


Spring 2018

# Application of Huffman Data Compression Algorithm in Hashing Computation

Lakshmi Narasimha Devulapalli Venkata,  
Western Kentucky University, lakshmi.devulapallivenkata974@topper.wku.edu

Follow this and additional works at: <https://digitalcommons.wku.edu/theses>

 Part of the [Digital Communications and Networking Commons](#), [Information Security Commons](#), and the [Theory and Algorithms Commons](#)

---

## Recommended Citation

Devulapalli Venkata,, Lakshmi Narasimha, "Application of Huffman Data Compression Algorithm in Hashing Computation" (2018).  
*Masters Theses & Specialist Projects*. Paper 2614.  
<https://digitalcommons.wku.edu/theses/2614>

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact [topscholar@wku.edu](mailto:topscholar@wku.edu).

APPLICATION OF HUFFMAN DATA COMPRESSION ALGORITHM IN HASHING  
COMPUTATION

A Thesis  
Presented to  
The Faculty of the School of Engineering and Applied Sciences  
Western Kentucky University  
Bowling Green, Kentucky

In Partial Fulfillment  
Of the Requirements for the Degree  
Master of Science

By  
Lakshmi N. Devulapalli Venkata

May 2018

## DEDICATION

To my mother, father, brother, and friends.

&

To the faculty of the Computer Science Department at Western Kentucky University.

## ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Mustafa Atici for his enormous support throughout my masters degree program. It's been an honor and privilege to work under his guidance. I truly appreciate his contributions of time, knowledge and funding that helped me in conducting this research and attending conferences. I can never forget the first example *mississippi* for explaining how Huffman Data Compression algorithm works. His energy and enthusiasm towards research work always motivated me during hard times, especially when I was stuck in understanding the SHA-1 code. Dr. Atici emails his office hours before hand so that it won't be hard to find him at his office. His doors are always open for students who are struggling in understanding assignments or concepts from lectures. He is understanding and never stopped me in implementing my ideas. No Matter where I am, I always remember the quality time that I spent working on this research under his guidance.

Dr. Atici suggested me to step into Artificial Intelligence area after receiving the feedback of my poster presentation at a conference in March 2018. I might reconsider this thesis work for my Ph.D to implement the same in AI with his permission and again under his guidance.

To Dr. Qi Li, Associate professor, Thank you. Dr. Li taught me Data mining, one of the core subjects trending today. Thank you for your help throughout my course work and for accepting my request to be one of the thesis committee members.

To Dr. Michael Galloway, Assistant professor. Dr. Galloway taught me Software Engineering course, his lectures inspired and motivated me to become a Scrum Master.

Thank you Dr. Galloway for your support and accepting my request to be one of the thesis committee members.

To the Faculty members, office staff of SEAS Department and rest of the Western Kentucky University faculty, Thank you. You are the reason that I came all the way to this university and had wonderful memories that I cherish to the rest of my life. I Sincerely apologize that I cannot thank everyone Individually. Thank you all for having me here. I truly appreciate it.

I would like to thank my parents for their continuous support throughout my masters program and special thanks to my roommate, friends and ISA committee, who supported and motivated me during my hard times. I truly appreciate everything you did.

Finally, I would like to thank SEAS department computer science division, the graduate school and Ogden college for providing financial support during the research work. Thank you for everything you gave me here at WKU.

## CONTENTS

1	INTRODUCTION . . . . .	1
2	DEFINITIONS . . . . .	6
2.1	Message . . . . .	6
2.2	Sender and Receiver . . . . .	6
2.3	Encryption and Decryption . . . . .	6
2.3.1	1-1 Function . . . . .	7
2.4	Protocol . . . . .	7
2.5	Cryptographic Primitives . . . . .	8
2.6	Cryptosystem . . . . .	8
2.7	Symmetric-key cryptography . . . . .	10
2.7.1	Stream ciphers . . . . .	11
2.7.2	Block ciphers . . . . .	11
2.8	Public-key cryptography . . . . .	12
2.9	Cryptanalysis . . . . .	12
3	CRYPTOGRAPHIC HASH FUNCTIONS . . . . .	14
3.1	Preimage resistant . . . . .	15
3.2	Secondary Preimage resistant . . . . .	15
3.3	Collision resistant . . . . .	15
3.4	Iterated Hash functions . . . . .	16
3.4.1	Outline of Compression function . . . . .	16
3.5	Merkle—Damgård construction . . . . .	18

