Design and Comparative Analysis of Quantum Hashing Algorithms using Qiskit

Prodipto Das, Sumit Biswas, Sandip Kanoo and Veeradittya Podder

Abstract—This research explores the quantum implementation of three crucial hashing algorithms, namely Secure Hash Algorithm 1 (SHA-1), Message Digest 5 (MD5), and Secure Hash Algorithm 256 (SHA-256), within the context of network security for future networks. Quantum Cryptography, an emerging field, combines Cryptology and Cryptanalysis, presenting exciting prospects for secure communication. Our study focuses on the design and implementation of SHA-1, MD5, and SHA-256 algorithms specifically tailored for Quantum Computers. The primary objective is to investigate the time required to construct a hash and the bit rate at which a hash value can be transmitted. A comprehensive analysis of these three quantum algorithms is performed, with a particular emphasis on their performance in comparison to their classical counterparts. Experimental comparisons reveal that the execution time for qSHA-1, qMD5 and qSHA-256 significantly exceeds that of classical parts. Notably, our findings indicate a dependency of the implemented algorithms' execution time on the processor's speed. This research sheds light on the potential capabilities of quantum algorithms in the realm of network security and contributes valuable insights to the ongoing discussions surrounding quantum cryptography. An intriguing discovery arising from this study is the observation that quantum MD5 exhibits quicker execution times than quantum SHA-1 and quantum SHA-256. However, contrasting the classical scenario, where cSHA-1 demonstrates quicker execution than cMD5 and cSHA-256, it becomes evident that classical and quantum performance across these algorithms diverges markedly. This highlights a notable contrast in the behavior of these algorithms, thereby underscoring the potential dissimilarity rather than similarity between classical and quantum performances. The results underscore the need for further exploration to optimize the performance of quantum hashing algorithms, ensuring their viability in practical applications for future networks.

Index Terms—Quantum Cryptography, Hash Function, Quantum SHA-1, Quantum SHA-256, Quantum MD5, Hash Function, Digital Certificate.

1 Introduction

The process of hashing is important to cryptography. Cryptography is a method for protecting messages and data that are transmitted over the internet. We are aware that the amount of data that is currently accessible over the world wide web is increasing exponentially, and in order to protect this kind of data, we will provide a fingerprint as proof of its genuineness. The communication Digest algorithm is one example of a method that can be used to build a master fingerprint for the purpose of providing an authentication code for a communication. (hash value) [28].

Various hashing algorithms exist, paralleling the diversity of encryption methods, though some are more commonly utilized than others. Examples of prevalent hashing algorithms encompass MD5, SHA-1, SHA-2.

The MD5 algorithm utilises a 128-bit message as a measuring tool for data integrity; the user provides this message in order to generate a fingerprint message; the message's length can vary, but the most important thing is that it be irreversible. The Father of this algorithm is Professor Ronald L. Rivest of MIT [23]. This algorithm works the best on devices that are either 32 bits or 16 bits. It is possible to expand the compatibility of this algorithm to include 64 bit machines as well; however, the architecture of this type of scheme makes it likely that it will be fairly slow. The MD5 algorithm is an expansion of the MD4 method, which was significantly faster due to the fact that it only had three rounds, whereas the MD5 algorithm has four rounds, making it significantly slower. The function is a one-way hash that deals with the security features of the data.

The hash value is frequently referred to as the message digest, and the data that need to be encoded as the message. The SHA algorithm is used for data integrity, message authentication, and digital certificates.The SHA algorithm is developed by N.I.S.T. and used with digital signature applications, is a fingerprint that identifies the data [10].

Succeeding SHA-0, SHA-1 is the subsequent iteration of the Secure Hash Algorithm series, yielding 160-bit outputs. As the limitations of MD5 became apparent, SHA-1 saw an increase in adoption. Notably, it received FIPS 140 compliance status.

Rather than being a single algorithm, SHA-2 is an ensemble of hashing methods, including SHA-224, SHA-256, SHA-384, and SHA-512, with each variant characterized by its output size. Though superior in terms of security features compared to SHA-1, SHA-2 hasn't achieved the same widespread adoption.

Because of the increasing number of people who use the internet on a daily basis, it is essential to ensure that a complete file has been successfully downloaded via peer-to-peer (P2P) servers or networks. Because there is already a file with the same name, locating the original can be fairly challenging; hence, the message digest plays a significant part in the type of downloads in question. It is possible for these kinds of files to be associated with a message

- *Prodipto Das is with the Department of Computer Science, Assam University, Silchar, India PIN 788011.*
  E-mail: prodipto.das@aus.ac.in
-

Manuscript received ..., 2023; revised ..., 2023.

authentication code, which demonstrates that the source has been verified. If this is not the case, the file types delay the warning that a verified source could not be located, or vice versa. Both algorithms operate according to the same principle, although their structures are different [23] [20].

Since the message is shorter than 264 bits, Secure Hash will be able to process it successfully. The result of SHA is the message digest, and the length of these kinds of communications is 160 bits. (32 bits extra than MD5).

The purpose of this study is to create three quantum algorithms based on the classical components that are already in existence. This will be accomplished by building a variety of quantum circuits and analysing their performance to determine whether or not the results are comparable to the classical components or whether or not they produce new results. Because genuine chips may be obtained without cost using the IBMQ platform, which makes use of NISQ technology, the experimental study that needs to be done on the proposed work is carried out using that platform.

1.1 SECURE HASHING ALGORITHM 1

The SHA algorithm is a cryptographic hash method used in digital certificates and for ensuring data integrity. It acts like a unique fingerprint for data and was crafted by N.I.S.T. as a U.S. Federal Information Processing Standard (FIPS). Its primary purpose is digital signature applications. [22]. SHA-1 or Secure Hash Algorithm 1 is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value known as a message digest. It is most often used to verify a file has been unaltered. SHA-1 is now considered insecure since 2005 [24].

Algorithm 1 SHA-1 [26]

1: **procedure** SHA-1(input string) ▷ Take the input string
2: **Padding:**▷ Add padding so the final message length is a 64-bit multiple of 512.
3: **Appending length:** ▷ Calculate the length to be appended
4: **Divide the Input into 512-bit blocks:**▷ Partition the input into blocks of 512 bits each
5: **Initialize chaining variables:** ▷ Set up 5 chaining variables, each 32 bits, totaling 160 bits.
6: **Process Blocks:**

- Duplicate the chaining variables
- Segment the 512 bits into 16 sub-blocks
- Execute 4 rounds, with each having 20 steps.

7: **end procedure**

1.2 MESSAGE DIGEST 5 ALGORITHM

The efficiency of this algorithm varies with the size of the message. Though it necessitates an 8-bit message length and works rapidly, it's also compatible with extensive messages.
Step 1: Padding bits and Append Length
It's essential to pad bits starting with '1' and ending with '0' until the length is congruent to 448 mod 512. Subsequently, a 64-bit integer representing the original message's bit length is added. The resultant message, after padding, has a length of 512N, with N being an integer.
Step 2: Divide the input into 512-bit blocks
The padded message gets divided into N continuous 512-bit blocks, denoted as m1, m2, ... mn.
Step 3: Initialize chaining variables
Four 32-bit numbers are initialized as chaining variables, symbolized as (A,B,C,D) in the hash:
A = 01 17 2d 43
B = 89 AB CD EF
C = FE DC BA 98
D = 76 54 32 10
Step 4: Process blocks
The contents of the buffers (A, B, C, and D) are combined with input words through four auxiliary functions (W, X, Y, Z). This process encompasses four rounds, with each having 16 fundamental operations. The processing block P interacts with the buffers (A, B, C, and D), leveraging message word M[i] and constant K[i]. "$<<< s$" symbolizes a binary left shift by s bits. Four IRF (info related functions) accept three 32-bit word inputs, yielding a 32-bit word output. They employ logical operators like AND, OR, NOT, and XOR:
Q (A, S, D) = AS v not (A) F
W (A, S, D) = AS v S not (F)
E (A, S, D) = A xor S xor F
R (A, S, D) = S xor (A v not (F))
Every bit of A, S, and D is totalitarian and helps to balance each bit of Q (A, S, D). Functions (A, S, and D) labeled as P operate in "bitwise parallel" to ensure that if similar bits of D, E, and F are autarchic and balanced, each bit of W (A, S, D), E (A, S, D), and R (A, S, D) will be totalitarian and balance.
Step 5: Hashed Output
MD5 undergoes four rounds, producing a 128-bit hash.

1.3 SECURE HASHING ALGORITHM 256

The SHA-256 is a variant of the SHA-2 (Secure Hash Algorithm 2) series, developed by the National Security Agency in 2001 to replace SHA-1. It is a patented cryptographic function delivering a 256-bit output. While MD5 yields 128-bit hash values, SHA-2 offers versions generating various hash lengths, with SHA-256 being the most prevalent, producing 256-bit results. Notably, compared to MD5, SHA-2 offers enhanced security, particularly regarding collision resistance [1]

1.4 PARAMETERS USED FOR MD5 AND SHA ALGORITHM

A. Parameters of MD5.
Below equation shows a single MD5 operation.
1)Default Parameters
a = b + ((a + Process P (b, c, d) + M[i] + t[k]) ¡¡¡ s)
Here:- a, b, c, d = are Chaining variables
Process P=A non linear operation
M[i] =For M[q x 16 + i], which is the ith 32-bit word in the qth 512-bit block of the message t[k]=a constant ¡¡¡s =circular-left shift by s bits [2].
2) Actual Parameters.
Key Length: 64 bits, 128 bits, 256 bits , 512 bits

Algorithm 2 MD5 [23]

$M = (Y0, Y1, \ldots\ldots\ldots\ldots, Yn-1)$ ▷ Message to hash after padding. Each Yi is a 32-bit word and N is a multiple of 16 MD5 (M)
(A,B,C,D) ← (0x67452301, 0xefab89 , 0x98badcfe , Ox10325476) ▷ initialize (A,B,C,D)
for $i \leftarrow 0$ to $1/16 - 1$ **do**
    $X_j \leftarrow Y16_{i+j}$ ▷ Copy block I to X
    for $j \leftarrow 0$ to 15 **do**
        $W_j \leftarrow X_\sigma(j)$ ▷ Copy X to W
        for $z \leftarrow 0$ to 63 **do**
            $(Q_4, Q_3, Q_2, Q_1) \leftarrow (A, D, C, B)$ ▷ initialize Q
            Round0(Q , W) Round1(Q , W) Round2(Q , W) Round3(Q, W) ▷ Rounds 0 , 1 , 2 and 3
            $(A, B, C, D) \leftarrow (Q60 + Q_4, Q63 + Q_1, Q62 + Q_1, Q61 + Q_3)$
        end for
    end for
end for
Require: $n \geq 0 \vee x \neq 0$
Ensure: $y = x^n$
$y \leftarrow 1$
if $n < 0$ **then**
    $X \leftarrow 1/x$
    $N \leftarrow -n$
else
    $X \leftarrow x$
    $N \leftarrow n$
end if
while $N \neq 0$ **do**
    if N is even **then**
        $X \leftarrow X \times X$
        $N \leftarrow N/2$
    else[N is odd]
        $y \leftarrow y \times X$
        $N \leftarrow N - 1$
    end if
end while

Block Size: 128 bits
Cryptanalysis: Resistance Strong against Digital Certificate and very fast on 32 bit machines Security Secure
Rounds: 4
Steps: 16
B. Parameters of SHA.
Below equation shows a single SHA operation.
1) Default Parameters.
abcde(e+process p_s5(a)+W[t]+k[t]),a,s30(b), c, d
Here:- a, b, c, d, e =chaining variables
Process p =status of logical operations st =¡¡¡
W[t] =derived other 32 bits bytes
K[t]=five additives constants are defined [2] [3].
2) Actual Parameters.
Key Length: 128 bits
Block Size: 160 bits
Cryptanalysis: Resistance Strong against Digital Certificate.
Rounds: 4
Total Steps: 20

Algorithm 3 SHA-256 [1]

procedure SHA-256(input string) ▷ Take the input string of varying length
    Padding: ▷ Add Padding to the end of the genuine message so that the length is 64 bits less than the exact multiple of multiple of 512.
    Appending length: ▷ In this step the excluding length is calculated and appended at the end of the original input text
    Divide the Input into 512-bit blocks: ▷ In this step Input is divided into 512 bit blocks
    Initialize chaining variables: ▷ In this Step 8 chaining variables are initialized. 8 chaining variables of 32 bit each=160 bit of total And 64 additive constants are also initialized.
    Process Blocks:

- Duplicate the chaining variables
- Segment the 512 bits into 16 smaller blocks
- Execute the sub-block operations over 64 cycles.

end procedure

1.5 Literature Review

After a decade of exhaustive research on quantum computing, it has become a reality, and researchers and industry experts are giving extensive focus to it. As reported by "Quantum Supremacy" [2], quantum computers easily solve problems in exponentially less time. On the other hand, quantum computers can expose the privacy and security of encrypted data for real life information and communication systems. Therefore, exploration of quantum computing and computers is advancing rapidly [18].

