



SHA-256 Algorithm

CRYPTOGRAPHY REPORT

DR. AHMED SALEM

Teams Member

- 1. Nada Khalid Mobarak**
- 2. Hazem Abdelsattar Mohammed Ali**
- 3. Mai Abdallah Abdulaziz Abdullah**
- 4. Hazem Mohamed Hassan AlRawi**
- 5. Mohamed Omran Yassin**

6th Semester in CS Department of FCI

Submitted to: Dr. Ahmed

Submission Date: 2023/5/18

Word Count: 7991

Table of Contents

Table of Contents.....	2
Table of Figures.....	4
1. Abstract.....	5
2. Introduction	6
2.1 Technical Terminologies	6
2.2 Aim and Objectives	7
3. Background Study	8
3.1 Bitcoin and Double Spending Problem	8
3.2 Blockchain Technology.....	8
3.3 Brief History.....	9
3.4 SHA-256 Previous Versions.....	9
3.5 Hash Function.....	9
3.6 Main Characteristic of SHA-256.....	11
3.7 How SHA 256 Algorithm Work.....	12
3.8 SHA-256 Pseudocode.....	13
3.9 Encrypt and Decrypt message example:.....	17
3.10 The Reason Why we can't Decrypt encrypted message using SHA-256.....	18
3.12 Flowchart for the Algorithm.....	19
4. Development.....	20
4.1 C Sharp Implementation.....	20
4.2 Discuss program	24
4.3 Steps of development	25
5. Testing	27
5.1 Perform 5 Example of Encrypting.....	27
5.2 Different test scenarios to validate algorithm.....	28
5.3 Issues Encountered while working on it.....	29
6. Evaluation	30
6.1 Level of Security:	30
6.2 Functionality:.....	30
6.3 Ease of Implementation:.....	30
6.4 Performance:	31
6.5 Pros and Cons	31
6.6 SHA-256 Limitation	32

6.7 Analyze security weaknesses:	32
6.7.1 Types of Attacks.....	32
6.7.2 Security Considerations:.....	33
6.8 Suggested Modification.....	34
6.9 Applications of SHA-256.....	35
7. Conclusion.....	37
8. References	38

Table of Figures

Figure 1 Verification is shared as a bundle with the user on the browser	10
Figure 2 recalculated the hash value shared with user to its true value again after reaching the user	11
Figure 3 Message Size and Digest Message Size of SHA Family	12
Figure 4 Array of Round Constants	14
Figure 5 How we preprocess the data by padding.....	14
Figure 6 Compressing Cycle	16
Figure 7 One Iteration in a SHA-2 Compression Function	17
Figure 8 Operations in SHA-256	17
Figure 9 Our Flowchart for SHA-256 Algorithm	19
Figure 10 Our website for encrypting SHA-256	26
Figure 11 Encryption Example 1	27
Figure 12 Encryption Example 2	27
Figure 13 Encryption Example 3	27
Figure 14 Encryption Example 4	28
Figure 15 Encryption Example 5	28
Figure 16 Comparison between SHA Family Algorithms in Security Strenght	32

1. Abstract

The SHA-256 algorithm is a widely used cryptographic hash function that generates a fixed-size output, or hash, from any input data. It is a member of the SHA-2 family of hash functions, which were developed by the National Security Agency (NSA) in the United States. SHA-256 is used in various applications, including digital signatures, password storage, and blockchain technology. It works by taking an input message and processing it through a series of mathematical operations to produce a unique 256-bit hash value. The security of SHA-256 is based on its resistance to collision attacks, which means that it is practically impossible to find two different messages that produce the same hash value. This report provides an in-depth analysis of the SHA-256 algorithm, including its history, design, implementation, and security properties. We also discuss some of the potential weaknesses and attacks that have been discovered in SHA-256, as well as its strengths and advantages over other hash functions. Overall, SHA-256 is a highly secure and reliable encryption algorithm that plays a crucial role in modern cryptography.

2. Introduction

Over the years, people were used to using computers for different purposes, especially in politics, business and so on. Due to the computing revolution and the large number of hacking techniques, experts decided that it is a must to have a way to protect data and prevent hackers from unwanted access and finally arrive at cryptography.

Cryptography is the process of hiding a specific information so that only the person who can access it is the writer of it and the person a message was intended for can read it.

There are Numerous algorithms of Cryptography like Multiplicative, Hill cipher, One-Time Pad cipher, SHA, etc.

Over the upcoming sections, we will acquire insight into SHA-256 and its descendants. Additionally, we will learn why SHA-256 is regarded as the most superior cryptographic algorithm as a result of its benefits. Furthermore, we will discuss how the algorithm operates, its schematic representation, and some illustrations to further elaborate

In the next few parts we will know about the history of SHA-256 and its family, why SHA-256 is considered as the best cryptography algorithm due to its advantages, how the algorithm works, the flowchart of the algorithm, evaluate it based on many different criteria, analyze its security weaknesses and cover its application areas.

2.1 Technical Terminologies

The major part that leads to comprehensive understanding that we need to know the main terminologies used in SHA-256 algorithm and hashing which include:

Message Digest: The output of a hash function, which represents the original data in a fixed-size format.

Collision: A situation where two different input messages produce the same hash value.

Hash Function: A mathematical algorithm that takes an input message and produces a fixed-size output hash value.

Block: The fixed-size input data that is processed by the hash function.

The proof of work: The process of transaction verification done in blockchain.

Nonce: In a “proof of work” consensus algorithm, it’s a random value used to vary the output of the hash value

Compression Function: The mathematical operation that takes a block of input data and produces a fixed-size output.

Padding: The process of adding extra bits to the input data to ensure that it is a multiple of the block size.

Chaining: The process of using the output of one compression function as the input to the next compression function.

Iteration: The number of times the compression function is applied to the input data in order to generate the final hash value.

Rounds: A round is a sequence of functions that are executed repeatedly to scramble the data beyond recognition. in SHA-256 algorithm we make 64 rounds.

Shift Amount: it's a fixed method that we use to shuffle the bits. In SHA-256, we divide blocks to eight segments of 32 bits. This shift randomizes the data

Additive Constants: The values added to the blocks. In SHA-256, we use 64 constants we add to the blocks. They are the cube roots of the first 64 prime numbers.

Salt: A random value added to the input message before hashing, which makes it more difficult to find matching hash values.

It is essential to understand these terms in order to get an understanding of how hashing algorithms and SHA algorithms work in general.

2.2 Aim and Objectives

In this report we aiming to

1. Explain the basics of SHA-256 algorithm then will dive in much more details
2. Mention the previous versions of SHA-256
3. Understanding how the SHA-algorithm works
4. Evaluate the strengths and weaknesses of SHA-256 and tell its application
5. Discuss its significance in modern cryptography and analyze the current and potential uses of SHA-256
6. And Finally show our implementation to this algorithm and discuss the program and its validation

3. Background Study

3.1 Bitcoin and Double Spending Problem

Bitcoin is a decentralized digital currency; it is a type of money that is completely virtual and is regarded as decentralized because it can be bought, sold, or exchanged without an intermediary such as a government or a banking institution. This is why it is said to operate under a peer-to-peer system where individuals can make transactions directly with each other. However, without the essential governing authority, how can these individuals prove their transactions? How can we be sure that they actually paid for something or, better yet, how can we even know that they had the money at all? This problem is called the **double spending problem**; it is the risk of digital currency being spent more than once. Being completely virtual, this form of money makes it possible for people, especially those with the knowledge and computing power, to simply produce copies of their digital money and spend it for multiple times. This is avoidable with traditional physical currencies because this type of money cannot easily be replicated. Furthermore, people have a more evident way of proving its authenticity and its past ownership. However, this is not the case with regards to virtual currencies. Fortunately, the creator of Bitcoin, under the name Satoshi Nakamoto, devised a solution in October 2008 called the **proof of work**. So, what is proof of work and how does it work? Proof of work is a consensus algorithm that is used by Bitcoin and other cryptocurrencies to verify transactions. In order to explain how this is applied, we need to introduce the concept of a blockchain for a better understanding of the blockchain technology.

