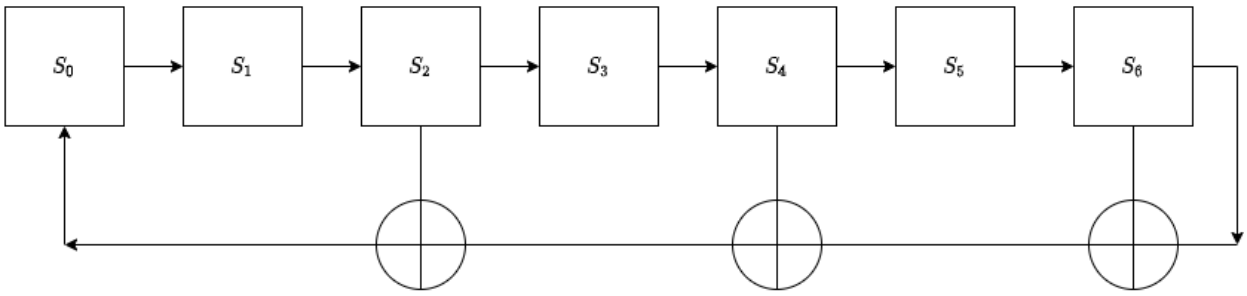

FIG. III.1. A sample LFSR with taps at 2,4 and 6

C. The Epoch, Nonce and Mining

The time stamp appended in a block (see II.1) is called an epoch. It is a 4-byte integer that represents the number of seconds passed from January 1, 1970 00:00 UTC.

The nonce is the arbitrary string that needs to be found by brute force algorithms to ensure the hash starts with a certain minimum number of zeroes. The process of finding the nonce is called "mining" in the language of crypto-currencies. Mining requires heavy use of resources like electricity and computing power. Bitcoin miners are often provided with incentives for the process. Since we have a 256-bit hash, mining success probability is of the order of 2^{-256}, a number that is astronomically small. Mining is a form of proof of work that ensures fraudulent individuals do not take advantage of a decentralized ledger. If the majority mines honest blocks, it becomes easy for someone or the other to get the nonce. In case of a fork in terms of two different blockchains with appropriate nonces, the longer blockchain gets accepted.

The hash function that we are using generates 8-bit long hashes. Our nonce can thus be 8-bit long, which can be represented by a single character.

IV. GROVER'S ALGORITHM

Grover's search algorithm[9] is an important quantum algorithm that provides an edge over all classical algorithms in database search. Grover's algorithm shall form the backbone for our search for the nonce. The algorithm uses repeated iterations of a quantum oracle (an oracle stands for a special function) and a diffuser on a superposition of $|0\rangle$ and $|1\rangle$.

A. Oracle

The oracle is the function that "marks" the states which are the successful search results by applying a negative sign on it. Consider the state $|\omega\rangle$ to be successful. Denote the oracle by U_ω so that

$$U_\omega |x\rangle = \begin{cases} |x\rangle & x \neq \omega \\ -|x\rangle & x = \omega \end{cases} \tag{IV.1}$$

The purpose of the oracle is to reflect the coefficient of "solution state ket" about 0, keeping the other coefficients intact. This will help the diffuser amplify amplitudes.

B. Diffuser

The diffuser is the key component of the Grover Search algorithm. The diffuser amplifies the amplitude of the solution states. It does this by reflecting the amplitudes of the states about the average amplitude. We denote the diffuser by U_s, so that

$$U_s = 2|s\rangle\langle s| - \mathbb{I} \tag{IV.2}$$

where $|s\rangle$ is some arbitrary state. This is indeed a reflection about the state $|s\rangle$. This is because any arbitrary state $|\psi\rangle$ can be written as $|\psi\rangle = \cos\alpha\,|s\rangle + \sin\alpha\,|s_\perp\rangle$ where $|s_\perp\rangle$ is orthogonal to

$|s\rangle$. Then we have

$$\begin{aligned} U_s|\psi\rangle &= U_s\left(\cos\alpha|s\rangle + \sin\alpha|s_\perp\rangle\right) \\ &= \left(2|s\rangle\langle s| - \mathbb{I}\right)\left(\cos\alpha|s\rangle + \sin\alpha|s_\perp\rangle\right) \\ &= \cos\alpha\left(2|s\rangle\langle s||s\rangle - |s\rangle\right) + \sin\alpha\left(2|s_\perp\rangle\langle s_\perp||s_\perp\rangle - |s_\perp\rangle\right) \\ &= \cos\alpha\left(2|s\rangle - |s\rangle\right) + \sin\alpha\left(-|s_\perp\rangle\right) \\ &= \cos\alpha|s\rangle - \sin\alpha|s_\perp\rangle \end{aligned} \tag{IV.3}$$

and we can clearly see that this is a reflection about the $|s\rangle$ axis.

Note that we can write $|s\rangle$ as a sum of two components, one linear function of $|\omega\rangle$ and the other a linear function of a state $|s\prime\rangle$ orthogonal to $|\omega\rangle$. Thus

$$|s\rangle = \sin\theta|\omega\rangle + \cos\theta|s\prime\rangle \tag{IV.4}$$

Each iteration will comprise U_sU_ω, i.e., reflection of $|s\rangle$ about $|s\prime\rangle$ followed by reflection about $|s\rangle$. This results in amplitude amplification.

C. The Method

The method of applying Grover's Search is quite simple.