In 2017, IBM Quantum Experience (IBMQ), the quantum computing research initiative of IBM, first launched a 5 qubit quantum computer in cloud service (QISKit, IBM, 2018). IBM QISKit is a software platform developed by IBM that accelerates research on QC. The dreams of Richard Feynman and Toffoli is becoming a reality [5]; [27]. Superposition, interference, and entanglement are exciting and spooky phenomena.

A great number of works have been documented in the most recent advancements in post-quantum cryptography (PQC). A variety of research projects, such as PQCrypto, SAFEcrypto, CryptoMathCREST, and PROMETHEUS, in addition to various standardisation initiatives, have been addressing the topic of post-quantum cryptography, which is currently a popular research topic. These initiatives have been laid out at a variety of different levels. Above all else, the National Institute of Standards and Technology (NIST) of the United States Government is working on developing post-quantum public-key crypto systems. To date, there have been two phases of this project, and it is anticipated that the first standard draughts will be delivered between the years 2022 and 2024 [4].

Despite the prevalence of quantum computers, traditional cryptographic algorithms such as codes, hashes, lattices, and multivariate techniques remain secure, there were challenges in cryptography in the 1990s decade brought about by the invention of the Shor and Grover algorithm.

These challenges allowed popular algorithms such as RSA (1978), Diffie-Hellman (2002), and Elliptical curve (1985) to be cracked [17].

A network platform that is immune to quantum attacks is essential at this point in time. In this vein of thought, other than the work being undertaken by NIST, the Internet Engineering Task Force (IETF) has published a Request for Comment (RFC) that can give a modification for quantum resistance to the Internet Key Exchange that is extensively used (IKE). In a similar vein, both the International Organization for Standardization (ISO) and the United States Federal Information Processing Standards (FIPS) have developed programs that check to see if cryptographic modules are used in a network in a way that is accurate and trustworthy. The International Organization for Standardization is participating in the Horizon 2020 project known as PQCRYPTO (Post-Quantum Cryptography for Long-Term Security). The Federal Information Processing Standards Organization (FIPS) has published a draught road map for post-quantum hardware/software module evaluations [30].

PQC algorithms are currently the primary area of concentration for research groups working in the field of quantum cryptography. The post-quantum algorithms use super singular isomorphy, ring learning with errors, and coding as its quantum-resistant cryptographic primitives [3]

In a research study by [8], the performance of three different hash function algorithms – SHA-1, SHA-256, and SHA-512 – was examined on an Intel Core Processor 2nd Generation. The goal was to figure out which algorithm works faster when applied to specific tasks.

Another investigation, discussed in [21], delved into the security of VSAT satellite communications using the MD5 algorithm. Additionally, a simulation of VSAT satellite communication was carried out on a regular computer using the Montogomery algorithm.

In the realm of [13], the focus was on developing a quantum computer simulator that mimics the behavior of a real quantum computer. This simulator is skilled at generating data that mimics quantum-level processes.

Advancements in quantum cryptography spurred an assessment of various cryptographic systems to understand their resistance against quantum attacks, as explained in [3].

For newcomers in the field of quantum mechanics, [7] introduced an innovative way to learn about quantum machines and their operations.

Exploring the role of a single photon particle in quantum cryptography, [9] highlighted how precisely controlled photon emissions from a laser can be used for secure communication.

In a retrospective analysis, [25] looked back at the state of quantum cryptography in 2009 and its impact on network security.

The use of photon particles and Shor's algorithm for secure cryptographic techniques was discussed in [16].

An examination of the potential of quantum cryptography to replace classical cryptography in the future has been discussed in [11].

Comparing Classical and Quantum Cryptography techniques using various algorithms was the main focus of [15].

The breakthrough algorithm designed by Shor, which exposes vulnerabilities in classical cryptography, was discussed in [6], along with protocols for secure communication using single photons.

[19] delved into the security requirements of communication systems from a quantum cryptography perspective.

In the book [12], Chapter 4 explored how the MD5 algorithm works and compared it to MD4.

Lastly, [14] covered essential hash functions such as MD-5, SHA-1, SHA-256, SHA-512, and SHA-384, offering a comprehensive overview of their features and roles in cryptography.

1.6 Contributions

This paper proposes the method of designing various quantum circuits by using available quantum gates and using these circuits to develop quantum versions of three classical algorithms. This paper discusses the detailed framework for designing quantum SHA-1, quantum MD5, and quantum SHA-256 algorithms. The algorithms for quantum versions of SHA-1, SHA-256, and MD5 are presented. Further, this paper presents the complete steps to designing various quantum circuits, and these circuits can be further used for designing any classical algorithm for quantum computers.

1.7 Organization of Paper

In this research work, quantum versions of the three classical algorithms are implemented to compare their performance. The algorithms are executed on the IBMQ quantum experience QISKit software platform. The proposed quantum circuits are implemented using a cloud service in the actual chip ibmq_16_melbourne, ibmq_belem, and ibmqx4 quantum computers (QISKit, IBM, 2018).

The rest of the paper is organized as follows.

2 Implementation

The reason for selecting SHA-1, SHA-256, and MD5 algorithms for this research work is that SHA-1, SHA-256, and MD5 algorithms are probably the most well-known and widely used hash functions. Since 2005, SHA-1 and MD5 algorithms have not been considered secure against well-funded opponents. The target of this work is to find any changes in quantum versions over the classical algorithms.

For the sake of working with quantum computers on the level of individual circuits, pulses, and algorithms, this article makes use of the open-source software development kit known as Qiskit. It offers tools for the creation and manipulation of quantum programmes, as well as the execution of these programmes on prototype quantum devices on IBM's Quantum Experience or on simulators running on a local computer. In order to construct the quantum SHA-1 algorithm on our local computer, we make use of IBM's QASM simulator. For scientific computing, we make use of Anaconda, and we use Jupyter in order to interface with Qiskit. This method takes as input a variety of various combinations of text, numerals, and special characters, each of which can be any length. The goal is to determine how much of a discrepancy in output time can be attributed to differences in the length of the inputs. The same input is carried out in a number of different ways so that the difference in execution time can be observed.

2.1 Implementation of Quantum Circuits

2.1.1 Implementation of Quantum–XOR Circuit

When performing an XOR operation, the result will always be zero if there are any two bits that are identical. If not, the answer is yes. We are able to create the XOR circuit by using the CNOT gate.

A C-not gate is utilised in the construction of the quantum XOR logic gate. The operation of a C-not gate is identical to the operation of a traditional XOR gate in every respect. Therefore, in order to create a quantum XOR gate, all we had to do was first put our input layer, which differs depending on the input of the user, and then put a C-not on the layer that came after it, which would result in XOR outputs depending on the input. This was all there was to it.

Therefore, if the inputs are the same, such as 00 or 11, the circuit will create the output 0, and if the inputs are not the same, such as 01 or 10, the circuit will produce the output 1. When performing an XOR operation, the result will always be zero if there are any two bits that are identical. If not, the answer is yes. We are able to create the XOR circuit by using the CNOT gate.

Algorithm 4 Quantum–XOR Algorithm

procedure QUANTUM–XOR(a, b) ▷ Two inputs a and b
  $qc \leftarrow QuantumCircuit(2, 1)$
  qc.reset(range(2))
  if $a == 1$ **then**
    $qc.x(0)$
  else
    if $b == 1$ **then**
      $qc.x(1)$
    end if
  end if
  $qc.cx(0, 1)$
  $qc.measure(1, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ ——
end procedure

In the Circuit diagram, first layer is an input layer.

Fig. 1: Circuit diagram of Q–XOR

2.1.2 Implementation of Quantum–AND Circuit

In the AND operation, if both bits are 1 only, then the output is one. Otherwise, it is 0. Using the CCNOT gate, we can design the AND circuit.

The quantum AND is designed with the help of three qubits and one classical bit. The first layer will be our input layer, followed by a CNOT gate in the next layer. The input layer will use a not gate depending on the input, and a reset gate will be used in the 3rd qubit, q2, which will reset its value after each round. The circuit is designed in such a way that when the output is 11, it will only result in 1 as output; otherwise, it will produce 0 as output for another combination of inputs.

Algorithm 5 Quantum–AND Algorithm

procedure QUANTUM–AND(a, b) ▷ Two inputs a and b
  $qc \leftarrow QuantumCircuit(3, 1)$
  qc.reset(range(3))
  if $a == 1$ **then**
    $qc.x(0)$
  else
    if $b == 1$ **then**
      $qc.x(1)$
    end if
  end if
  $qc.ccx(0, 1, 2)$
  $qc.measure(2, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ Return a&&b
end procedure

Circuit Diagram:
In the following diagram the first layer is an input layer.

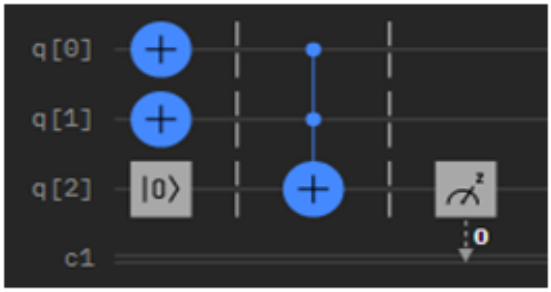


Fig. 2: Circuit diagram of Q-AND

2.1.3 Implementation of Quantum–NOT Circuit

The NOT operation simply alters the qubit, i.e., if we input 1, then the output is 0, and vice versa. Using the concept of the CCNOT gate and some helping qubits, we can design the NOT gate. A total of 3 qubits are necessary for NOT gate implementation.

For attaining the properties of quantum NOT, the use of the given NOT gate is sufficient. But we have designed our own quantum NOT gate, which consists of 3 qubits and 1 classical bit. The qubits q1 and q2 are just helping qubits, while the qubit q0 will intake the input given by a user. So, by arranging the qubits as above, when the input is 0, the output will be 1, and when the input is 1, the output will be 0.