3.2 Blockchain Technology

Blockchain technology is a public ledger that contains the history of every Bitcoin transaction that ever occurred. It is composed of a series of blocks, each containing the following data: about previous transactions, a cryptographic hash of the block that it extends, or the hash of the previous transaction, and the timestamp showing the time of the block's creation. So, how is the blockchain related to the proof of work? Well, before a block can be added to the blockchain, it must first have a valid proof of work, which can only be achieved by solving cryptographic problems. This activity is called **Bitcoin mining**, and the people who perform this are called **miners**. Any network participant can be a miner, and the miner who generates a valid proof of work first will be given a free Bitcoin as a reward and compensation for the computing power that he or she spent. However, it must be noted that only those with very strong computing power are likely to solve these highly complex cryptographic problems. Especially with the conception of large Bitcoin mining farms with expensive and fast computing resources, mining has become more and more challenging and costly. But what are these cryptographic problems? In order to find a valid block, a miner is tasked to generate a hash that satisfies the specific condition provided by the network protocol. He or she must hash three things: the hash of the previous block, the data of the set of transactions that will be added to the blockchain, and a **nonce**, which is short for number only, used once. This nonce is repeatedly incremented in the block until its value gives the block's hash the required number of zero bits that are specified by the network protocol, also known as the target. The particular hash function used by Bitcoin is known as SHA-256.

3.3 Brief History

SHA-256 is a part of the hash algorithms group known as SHA-2. The NSA created it in 2001 as a successor for SHA-1 which is not no longer used due to its vulnerability towards brute force attacks. This cryptographic hash function is patented and was invented by Glenn M Lilly as listed in the patent application filed by the NSA on March 5th, 2001. The patent was later granted and published on December 7th, 2004.

3.4 SHA-256 Previous Versions

SHA-256 is actually just one type under the SHA family of hash functions which was first introduced in 1993 by the national security agency starting with SHA-0 after two years the national institute of standards and technology developed the SHA-1 to fix the security issues found in its predecessor however weaknesses were still found in SHA-1 which led to the creation of the SHA-2 family in 2001. it is called a family as it has several variants, it consists of a total of six hash functions each with different digests or hash values or term sizes which are indicated in their numbers so SHA-224 is 224 bit long SHA-256 is 256 bit long and so on, SHA-512/224, SHA-512/256 are just truncated versions of the others

All of them are based on the same design principles but using different constants as SHA-1 but SHA-256 with significantly stronger security properties, SHA-256 in particular is widely used for secure data transmission and storage digital signatures and blockchain technology and it's the most widely used type in authentication protocols and employed by bitcoin so it's is considered to be the most secure hash function of them

so how is the SHA-256 more secure compared to its predecessors will discuss that in the few next sections

3.5 Hash Function

Besides symmetric and asymmetric cryptography, hash functions represent a third type of cryptography, which we can call keyless cryptography.

A hash function, also known as a message digest, does not use a key, but instead creates an essentially unique fixed-length hash value, which is called a hash, based on the original message, similar to a fingerprint. Any slight change in the message changes the whole output completely.

The meaning of 'hash' is to chop or scramble so we can say that hashing is the process of scrambling original data to the point that it can't be reproduced in its original form is called a hash digest. By taking a piece of data and passes it through a hash function and converting it to separate hash digest

The Main characteristic of hash function

- **One-way**

They are designed to be irreversible, which means it's insolvable to get the original plaintext back, so we can say that hashing aims to provide confidentiality in the first place, but not integrity

- **Collision-free**

there is no two strings map to the same hash output by using unique hash values, accordingly, there are no “collisions” between the output strings. Consequently there is no duplicate output string produced to prevent this kind of collisions programmers use advanced technologies like salting, key scratching, Merkle trees, Message authentication code etc.

- **Lightning-fast**

Taking less in computing time is a major characteristic for hash function as it's used in databases for this reason hash values are stored in so-called hash tables to ensure fast access

Hashing main Applications

They are used over numerous regions of computer science, such as:

- adding digital signatures to emails
- To encrypt communication between web servers and browsers, and generate session IDs for internet applications and data caching
- To protect sensitive data such as passwords and payment details
- Integrity Verification as hash is shared with the user as a bundle and then recalculated to its true value again after reaching the user



Figure 1 Verification is shared as a bundle with the user on the browser

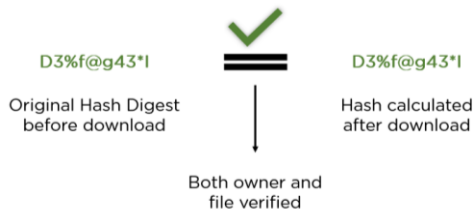


Figure 2 recalculated the hash value shared with user to its true value again after reaching the user

Now after understanding the working of hash functions, we can go to our main topic which is SHA-256 algorithm.

3.6 Main Characteristic of SHA-256

- **Deterministic:** The 256 featured in the name signifies the ultimate the output of hash digest is consistently 256 bits, so it will always produce the same hash or the same set of 64 alphanumeric characters for the same input no matter what is the length of the plaintext size and this is important because if the results differed every time then we will have no way of keeping track of the input data that we hashed
- **pre-image resistance:** means that it is infeasible to determine the original input data from the output hash, we used the word infeasible because determining the input is not entirely impossible since the hash generated for a particular input will always be the same so we it can be determine by brute force or trying every possible combination of the input data
- **Avalanche effect:** this means that a slight change even just changing a character from lowercase to uppercase and vice versa will completely change the resulting hash this property makes the hash pre-image resistant
- **Digest Length:** hash digest should be 256 bits in SHA-256 algorithm, 512 bits in SHA-512, and so on. Noting that larger digest leads to higher calculating cost
- **Message Length:** The length of the plaintext should be less than 264 bits.
- **Resistance to Collision:** collision happens when two unique inputs have the same hash value so the size limit of the input value for the SHA-256 function is 2^{64} which is infinitely large this means that almost anything can be hashed using SHA-256 and all of these things regardless of their size will all be converted to a 256-bit string

- **Irreversible:** As it's a form of hashing function, so we can't take back our input from the generated text from hash function since SHA algorithm is not a form of encryption it is designed to be more of an authentication method and not as an encryption that must be decoded that's why also is quick to compute

Algorithm	Message Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	160
SHA-224	$< 2^{64}$	224
SHA-256	$< 2^{64}$	256
SHA-384	$< 2^{128}$	384
SHA-512	$< 2^{128}$	512

Source: Nahari & Krutz, 2011

Figure 3 Message Size and Digest Message Size of SHA Family

3.7 How SHA 256 Algorithm Work

We can describe SHA algorithms by covering the three essential stages:

- **Preprocessing** - This is where messages are padded, broken down into smaller chunks, and initialized values are set.
- **Buffer initialization** - Before making any computations, we initialize some buffer values. They are fixed constants which represent hash values.
- **Hash Computation** - This process involves a sequence of operations that produce a set of hash values. The 256-bit hash digest we get is generated by computing these different hash values.