1. Initialize $|s\rangle = \dfrac{|0\rangle + |1\rangle}{2}$.

2. Perform U_ω on the state to reflect component along $|\omega\rangle$, keeping components orthogonal to $|\omega\rangle$ intact.

3. Apply U_s on this state to reflect the state about $|s\rangle$

4. Repeat steps 2 and 3 till search is complete.

The above steps are illustrated in Now will we actually implement the oracle and the diffuser using quantum logic gates. First let us take a look at the oracle.

Implementation of the Oracle

The oracle is implemented using a phase kickback mechanism. Consider the CNOT gate in IV.1. The CNOT gate acts as shown below.

$$\begin{aligned} CX\,|00\rangle &= |00\rangle \\ CX\,|01\rangle &= |01\rangle \\ CX\,|10\rangle &= |11\rangle \\ CX\,|11\rangle &= |10\rangle \end{aligned} \tag{IV.5}$$

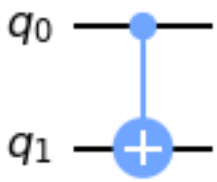

FIG. IV.1. CNOT gate

Now, if we have our classical function f behaving as

$$\begin{aligned} f(\omega) &= 1 \\ f(s) &= 0 \quad \forall s \neq \omega \end{aligned} \tag{IV.6}$$

Then we can set up an output qubit as

$$|out\rangle = |-\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

so that

$$CX\,|f(x)\ out\rangle = \begin{cases} |f(x)\ out\rangle & f(x) = 0 \\ -\,|f(x)\ out\rangle & f(x) = 1 \end{cases} \tag{IV.7}$$

Thus we get our oracle. The oracle circuit is shown in IV.2.

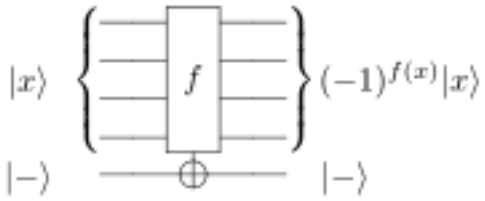

FIG. IV.2. The Oracle

Implementation of the Diffuser

The diffuser has the equation as per equation IV.3. To implement the diffuser on the state $|000\cdots0\rangle$, we just need to flip all qubits using X gates and then apply a multi-controlled Z-gate. Since we are starting with the state $|+++\cdots+\rangle$, we need to apply a H-gate in the beginning and at the end to got to $|000\cdots0\rangle$ and revert back. Figure IV.3 shows the diffuser circuit for 8 qubits.

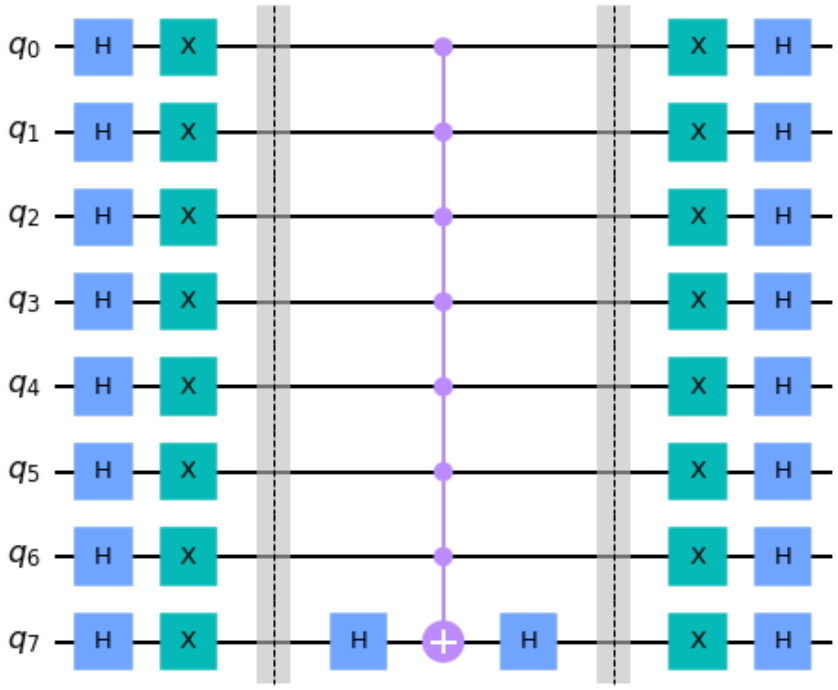

FIG. IV.3. The Diffuser for 8 qubits

D. Number of Iterations

Let our register consist of n qubits. The number of pure states possible is then $2^n = N$. Consider the initial state of the system, where the system is initialized to $|+++\cdots+\rangle$. This is nothing but an equal superposition of the N pure states. Since $|\omega\rangle$ is one of these states and all states occur with equal probabilities, probability of finding $|\omega\rangle$ is $\frac{1}{N} = \frac{1}{2^n}$. Now, from equation IV.4, the probability of measuring $|\omega\rangle$ is $\sin^2\theta$. Thus we have,

$$\begin{aligned} \sin^2\theta &= \frac{1}{2^n} \\ \Rightarrow \sin\theta &= \sqrt{\frac{1}{2^n}} \end{aligned} \tag{IV.8}$$