Circuit diagram: In the following diagram, the first layer is the helping layer, which is fixed. Here, we have used the

Algorithm 6 Quantum–NOT Algorithm

procedure QUANTUM–NOT(a) ▷ Input a
  $qc \leftarrow QuantumCircuit(3, 1)$
  qc.reset(range(3))
  $qc.x(1)$
  $qc.x(2)$
  if $a == 1$ **then**
    $qc.x(0)$
  end if
  $qc.ccx(0, 1, 2)$
  $qc.measure(2, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ Return !a
end procedure

identity gate, which is actually the absence of a gate. It does not impact the output result. The second layer is our input layer, where the first qubit will hold the input bit.

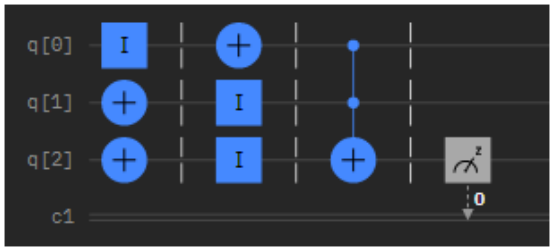


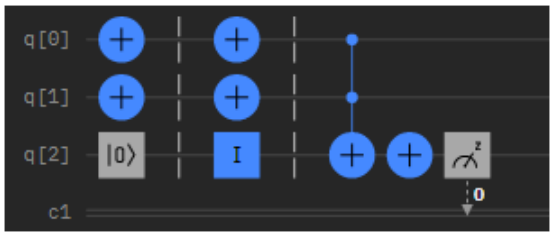
Fig. 3: Circuit diagram of Q-NOT

2.1.4 Implementation of Quantum–OR Circuit

In the OR operation, the output is zero only when both input bits are 0. Otherwise, for all combinations of input bits, the output is always 1. Using the CCNOT gate, we can design the OR gate.

The quantum OR gate is designed with the help of three qubits and one classical bit. The first layer is the same as that of quantum AND. But there is an inclusion of a helping layer in quantum OR, which consists of two not gates in qubits q0 and q1, and an identity gate in q2, which will help convert the inputs into our desired outputs. The output of this layer will pass through a cc-not gate and a c-not gate. The result of designing such a gate will be that when the inputs have at least one 1, the output will always be 1, and only when the input is 00, the output will be 0.

The quantum OR circuit is shown in Fig. 4, where the first layer is a helping layer and the second layer is our output layer. Finally, after applying the CCNOT, we are just inverting the output value using the Pauli-X gate.



Algorithm 7 Quantum–OR Algorithm

procedure QUANTUM–AND(a, b) ▷ Two inputs a and b
  $qc \leftarrow QuantumCircuit(3, 1)$
  qc.reset(range(3))
  $qc.x(0)$
  $qc.x(1)$
  if $a == 1$ **then**
    $qc.x(0)$
  else
    if $b == 1$ **then**
      $qc.x(1)$
    end if
  end if
  $qc.ccx(0, 1, 2)$
  $qc.x(2)$
  $qc.measure(2, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ Return $a|||b$
end procedure

Fig. 4: Circuit diagram of q-OR

2.1.5 Implementation of Quantum–Addition Circuit

To implement the adder, we have used the concept of a full adder. The Full Adder is the adder that adds three inputs and produces two outputs. The first two inputs are A and B, and the third input is an input carried as C-IN. The output carry is designated as C-OUT, and the normal output is designated as S, which is SUM. A full adder logic is designed in such a manner that it can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to another.

To execute quantum addition, two circuits have been devised, the Sum Circuit and the Carry Circuit.

The Carry Circuit is shown in Fig. 5, where the first layer is the input layer. The q[0] qubit will hold the carry bit, and q[1] and q[2] will hold the input bits. q[3] is a helping qubit.

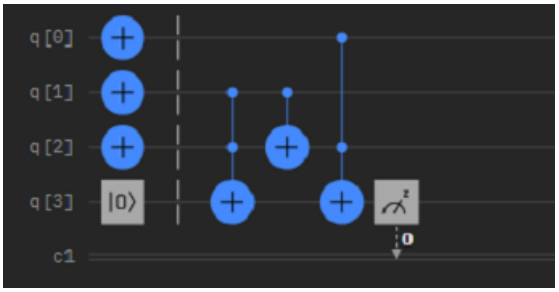


Fig. 5: Circuit diagram of Carry

On the identical inputs, the carry circuit will compute the carry and the sum circuit will compute the sum.

Circuit Design: In the following diagram, q[0] holds the carry bit, q[1] and q[4] will have the same input bits, and q[2] and q[5] will hold the same input bits.

One of the more difficult circuits to construct was the quantum addition circuit. It has six qubits and two classical

Algorithm 8 Quantum–CARRY Algorithm

procedure QUANTUM–CARRY($a, b, output$) ▷ Three inputs a, b and output
  $qc \leftarrow QuantumCircuit(4, 1)$
  qc.reset(range(4))
  if $output == 0$ **then**
    if $a == 1$ **and** $b == 1$ **then**
      $qc.x(1)$
      $qc.x(2)$
    else if $a == 1$ **and** $b == 0$ **then**
      $qc.x(1)$
    else if $a == 0$ **and** $b == 1$ **then**
      $qc.x(2)$
    end if
  else if $output == 1$ **then**
    if $a == 1$ **and** $b == 1$ **then**
      $qc.x(0)$
      $qc.x(1)$
      $qc.x(2)$
    else if $a == 1$ **and** $b == 0$ **then**
      $qc.x(0)$
      $qc.x(1)$
    else if $a == 0$ **and** $b == 1$ **then**
      $qc.x(0)$
      $qc.x(2)$
    else if $a == 0$ **and** $b == 0$ **then**
      $qc.x(0)$
    end if
  end if
  $qc.barrier()$
  $qc.ccx(1, 2, 3)$
  $qc.cx(1, 2)$
  $qc.ccx(0, 2, 3)$
  $qc.barrier()$
  $qc.measure(3, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ Return CARRY
end procedure

Algorithm 9 Quantum–SUM Algorithm

procedure QUANTUM–SUM($a, b, carry$) ▷ Three inputs a, b and carry
  $qc \leftarrow QuantumCircuit(3, 1)$
  qc.reset(range(3))
  if $carry == 0$ **then**
    if $a == 1$ **and** $b == 1$ **then**
      $qc.x(1)$
      $qc.x(2)$
    else if $a == 1$ **and** $b == 0$ **then**
      $qc.x(1)$
    else if $a == 0$ **and** $b == 1$ **then**
      $qc.x(2)$
    end if
  else if $carry == 1$ **then**
    if $a == 1$ **and** $b == 1$ **then**
      $qc.x(0)$
      $qc.x(1)$
      $qc.x(2)$
    else if $a == 1$ **and** $b == 0$ **then**
      $qc.x(0)$
      $qc.x(1)$
    else if $a == 0$ **and** $b == 1$ **then**
      $qc.x(0)$
      $qc.x(2)$
    else if $a == 0$ **and** $b == 0$ **then**
      $qc.x(0)$
    end if
  end if
  $qc.cx(1, 2)$
  $qc.cx(0, 2)$
  $qc.barrier()$
  $qc.measure(2, 0)$
  $backend \leftarrow Aer.get\_backend('aer\_simulator')$
  $job \leftarrow backend.run(qc, shots \leftarrow 1, memory \leftarrow True)$
  $output \leftarrow job.result().get\_memory()[0]$
  return $qc, output$ ▷ Return SUM
end procedure

bits. The q0 is used to hold the addition's carry bits, while the designs of q1 and q2 are identical to those of q4 and q5. The carry is calculated using the qubits q1, q2, and q3, as well as two cc-not gates and one c-not gate. And the sum is accomplished with the assistance of qubits q4 and q5, as well as two c-not gates.

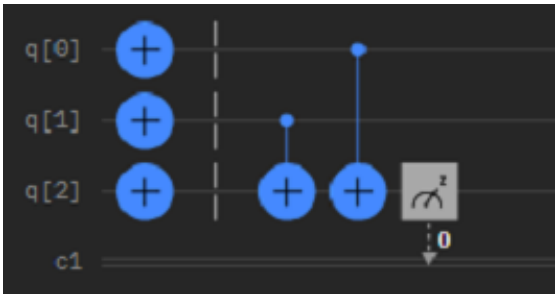


Fig. 6: Circuit diagram of Sum

The sum circuit is shown in Fig. 6, where the first layer is the input layer. The q[0] qubit will hold the carry bit generated from the carry circuit, and q[1], q[2] will hold the input bits to compute the sum.

Modified Quantum Adder circuit is created by joining the Sum and Carry circuits together.

Algorithm 10 Quantum–ADDER Algorithm

```
procedure QUANTUM–ADDER(a, b, output)  ▷ Three inputs a, b and output
    qc ← QuantumCircuit(6, 2)
    qc.reset(range(6))
    if output == 0 then
        if a == 1 and b == 1 then
            qc.x(1)
            qc.x(2)
            qc.x(4)
            qc.x(5)
        else if a == 1 and b == 0 then
            qc.x(1)
            qc.x(4)
        else if a == 0 and b == 1 then
            qc.x(2)
            qc.x(5)
        end if
    else if output == 1 then
        if a == 1 and b == 1 then
            qc.x(0)
            qc.x(1)
```

Algorithm 11 Quantum–ADDER Continue

```
            qc.x(2)
            qc.x(4)
            qc.x(5)
        else if a == 1 and b == 0 then
            qc.x(0)
            qc.x(1)
            qc.x(4)
        else if a == 0 and b == 1 then
            qc.x(0)
            qc.x(2)
            qc.x(5)
        else if a == 0 and b == 0 then
            qc.x(0)
        end if
    end if
    qc.barrier()
    qc.ccx(1, 2, 3)
    qc.cx(1, 2)
    qc.ccx(0, 2, 3)
    qc.barrier()
    qc.cx(4, 5)
    qc.cx(0, 5)
    qc.barrier()
    qc.measure(3, 0)
    qc.measure(5, 1)
    backend ← Aer.get_backend('aer_simulator')
    job ← backend.run(qc, shots ← 1, memory ← True)
    output ← job.result().get_memory()[0]
    add = output[0 : 1]
    output = output[1 : 2]
    return qc, output  ▷ Return
end procedure
```

Circuit Design: In the following diagram, q[0] holds the carry bit, q[1] and q[4] will have the same input bits, and q[2] and q[5] will hold the same input bits.

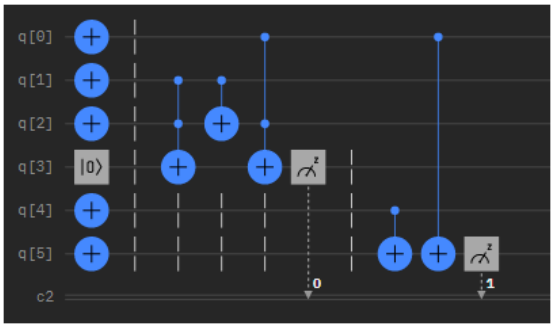


Fig. 7: Circuit diagram of modified q-Adder

2.1.6 *Quantum Mod Algorithm*

When we are trying to find mod 2^n, then using the AND operation, we can easily compute the remainder. Suppose we want to find X mod 2^n. For this, we need to follow the following procedure:

1) Find $2^n - 1$
2) Then do X AND $2^n - 1$

This method work only if we are finding the mod 2^n.

2.2 Implementation of Quantum SHA–1 Algorithm

The input taken for this algorithm consists of different combinations of text, numbers, and special characters of different lengths. The motive is to check the difference in output time due to variations in the length of inputs. The same input is executed multiple times to visualise the difference in execution time.