SHA 256 follows the steps given below:

1. First, data is converted into binary.
2. We divided binary data to blocks of 512 bits. In case the size of the block is under 512, it will be subjected to padding. In case of its size being larger, it will be divided into chunks consisting of 512 bits.
3. The message is divided into smaller blocks each one of it is 32 bits.
4. 64 rounds of compression functions are performed, wherein the hash values generated are rotated and additional data is added.
5. From the output of previous operations, new hash values are created
6. In the last round, one final 256-bit hash value is produced this hash digest is the end product of SHA 256.

3.8 SHA-256 Pseudocode

If you want to see all the detailed steps we just mentioned above in pseudocode form, then here it is, straight from Wikipedia:

Note 1: All variables are 32 bit unsigned integers and addition is calculated modulo 2³²

Note 2: For each round, there is one round constant $k[i]$ and one entry in the message schedule array $w[i]$, $0 \leq i \leq 63$

Note 3: The compression function uses 8 working variables, a through h

Note 4: Big-endian convention is used when expressing the constants in this pseudocode,

and when parsing message block data from bytes to words, for example,

the first word of the input message "abc" after padding is 0x61626380

Initialize hash values:

(first 32 bits of the fractional parts of the square roots of the first 8 primes 2 to 19):

$h_0 := 0x6a09e667$

$h_1 := 0xbb67ae85$

$h_2 := 0x3c6ef372$

$h_3 := 0xa54ff53a$

$h_4 := 0x510e527f$

$h_5 := 0x9b05688c$

$h_6 := 0x1f83d9ab$

$h_7 := 0x5be0cd19$

Initialize array of round constants:

(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2 to 311):

```

k[0..63] =
0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b
0x59f111f1 0x923f82a4 0xab1c5ed5 0xd807aa98 0x12835b01
0x243185be 0x550c7dc3 0x72be5d74 0x80deb1fe 0x9bdc06a7
0xc19bf174 0xe49b69c1 0xefbe4786 0x0fc19dc6 0x240ca1cc
0x2de92c6f 0x4a7484aa 0x5cb0a9dc 0x76f988da 0x983e5152
0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147
0x06ca6351 0x14292967 0x27b70a85 0x2e1b2138 0x4d2c6dfc
0x53380d13 0x650a7354 0x766a0abb 0x81c2c92e 0x92722c85
0xa2bfe8a1 0xa81a664b 0xc24b8b70 0xc76c51a3 0xd192e819
0xd6990624 0xf40e3585 0x106aa070 0x19a4c116 0x1e376c08
0x2748774c 0x34b0bcb5 0x391c0cb3 0x4ed8aa4a 0x5b9cca4f
0x682e6ff3 0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208
0x90beffffa 0xa4506ceb 0xbef9a3f7 0xc67178f2

```

Figure 4 Array of Round Constants

Pre-processing (Padding):

begin with the original message of length L bits

append a single '1' bit

append K '0' bits, where K is the minimum number ≥ 0 such that $L + 1 + K + 64$ is a multiple of 512

append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits

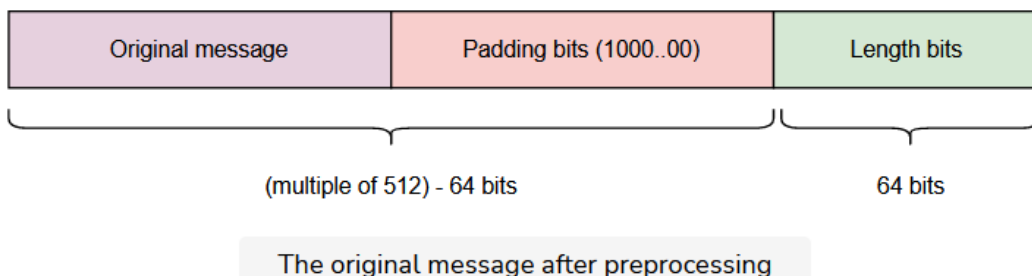


Figure 5 How we preprocess the data by padding

Process the message in successive 512-bit chunks:

break message into 512-bit chunks

for each chunk

create a 64-entry message schedule array $w[0..63]$ of 32-bit words

(The initial values in $w[0..63]$ don't matter, so many implementations zero them here)

copy chunk into first 16 words $w[0..15]$ of the message schedule array

Extend the first 16 words into the remaining 48 words $w[16..63]$ of the message schedule array:

for i from 16 to 63

$s0 := (w[i-15] \text{ rightrotate } 7) \text{ xor } (w[i-15] \text{ rightrotate } 18) \text{ xor } (w[i-15] \text{ rightshift } 3)$

$s1 := (w[i-2] \text{ rightrotate } 17) \text{ xor } (w[i-2] \text{ rightrotate } 19) \text{ xor } (w[i-2] \text{ rightshift } 10)$

$w[i] := w[i-16] + s0 + w[i-7] + s1$

Initialize working variables to current hash value:

$a := h0$

$b := h1$

$c := h2$

$d := h3$

$e := h4$

$f := h5$

$g := h6$

$h := h7$

Compression function main loop:

for i from 0 to 63

$S1 := (e \text{ rightrotate } 6) \text{ xor } (e \text{ rightrotate } 11) \text{ xor } (e \text{ rightrotate } 25)$

$ch := (e \text{ and } f) \text{ xor } ((\text{not } e) \text{ and } g)$

$\text{temp1} := h + S1 + ch + k[i] + w[i]$

$S0 := (a \text{ rightrotate } 2) \text{ xor } (a \text{ rightrotate } 13) \text{ xor } (a \text{ rightrotate } 22)$

$\text{maj} := (a \text{ and } b) \text{ xor } (a \text{ and } c) \text{ xor } (b \text{ and } c)$

$\text{temp2} := S0 + \text{maj}$

$h := g$

$g := f$

$f := e$

$e := d + \text{temp1}$


```

d := c
c := b
b := a
a := temp1 + temp2

```

Add the compressed chunk to the current hash value:

```

h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h

```

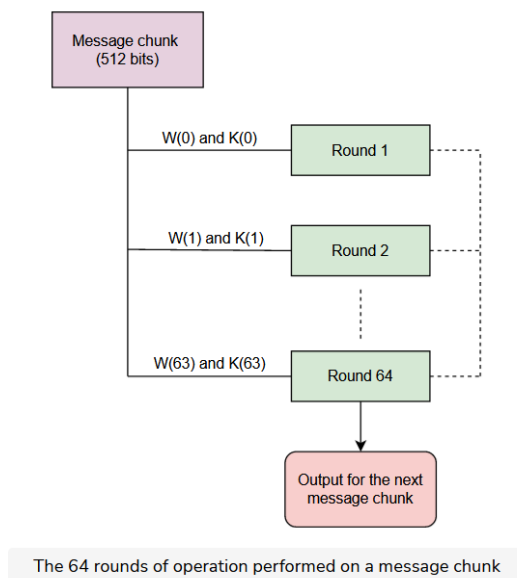


Figure 6 Compressing Cycle

The image illustrates the complete cycle of compression:

Produce the final hash value (big-endian):

```

digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7