At each iteration we reflect by 2θ. Thus, after k iterations, we arrive at angle $2k\theta$. For complete search, $2k\theta$ should be as close as possible to $\frac{\pi}{2}$ but less than that (to avoid overshoot). For small $theta$, $\sin\theta \approx \theta$. Thus, we have

$$\begin{aligned} 2k\theta &= \frac{\pi}{2} \\ \Rightarrow k\sqrt{\frac{1}{2^n}} &= \frac{\pi}{4} \\ \Rightarrow k &= \frac{\pi}{4}\sqrt{2^n} \\ \Rightarrow k &= \frac{\pi}{4}\sqrt{N} \end{aligned} \tag{IV.9}$$

Thus, our complexity of search is $\mathcal{O}(\sqrt{N})$ as compared to $\mathcal{O}(N)$ for a classical brute force method. Further, if there are M solutions instead of one and we are interested in finding at least one solution, the complexity of Grover's algorithm is $\mathcal{O}\left(\sqrt{\frac{N}{M}}\right)$.

E. Generalized Grover Search

Generalized Grover search[10] dwells on the main Grover Search algorithm, except that it reduces the number of gates required by using suitable alternate unitary gates. This further increases the efficiency of the algorithm.

V. OUR WORK

In this section, we shall present our work. Our work essentially includes designing a suitable hash function and then cracking it using Generalized Grover Search algorithm. The entire code is available at We are working on **8-bit hashes**. Note that we will use the **little-endian system** throughout, i.e., S_0 represents the Most Significant Bit and S_7 the Least Significant Bit.

A. Classical Hashing Algorithm

The hashing algorithm followed is the 8-bit Pearson Hash generation using 2 LFSRs. The taps for the first LFSR are at indices 3, 4, 5 and 7. The taps for the second LFSR are at indices 1, 2, 4 and 7. The diagrams for the same are presented in figures V.1 and V.2 respectively. Characters are

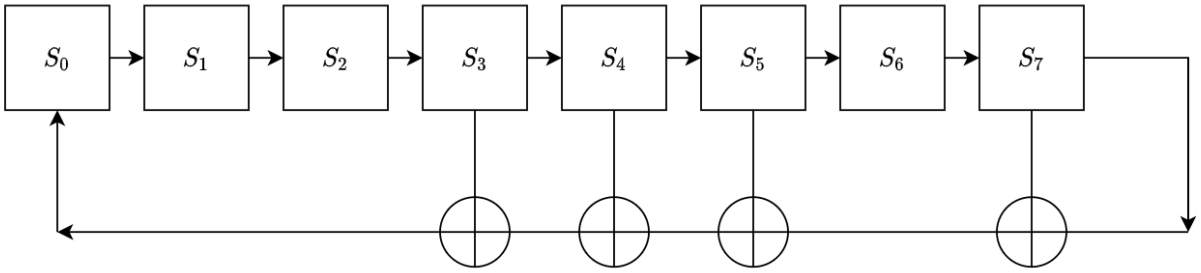

FIG. V.1. The First LFSR

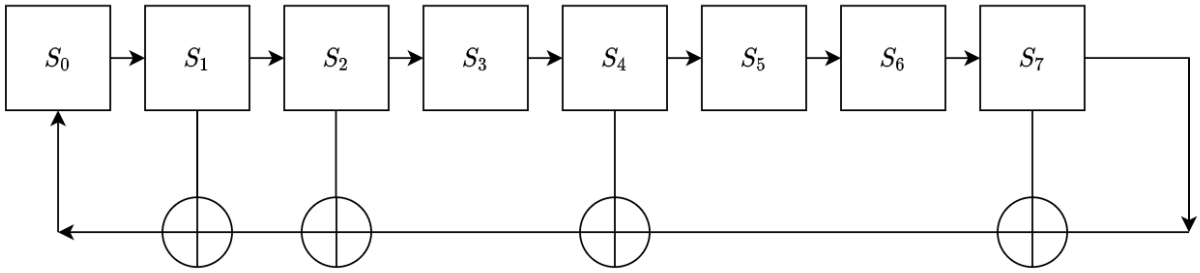

FIG. V.2. The Second LFSR

read from the message one at a time. The register is updates by taking its XOR with the ASCII of the character. The new value obtained is then passed through the first LFSR, followed by the second LFSR. The process is repeated for all characters till the end of the message. The code for the same is shown below.

```
def LFSR1_Classical(x):
    bit = ((x >> 0) ^ (x >> 2) ^ (x >> 3) ^ (x >> 4)) & 1
```

```
    x = ((x >> 1)|(bit<<7))%256;
    return x
def LFSR2_Classical(x):
    bit = ((x >> 0) ^ (x >> 3) ^ (x >> 5) ^ (x >> 6)) & 1
    x = ((x >> 1)|(bit<<7))%256;
    return x
def hash_Classical(message):
    x=0
    for i in message:
        x=x^ord(i)
        x=LFSR1_Classical(x)
        x=LFSR2_Classical(x)
    return x

message=input().rstrip()
print(hash_Classical(message))
```

Now we shall check the values of the hash for some sample strings. This can be done by running the code given above.

Message: `Hello World`

Hash: `15`

Making a slight change

Message: `Hello Workd`

Hash: `239`

We see that the hash changes significantly.

B. Quantum Hashing Algorithm

Now we will implement the same hashing algorithm on a quantum computer. This will make use of one quantum register comprising 8 qubits. The nonce itself will be stored on a register of 8 qubits. One qubit will be the output qubit that decides if a given hash is valid or not. Thus, our circuit will make use of 17 qubits in total, along with some classical bits.