Algorithm 12 Quantum–SHA-1 Algorithm

procedure QUANTUM–SHA-1($InputMessage$) ▷ Any kind of string of any length
  Take the input string and convert into its corresponding equivalent binary code.
  Add Padding
  $ap \leftarrow dig + 1$
  while $len(ap)\%512 \neq 448$ **do**
    $Ap \leftarrow Ap + 0$
  end while
  Count the total length of the message and convert it into binary equivalent.
  $ap\_length \leftarrow msg\_length \bmod 2^{64}$
  $tot\_length \leftarrow ap + ap\_length$
  $n \leftarrow 512$
  $Chunks \leftarrow [tot\_length[i : i + n]foriinrange(0, len(tot\_length), n)]$
  Initialize 5 chaining variables and 4 additive constants.
  $cv0 \leftarrow' 01100111010001010010001100000001'$
  $cv1 \leftarrow' 11101111110011011010101110001001'$
  $cv2 \leftarrow' 10011000101110101101110011111110'$
  $cv3 \leftarrow' 00010000001100100101010001110110'$
  $cv4 \leftarrow' 11000011110100101110000111110000'$
  $f1 \leftarrow cv0$
  $f2 \leftarrow cv1$
  $f3 \leftarrow cv2$
  $f4 \leftarrow cv3$
  $f5 \leftarrow cv4$
  $ck0 \leftarrow' 01011010100000100111100110011001'[0 \leq t \leq 19]$
  $ck1 \leftarrow' 01101110110110011110101110100001'[20 \leq t \leq 39]$
  $ck2 \leftarrow' 10001111000110111011110011011100'[40 \leq t \leq 59]$
  $ck3 \leftarrow' 11001010011000101100000111010110'[60 \leq t \leq 79]$

Algorithm 13 Quantum SHA–1 Continue

  for $l = 0$ to $len(Chunks)$ **do**
    $n \leftarrow 32$
    $ch \leftarrow chunks[l]$
    for $l = 0$ to $len(ch, n)$ **do**
      $chu = ch[i : i + n]$
    end for
    for $t = 16$ to 80 **do**
      $M[t] \leftarrow ROTL1(M[t-3] \bigotimes M[t-8] \bigotimes M[t-14] \bigotimes M[t-16])$
    end for
    for $t = 0$ to 80 **do**
      if $t \leq 19$ **then**
        $F \leftarrow (B\&\&C)||\neg(B\&\&D)$
      else if $(t > 19$ && $t \leq 39)$ **then**
        $F \leftarrow B \bigotimes C \bigotimes D$
      else if $(t > 39$ && $t \leq 59)$ **then**
        $F \leftarrow (B\&\&C)||(B\&\&D)||(C\&\&D)$
      else if $(t > 59$ && $t \leq 79)$ **then**
        $F \leftarrow (B \bigotimes C \bigotimes D)$
      end if
      $Add1 \leftarrow (F + cv4) mod 2^{32}$
      $Shft5 \leftarrow ROTL5(cv0)$
      $Add2 \leftarrow (Add1 + Shft5) mod 2^{32}$
      $cv1 \leftarrow cv0$
      $Add3 \leftarrow (Add2 + M[t]) mod 2^{32}$
      $Add4 \leftarrow (Add3 + K[t]) mod 2^{32}$
      $cv0 \leftarrow Add4$
      $Shft30 \leftarrow ROTL30(cv1)$
      $cv4 \leftarrow cv3$
      $cv3 \leftarrow cv2$
      $cv2 \leftarrow Shft30endfor(0, 80)$
    end for
    $cv0 \leftarrow (cv0 + f1) mod 2^{32}$
    $cv1 \leftarrow (cv1 + f2) mod 2^{32}$
    $cv2 \leftarrow (cv3 + f3) mod 2^{32}$
    $cv3 \leftarrow (cv4 + f4) mod 2^{32}$
    $cv4 = (cv5 + f5) mod 2^{32}$
  end for
  $f1 \leftarrow cv0$
  $f2 \leftarrow cv1$
  $f3 \leftarrow cv2$
  $f4 \leftarrow cv3$
  $f5 \leftarrow cv4$
  return $output$ ▷ Return 160 Bit Hash
end procedure

The algorithm described above is the same as the classical SHA1. The only difference is that instead of performing simple classical addition, XOR, AND, NOT, OR, and mod, we used quantum addition, XOR, AND, NOT, OR, and mod operations.

2.3 Implementation of Quantum MD–5 Algorithm

The implementation of Quantum MD5 is done with the help of IBM Quantum and Qiskit. IBM Quantum is a platform that allows users to access quantum computers and build quantum circuits. Qiskit is an SDK tool that can be used in Python to implement quantum circuits. With the help of

these two, the required quantum circuits were designed and implemented.

Algorithm 14 Quantum–MD5 Algorithm

procedure QUANTUM–MD5($InputMessage$) ▷ Any kind of string of any length
  Take the input string and convert into its corresponding equivalent binary code.
  $ap \leftarrow dig + 1$ ▷ Add Padding
  while $len(ap)\%512 \neq 448$ **do**
    $Ap \leftarrow Ap + 0$
  end while
  Count the total length of the message and convert it into binary equivalent.
  $ap\_length \leftarrow msg\_length \bmod 2^{64}$
  $tot\_length \leftarrow ap + ap\_length$
  $n \leftarrow 512$
  $Chunks \leftarrow [tot\_length[i : i + n]foriinrange(0, len(tot\_length), n)]$
  Initialize 4 chaining variables and 64 additive constants.
  $cv0 \leftarrow' 01100111010001010010001100000001'$
  $cv1 \leftarrow' 11101111110011011010101110001001'$
  $cv2 \leftarrow' 10011000101110101101110011111110'$
  $cv3 \leftarrow' 00010000001100100101010001110110'$
  $f1 \leftarrow cv0$
  $f2 \leftarrow cv1$
  $f3 \leftarrow cv2$
  $f4 \leftarrow cv3$
  $K \leftarrow []$
  for $l = 0$ to $len(Chunks)$ **do**
    $n \leftarrow 32$
    $ch \leftarrow chunks[l]$
    for $l = 0$ to $len(ch, n)$ **do**
      $chu = ch[i : i + n]$
    end for
    for $t = 16$ to 80 **do**
      $M[t] \leftarrow ROTL1(M[t-3] \bigotimes M[t-8] \bigotimes M[t-14] \bigotimes M[t-16])$
    end for
    for $t = 0$ to 80 **do**
      if $t \leq 19$ **then**
        $F \leftarrow (B\&\&C)||\neg(B\&\&D)$
      else if $(t > 19$ && $t \leq 39)$ **then**
        $F \leftarrow B \bigotimes C \bigotimes D$
      else if $(t > 39$ && $t \leq 59)$ **then**
        $F \leftarrow (B\&\&C)||(B\&\&D)||(C\&\&D)$
      else if $(t > 59$ && $t \leq 79)$ **then**
        $F \leftarrow (B \bigotimes C \bigotimes D)$
      end if
      $Add1 \leftarrow (F + cv4) mod 2^{32}$
      $Shft5 \leftarrow ROTL5(cv0)$
      $Add2 \leftarrow (Add1 + Shft5) mod 2^{32}$
      $cv1 \leftarrow cv0$

Algorithm 15 Quantum MD5 Continue

      $Add3 \leftarrow (Add2 + M[t]) mod 2^{32}$
      $Add4 \leftarrow (Add3 + K[t]) mod 2^{32}$
      $cv0 \leftarrow Add4$
      $Shft30 \leftarrow ROTL30(cv1)$
      $cv4 \leftarrow cv3$
      $cv3 \leftarrow cv2$
      $cv2 \leftarrow Shft30endfor(0, 80)$
    end for
    $cv0 \leftarrow (cv0 + f1) mod 2^{32}$
    $cv1 \leftarrow (cv1 + f2) mod 2^{32}$
    $cv2 \leftarrow (cv3 + f3) mod 2^{32}$
    $cv3 \leftarrow (cv4 + f4) mod 2^{32}$
    $cv4 = (cv5 + f5) mod 2^{32}$
  end for
  $f1 \leftarrow cv0$
  $f2 \leftarrow cv1$
  $f3 \leftarrow cv2$
  $f4 \leftarrow cv3$
  $f5 \leftarrow cv4$
  return $output$ ▷ Return 128 Bit Hash
end procedure

The aim of the work is to implement the MD5 algorithm in a quantum computer, report the resulting hash value along with the execution time after executing on multiple different processors, and compare the execution time with qSHA-1 and qSHA-256 .

The aforementioned algorithm is the same as standard MD5. The sole difference is that instead of performing simple classical addition, XOR, AND, NOT, and OR operations, quantum addition, Q-XOR, Q-AND, Q-NOT, and Q-OR operations are used.

Quantum MD–5 Algorithm Steps

Steps to implement MD5 algorithm in quantum computer are:

Step 1: Padding

Before starting the MD 5 algorithm, padding of the input message is needed. The input message needs to be a multiple of 512. If the message is not a multiple of 512 then we calculate how many bits are missing.

Case I:

If the missing bits are more than 64 bits from it being a multiple of 512 then we pad the message by adding 1 followed by 0s until the message is a 64 bit less than a multiple of 512. In the remaining 64 bits we append the original size of the message block. This is done to increase the complexity of the input message block.

Case II:

If the missing bits is less than 64 bits from it being a multiple of 512 then we pad the message by 1 followed by 0s until the message is 64 bits less than the next multiple of 512. (Eg: if message is of size 450 bits than we will pad it with 1 followed by 0s till its of the size 960 so it can be exactly 64 bits less than the next multiple of 512 that being 1024 in this case.)

Step 2: Chunking up of the Padded Message

1) After the padding is done the message block is divided into chunks each of size 512 bits.

2) Each 512 bits chunk is then divided into 16 sub-block each of size 32 bits.
3) In this algorithm we save those chunk values in an array M.

Step 3: Initialize buffer, shift values and constant values

The MD 5 algorithm uses 4 chaining variables or buffers each of size 32 bits each. These buffers have hexadecimal values. These buffers are:

cv1:0x67452301('01100111010001010010001100000001')
cv2:0xefcdab89 ('11101111110011011010101110001001')
cv3:0x98badcfe ('10011000101111010110111001111110')
cv4:0x10325476 ('00010000001100100101010001110110')

and also save these values in final_cv1 = cv1 final_cv2 = cv2, final_cv3 = cv3 and final_cv4 = cv4 for later use. We initialize these buffer values in binary for our easy of execution later. Similar to the chaining variables we also initialize the number of bits that needs to be circular left shifted for every round in an array.

shift = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

And all the 64 different constant that needs to be used for every round in the algorithm.

K= [0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8, 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665, 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1, 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391]

L=[1,6,11,0,5,10,15,4,9,14,3,8,13,2,7,12,5,8,11,14,1,4,7,10,13, 0,3,6,9,12,15,2,0,7,14,5,12,3,10,1,8,15,6,13,4,11,2,9]

Step 4: Start the loop and implement the operation of per round

1) We start a loop for each 512 bits chunk.
2) Inside that loop, we start another loop for each round from 0-64.
3) Under that loop we start to implement operation as round recommendation.

For round (0-15):

We first perform the operation F(B,C,D) = (B AND C) OR (NOT B AND D).
(B AND C) is done by using the values cv2 and cv3 and using algorithm 4 to generate “finalAND”.
(NOT B) is done by using the value cv2 and using the algorithm 2 to generate “finalNOT”.
(NOT B AND D) is done by using the value finalNOT and cv4 and using algorithm 4 to generate “finalAND2”.
(B AND C) OR (NOT B AND D) is done by using the value finalAND and finalAND2 and using the algorithm 3 to generate “finalPer”.