```

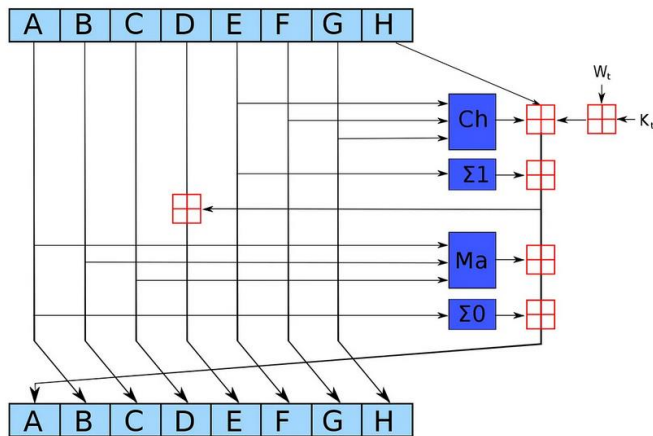


Figure 7 One Iteration in a SHA-2 Compression Function

Where the blue components perform the following operations

$$\begin{aligned}
 \text{Ch}(E, F, G) &= (E \text{ AND } F) \text{ XOR } ((\text{NOT } E) \text{ AND } G) \\
 \text{Ma}(A, B, C) &= (A \text{ AND } B) \text{ XOR } (A \text{ AND } C) \text{ XOR } (B \text{ AND } C) \\
 \Sigma(A) &= (A \ggg 2) \text{ XOR } (A \ggg 13) \text{ XOR } (A \ggg 22) \\
 \Sigma(E) &= (E \ggg 6) \text{ XOR } (E \ggg 11) \text{ XOR } (E \ggg 25) \\
 + &= \text{addition modulo } 2^{32}
 \end{aligned}$$

Figure 8 Operations in SHA-256

The red component is addition modulo 2^{32} for SHA-256

3.9 Encrypt and Decrypt message example:

Encrypt message example: -

- Plain text = Hello world
- Encrypted message =
fe1618bfdd6e70a2969f51867d9c2abbb27dfb67a9a04f70801c9a6bd4bf4ab0

Decrypt message example: -

- Encrypted message =
fe1618bfdd6e70a2969f51867d9c2abbb27dfb67a9a04f70801c9a6bd4bf4ab0
- Encrypted message = Can't Be Decoded!

3.10 The Reason Why we can't Decrypt encrypted message using SHA-256

It is mathematically impossible to reconstruct the original message by 'decrypting' a hash value.

since SHA is not a form of encryption it is designed to be more of an authentication method and not as an encryption that must be decoded

As we discussed and explained earlier, hashing is irreversible process as it scrambles data to the point that we can't be reproduced it to its original form is called a hash digest converting it to separate hash digest, consequently original form can't be reconstructed again and we mentioned that, hashing aims to provide confidentiality in the first place, but not integrity

3.12 Flowchart for the Algorithm

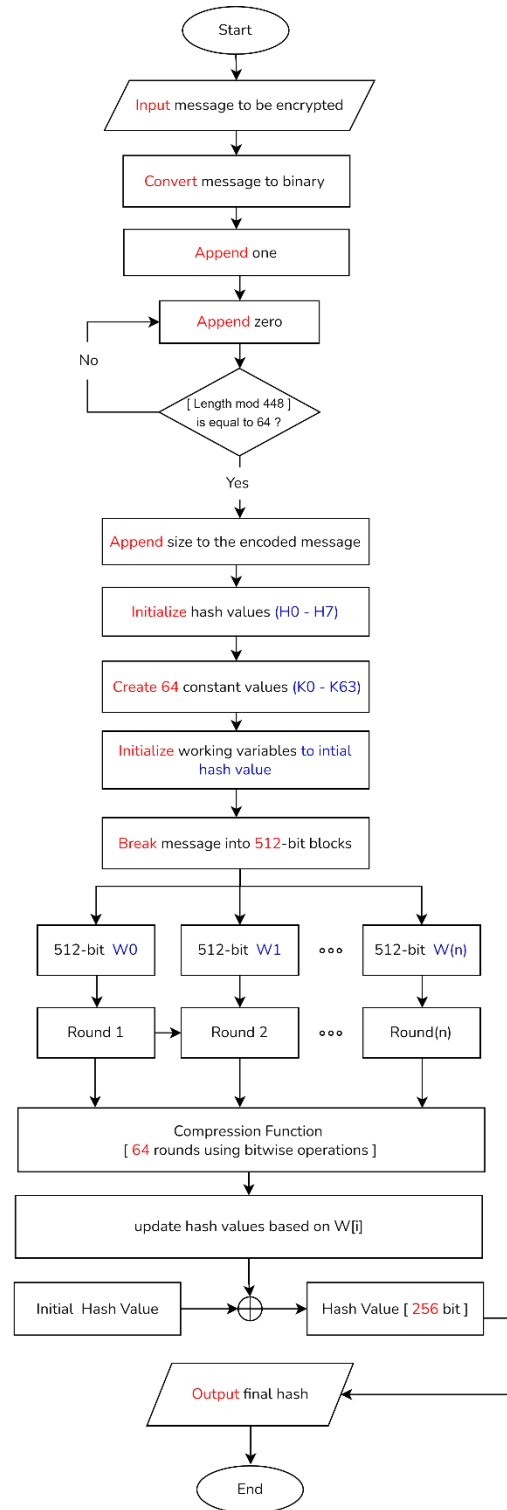


Figure 9 Our Flowchart for SHA-256 Algorithm

4. Development

4.1 C Sharp Implementation

Here is our implementation to SHA-256 from scratch in C# language

```
using Microsoft.VisualBasic;
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Http.Headers;
using System.Numerics;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;

namespace SHA256Alog
{
    class Message
    {
        public String OriginalMessage;
        public String EncryptedMessage;

        Int64 [] BlocksTemp = { 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f,
                                0x9b05688c, 0x1f83d9ab, 0x5be0cd19 };

        Int64[] WordTemp =
        {
            0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
            0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
            0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
            0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
            0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
            0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
            0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
            0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90bffffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
        };

        List<List<int>> Word = new List<List<int>>();
        List<List<int>> Blocks = new List<List<int>>();
        List<List<int>> HashValues = new List<List<int>>();
        public Message(String OriginalMessage)
        {
            this.OriginalMessage = OriginalMessage;
            Encrypt();
        }
    }
}
```

```

    }
    public void PrintEncryptedMessage()
    {
        Console.WriteLine(EncryptedMessage);
    }
    private void UpdateHashValues()
    {
        for(int i=0;i<8;i++)
        {
            HashValues[i] = Add(new List<int>(HashValues[i]), new
List<int>(Blocks[i]));
        }
        for(int i=0;i<8;i++)
            for(int j=0;j<32;j+=4)
            {
                int shift = 3;
                int num = 0;
                for(int k=j;k<j+4;k++)
                {
                    if (HashValues[i][k] == 1)
                        num += (1 << shift);
                    shift--;
                }
                char temp;
                if(num<10)
                {
                    num += 48;
                    temp =(char)num;
                }else
                {
                    num -= 10;
                    temp= (char)(num + 'A');
                }
                EncryptedMessage += temp;

            }

    }

    private List<int> Add(List<int>Operand1, List<int>Operand2)
    {
        List<int> Value=new List<int>();
        int cur = 0;
        for(int i=31;i>=0;i--)
        {
            cur += Operand1[i] + Operand2[i];
            Value.Add(cur % 2);
            cur /= 2;
        }
        Value.Reverse();
        return Value;
    }
    private List<int> RotateRight(List<int>Value, int cnt)
    {
        List<int> Result = new List<int>();
        for (int i = 32-cnt, j = 0; j < 32; j++, i = (i + 1) % 32)
            Result.Add(Value[i]);
        return Result;
    }
    private List<int> Invert(List<int> Value)