The Two LFSRs and Their Inverses

Let us first take a look at the two LFSR circuits, given in figures V.3 and V.4 respectively. These LFSRs perform the same function as the classical LFSRs, except that they do it on qubits.

For a quantum computer, it is imperative that after each iteration, the registers are restored to the original state for the next iteration to take place successfully. We need to design the circuits for inverting the LFSRs as well. These circuits are shown in figures V.5 and V.6 respectively.

The Hashing Operation

The hashing is performed in the following steps.

1. The classical algorithm is used to compute the hash till the end of the message block, just before the nonce.
2. The quantum hash register of 8 qubits is initialized with the hash generated.
3. The nonce is XORed with the hash register using CX gates.
4. The first LFSR with taps at 3, 4, 5 and 7 is operated once on the hash register.
5. The second LFSR with taps at 1, 2, 4 and 7 is operated once on the hash register.

The circuit for the entire hashing operator is shown in figure V.7.

The Oracle

It is now time to put everything together. We shall assume that a valid nonce is one that starts with 5 zeroes. For this, we need to check if the registers at indices 0, 1, 2, 3 and 4 are $|0\rangle$. A multi-controlled Toffoli gate from these register qubits to the output will flip the qubits of the output if and only if all of the 5 qubits were $|1\rangle$. Further, if the output is initialized to $|out\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$, we will get the output to flip sign only when all of the 5 qubits are $|1\rangle$ or else output stays the same. Since we are interested in checking if all the 5 qubits are $|0\rangle$, we must use X gates on them before passing them through the multi-controlled Toffoli gate. After the check, we must restore the registers by inverting the operations applied. This invovles taking an X gate on qubits 0, 1, 2, 3 and 4, followed by the inverse of the second LFSR, followed by the inverse of the first LFSR. Figure V.8 shows the complete oracle circuit.