3.6	Description of SHA-1 . . . . .	20
3.6.1	SHA-1 overview . . . . .	20
3.6.2	SHA-1 Padding scheme . . . . .	20
3.6.3	SHA-1 Compression function . . . . .	22
3.6.4	SHA-1 Message Digest computation . . . . .	24
3.7	Applications of SHA-1 . . . . .	26
3.7.1	Data Integrity . . . . .	26
3.7.2	Digital Signature . . . . .	27
4	HUFFMAN COMPRESSION ALGORITHM . . . . .	28
4.1	File compression . . . . .	28
4.2	Optimal compression . . . . .	29
4.3	Huffman Data compression . . . . .	30
4.3.1	Construction of algorithm . . . . .	31
4.3.2	Huffman Tree . . . . .	31
5	NEW SEED CONSTRUCTION . . . . .	34
5.1	Base . . . . .	34
5.1.1	SHA-1 as base . . . . .	34
5.2	New Approach . . . . .	34
5.2.1	Why Huffman compressed code . . . . .	34
5.3	New Seed Construction . . . . .	35
5.3.1	Steps involved in construction . . . . .	36
5.4	Observations . . . . .	37

6 CONCLUSION . . . . .	42
REFERENCES . . . . .	43
APPENDICES . . . . .	47
A APPENDIX A: SETTING UP VISUAL STUDIO . . . . .	48
A.1 Installation guide for Visual Studio Community 2017 . . . . .	48
B APPENDIX B: SHA-1 OF HUFFMAN COMPRESSED CODES . . . . .	50
C APPENDIX C: NEW SEED CONSTRUCTION SOURCE . . . . .	60



## LIST OF TABLES

4.1	Character frequency table for Encoded Tree Structure . . . . .	30
4.2	Character frequency table for Encoded Tree Structure – 2 . . . . .	30
4.3	Character frequency table of Huffman Tree . . . . .	33
5.1	Character frequency table of Huffman Tree – seed construction . . . . .	36
5.2	Observation Table 1 . . . . .	39
5.3	Observation Table 2 . . . . .	39
5.4	Observation Table 3 . . . . .	39

## LIST OF FIGURES

2.1	Encryption and Decryption . . . . .	7
2.2	One-One function from Domain X to Range Y . . . . .	7
2.3	Encryption and Decryption with key . . . . .	9
2.4	Symmetric-key cyrptography . . . . .	10
2.5	Stream cipher encryption . . . . .	10
2.6	Stream cipher decryption . . . . .	11
2.7	Block cipher . . . . .	12
2.8	Public-key cryptography . . . . .	13
3.1	Domain X with N values and Range Y with M values . . . . .	14
3.2	Compression function . . . . .	17
3.3	Merkle—Damgård construction . . . . .	19
3.4	Data Integrity . . . . .	26
3.5	Digital Signature . . . . .	27
4.1	Encoded Tree structure . . . . .	29
4.2	Encoded Tree structure – 2 . . . . .	30
4.3	Huffman level 1 . . . . .	32
4.4	Huffman Tree . . . . .	32
5.1	Huffman Tree – seed construction . . . . .	36
5.2	Compression function output . . . . .	38
5.3	New Seed construction output – our observation . . . . .	38
5.4	Google’s sample message 1 . . . . .	40

5.5	Google’s sample message 2 . . . . .	41
A.1	Visual Studio downloads page . . . . .	48
A.2	Installation policy . . . . .	48
A.3	Installation packages . . . . .	49
B.1	<i>SHA1-Header.h</i> . . . . .	50
B.2	<i>SHA-1 Source.cpp</i> . . . . .	51
B.3	<i>SHA-1 Source.cpp continuation-1</i> . . . . .	52
B.4	<i>SHA-1 Source.cpp continuation-2</i> . . . . .	53
B.5	<i>SHA-1 Source.cpp continuation-3</i> . . . . .	54
B.6	<i>SHA-1 Source.cpp continuation-4</i> . . . . .	55
B.7	<i>Huffman.h</i> . . . . .	56
B.8	<i>Huffman.h continuation - 1</i> . . . . .	57
B.9	<i>Compression.cpp</i> . . . . .	58
B.10	<i>Compression.cpp continuation - 1</i> . . . . .	59
B.11	<i>Input.txt</i> . . . . .	59
B.12	<i>HuffmanCodes.txt</i> . . . . .	59
C.1	<i>NewSeedHeader.cpp</i> . . . . .	60
C.2	<i>NewSeedSource.cpp</i> . . . . .	61
C.3	<i>NewSeedSource.cpp continuation-1</i> . . . . .	62
C.4	<i>NewSeedSource.cpp Continuation-2</i> . . . . .	63
C.5	<i>NewSeedSource.cpp Continuation-3</i> . . . . .	64
C.6	<i>NewSeedSource.cpp Continuation-4</i> . . . . .	65