```

```

    {
        for (int i = 0; i < 32; i++)
            Value[i] = 1 - Value[i];
        return Value;
    }
    private List<int> XOR(List<int> Operand1, List<int> Operand2)
    {
        List<int> Result = new List<int>();
        for (int i = 0; i < 32; i++)
            Result.Add((Operand1[i] ^ Operand2[i]));
        return Result;
    }
    private List<int> AND(List<int> Operand1, List<int> Operand2)
    {
        List<int> Result = new List<int>();
        for (int i = 0; i < 32; i++)
            Result.Add((Operand1[i] & Operand2[i]));
        return Result;
    }

    private List<int> RightShift(List<int> Operand, int cnt)
    {
        List<int> Result = new List<int>();
        for(int i=0;i<cnt;i++)
            Result.Add(0);

        for (int i=0;i<32&&Result.Count<32;i++)
        {
            Result.Add(Operand[i]);
        }
        return Result;
    }
    private List<int> Sigma0Σ(List<int> Value)
    {
        return XOR(XOR(RotateRight(Value, 2) , RotateRight(Value, 13)),
        RotateRight(Value, 22));
    }
    private List<int> Sigma1Σ(List<int> Value)
    {
        return XOR(XOR(RotateRight(Value, 6) , RotateRight(Value, 11)),
        RotateRight(Value, 25));
    }
    private List<int> Sigma0σ(List<int> Value)
    {
        return XOR(XOR(RotateRight(Value, 7),RotateRight(Value,
        18)),RightShift(Value,3));
    }
    private List<int> Sigma1σ(List<int> Value)
    {
        return XOR(XOR(RotateRight(Value, 17), RotateRight(Value, 19)),
        RightShift(Value, 10));
    }
    private List<List<int>> InitalizeW(List<List<int>>> W)
    {
        for (int i = 16; i < 64; i++)
            W[i] = Add(Add(W[i - 16], Sigma0σ(W[i - 15])), Add(W[i - 7], Sigma1σ(W[i
        - 2])));
        return W;
    }
    private void UpdateBlocks(List<List<int>>> W)
    {

```

```

        for (int i = 0; i < 64; i++)
        {
            List<int> Choice = XOR(AND(new List<int>(Blocks[4]), new
List<int>(Blocks[5])), AND(Invert(new List<int>(Blocks[4])), new List<int>(Blocks[6])));
            List<int> Majority = XOR(XOR(AND(new List<int>(Blocks[0]), new
List<int>(Blocks[1])), AND(new List<int>(Blocks[0]), new List<int>(Blocks[2]))), AND(new
List<int>(Blocks[1]), new List<int>(Blocks[2])));
            List<int> temp1 = Add(Add(Add(new List<int>(Blocks[7]), Sigma1Σ(new
List<int>(Blocks[4])), Add(new List<int>(Word[i]), new List<int>(W[i]))), new
List<int>(Choice)));
            List<int> temp2 = Add(Sigma0Σ(new List<int>(Blocks[0])), new
List<int>(Majority));
            Blocks[7] = new List<int>(new List<int>(Blocks[6]));
            Blocks[6] = new List<int>(new List<int>(Blocks[5]));
            Blocks[5] = new List<int>(new List<int>(Blocks[4]));
            Blocks[4] = new List<int>(Add(new List<int>(Blocks[3]) , new
List<int>(temp1)));
            Blocks[3] = new List<int>(new List<int>(Blocks[2]));
            Blocks[2] = new List<int>(new List<int>(Blocks[1]));
            Blocks[1] = new List<int>(new List<int>(Blocks[0]));
            Blocks[0] = new List<int>(Add(new List<int>(temp1) , new
List<int>(temp2)));
        }
    }
    private void initializeWandBlocks()
    {
        for(int i=0;i<64;i++)
        {
            List<int> Value = DecimalToBinary(WordTemp[i],32);
            Word.Add(Value);
        }
        for(int i=0;i<8;i++)
        {
            List<int> Value = DecimalToBinary(BlocksTemp[i], 32);
            Blocks.Add(Value);
            HashValues.Add(Value);
        }
    }
    private List<int> DecimalToBinary(Int64 DecimalValue,int size)
    {
        List<int> BinaryRepresentation = new List<int>();
        while (DecimalValue > 0)
        {
            if (DecimalValue % 2 == 1)
                BinaryRepresentation.Add(1);
            else BinaryRepresentation.Add(0);
            DecimalValue /= 2;
        }
        while (BinaryRepresentation.Count < size)
        {
            BinaryRepresentation.Add(0);
        }
        BinaryRepresentation.Reverse();
        return BinaryRepresentation;
    }
    private void Encrypt()
    {
        initializeWandBlocks();
        List<int> Numbers=new List<int> ();

```



```

for(int i=0;i<OriginalMessage.Length;i++)
{
    int DecimalValue =(int)OriginalMessage[i];
    List<int> BinaryRepresentation = DecimalToBinary(DecimalValue,8);
    for (int j = 0; j < 8; j++)
        Numbers.Add(BinaryRepresentation[j]);
}
Numbers.Add(1);
while(Numbers.Count%512!=448)
{
    Numbers.Add(0);
}
int size = 8 * OriginalMessage.Length;
List<int> BinaryRepresentationOfSize= DecimalToBinary(size,64);
for (int i = 0; i < 64; i++)
    Numbers.Add(BinaryRepresentationOfSize[i]);
for (int i = 0; i < Numbers.Count; i += 512)
{
    List<List<int>> W = new List<List<int>>();
    for (int j = i; j < i + 512; j += 32)
    {
        List<int> N = new List<int>();
        for (int k = j; k < j + 32; k++)
        {
            N.Add(Numbers[k]);
        }

        W.Add(N);
    }
    List<int> Num = new List<int>();
    for (int k = 0; k < 32; k++)
        Num.Add(0);
    while (W.Count < 64)
        W.Add(Num);
    W = InitializeW(W);
    UpdateBlocks(W);
}
UpdateHashValues();
}
}
}

```

4.2 Discuss program

- The program consists of number of functions and by describing the [Encrypt function](#) we can describe the whole process of SHA-256 algorithms
we did go to deeper details in the pseudocode section 3.10

- Encrypt Function Contains 7 Mandatory function each one of them does only one step of SHA256 Encrypting algorithm:
 1. `inititalizeWordsandHashValues()`
 2. `Padding()`
 3. `GetChunk()`
 4. `ChunkInitalizer()`
 5. `UpdateWorkingVariables()`
 6. `UpdateHashValues()`
 7. `ConvertHashValuesToHexadicimal()`
- `inititalizeWordsandHashValues()`:
initiate words and hash values which are predetermined values.
Hash Values equal to first 32 bits of the fractional parts of the square roots of the first 8 primes (2..19) and words values first 32 bits of the fractional parts of the cube roots of the first 64 primes (2..311)
- `Padding()`:
convert the original message to binary string and pad the message to make it size is a multiple of 512.
- `GetChunk()`:
we gave this function the index of the chunk that have to update the Hash Values.
- `ChunkInitalizer()`:
Do Calculations to transform the shape of the chunk.
- `UpdateWorkingVariables()`:
is concerned in updating working variables based on the current chunk
- `UpdateHashValues()`:
is concerned in updating hash values to encrypt the original message.
- `ConvertHashValuesToHexadicimal()`:
is concerned in converting Hash Values to Hexadecimal Values to produce the final shape of the encrypted message.