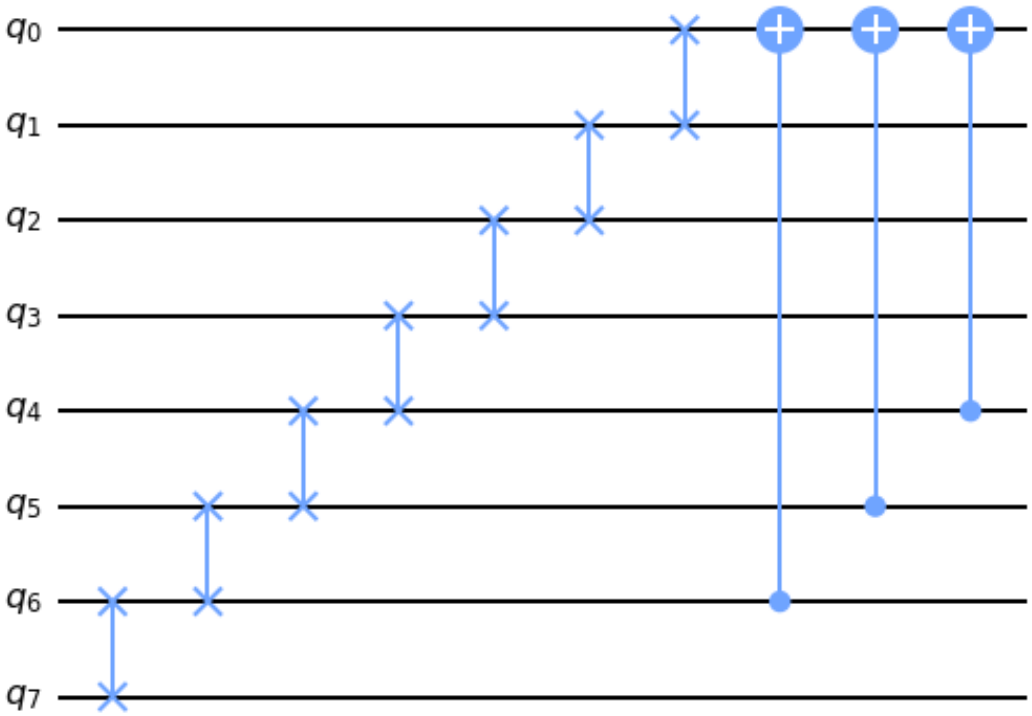

FIG. V.3. The First LFSR

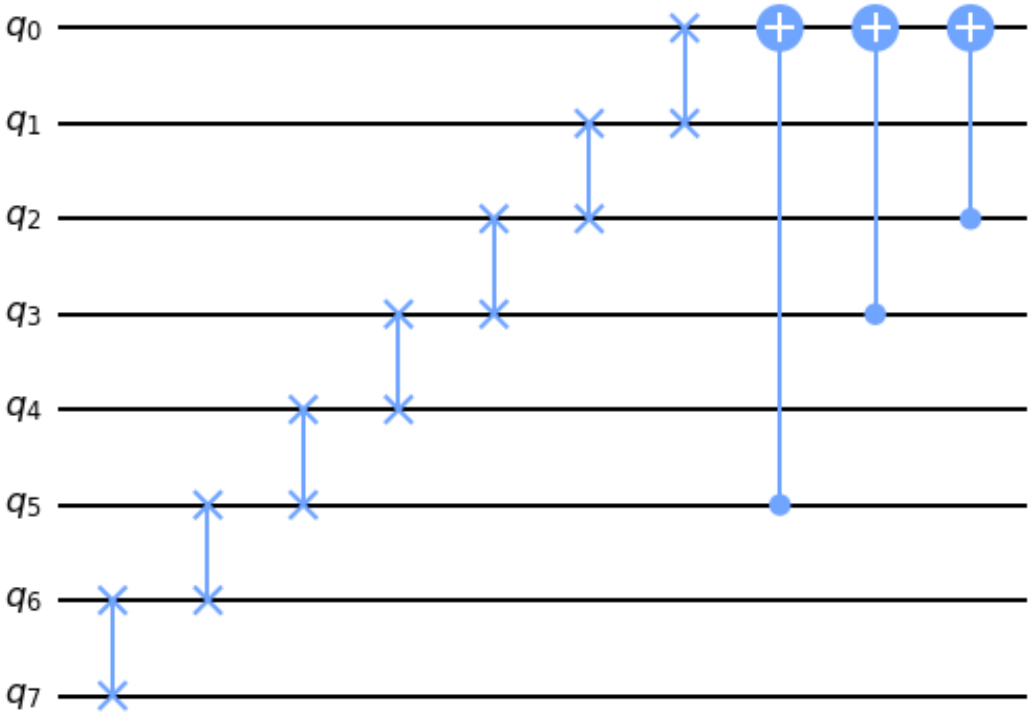

FIG. V.4. The Second LFSR

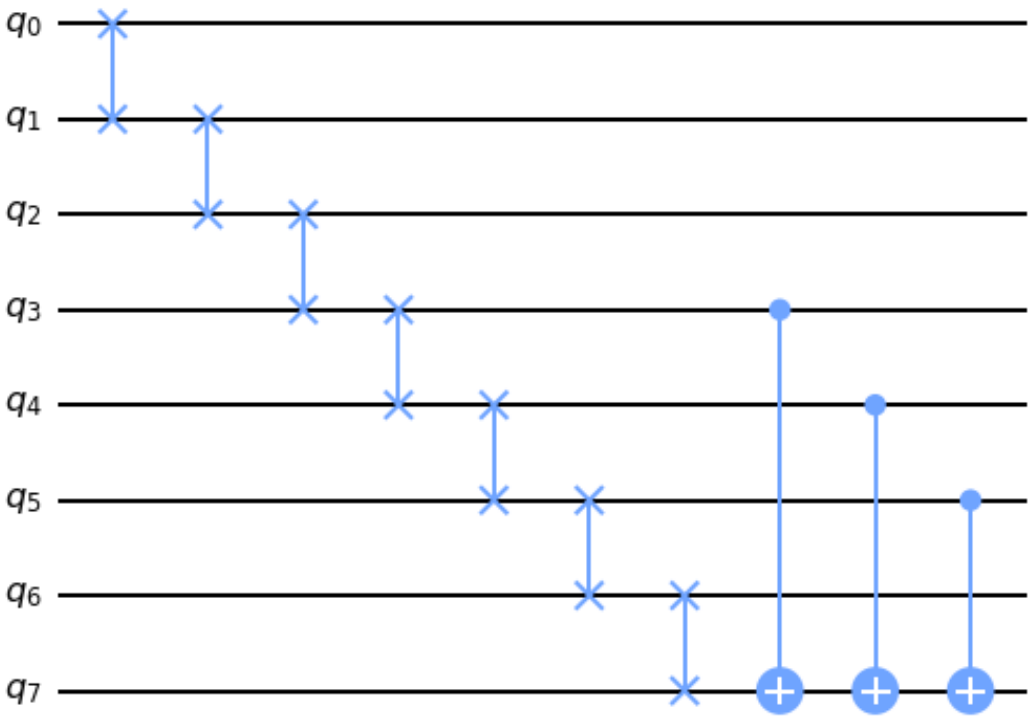

FIG. V.5. Inverse of the First LFSR

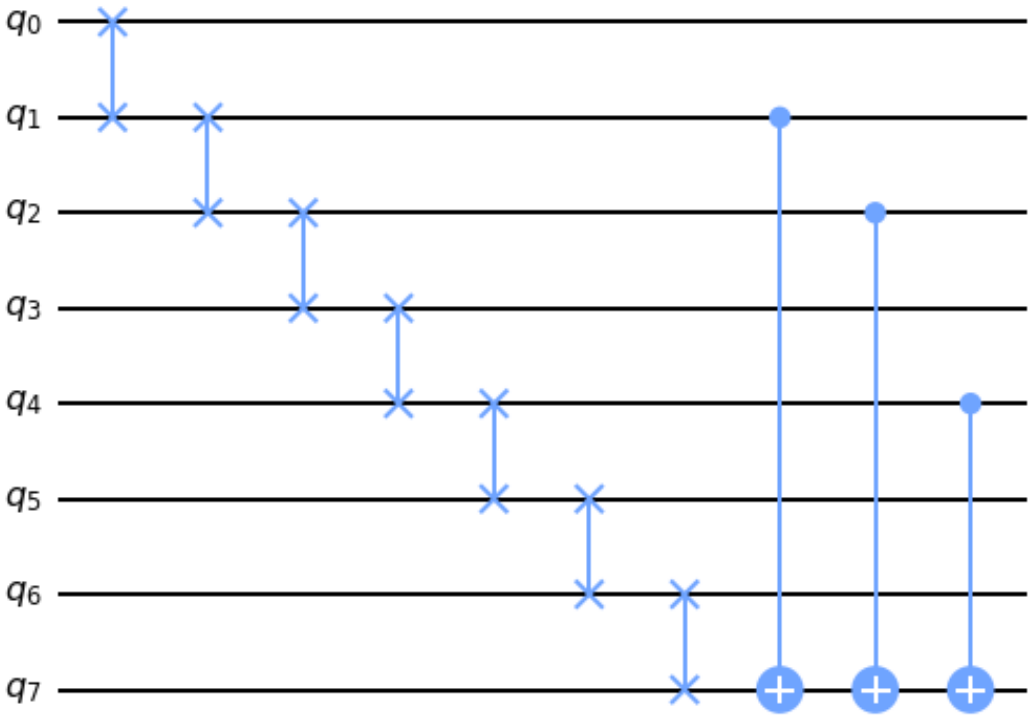

FIG. V.6. Inverse of the Second LFSR

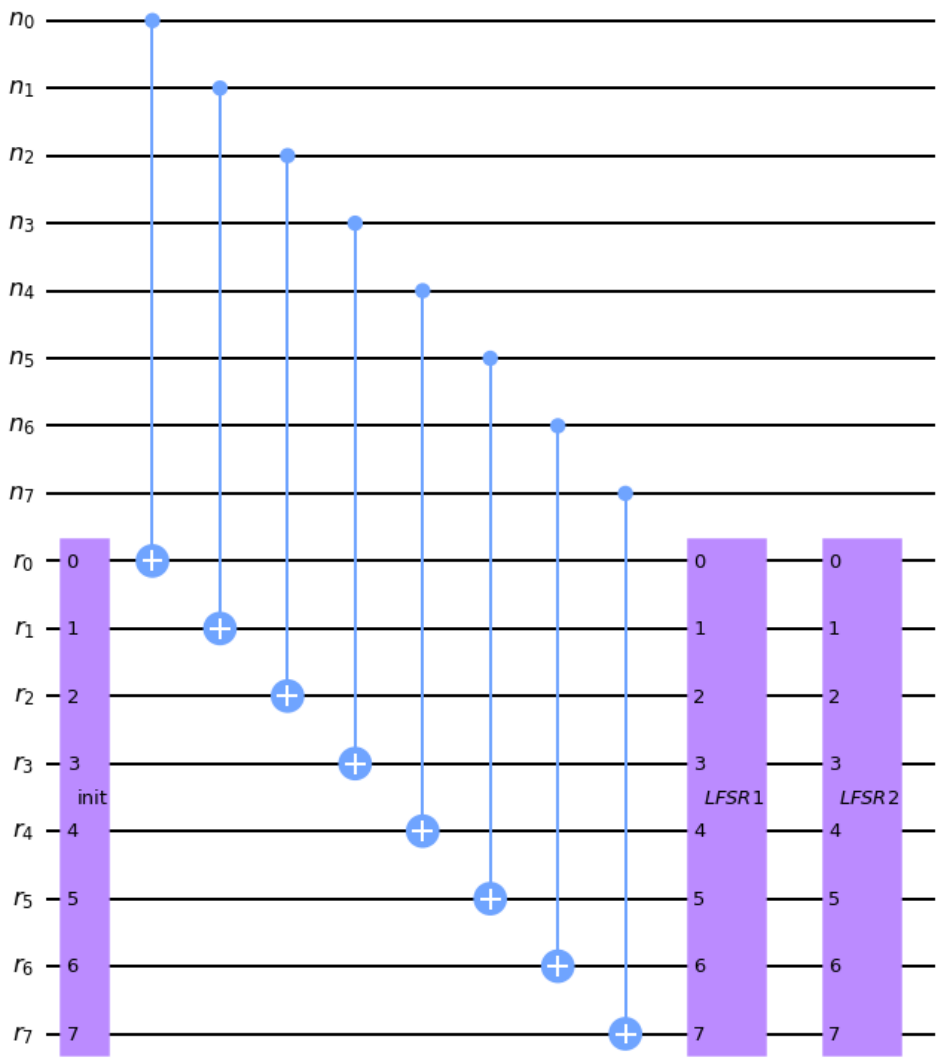

FIG. V.7. Hashing Circuit

Grover Search

The grover search involves setting up the nonce to a superposition of all states using a H gate on all the qubits followed by the repeated application of the oracle followed by the diffuser. The diffuser used has been discussed in the section preceding figure IV.3. The number of iterations required is given by equation IV.9, where, we now have $M = 8$ solutions instead of 1. Thus, number of iterations, k, is

$$k = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rfloor = \left\lfloor \frac{\pi}{4}\sqrt{\frac{256}{8}} \right\rfloor = 4 \tag{V.1}$$