For round (16-31):

We perform the operation G(B,C,D) = (D AND B) OR ((NOT D) AND C)
(D AND B) is done by using the values cv4 and cv2 and using algorithm 4 to generate “finalAND”.
(NOT D) is done by using the values cv4 and algorithm 2 to generate “finalNOT”.
(NOTD AND C) is done by using the values finalNOT and cv3 and using algorithm 4 to generate “finalAND1”.
(D AND B) OR ((NOT D) AND C) is done by using the values “finalAND” and “finalAND1” and using the algorithm 3 to generate “finalPer”.

For round (32 – 47):

We perform the operation H(B,C,D) = B XOR C XOR D
(B XOR C) is done by using the values cv2 and cv3 and using algorithm 1 to generate “finalXOR1”.
(B XOR C XOR D) is done by using the values finalXOR1 and cv4 to generate “finalPer”.

For round (48 – 63):

We perform the operation I(B,C,D) = C XOR (B OR NOTD)
(NOT D) is done by using the value cv4 and algorithm 2 to generate “finalNOT’.
(B OR NOT D) is done by using the values cv2 and finalNOT and using algorithm 3 to generate “finalPer1”.
(C XOR (B OR NOT D) is done by using the values cv3 and finalPer1 and using algorithm 1 to generate “finalPer”.

First Addition

We perform the operation A + (result of F(B,C,D) or G(B,C,D) or H(B,C,D) or I(B,C,D)) by using the values cv1 and final Per and algorithm 5 to generate “sm”.
Modulo operation is done by using the values sm and (01111111111111111111111111111111) and using algorithm 4 to generate ‘final’.

Second Addition

We perform the operation (result of first addition) + M.
For round (m = 0-15):
We take the value of M to M[m] and using the value of final and algorithm 5 to generate “sm”.
For round (m = 16-63):
We take the value of M to M[L[m – 16]] and using the value of final and algorithm 5 to generate “sm”.
Modulo operation is done by using the values sm and (01111111111111111111111111111111) and using algorithm 4 to generate ‘final’.

Third Addition

We perform the operation (result of second addition) + K.
Here we using a function namely “binaryNUM’ implemented earlier which converts the hexadecimal constant values to binary.
So, we take the values binaryNUM(K[m]) and final and using algorithm 5 to generate “sm”.
Modulo operation is done by using the values sm and (01111111111111111111111111111111) and using algorithm 4 to generate ‘final’.
Then we circular left shift the “final” value by shift[m] values to generate “shft30”.

Fourth Addition

We perform the operation (shft30) + cv2 by using the values shft30 and cv2 and using the algorithm 5 and 6 to generate "sm".

Modulo operation is done by using the values sm and (01111111111111111111111111111111) and using algorithm 4 to generate 'final'.

After fourth addition we save the result as following:

cv1 = cv4

cv4 = cv3

cv3 = cv2

cv2 = final

This process is repeated for 64 rounds, after that the final values of cv1,cv2,cv3 and cv4 go through the following process:

Addition between cv1 and final_cv1 is done by take the value cv1 and final_cv1 and using the algorithm 5 to generate "sm".

Modulo operation is done by using the values sm and (01111111111111111111111111111111) and using algorithm 4 to generate 'final'. The final value is saved in both cv1 and final_cv1.

This process is repeated for cv2 and final_cv2, cv3 and final_cv3, cv4 and final_cv4.

This concludes the operation for chunk – 1. If there are multiple chunks then this whole process is repeated again for each chunk.

After that the values of final_cv1, final_cv2, final_cv3 and final_cv4 is appended together to generate the resulting hash value of the algorithm.

2.4 Implementation of Quantum SHA–256 Algorithm

To implement SHA–256 in a quantum computer, the below circuit 8 is used to perform the addition with the help of 4 qubits and 2 classical bits. The first layer is the input layer, where qubits q[0] and q[1] are used to hold the two input bits, and qubit q[2] is used to hold the carry bit. The qubit q[3] is a helping qubit that is initially initialised to zero. At first, the CCNOT gate was applied to q[0], q[1], and q[3], then the CNOT gate was applied to q[0] and q[1]. After that, again, the CCNOT gate was applied to q[1], q[2], and q[3], followed by the CNOT gate, which was applied to q[1] and q[2]. After applying the gate, the next task is to measure the qubits. To do so, the measure gate is applied to q[3] and q[2], and their values are stored in the c[0] and c[1] classical bits, respectively. The classical bits c[0] and c[1] store the values of sum and carry, respectively.

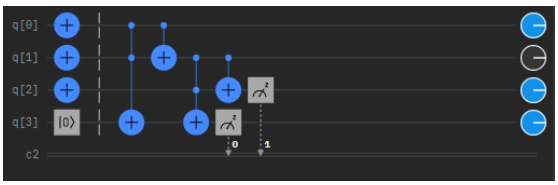


Fig. 8: Circuit diagram of q-Adder for SHA–256

Algorithm 16 Quantum–SHA-256 Algorithm

procedure QUANTUM–SHA-256($InputString$) ▷ Any kind of string of any length

Take the input string and convert into its corresponding equivalent binary code.

Add Padding

$ap \leftarrow dig + 1$

while $len(ap)\%512 \neq 448$ **do**

$Ap \leftarrow Ap + 0$

end while

Append length

Count the total length of the message and convert it into binary equivalent.

$ap\_length \leftarrow msg\_length \bmod 2^{64}$

$tot\_length \leftarrow ap + ap\_length$

Divide the input into 512 bit blocks

$n \leftarrow 512$

$Chunks \leftarrow [tot\_length[i : i + n] for i in range(0, len(tot\_length), n)]$

Initialize 7 chaining variables and 64 additive constants.

$cv0 \leftarrow'$ 01101010000010011110011001100111′

$cv1 \leftarrow'$ 10111011011001111010111010000101′

$cv2 \leftarrow'$ 00111100011011101111001101110010′

$cv3 \leftarrow'$ 10100101010011111111010100111010′

$cv4 \leftarrow'$ 01010001000011100101001001111111′

$cv5 \leftarrow'$ 10011011000001010110100010001100′

$cv6 \leftarrow'$ 00011111100000111101100110101011′

$cv7 \leftarrow'$ 01011011111000001100110100011001′

$f1 \leftarrow cv0$

$f2 \leftarrow cv1$

$f3 \leftarrow cv2$

$f4 \leftarrow cv3$

$f5 \leftarrow cv4$

$f6 \leftarrow cv5$

$f7 \leftarrow cv6$

$f8 \leftarrow cv7$

$k \leftarrow$ ['01000010100001010001011111001100 0′,'01110001001101110100010010010001′,'10110101110000001111101111001111′,'11101001101101011101101110100101′,'00111001010101101100001001011011′,'01011001111100010001000111110001′,'10010010001111111000001010100100′,'10010000101111101111111111110′,'101001 00010100000110110011101011′,'101111101111001101000111110111′, ...'11000110011100010111100011110010′] ▷ 64 constants K, one for each step. First 32 bits of the fractional sections of the cube roots of the first 64 prime numbers, from 2 to 311

Process the blocks

for $l = 0$ to $len(Chunks)$ **do**

$n \leftarrow 32$

$ch \leftarrow chunks[l]$

$Chunks \leftarrow [ch[i : i + n] for i in range(0, len(ch), n)]$

Preparing the Message Schedule

for $t = 16$ to 64 **do**

$msg \leftarrow \prod_1^{256}(chu_{t-2}) + chu_{t-7} + \prod_0^{256}(chu_{t-7}) + chu_{t-16}$

$chu.append(msg)$

end for

for $m = 0$ to 64 **do**
$\alpha(cv0, cv1, cv2) \leftarrow (cv0 \wedge cv1) \veebar (\neg cv0 \wedge cv2)$
$\beta(cv4, cv5, cv6) \leftarrow (cv4 \wedge cv5) \veebar (cv4 \wedge cv6) \veebar (cv5 \wedge cv6)$
$add1 \leftarrow \Sigma_{i=0}^{256} kn + \beta(cv4, cv5, cv6)) \% 4294967296$
$add2 \leftarrow \alpha(cv0, cv1, cv2) + cv7) \% 4294967296$
$add3 \leftarrow (add2 + \Sigma_{i=1}^{256} cv4) \% 4294967296$
$add4 \leftarrow (add3 + K[m]) \% 4294967296$
$add5 \leftarrow (add4 + chu[m]) \% 4294967296$
$add6 \leftarrow (add5 + cv3) \% 4294967296$
$add7 \leftarrow (add5 + add1) \% 4294967296$
$cv7 \leftarrow cv6$
$cv6 \leftarrow cv5$
$cv5 \leftarrow cv4$
$cv4 \leftarrow add6$
$cv3 \leftarrow cv2$
$cv2 \leftarrow cv1$
$cv1 \leftarrow cv0$
$cv0 \leftarrow add7$
end for
end for
$cv0 \leftarrow (cv0 + f1) \% 4294967296$
$f1 \leftarrow cv0$
$cv1 \leftarrow (cv1 + f2) \% 4294967296$
$f2 \leftarrow cv1$
$cv2 \leftarrow (cv2 + f3) \% 4294967296$
$f3 \leftarrow cv2$
$cv3 \leftarrow (cv3 + f4) \% 4294967296$
$f4 \leftarrow cv3$
$cv4 \leftarrow (cv4 + f5) \% 4294967296$
$f5 \leftarrow cv4$
$cv5 \leftarrow (cv5 + f6) \% 4294967296$
$f6 \leftarrow cv5$
$cv6 \leftarrow (cv6 + f7) \% 4294967296$
$f7 \leftarrow cv6$
$cv7 \leftarrow (cv7 + f8) \% 4294967296$
$f8 \leftarrow cv7$
return *output* ▷ Return 256 Bit Hash
end procedure

3 Experimental Results and Analysis

The execution times of the algorithms that were implemented using Qiskit (qSHA-1, qMD5, and qSHA-256) are much slower when compared to the execution times of the traditional algorithms. It's possible that the execution of quantum circuits in a classical computer is one of the reasons why quantum computers have a slower execution time. Another possibility is that quantum computers solve their operations bit-by-bit, which takes a significant amount of time and is another reason why quantum computers have a slower execution time. However, the prospect of running encryption on a quantum computer rather than a classical computer opens the door to future research into the enhancement and invention of encryption methods that are both better and more robust.

3.1 Experimental Results of Quantum MD–5

It has been confirmed that the MD5 algorithm can be implemented on a quantum computer after implementation. However, the execution time is longer when compared to traditional MD5. This could be due to a variety of factors influencing the execution time.