4.3 Steps of development

We first implement the algorithm from scratch in C Sharp Console application as we demonstrate in the 5 examples in the next section

Then after making sure that the code is logically right and working properly we created MVC Project with front view so it's easy to deal with

And finally we deployed our small website and published it online so that anyone can use it

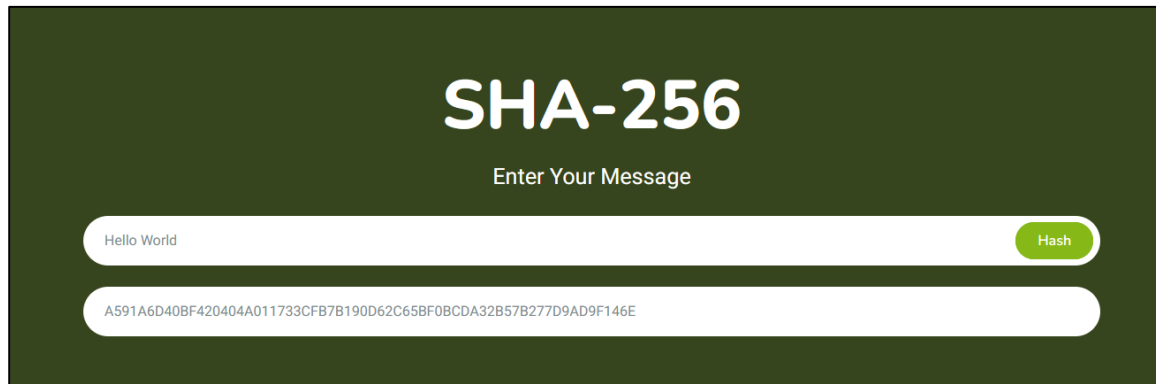


Figure 10 Our website for encrypting SHA-256

Here is the Link of our small website :

<http://sha-256.somee.com/>

5. Testing

5.1 Perform 5 Example of Encrypting

Example. 1


```

1  using Microsoft.VisualBasic;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7  using SHA256 = SHA256Algorithm;
8
9  namespace SHA256Namespace
10 {
11     class Program
12     {
13         static void Main(string[] args)
14         {
15             Console.WriteLine("SHA256 is Generated Successfully!");
16         }
17     }
18 }

```

Figure 11 Encryption Example 1

Example. 2



The screenshot shows the Microsoft Visual Studio Debug Console. The title bar reads "Microsoft Visual Studio Debug Console". The console output shows a test execution for SHA256. The test name is "test". The test result is "Passed". The test output is "SHA256 is Generated Successfully!". The console also shows the command prompt "C:\Users\hazem\Desktop\SHA256\SHA256\bin\Debug\net6.0\SHA256Namespace.exe (process 24684) exited with code 0". The console text is as follows:

```
test
SHA256 is Generated Successfully!

C:\Users\hazem\Desktop\SHA256\SHA256\bin\Debug\net6.0\SHA256Namespace.exe (process 24684) exited with code 0
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
close when debugging stops.
Press any key to close this window . . .
```

Figure 12 Encryption Example 2

Example. 3

[illegible]

Figure 13 Encryption Example 3

Example. 4

```

View? Microsoft Visual Studio Debug Console
56Na
1 the transaction credit card number is 1234
2 539F69F649D7692828D213D196302291A606C465CF97CAE404CE2DA2C048C9DE
3 SHA256 is Generated Successfully!
4
5 C:\Users\hazem\Desktop\SHA256\SHA256\bin\Debug\net6.0\SHA256Namespace.exe (process 16028) exited with code 0.
6 To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close
7 when debugging stops.
8 Press any key to close this window . . .
9
10
11
12

```

Figure 14 Encryption Example 4

Example. 5

```

using System.Collections.Specialized;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using System.Security.Cryptography;
using SHA256Alog;
namespace SHA256Namespace
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("we will conquer mars planet after 4 am");
            string input = "we will conquer mars planet after 4 am";
            string sha256 = SHA256Alog.SHA256(input);
            Console.WriteLine(sha256);
            Console.WriteLine("SHA256 is Generated Successfully!");
        }
    }
}

```

Figure 15 Encryption Example 5

5.2 Different test scenarios to validate algorithm

- In order to validate the output, we set a function that compares the output of our implementation with the output of other built-in function SHA256 in C Sharp class [System.Security.Cryptography.HashAlgorithm](#)

If the two strings are the same then the output is valid and we output that the “[SHA is generated successfully](#)” as it’s shown in the figures we include for the 5 examples above

- And we also double checked the validation by comparing the obtained output with the expected one from other known online implementations for a particular input message. If they match, then the implementation is valid.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Security.Cryptography;
using SHA256Alog;
namespace SHA256Namespace
{

```

```

class SHA256EncryptionAlgorithm
{
    public static void Main()
    {
        String MessageYouWantToEncrypt= Console.ReadLine();
        Message Key = new Message(MessageYouWantToEncrypt);

        string hash = String.Empty;
        // Initialize a SHA256 hash object
        using (SHA256 sha256 = SHA256.Create())
        {
            // Compute the hash of the given string
            byte[] hashValue =
sha256.ComputeHash(Encoding.UTF8.GetBytes(MessageYouWantToEncrypt));

            // Convert the byte array to string format
            foreach (byte b in hashValue)
            {
                hash += $"{b:X2}";
            }
        }