C.7	<i>NewSeedSource.cpp Continuation-5</i>	66
C.8	<i>NewSeedMain.cpp</i>	67
C.9	<i>HuffmanSHA1input.txt</i>	68

# APPLICATION OF HUFFMAN DATA COMPRESSION ALGORITHM IN HASHING COMPUTATION

Lakshmi N. Devulapalli Venkata

May 2018

68 Pages

Directed by: Dr. Mustafa Atici, Dr. Qi Li, Dr. Michael Galloway

School of Engineering and Applied Sciences

Western Kentucky University

Cryptography is the art of protecting information by encrypting the original message into an unreadable format. A cryptographic hash function is a hash function which takes an arbitrary length of the text message as input and converts that text into a fixed length of encrypted characters which is infeasible to invert. The values returned by the hash function are called as the *message digest* or *simply hash values*. Because of its versatility, hash functions are used in many applications such as message authentication, digital signatures, and password hashing [Thomsen and Knudsen, 2005].

The purpose of this study is to apply Huffman data compression algorithm to the SHA-1 hash function in cryptography. Huffman data compression algorithm is an optimal compression or prefix algorithm where the frequencies of the letters are used to compress the data [Huffman, 1952]. An integrated approach is applied to achieve new compressed hash function by integrating Huffman compressed codes in the core functionality of hashing computation of the original hash function.

## Chapter 1

### INTRODUCTION

In recent years, the Internet has taken over the globe with the technological revolution in many industries as it provides communication between millions of people and different nations, there will be a huge demand for security to protect the confidentiality of the information. This can be achieved through cryptography. As we know, cryptography is derived from ancient Greek words *kryptos* and *graphein*, meaning hidden writing. If we observe the traces of ancient history, this art was documented between 1900 and 2000 BC in ancient Egypt in the form of Hieroglyphs [Hat, 2013].

Another strong evidence that the early civilizations used cryptography is the *Arthashastra*, which is also known as the science of politics, written by Kautilya between 2nd BC and 3rd BC for military strategy and communication with spies in hidden codes [Boesche, 2002]. In 100 BC, Roman dictator Julius Caesar used an encrypted form to communicate with his army generals. This famous method of secret writing was known as Caesar cipher. The algorithm was used to encrypt and decrypt the plaintext message by shifting each character to 3 places. This algorithm was not secure as the cipher can be broken if we know the frequency shift of each character. Later in 16th century, A French scientist named Blaise de Vigenère introduced first auto key (encryption key) ciphers. In this algorithm, a key is repeated several times and a modulo operation is performed on a plaintext to produce a ciphertext [David, 1999]. However, this algorithm was broken in 19th century, but the notion of using autokey or encryption key has created new roots for modern age cryptography. This period was also known for industrial revolutions, scientists designed electro-mechanical machines in which the motors are responsible for generating ciphertext

when the key is set to a value. This was initially used for military purposes, post-World War II, many cryptographers were attracted by commercial industries with the intention of securing the data from their competitors [Hat, 2013].

On observing the history, cryptographers stated that security should depend on the secrecy of a key but not an encryption algorithm. Modern cryptography includes mathematical problems which are difficult to solve, and the algorithms are a complex, time-consuming process for humans to solve and are executed by computers with powerful hardware devices [Simpson, 1997]. The most important objectives of cryptography are *data integrity*, *authentication* and *confidentiality*. In Data Integrity, the data cannot be altered when stored in an insecure place or when it is being transmitted through an insecure channel. Authentication deals with the user identity and confidentiality, the data stored by an individual can be encrypted in an unreadable format and later it can be decrypted.

Modern cryptography is further divided into different algorithms such as symmetric-key cryptography, this algorithm refers to an encryption technique where both sender and receiver share the same key and public-key cryptography, unlike symmetric-key this algorithm uses a public-key and a private-key, though they are different they are related mathematically. The public-key is designed in such a way that computing a private-key is infeasible from it, even though they are related [Diffie and Hellman, 1976]. Another branch of cryptography is the cryptographic hash functions.

These hash functions are considered as a special class in this field as they are designed to be one-way functions, which are not feasible to invert but relatively easy to compute. Hash functions take an arbitrary length of data as input and generate a message digest (hash values) of fixed length between 128 and 512 bits. These hash values are used in

digital signatures to verify the data integrity. If the data is altered, then the hash value is no longer valid. Though the data is stored in an insecure environment, its integrity can be checked from time to time by computing the hash value and confirm that it is not changed. A detailed description about symmetric-key, public-key cryptography and cryptographic hash functions is mentioned in chapter 2 and chapter 3.