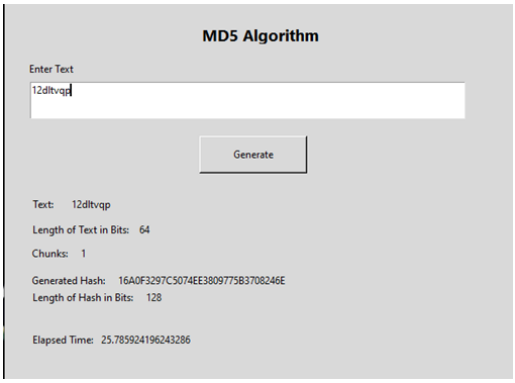


Fig. 9: Result 1

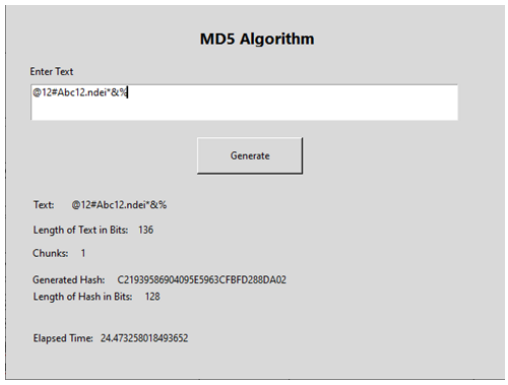


Fig. 10: Result 2

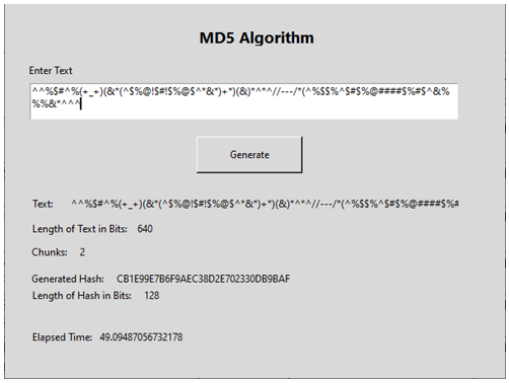


Fig. 11: Result 3

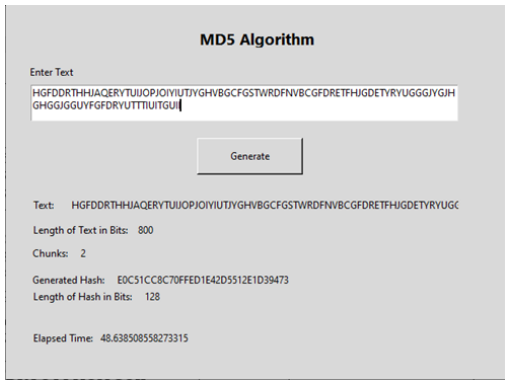


Fig. 12: Result 4

Figures 9, 10, 11, and 12 with various lengths of inputs always provide a fixed length of 128 bits of output. There will be different hashes for different inputs.

In table 1, the output is calculated using various sizes of input. Regardless of the input sizes, the output of the q-MD5 method is always a fixed size of 128 bits with hexadecimal digits. For each input, a separate hash is used. Table 2 shows the execution timings for the same input when run on different CPUs. A faster CPU will reduce the amount of time required. The execution time is determined by the machine's power. Two distinct processors use the same input and carry it out; the execution times of these two processors are shown in figure 13. According to the findings, it appears that the duration of the execution is directly proportional to the type of processors that were employed. This means that a powerful processor should make the execution time shorter. The execution time is determined by the machine's power.

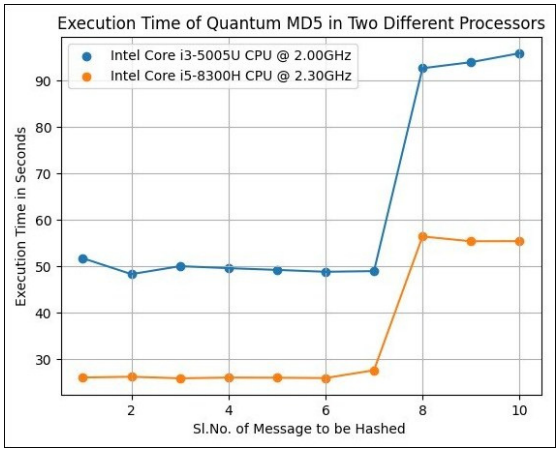


Fig. 13: Execution time for different processors

In table 3, several inputs are taken and run many times, with the execution time recorded. The execution time doubles as the number of 512-bit chunks rises.

From the above results, we can come to the conclusion that MD5 can be implemented in a quantum environment. Both properties of MD5, i.e., producing a 128-bit output hash for any given input length and also producing the same output hash for the same input, are being followed by the designed algorithm. The execution time is directly dependent on the number of chunks formed by the input message. The execution times almost double when the number of chunks is increased from 1 to 2. The execution time also depends on the processor's power. There is a significant decrease in execution time just by switching to a better and faster-performing processor.

3.2 Experimental Results of Quantum SHA–1

For the sake of dealing with quantum computers on the level of individual circuits, pulses, and algorithms, this article makes use of an open source software development kit called Qiskit. It is designed to help developers create, edit, and test quantum programmes that can be executed on IBM's Quantum Experience prototype devices or locally on computer simulators. These programmes can also be performed on IBM's Quantum Experience. In order to implement the quantum SHA-1 algorithm on a local computer, the IBM QASM simulator is used.

The QASM simulator, which is included in QiskitAer, is used to simulate the quantum circuit. It is necessary to execute the circuit a great number of times so that a compilation of statistics regarding the distribution of the bitstrings can be made. Through the use of the shots keyword, the excute function allows the user to choose the number of iterations that will be performed on the circuit. 1024 different shots were taken for this piece of work.

The SHA-1 algorithm that was discussed earlier is identical to the one that was just described. The only thing that is different is that rather than executing a straightforward classical addition, XOR, AND, NOT, OR, AND MOD, quantum addition is employed along with the corresponding operations of XOR, AND, NOT, OR, and MOD. The process of doing quantum addition makes use of the idea of a Full Adder, in which two circuits were created, one for the purpose of computing CARRY and the other for the purpose of computing SUM. The SUM and CARRY circuits can do quantum addition.

The Qiskit implementation of the SHA-1 algorithm executes far more slowly than the regular SHA-1 method, which is denoted by the notation q-SHA-1. The execution of quantum circuits on a conventional computer is most likely one of the elements behind the slower processing speed of quantum computers.

Due to a phenomenon known as quantum parallelism, which results from superposition, quantum computers can achieve quadratic or exponential gains in solution speed when compared to classical computers. Some probabilistic operations can be carried out faster by quantum parallelism than by traditional methods. However, quantum computing does not provide such a boost for all problems, and researchers are still figuring out where it is most useful. For some types of difficulties, quantum computing has the potential to provide astonishing results [29].

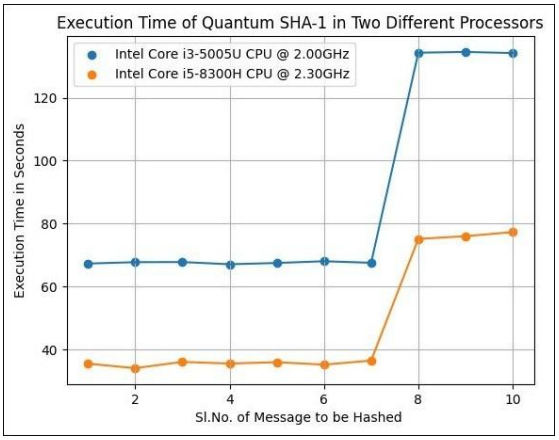


Fig. 14: Execution time for different processors

In table 4, execution time is calculated for different inputs. When the number of 512-bit blocks increases, execution time doubles.

In table 5, for the same input, when we execute the same input on different processors, we get two different execution times. A higher processor will take less time. Execution time is dependent on the power of the machine.

Two distinct processors use the same input and carry it out; the execution times of these two processors are shown

TABLE 1: Hashes of qMD5 with Various Inputs

SL. No.	INPUT	INPUT LENGTH (bits)	OUTPUT HASH (hexadecimal)	OUTPUT LENGTH (bits)
1.	12dltvqp	64	16A0F3297C5074EE3809775B3708246E	128
2.	@12#Abc12.ndei*&%	136	C21939586904095E5963CFBFD288DA02	128
3.	%$#$$%*&%(^^&%$&$% ^&#*(&)(+)	280	BE27D500AF1DFDAFC0577FA3573B9832	128
4.	ABNYGFTRVVHGFGFGFGCGHD FDDDDGFDCCFDD	280	BE65B41172A064C8B5D671FAF8630F4A	128
5.	12489489746546546878789787468787987	280	AEB56E86B870B41A6361DE5ADAB26176	128
6.	abbdcgufrhurfghvdnbvcfdsndcgsd gchgs	280	590F31282042 222BB4D6063 43E52A869	128
7.	abbdcgufrhurfghvdnbvcfdsndcgs dgchgsdcnvdbcmbdgbwdbnbvdv	440	73102D5A44357 6AD7A9DA434 0F5304C	128
8.	1556787128798600364712149535487 878794554547845457874211647 244	488	3221CF855982B8126535384462CB0771	128
9.	^^%$#^%(+_+)(&*(^$%$#$%$^*&*)+*) (&)*^*^//—/*(^%$$%^$# $%####$% #$^&%%%&*^^^	640	CB1E99E7B6F9AEC38D2E702330DB9BAF	128
10.	HGFDDRTHHJAQERYTUIJOPJOIY IUTJYGHVBGCFGSTWRDFNV BCGFDRETFHJGDETYRYUGGGJ YGJHGHGGJGGUYFGFDRYUTTT IUITGUII	800	E0C51CC8C70FFED1E42D5512E1D39473	128

TABLE 2: Execution Time of Q-MD5 in two Different Processors

SL. No.	INPUT	Intel® Core™ i3-50005U CPU @ 2.00 GHz	Intel® Core™ i5-83005H CPU @ 2.30 GHz
1.	12dltvqp	51.76	26.11
2.	@12#Abc1 2.ndei*&%	48.34	26.27
3.	%$#$^$%*&%(^*^*&%$&^$%*^*&#*(&)(+)	50.05	25.94
4.	ABNYGFTRVV HGFGFGFGCG HDFDDDDGFDC CFDD	49.63	26.09
5.	1248948974 6546546878 789787468 787987	49.25	26.06
6.	abbdcgufrh urfghvdnbv cfdsndcgsd gchgs	48.85	25.99
7.	Abbdcgufrh urfghvdnbv cfdsndcgsd gchgsdcnvd bcmbdgbwd bnbvdv	48.99	27.67
8.	1556787128798600364712149535487878794554547845457874211647244	92.63	56.44
9.	^^^%$#^%(+_+)(&*(^$%$#%$^*&*)+*)(&)*^*^//—/*(^%$$%^$#$% ####$%#$^&%%% &*^^^	93.93	55.41
10.	HGFDDRTHHJA QERYTUIJOPJ OIYIUTJYGHV BGCFGSTWRDF NVBCGFDRETF HJGDETYRYUGG GJYGJHGHGG JGGUYFGFDR YUTTTIUIT GUII	95.85	55.44

in figure 14. According to the findings, it appears that the duration of the execution is directly proportional to the type of processors that were employed. This means that a powerful processor should make the execution time shorter. The execution time is determined by the machine's power.

3.3 Experimental Results of Quantum SHA–256

The Qiskit implementation of the SHA-256 algorithm executes far more slowly than the regular SHA-256 method, which is denoted by the notation q-SHA-256. The execution of quantum circuits on a conventional computer is most likely one of the elements behind the slower processing speed of quantum computers. In table 6, the output is calculated using various sizes of input. Regardless of the input sizes, the output of the q-SHA-256 method is always a fixed size of 256 bits with hexadecimal digits. For each input, a separate hash is used. In table 6, for the same input, when we execute the same input on different processors, we get two different execution times. A higher processor will take less time. Execution time is dependent on the power of the machine.

Two distinct processors use the same input and carry it out; the execution times of these two processors are shown in figure 15. According to the findings, it appears that the duration of the execution is directly proportional to the type of processors that were employed. This means that a powerful processor should make the execution time shorter. The execution time is determined by the machine's power.