        if (hash == Key.EncryptedMessage)
        {
            Console.WriteLine(Key.EncryptedMessage);
            Console.WriteLine("SHA256 is Generated Successfully!");
        }
        else Console.WriteLine("Not Generated :(");
    }
}

```

5.3 Issues Encountered while working on it

- 32-bit size operands:**
 Implementing SHA265 algorithm require operating on binary representation of the message so doing operations on the 31th bit is a little bit hard because of it is sign bit so we had to build from scratch the operations which can be done in the algorithm to avoid ambiguous results.
- Tracing operations:**
 Operations applied on SHA-256 algorithm are complicated and with having big stream of 0's and 1's makes debugging the code challenging to trace and it was difficult make sure each operations works properly.

6. Evaluation

Here is an evaluation of SHA-256 based on different criteria:

6.1 Level of Security:

Defining the security of SHA-256 are three essential properties:

- With 2^{256} possible outputs it's almost impossible to reform the original data from the hash value as it leads the brute-force attacks to make 2^{256} attempts just to reach the original message
- It is computationally infeasible for two different inputs to result in the same hash output.
- A subtle change to the input should result in a significant change to the hash output which makes it complicated for the attacker due to the avalanche effect.
- High collision resistance makes it exceedingly rare to have two messages with identical hash values.

Although there have been some attacks that have been successful in breaking SHA-256, they are still very difficult to execute in practice. But still SHA-256 is one of the best algorithms that provides a high level of security for message integrity verification and digital signatures.

6.2 Functionality:

The main functionality of SHA-256 includes:

1. **message integrity verification**: SHA-256 ensures that the data has not been tampered with by creating a unique digital signature for each block of data.
2. **Message authentication**: SHA-256 provides an assurance that the message was sent by the claimed sender and has not been modified in transit.
3. **Password hashing**: SHA-256 is often used to securely store passwords by generating a one-way hash. This makes it difficult for attackers to reverse engineer the original password from the stored hash.
4. **Digital signatures**: SHA-256 is a widely used algorithm for digital signatures, which are used to verify the authenticity of documents and transactions in a secure manner.
5. **Enabling Proof of Work Mining**: SHA-256 hashing algorithm was initially used to validate transactions on a blockchain network by a Proof of Work agreement mechanism.

6.3 Ease of Implementation:

SHA-256 is a widely used and well-documented hash function, which makes it relatively easy to implement in software and hardware. Although it's a little complicated if you are making it from scratch but there are also many libraries and tools available that provide implementations of SHA-256, which can make it easier to integrate into applications.

6.4 Performance:

- In the terms of performance SHA-256 is a relatively fast hash function, with typical speeds of several hundred megabytes per second on modern processors.
- hash function takes only a few seconds to process a million characters. However, larger data sets or systems with lower computational power might take longer to compute the hash
- When it comes to the time it takes to compute the SHA-256 hash value, it depends on a few factors such as the size of the input data, the computing power of the system, and the implementation of the hash algorithm.

Generally, SHA-256 is a secure, functional, and widely used cryptographic hash function. Its methods of operation are well-defined and its performance is relatively fast, making it a good choice for many applications. Its ease of implementation also makes it accessible to developers who want to integrate it into their applications.

6.5 Pros and Cons

Pros	cons
widely supported for cryptographic purposes	SHA-256 is slower than its predecessors
used by large organization such as NIST, many governments such as the USA and Australia	to support SHA-2 encryption some software may need to update
resistant to collision, pre-image and also second-preimage attacks	
SHA-256 is supported by the latest browsers, OS platforms and mobile devices	
It addresses SHA-1's weaknesses	
it's much more secure than MD5 and SHA-1 since it has a larger hash size	

The figure below demonstrates the difference between SHA family algorithm and their strength security

Function	Output Size	Security Strengths in Bits		
		Collision	Preimage	2nd Preimage
SHA-1	160	< 80	160	$160 - L(M)$
SHA-224	224	112	224	$\min(224, 256 - L(M))$
SHA-512/224	224	112	224	224
SHA-256	256	128	256	$256 - L(M)$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	$512 - L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	$\min(d/2, 128)$	$\geq \min(d, 128)$	$\min(d, 128)$
SHAKE256	d	$\min(d/2, 256)$	$\geq \min(d, 256)$	$\min(d, 256)$

Table 4: Security strengths of the SHA-1, SHA-2, and SHA-3 functions

Figure 16 Comparison between SHA Family Algorithms in Security Strenght

6.6 SHA-256 Limitation

- **ASIC Dominance:**

After Bitcoin's introduction, the debate over whether blockchain networks should be resistant to ASIC arose. However, the SHA-256 hashing algorithm wasn't meant to withstand the strength of powerful machines.

- **Current Mining Profitability:**

To mine profitable SHA-256 cryptocurrencies, a powerful ASIC mining rig is necessary. However, not all ASICs produce the same results. Despite the potential for higher profits, the S19 Pro is much more expensive to purchase. As new and even more powerful machines are released, ASIC mining is becoming increasingly competitive and requires significant financial investment.

- **Emergence of SHA-3 And Other Hashing Algorithms:**

As we mentioned earlier, SHA2 family is not the latest version of secure hash algorithms. SHA-3, is known as Keccak, SHA-3 is ASIC friendly same as sha-256, however SHA-3 is much faster and secure than SHA-256. SHA-3 is used by Nexus, Smart Cash, and a few other blockchains.

6.7 Analyze security weaknesses:

6.7.1 Types of Attacks

1. **brute force and dictionary attacks to attack passwords:** it is known to be an exhaustive check as it operates every possible combination of characters. In the dictionary attack, the attacker uses a file with all words, phrases, and common passwords used. Each word in the file is hashed and is compared to the password hash in the database. If they matched, then word is the password. So, by either using brute-force or comparing hashes to known password, plaintext will be recovered
2. **Collision attacks:** A collision attack is an attack in which an attacker tries to find two different messages that produce the same hash value. While SHA-256 is considered to be

resistant to collision attacks, quantum collision attacks on reduced SHA-256 have been demonstrated to significantly improve the classical attacks

3. **Known plaintext attacks:** A known plaintext attack is an attack in which an attacker has access to both the plaintext and the corresponding hash value. While SHA-256 is not vulnerable to known plaintext attacks in general, there may be specific instances where it can be vulnerable
4. **Bit flip attacks:** A bit flip attack is an attack in which an attacker tries to modify the input message in such a way that the resulting hash value is different from the original hash value. While SHA-256 is not vulnerable to bit flip attacks in general, there may be specific instances where it can be vulnerable.
5. **Preimage attacks:** A preimage attack is an attack in which an attacker tries to find a message that produces a given hash value. While SHA-256 is considered to be resistant to preimage attacks, advances in computing have reduced the cost of information processing and data storage to retain effective security.
6. **Length extension attack:** SHA-256 is vulnerable to a length extension attack, which can be used to generate a new hash that appears to be related to an existing hash. This attack is possible because SHA-256 uses a Merkle-Damgard construction, which makes it vulnerable to this type of attack.
7. **Attacks using quantum computers:** SHA-256 is vulnerable to attacks using quantum computers. While this is still a theoretical threat, it is important to consider as quantum computing technology continues to advance
8. **birthday attacks:** A birthday attack is a type of attack where an attacker tries to find two different inputs that produce the same hash output with a high probability. This type of attack is based on the birthday paradox, which states that in a group of 23 people, there is a 50% chance that two people have the same birthday. It's based on the fact that the number of possible hash outputs is 2^{256} , which is a very large number. However, if an attacker hashes a large number of random inputs (around 2^{128}), there is a high probability that two inputs will result in the same hash output. This attack can be resolved by using a technique called salting which involves adding a random value to the input message before hashing, which makes it harder for an attacker to find two inputs that produce the same hash output.
9. **Coordinated attacks:** If multiple parties use the same secret key to hash the same input data, it's possible for an attacker to collude with one of these parties to obtain the original input data and calculate the secret key.

6.7.2 Security Considerations:

security implications which includes the importance of key management, secure implementation practices, and the use of other security measures that we need to know about SHA-256:

1. **Key management:** The security of any cryptographic algorithm depends heavily on key management. In the context of SHA-256, key management refers to the proper generation,

distribution, and storage of keys used in the algorithm. If a key is compromised, an attacker can easily break the encryption or tamper with the integrity of the data being protected.

2. **Secure implementation practices:** Proper implementation of SHA-256 is critical to maintaining its security. Developers must follow best practices for secure coding and avoid common implementation pitfalls that can weaken the algorithm's security. For example, developers should use a different initialization vector (IV) for each message to prevent attacks such as birthday attacks.
3. **Use of other security measures:** SHA-256 is just one component of a comprehensive security architecture. Other security measures, such as access control, secure communication protocols, and intrusion detection systems, also play a critical role in ensuring the security of data being hashed.