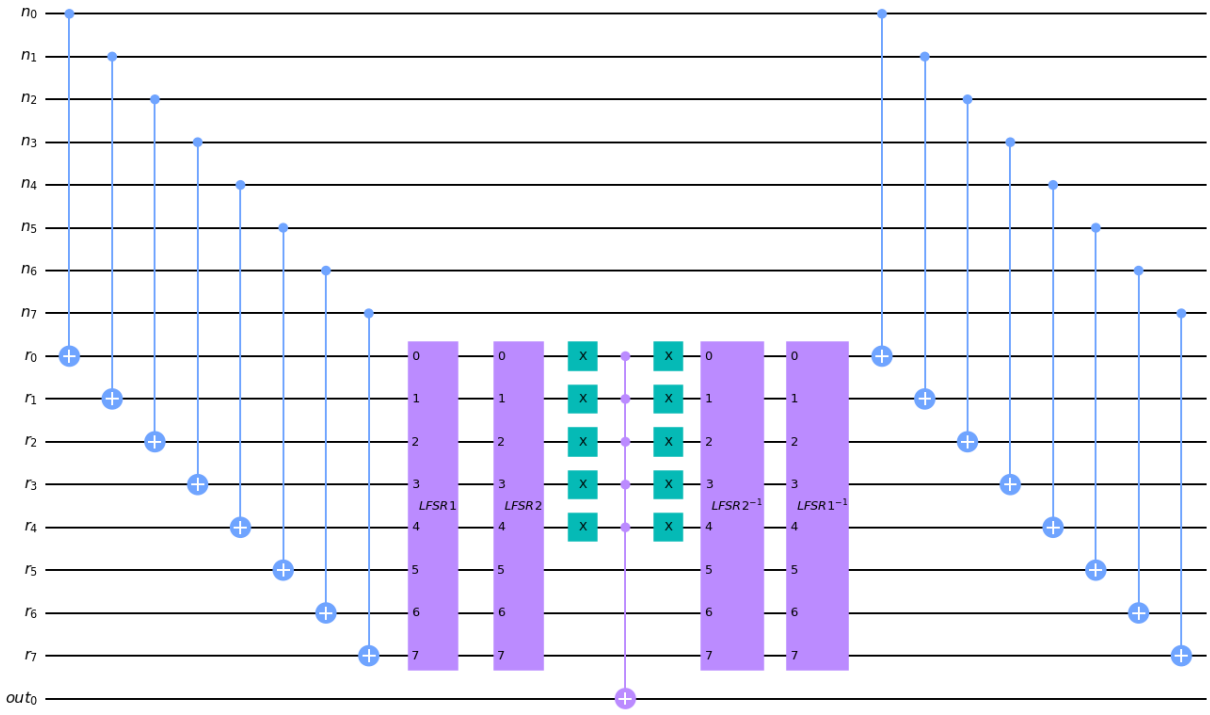

FIG. V.8. Circuit for the Oracle

Finally, we perform a measurement of the nonce. Figure V.9 shows the entire circuit.

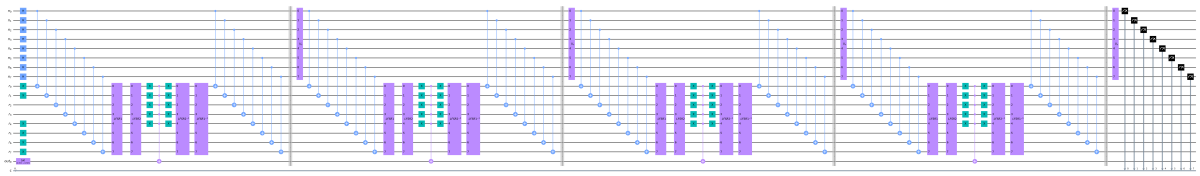

FIG. V.9. Grover Search Circuit

Generalized Grover Search

Here, we replace the H gate followed by X gate by a single unitary gate. $R_y\left(\frac{\pi}{2}\right)$ is a suitable unitary transformation substitute for HX. Its inverse is $R_y\left(-\frac{\pi}{2}\right)$. We can then simplify our diffuser to the one given in figure V.10 thereby reducing 16 gates per iteration. This reduction in the number of gates is very helpful. We will do an analysis of the same in the next section.

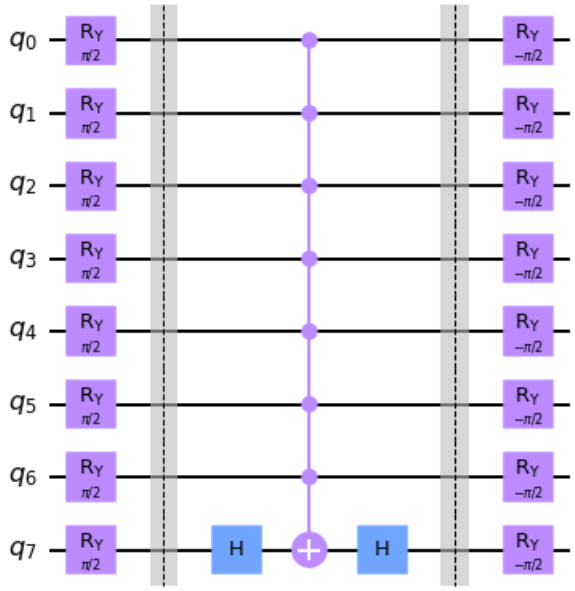

FIG. V.10. Generalised Grover Diffuser

VI. RESULTS

In this section, we shall see the results obtained after performing the Grover Search. Since we are using the **little-endian** system, the output qubits obtained must be read from right to left to get the ASCII value of the nonce-character. We will then append the nonce to the message and check if its classical hash indeed starts with 5 zeroes when written in binary.