TABLE 3: Execution Time for different inputs in Q-MD5

SL. No.	INPUT	No. of 512-bit chucks	T-1	T-2	T-3	T-4	T-5	Avg Time
1.	12dltvqp	1	27.22	26.44	25.92	25.59	25.39	26.11
2.	@12#Abc12.ndei&%	1	25.90	25.37	26.26	27.80	26.05	26.27
3.	%$#$$%&%(	1	25.67	25.68	27.41	24.45	26.55	25.94
4.	ABNYGFTRVVHGFGFGFGC GHDFDDDDGFDCCFDD	1	24.76	26.21	25.81	26.40	27.31	26.09
5.	124894897465465468787897 87468787987	1	26.48	24.55	27.81	25.58	25.87	26.06
6.	abbdcgufrhurfghvdnbvcfdsnd cgsdgchgs	1	25.07	26.19	26.78	27.00	24.93	25.99
7.	abbdcgufrhurfghvdnbvcfdsnd cgsdgchgsdcnvdbcmbdgbwdb nbvdv	1	25.38	26.71	28.72	29.96	27.61	27.67
8.	155678712879860036471214953 548787879455454784545787421 1647244	2	57.31	56.17	55.59	58.92	54.21	56.44
9.	^^%$#^%(+_+)(&*(^$%$#$%$^*& *)+*)(&)*^*^///*(^%$$%^$#$% ####$%#$^&%%%&*^^^—	2	53.81	53.79	54.21	55.61	59.66	55.41
10.	HGFDDRTHHJAQERYTUIJO PJOIYIUTJYGHVBGCFGSTW RDFNVBCGFDRETFHJGDET YRYUGGGJYGJHGHGGJGGU YFGFDRYUTTTIUITGUII	2	55.72	57.12	55.85	53.59	54.95	55.44

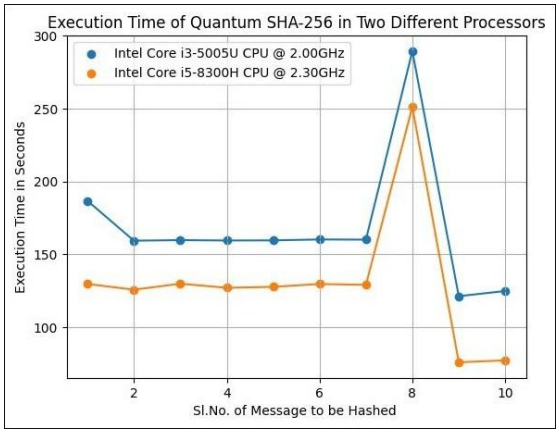


Fig. 15: Execution time of qSHA-256 for different processors

3.4 Performance Analysis Between Quantum SHA–1 , Quantum MD–5 and Quantum SHA–256

Fig. 16: Execution time of cSHA-1, cMD5 and cSHA-256 for the same inputs

Execution time difference between classical SHA-1, classical MD5 and classical SHA-256 for the same inputs is given in the figure 16. In table 9, it can be seen that qSHA-256 is taking more time then qMD5 and qSHA-1 for the same arbitrary input.

TABLE 4: Quantum SHA-1 execution time with respect to no. of 512 bit blocks

SL. No.	Message To be Hashed	No. of 512 bit blocks	Execution Time in Second	Output
1.	12dltvqp	1	35.43	F6AD80FDFC7B71BB6DA3FE5360120C9C30CEAB2C
2.	@12#Abc12.ndei&%	1	35.14	DF8B25E1F725044094BDE19EF8 9B63F944B37C4
3.	%$#$$%&%(^^&%$&$%^&#(&)(+)	1	35.99	5C7F397FD10ADA476AA09F6BE4C0411B36376EF9
4.	ABNYGFTRVVHGFGFGFGCGHDFD DDDGFDCCFDD	1	35.47	9A94549E690300CE7E74278EB7BD2E0BD8911AAB
5.	124894897465465468787897 87468787987	1	35.90	1573BF989E94CA61B489D5AA61ED4BF7E9A89161
6.	Abbdcgufrhurfghvdnbvcfdsndcgsdgchgs	1	35.12	A215E938CAE939D8D1CF14B160EC3A5C29D78273
7.	Abbdcgufrhurfghvdnbvcfdsndcgsdgc hgsdcnvdbcmbdgbwdbbvdv	1	36.41	6E670ABBA04F9CC4F40863D957BF57C54B215A87
8.	1556787128798600364712149535487878 794554547845457874211647 244	2	75.11	1A6A96068298FEF1627426E74249E613A02460C1
9.	e$	2	75.98	8BA00DD2C87C7D8555414B7FBD5EF06B331EA6A2
10.	HGFDDRTHHJAQERYTUIJOPJOIYI UTJYGHVBGCFGSTWRDFNVBCG FDRETFHJGDETYRYUGGGJYGJHG HGGJGGUYFGFDRYUTTTIUITGUII	2	77.28	3AEE40FCC411779C9CDE43690 D0CB896B890237

TABLE 5: Quantum SHA-1 execution time in two different Processors

Sl. No.	Message To be Hashed	Execution Time in Seconds of two different Processors	
		Intel® Core™ i3-5005U CPU @ 2.00GHz	Intel® Core™ i5-8300H CPU @ 2.30GHz
1.	12dltvqp	67.27	35.43
2.	@12Abc12.ndei*&%	67.71	34
3.	%$#$^$%*&% (^^*&%$@& ^$%*^*& #*(&)(+)	67.75	35.99
4.	ABNYGFTRVVHG FGFGFGCGHDFD DDDGFDCCFDD	67.05	35.47
5.	124894897465 465468787897 87468787987	67.45	35.90
6.	Abbdcgufrhur fghvdnbvcfd sndcgsdgchgs	68.01	35.12
7.	Abbdcgufrhurfghvdnbvcfdsndcgsdgchgsdcnvdbcmb dgbwdbnbvdv	67.51	36.41
8.	155678712879860036471214953548787879455454784545 78 74211647244	134.34	75.11
9.	^^%$#^%(+_+)(&*(^$%$#$%@$^*&*)+*)(&)*^*^//—/*(^%$ $%^$#$%@####$%# $^&%%%&*^^^	134.58	75.98
10.	HGFDDRTHHJAQE RYTUIJOPJOIYI UTJYGHVBGCFGS TWRDFN-VBCGFD RETFHJGDETYRY UGGGJYGJHGHGG JGGUYFGFDRYUT TTIUITGUII	134.22	77.28

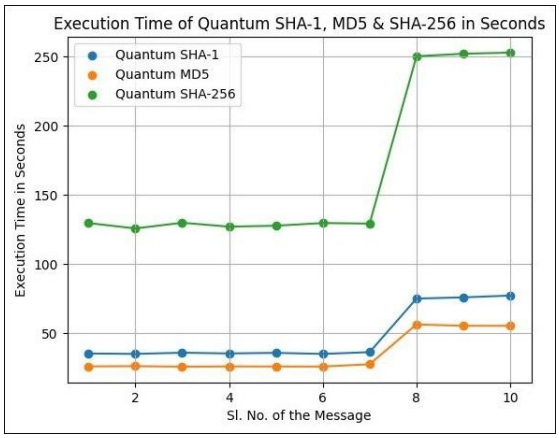


Fig. 17: Execution time of qSHA-1, qMD5 and qSHA-256 for the same inputs

The execution time difference between qSHA-1, qSHA-256, and qMD5 for the same inputs is given in figure 17. If we observe the figure 16, then classical SHA-1 is taking less time than classical MD5 and classical SHA-256, but in the case of quantum version comparison, quantum MD5 is taking less time than quantum SHA-1 and quantum SHA-256 in figure 17, which is the reverse of classical comparison. A new thing that was found through this work is that qMD5 is taking less time than qSHA-1 and qSHA-256. But in the case of classical comparison, cSHA-1 takes less time than cMD5 and cSHA-256. One thing that can be said from this work is that the classical performance and quantum performance of any algorithm may not be similar, but rather opposite.

TABLE 6: Output Hash With Various Inputs of SHA-256

SL. No.	INPUT	INPUT LENGTH (bits)	OUTPUT HASH (hexadecimal)	OUTPUT LENGTH (bits)
1.	12dltvqp	64	127A9EC06672D107D40CEEEF6F83D3A16EB646A5929CBC1D47E0A995715CADF7	256
2.	@12#Abc12.ndei*&%	136	B6728E5F88E71761ADAEC00DF9E80B077942240E86AE8BBB9D68542A617F54EB	256
3.	%$#$$%*&%(*^&%$&$%^&#*(&)(+)	280	351A2322829AED78FA29B0B1AC34E0A9325461EF4492FB8A8BC0BF35E3F514B2	256
4.	ABNYGFTRVVHGFGFGFGCGHDFDDDDGFDCCFDD	280	BBFEF4BBC89E506A7418E0FB730C90A3CECDCEC4552B060A7AC98238EE09A30C	256
5.	12489489746546546878789787468787987	280	D9B9E71A2128332CCFA6FC6D6C912AF0267C7819ED3CBE4EF1E2BADD8A3AE4FB	256
6.	abbdcgufrhurfghvdnbvcfdsndcgsdgchgs	280	39C8B2B6AA43549292CFA649EB06C946D349CE0A8BFD18CF33DF89A497A5BB07	256
7.	abbdcgufrhurfghvdnbvcfdsndcgsdgchgsdcnvdbcmbdgbwdbnbvdv	440	1B25E1F3F4C5B3324064B557484466A3D1C3CA4A114916002C959CC73E256E82	256
8.	1556787128798600364712149535487878794554547845457874211647	244	A1314F2708AC51CA451196A4CDA092DCC7B655A97B6E5BC1B826ED48FA86C214	256
9.	^^%$#^%(+_+)(&*(^$%$#$%$^*&*)+*)(&)*^*^//—/*(^%$$%^$# $%####$%#$^&%%%&*^^^	640	FB5B840462711405E6EB50D275BBC93CA05AAF010F8B7F98563379EDA7553C5F	256
10.	HGFDDRTHHJAQERYTUIJOPJOIYIUTJYGHVBGCFGSTWRDFNVBCGFDRETFHJGDETYRYUGGGJYGJHGHGGJGGUYFGFDRYUTTTIUITGUII	800	56DF90F802B0513C0F8C624FB359CDBE456A014D76D52D844AADC9D00AE65711	256

4 Conclusion

From the experimental results, it can be concluded that the proposed qSHA1, qSHA-256, and MD5 run successfully in a quantum environment. The execution time of the proposed algorithms is directly proportional to the number of chunks formed by the input message. The execution time doubles when the number of chunks is increased from 1 to 2. The execution time also depends on the processor's speed. In the experiment, it is observed that the classical part is faster than the quantum part. However, the number of cycles per unit time in the proposed quantum algorithms is higher than that of the classical algorithms. Hence, there is a possibility to reduce the execution time of the proposed algorithm by using optimization techniques. The reason for the longer execution of the proposed algorithms may be due to the execution mix-up between classical computers and quantum computers. Since the inputs are sent from the classical computer to the quantum simulator, the results are coming back to the classical computer. In the future, when a pure quantum computer is implemented, the real-time execution may be measured.

In this paper, the detailed comparison of quantum versions of the SHA-1, SHA256, and MD5 algorithms is presented in Table 9. Interestingly, the proposed quantum algorithms have the same message digest as their classical counterparts. The major contributions of this paper are the implementation of certain functions and the design of their circuits purely using quantum gates. The functions implemented for basic qubit additions, XOR, AND, NOT, and OR operations are available in a classical computer and are often used for the implementation of hash algorithms. Further, from Table 4, it is clear that execution time gets almost doubled, when the number of blocks increases from 1 to 2. Also, from Table 5, we can conclude that execution time varies greatly depending on the type of machine. In the Intel processor i5, the execution time is almost reduced to half as compared to that of the Intel i3 processor. Finally, it is concluded that the proposed algorithms take between 35 and 70 seconds on average. In the future, when the quantum computer is fully functional, these algorithms will run efficiently. Future research can be further extended to optimize the proposed algorithms to reduce the overall execution time.