There are many cryptographic hash functions which are broken previously and some of them are used today in real-world applications. In 1989, Ronal Rivest first introduced us to Message Digest 2 (MD2) produces 128-bit message digest [[networks, networks], [Kaliski, 1992]]. He improved on MD2 and developed MD4 [[Rivest, 1990], [Schneier, 2005]]. In 1991, Security weakness was found in this algorithm which led to the development of MD5 [Rivest, 1992] (introduced in 1992) and most of the industries deployed it in their applications. In 1993, National Institute of Standards and Technology (NIST) introduced SHA-0 [Barker, 1993] very similar to MD5, but it was withdrawn in a short period by NIST without disclosing the reason and Introduced SHA-1 under FIPS 180-1. This algorithm produces 160-bit message digest when an arbitrary length of message less than  $2^{64}$  is given as input [FIPS, 1995].

According to NIST, all these hash functions including SHA-1 share common phenomena, *Security Attacks*. In 2005, a group of researchers broke the collision resistance of MD5 hash function [Wang and Yu, 2005]. This incredible work inspired many cryptographers to develop perfect hash functions and cryptanalysts to design algorithms that can expose the weakness of the hash functions. The same team showed first theoretical attack on SHA-1, which is widely used after MD5 [Wang, Yin, and Yu, 2005]. Based on this groundbreaking work, Dr. ir. Marc Stevens and his team demonstrated in [Stevens,



2013] at EUROCRYPT 2013 the best collision attack at  $2^{61}$  calls to SHA-1 hash function. As proof of concept, Karpman demonstrated a 76-round practical collision for SHA-1 in [Karpman, Peyrin, and Stevens, 2015]. Dr. Steven's stated that they improvised the start-from-the-middle approach [Karpman et al., 2015] using [Joux and Peyrin, 2007] auxiliary paths speed up technique to find first colliding pair for full SHA-1, which led to freestart collision [Stevens, Karpman, and Peyrin, Stevens et al.]. As we know, the hash functions are constructed based on Merkle—Damgård paradigm [[Merkle, Charles, et al., 1979], [Damgård, 1989]], they take message block and chaining value (Initialization vector, IV) as input to build the hash function by iterating the compression function. In [Stevens, Karpman, and Peyrin, Stevens et al.] Dr. Steven's stated that the attacker can choose the IV for a freestart collision, they found to IV's with slight difference (say two bits) and fed them to the process block. This freestart collision gave a standard basis for full SHA-1 collision and the same team came up with the first practical collision of SHA-1 in 2017, this attack took 6500 years of single CPU and required around 9 quintillion SHA-1 computations [Stevens, Bursztein, Karpman, Albertini, and Markov, 2017]. They warned all the industry actors to migrate to safe standards before any real-time attacks, as SHA-1 is no longer safe to use for digital signatures [Goodin, 2017]. Even before the first practical collision of SHA-1, Microsoft announced in 2016 to deprecate and replace SHA-1 by SHA-256 by January 2017 for security reasons to guarantee the SSL durability, even though the Certification Authorities (CA) did not recommend any changes. Since then most of the industry actors decided to agree on not issuing and trusting SHA-1 certificates [tbs internet, 2018]. However, due to the increased cryptanalytic attacks and computational power, there is always a need for

secure hash function. For this very reason, every cryptographer and researcher including us are striving to develop a secure and perfect hash function.

We started this study by understanding the basic concepts of cryptography and cryptographic hash functions. After studying recent cryptanalytic attacks, we understood that for any hash function preprocessing stage plays an important role. First we introduced the concept of *salt*, that is, adding extra bits to the hash value of the message. Later, we decided to introduce Huffman compressed codes in to preprocessing stage by eliminating the concept of *salt*, as adding this does not provide enough security to strengthen the hash algorithm.

This thesis is organized as following. In chapter 2, we introduced you to the basic definitions involved in cryptography. In chapter 3, we explain about cryptographic hash functions, construction of Merkle—Damgård paradigm and SHA-1 hash function and its applications. Chapter 4 explains about the Huffman compression Algorithm and in chapter 5 we introduce you to our algorithm, that is, the seed construction based on SHA-1 using the Huffman compressed codes. The remaining chapters are conclusion, appendix, and references.

## Chapter 2

### DEFINITIONS

#### 2.1 Message

A message  $\mathcal{M}$  is plaintext used for communication between sender and receiver and easily readable by humans. This message is of arbitrary length and used as input for any cryptographic hash functions.