Number of Gates

First we observe the difference in the number of gates used in the Grover and the Generalized Grover Search algorithms. To count the gates, we split them as follows

Each LFSR (or LFSR inverse) has

7 $SWAP$ gates

3 CX gates

Each Diffuser has

18 H gates

16 X gates

1 MCT gate

Each Generalized Diffuser has

16 R_y gates

2 H gates

1 MCT gate

Each iteration has

2 LFSRs

2 Inverse LFSRs

1 Diffuser (or Generalized Diffuser)

16 CX gates

10 X gates

1 MCT gate

There are 8 H gates at the start followed by 4 iterations. Thus the total number of gates can be summarized by table I.

SUMMARY OF THE NUMBER OF GATES USED							
Algorithm	**H**	**X**	**CX**	**SWAP**	**MCT**	**R_y**	**TOTAL**
Grover Search	80	104	112	112	8	0	416
Generalized Grover Search	16	40	112	112	8	64	352

TABLE I. Table summarizing the number of gates required

We observe that the Generalized Grover Search algorithm uses significantly less number of gates.

Qiskit Simulation

Here, we will use the **aer_simulator** provided by qiskit to run our code. For testing purposes our message is

Message: `Hello World`

This is then passed on to our program. A simulation of 1024 measurements is done. The result is shown in figure VI.1. Note that since all since we are using the **little-endian** system, output

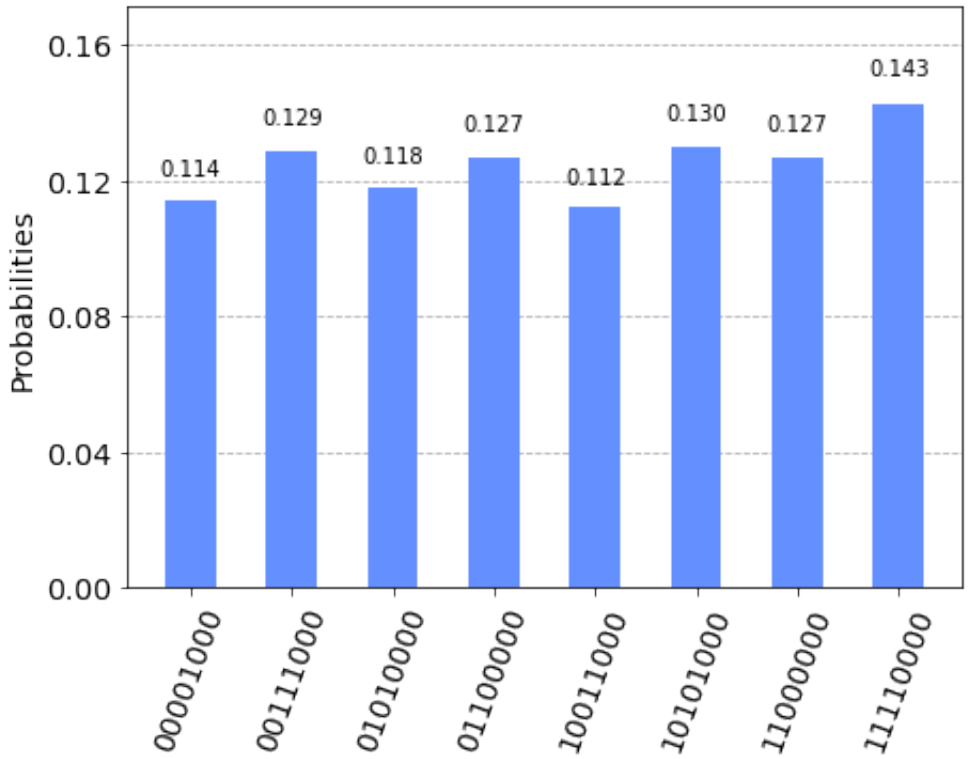

FIG. VI.1. Nonce values obtained and their probabilities

must be read from right to left. Converting the binary to decimal, we get the ASCII of the nonce to be appended as

```
28
3
21
25
15
6
10
16
```

We append the nonce and test the output hashes. The output hashes are (in the same order as the nonce values)

```
4
3
6
5
0
2
1
7
```

Thus, we conclude that our algorithm works successfully.

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system,"

[2] V. M. et. al., "The impact of quantum computing on present cryptography," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 3, 2018.

[3] A. L. Selvakumar and C. S. Ganadhas, "The evaluation report of sha-256 crypt analysis hash function," in *2009 International Conference on Communication Software and Networks*, pp. 588–592, 2009.

[4] "Blockchain technology basics." `https://www.spheregen.com/blockchain-technology-basics/`. Accessed: 2021-09-10.

[5] R. R. A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems,"

[6] P. K. Pearson, "Fast hashing of variablelength text strings,"

[7] H. Krawczyk, "Lfsr-based hashing and authentication," in *Advances in Cryptology — CRYPTO '94* (Y. G. Desmedt, ed.), (Berlin, Heidelberg), pp. 129–139, Springer Berlin Heidelberg, 1994.

[8] R. Khalaf and A. Abdullah, "Generate quantum key by using quantum shift register," *International Journal of Computer Networks and Communications Security*, vol. 3, pp. 248–252, 06 2015.

[9] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, (New York, NY, USA), p. 212–219, Association for Computing Machinery, 1996.

[10] A. Gilliam, M. Pistoia, and C. Gonciulea, "Optimizing quantum search using a generalized version of grover's algorithm," 2020.