4.1 Future Work

It's paramount to highlight that the circuits constructed within this research lay a robust foundation for the evolution of novel algorithms in the upcoming years. These circuits aren't just static; they offer avenues for optimization through potential rewiring to boost their execution speeds.

This work provides a crucial stepping stone for quantum cryptography enthusiasts and researchers. In an era where Post Quantum Encryption (PVE) is becoming increasingly vital, our algorithm acts as a pivotal reference. We eagerly invite the academic and research community to leverage our foundational quantum circuits, not only to amplify their efficiency but to innovate, pushing boundaries to attain optimal execution times and fortify quantum cryptographic methodologies.

TABLE 7: Comparison Table of Quantum SHA-256 execution time in two different Processors

Sl. No.	Message To be Hashed	Execution Time in Seconds of two different Processors	
		Intel® Core™ i3-5005U CPU @ 2.00GHz	Intel® Core™ i5-8300H CPU @ 2.30GHz
1.	123456	186.69	129.66
2.	abcfbcewhiufhiwue kbdfhebfhb	159.38	125.77
3.	13135446ddffbjhk hghggkugkgh	159.77	129.875
4.	CGJFGFFRYTEGJHLK UUKBCDDYFJOJUOU	159.52	127.07
5.	SHTSTEYJBJPJOJHF12 51354cxgfdtgdg$$&	159.59	127.72
6.	*)&(%&$#Q@#$% _)(*&%$$@$@%#$$	160.23	129.67
7.	hfbhkswegAHGIGG*&(* %%%%&*16547 454546478AGhhdgygh	160.11	129.07
8.	Hediywgerhbqwdigd3 14987448784R%$%#4 6585889848r27676 4685$%&$9	289.41	251.11
9.	^^%$#^%(+_+)(&*(^$%$#$%$@$$^*&*)+*)(&)*^*^//—/*(^%$ $%^$#$%$@$####$%# $^&%%%&*^^^	121.32	75.98
10.	HGFDDRTHHJAQE RYTUIJOPJOIYI UTJYGHVBGCFGS TWRDFN-VBCGFD RETFHJGDETYRY UGGGJYGJHGHGG JGGUYFGFDRYUT TTIUITGUII	124.85	77.28

TABLE 8: Execution Time Comparison between three Hashing Algorithms

SL. No.	INPUT	SHA-1 @ 2.00 GHz	MD5	SHA-256
1.	12dltvqp	0.0039970	0.0059965	0.00492545
2.	$12\#Abc12.ndei\&\%$	0.0039784	0.0060489	0.0049682
3.	%$#$$%*&%(*&%$@&$%**& #*(&)(+)	0.0039149	0.0059879	0.0049334
4.	ABNYGFTRVV HGFGFGFGCGH DFDDDDGFDC CFDD	0.0039153	0.0059733	0.0049799
5.	124894897 46546546878 78978746 8787987	0.0039633	0.0060735	0.0049166
6.	abbdcgufrh urfghvdnbvc fdsndcgsdg chgs	0.0039975	0.0059971	0.0049993
7.	Abbdcgufrh urfghvdnbv cfdsndcgsd gchgsdcnvd bcmbdgbwdb nbvdv	0.0039982	0.0060167	0.0049970
8.	1556787128 79860036471 21495354878 7879455454 7845457874 211647244	0.0059135	0.0088956	0.0078251
9.	%$#%(+ +)(&*($%$#$%@$*&*)+*) (&)**//—/*(%$ $ %$#$%@### $%# $&%%%& *	0.0059950	0.0089937	0.0079805
10.	HGFDDRTHHJ AQERYTUIJO PJOIYIUTJYG HVBGCFGSTW RDFNVBCGF DRETFHJGD ETYRYUGGG JYGJHGHGG JGGUYFGFDR YUTTTIUITGUII	0.0059728	0.0089765	0.0079273

REFERENCES

[1] *Secure hash standard*. National Institute of Standards and Technology, Washington, 2002. URL: http://csrc.nist.gov/publications/fips/. Note: Federal Information Processing Standard 180-2.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[3] Fábio Borges, Paulo Ricardo Reis, and Diogo Pereira. A comparison of security and its performance for key agreements in post-quantum cryptography. *IEEE Access*, 8:142413–142422, 2020.

[4] Tiago M. Fernández-Caramés and Paula Fraga-Lamas. Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks. *IEEE Access*, 8:21091–21116, 2020.

[5] Richard P. Feynman. Quantum mechanical computers. *Foundations of Physics*, 16(6):507–531, 1986.

[6] Gottesman. Private key and public key quantum cryptography. In *2002 Summaries of Papers Presented at the Quantum Electronics and Laser Science Conference*, pages 189–, 2002.

[7] B.C. Grau. How to teach basic quantum mechanics to computer scientists and electrical engineers. *IEEE Transactions on Education*, 47(2):220–226, 2004.

[8] Shay Gueron. Speeding up sha-1, sha-256 and sha-512 on the 2nd generation intel® core™ processors. In *2012 Ninth International Conference on Information Technology - New Generations*, pages 824–826, 2012.

[9] O.L. Guerreau, F.J. Malassenet, S.W. McLaughlin, and J.-M. Merolla. Quantum key distribution without a single-photon source using a strong reference. *IEEE Photonics Technology Letters*, 17(8):1755–1757, 2005.

[10] Helena Handschuh. *SHA Family (Secure Hash Algorithm)*, pages 565–567. Springer US, Boston, MA, 2005.

[11] Vladislav S. Igumnov and Vadim N. Lis. Influence of quantum computers on classical cryptography. In *2007 8th Siberian Russian Workshop and Tutorial on Electron Devices and Materials*, pages 220–224, 2007.

[12] A. Kahate. *Cryptography and Network Security*. McGraw Hill

TABLE 9: Execution Time Comparison between three quantum Hashing Algorithms

SL. No.	INPUT	Quantum SHA-1 @ 2.00 GHz	Quantum MD5	Quantum SHA-256
1.	12dltvqp	35.43	26.11	129.66
2.	$12\#Abc12.ndei\&\%$	35.14	26.27	125.77
3.	%$#$$%*&%(*&%$@&$%**& #*(&)(+)	35.99	25.94	129.87
4.	ABNYGFTRVV HGFGFGFGCGH DFDDDDGFDC CFDD	35.47	26.09	127.07
5.	124894897 46546546878 78978746 8787987	35.90	26.06	127.72
6.	abbdcgufrh urfghvdnbvc fdsndcgsdg chgs	35.12	25.99	129.67
7.	Abbdcgufrh urfghvdnbv cfdsndcgsd gchgsdcnvd bcmbdgbwdb nbvdv	36.41	27.67	129.27
8.	1556787128 79860036471 21495354878 7879455454 7845457874 211647244	75.11	56.44	250.16
9.	%$#%(+ +)(&*($%$#$%@$*&*)+*) (&)**//—/*(%$ $ %$#$%@### $%# $&%%%& *	75.98	55.41	251.95
10.	HGFDDRTHHJ AQERYTUIJO PJOIYIUTJYG HVBGCFGSTW RDFNVBCGF DRETFHJGD ETYRYUGGG JYGJHGHGG JGGUYFGFDR YUTTTIUITGUII	77.28	55.44	252.79

Education, 2020.

[13] I.G. Karafyllidis. Quantum computer simulator based on the circuit model of quantum computation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(8):1590–1596, 2005.

[14] Gary C Kessler. An overview of cryptography. 2003.

[15] P. Siva Lakshmi and G. Murali. Comparison of classical and quantum cryptography using qkd simulator. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 3543–3547, 2017.

[16] B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, M. P. de Almeida, A. Gilchrist, D. F. V. James, and A. G. White. Photonic quantum computing: Shor's algorithm and the road to fault-tolerance. In *2008 Conference on Lasers and Electro-Optics and 2008 Conference on Quantum Electronics and Laser Science*, pages 1–2, 2008.

[17] Logan O. Mailloux, Charlton D. Lewis, Casey Riggs, and Michael R. Grimaila. Post-quantum cryptography: What advancements in quantum computing mean for it professionals. *IT Professional*, 18:42–47, 2016.

[18] Surya Teja Marella and Hemanth Sai Kumar Parisa. Introduction to quantum computing. In Yongli Zhao, editor, *Quantum Computing and Communications*, chapter 5. IntechOpen, Rijeka, 2020.

[19] Marcin Niemiec. Quantum cryptography - the analysis of security requirements. In *2009 11th International Conference on Transparent Optical Networks*, pages 1–4, 2009.

[20] Vandana Pandey and Vinod Kumar Mishra. Architecture based on md5 and md5-512 bit applications. *International Journal of Computer Applications*, 74:29–33, 07 2013.

[21] Jeong-Hyun Park and Sun-Bae Lim. Key distribution for secure vsat satellite communications. *IEEE Transactions on Broadcasting*, 44(3):274–277, 1998.

[22] Anak Agung Putri Ratna, Prima Dewi Purnamasari, Ahmad Shaugi, and Muhammad Salman. Analysis and comparison of md5 and sha-1 algorithm implementation in simple-o authentication based security system. In *2013 International Conference on QiR*, pages 99–104, 2013.

[23] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[24] Bruce Schneier. Schneier on security: Cryptanalysis of sha-1, 02 2005.

[25] Mehrdad S. Sharbaf. Quantum cryptography: A new generation of information technology security system. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 1644–1648, 2009.

[26] Secure Hash Standard. Fips pub 180-1. *National Institute of Standards and Technology*, 17(180):15, 1995.

[27] Tommaso Toffoli. Reversible computing. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 632–644, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[28] S. William. *Cryptography and Network Security - Principles and Practice, 7th Edition*. Pearson Education India.

[29] Peter Wittek. 5 - unsupervised learning. In Peter Wittek, editor, *Quantum Machine Learning*, pages 57–62. Academic Press, Boston, 2014.

[30] Engin Zeydan, Yekta Turk, Berkin Aksoy, and S. Bugrahan Ozturk. Recent advances in post-quantum cryptography for networks: A survey. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–8, 2022.

Prodipto Das is presently working as an Associate Professor in the Department of Computer Science, Assam University, Silchar. He received the M.Sc. and Ph.D. degrees in computer science from Assam University, Silchar, India, in 2003 and 2011, respectively. He received his M.Tech. degree in Computer Science and Engineering from Mizoram University, Aizawl, India, in 2021. He received a Junior Research Fellowship at UGC, India, in 2006. His research interests include quantum cryptography, mobile computing network security, and machine learning. He is a member of the IEEE.

Sumit Biswas is a research scholar in the Department of Computer Science, Assam University, Silchar. He received his M.Tech. degree from NIT Meghalaya. His research interests include quantum cryptography, network security, and machine learning. He is a student member of the IEEE.

Sandip Kanoo is a postgraduate student in the Department of Computer Science, Assam University, Silchar. He received his B.Sc. degree in computer science from Karimganj College. His research interests include quantum cryptography.

Veeradittya Podder is a graduate student from the Department of Physics, St. Stephen's College, University of Delhi. He was awarded the Professor Nagpaul Fellowship for research by the Department of Mathematics at St. Stephen's College. His interests include machine learning, quantum mechanics, quantum computing, and its applications in cryptography and natural language processing.