By prioritizing these considerations, organizations can minimize the risk of data breaches and unauthorized access to sensitive information.

6.8 Suggested Modification

Here are modifications that can be applied to it to improve its security:

1. Changing the **number of rounds** would impact the large amount of performance analysis consequently, all performance data would need to be either estimated or performed again especially in hardware and in memory-restricted environments
2. **Savings optimizing mode** in SHA-256 is a technique used to optimize the computation and memory requirements of the algorithm while still maintaining its security properties. The savings optimizing mode involves reducing the number of intermediate hash values that need to be stored in memory during the hashing process.

The savings optimizing mode achieves this reduction in memory requirements by changing the way that the compression function is computed. In standard SHA-256, the compression function takes as input the current intermediate hash values and the next block of input data, and produces a new set of intermediate hash values. In the savings optimizing mode, the compression function takes as input the current intermediate hash value and the next block of input data, and produces a new intermediate hash value directly.

The savings optimizing mode still maintains the security properties of SHA-256, including its resistance to collision attacks and pre-image attacks. However, it does result in a slightly slower hash function due to the additional computations required to produce the final hash value.

3. **HMAC-SHA256:** it's is a specific construction that uses SHA-256 in combination with a secret key to provide message authentication. This modification can help to prevent length extension attacks.
4. **SHA-3:** there have been some proposals for modifications to SHA-256, such as SHA-3, which is a new hash function that was designed to be more resistant to attacks than SHA-256. SHA-3 uses a different construction, called the sponge construction, which is based on a

permutation rather than a compression function. This makes it more resistant to certain types of attacks, such as length extension attacks.

5. variant of SHA-256 that is based [on elliptic curves](#), which are mathematical structures that can be used to perform cryptographic operations which provides stronger security guarantees than traditional SHA-256.
6. [Salted SHA-256](#): involves adding a random value (known as a salt) to the input before hashing it with SHA-256. This modification can help to prevent precomputed attacks.
7. [Iterated SHA-256](#): involves applying SHA-256 multiple times to the input. This modification can increase the computational complexity of finding collisions or preimages.

Although there are some suggested modifications but still there are not any logical or mathematical modifications to SHA-256 that are globally accepted and implemented. However, it is worth to say that any modifications to SHA-256 would need to be thoroughly tested and analyzed to ensure that they do not introduce new vulnerabilities or weaken the security of the algorithm.

6.9 Applications of SHA-256

- [Data integrity checks](#)

SHA-256 guarantees data integrity and the authenticity of information, providing both parties with confidence that the transmitted information is indeed coming from the expected individual.

The recipient device creates a hash of the original message then compares it to with one sent by the sender. In case both hash values are equal, then we're sure that message hasn't been altered with during transit.

- [Verifying Digital Signatures](#)

A digital signature is a way of signing digital documents or software which is verifiable by the other recipient, the person who receives or uses the files can check that the signature is real. This helps them figure out if you made or approved the document or file, or if someone else changed it.

A digital signature is created by applying a hash to the file and then encrypt it via private and public keys. The private key is used when the signature owner signs the document while the public key is used by the recipient when he decrypting the message

- [Verifying Blockchain Transactions](#)

Some of most popular blockchain applications, like Bitcoin, use SHA-256.

One of the most critical parts of blockchains is Block headers since they link one block of transactions to the next in a certain arrange, SHA-256 hash helps guarantee that there's no previous blocks has changed without altering the new block's header.

- **Digital Signature Verification**

Hash algorithms such as SHA-256 help guarantee that the signature is verified, it follows asymmetric encryption methodology to verify the authenticity of a file.

- **Password Hashing**

Websites store the user passwords in hashed format for two benefits. It helps foster a sense of confidentiality and privacy, and it reduces the load on the central database by maintaining all the digests are of consistent size.

- **SSL Handshake**

It's the method of establishing a secure and secure encrypted communication channel between the client (user's browser) and the server (web server) so that they can agree on encryption keys and hashing authentication

- **Password securing using hash**

also applied by companies when storing their user's passwords when a user enters a password the password usually runs through a hash algorithm and the output hash value is compared to the company's user database also used in Unix and Linux for also used in most popular encryption and authentication protocols:

SSL: secure sockets layer

TLS: transport layer security

IPsec: internet protocol security

SSH: secure shell

- **Validating Software**

It's used in validating downloaded software from websites also used in preventing software piracy and detecting unauthorized file modification such as web page defacement signing database keys system file modification virus signature update and so on

7. Conclusion

In conclusion, SHA-256 With its vital role in authentication and security is a well-established cryptographic hash function that's widely used in various industries, including blockchain and cybersecurity. Its design is based on the principles of collision resistance, pre-image resistance, and second pre-image resistance, making it extremely difficult to reverse engineer. SHA-256 has been thoroughly analyzed by the cryptographic community, ensuring its reliability and security. Its implementation is efficient, fast, and available on almost all modern platforms and programming languages. However, as computing power increases, SHA-256 may become vulnerable to attacks or brute-force attacks, necessitating the development of even more secure hash functions. Nonetheless, SHA-256 remains a crucial component in the foundation of modern cryptography and cybersecurity. In this report We managed to discuss all of these points:

- The SHA-256 algorithm is a cryptography algorithm which uses a cryptographic hash function that generates a fixed-size output from any input data.
- Bitcoin is a decentralized digital currency it is a type of money that is completely virtual and is regarded as decentralized.
- Blockchain and bitcoin and how they use hash functions.
- Characteristics of SHA-256.
- How the SHA-256 algorithm works.
- The reason why we can't decrypt encrypted messages using SHA-256.
- Pros and cons of the algorithm.
- Applications of the SHA-256.

8. References

The Basics of Information Security Understanding the Fundamentals of Infosec Book

What Is Bitcoin? How to Mine, Buy, and Use It

<https://www.investopedia.com/terms/b/bitcoin.asp>

What is a hash function? Definition, usage, and examples

<https://www.ionos.com/digitalguide/server/security/hash-function/>

Hash Function - an overview | ScienceDirect Topics

<https://www.sciencedirect.com/topics/computer-science/hash-function>

Golden SHA-256 - Wiki

<https://golden.com/wiki/SHA-256-XKEJ8AB>

SHA-256 Cryptographic Hash Algorithm

<https://komodoplatform.com/en/academy/sha-256-algorithm/#who-invented-sha-256>

SHA 256 Algorithm Explained by a Cyber Security Consultant

<https://sectigostore.com/blog/sha-256-algorithm-explained-by-a-cyber-security-consultant/>

What is SHA-256?

<https://blog.boot.dev/cryptography/how-sha-2-works-step-by-step-sha-256/>

Movable Type Scripts

<https://www.movable-type.co.uk/scripts/sha256.html>

Learn basics of hashing algorithms and how they secure data

<https://guptadeepak.com/understanding-hashing-algorithms-a-beginners-guide/>

What are the pros and cons of using sha256 to hash a password before passing it to bcrypt?

<https://security.stackexchange.com/questions/92175/what-are-the-pros-and-cons-of-using-sha256-to-hash-a-password-before-passing-it>

Is there a limit on the message size for SHA-256?

<https://stackoverflow.com/questions/17388177/is-there-a-limit-on-the-message-size-for-sha-256>

Length extension attack

https://en.wikipedia.org/wiki/Length_extension_attack#cite_note-7

Collision attack

https://en.wikipedia.org/wiki/Collision_attack#Hash_functions

HMAC - wiki

<https://en.wikipedia.org/wiki/HMAC>

SHA-3 - Wiki

<https://en.wikipedia.org/wiki/SHA-3>

Security Analysis of SHA-256 and Sisters

https://link.springer.com/chapter/10.1007/978-3-540-24654-1_13

Quantum Collision Attacks on Reduced SHA-256 and SHA-512

https://link.springer.com/chapter/10.1007/978-3-030-84242-0_22

Known text attack on Hash function (SHA 256 or SHA512)

<https://crypto.stackexchange.com/questions/89561/known-text-attack-on-hash-function-sha-256-or-sha512>

Attacks on and Advances in Secure Hash Algorithms, - ResearchGate

https://www.researchgate.net/publication/307415004_Attacks_on_and_Advances_in_Secure_Hash_Algorithms

Next Generation Cryptography

https://sec.cloudapps.cisco.com/security/center/resources/next_generation_cryptography

Length extension attack

https://en.wikipedia.org/wiki/Length_extension_attack

Schneier on Security

https://www.schneier.com/blog/archives/2017/02/sha-1_collision.html

What is a Collision Attack?

<https://www.comparitech.com/blog/information-security/what-is-a-collision-attack/>

Python SHA256: Implementation and Explanation

<https://www.pythonpool.com/python-sha256/>

Python Implementation from scratch

<https://medium.com/@domspaulo/python-implementation-of-sha-256-from-scratch-924f660c5d57>

Deep dive into SHA-256 (Secure Hash Algorithm 256-bit)

<https://enlear.academy/blockchain-deep-dive-into-sha-256-secure-hash-algorithm-256-bit-824ac0e90b24>

Microsoft SHA-256 class

<https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha256?view=net-7.0>