#### 2.2 Sender and Receiver

The two ends of a communication channels are called sender and receiver. Sender transfers a message to the receiver on the other end of the channel. This sender wants to transfer the message securely to make sure no attacker reads or modifies the original message.

#### 2.3 Encryption and Decryption

From 2.1 we stated that a message is plaintext, manipulating this plaintext in such a way to hide it from the attacker is called encryption ( $\mathcal{E}$ ). This encrypted message is called ciphertext  $\mathcal{C}$ . Processing the ciphertext back to original plaintext is called decryption ( $\mathcal{D}$ ). Let plaintext be denoted by  $\mathcal{M}$ , manipulated by encryption algorithm  $\mathcal{E}$  to produce ciphertext  $\mathcal{C}$ . This ciphertext can be larger or as same as the  $\mathcal{M}$ .

$$\therefore \mathcal{E}(\mathcal{M}) = \mathcal{C}$$

To recover the plaintext  $\mathcal{M}$  from  $\mathcal{C}$ , we need to do the reverse process called decryption  $\mathcal{D}$  on  $\mathcal{C}$ . Figure 2.1 describes the encryption and decryption process.

$$\therefore \mathcal{D}(\mathcal{C}) = \mathcal{M}$$

All the encryption and decryption functions must be 1-1 functions.

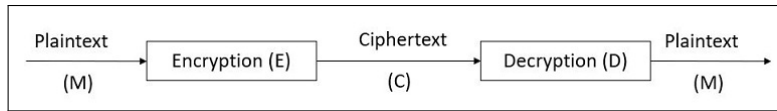


Figure 2.1: Encryption and Decryption

### 2.3.1 1-1 Function

A function is said to be 1-1(one-one) function, if each element in the range  $Y$  is mapped with at most one element in the domain  $X$  [Menezes, Van Oorschot, and Vanstone, 1996]. For example, consider the set  $X$  as  $\{a, b, c, d, e\}$  and set  $Y$  as  $\{1, 2, 3, 4, 5, 6\}$ , and the function from  $X$  to  $Y$  is shown in Figure 2.2. If  $f$  is the function and  $f: X \rightarrow Y$  is 1-1 function then the image of the function  $\text{Im}(f) = Y$ , where function  $f$  is called bijection [Schneier, Schneier].

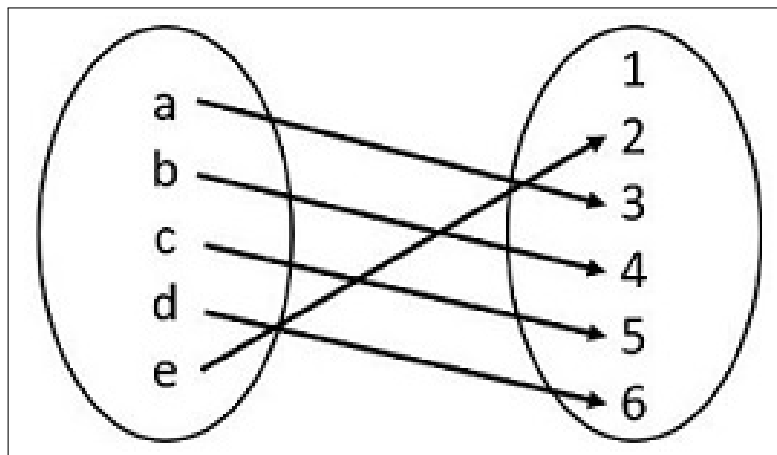


Figure 2.2: One-One function from Domain  $X$  to Range  $Y$

## 2.4 Protocol

Communication systems involving two or more parties to transmit information through any medium is called a Protocol. If the communication system involves only one party, then it is called a Procedure. For example, let us suppose that we have two players Alice and Bob. The purpose of the protocol is to flip a coin, based on the result one of

the players will plan of their next move. After choosing the sides, one of the players will flip the coin. If Alice wins, he decides the next event to do, else Bob makes the decision [Buchmann, 2013].

Let us consider another example with same players. Imagine, a cellular phone is acting as medium between the players and they intend to play the same coin flipping game over the phone. If Alice offers a side to Bob and flips the coin and tell Bob the he lost or won the game, Bob may disagree with decision as he cannot verify the result. In cryptographic point of view, this example can be termed as cryptographic protocol or security protocol [Buchmann, 2013].

## 2.5 Cryptographic Primitives

Cryptographic primitives are used as basic building blocks by cryptographers to construct cryptosystem for security purposes. These primitives are designed for a specific task. For example, digital signatures are one of the primitives widely used. Its task is to verify the authenticity of the documents.

## 2.6 Cryptosystem

A Cryptosystem is a package of cryptographic algorithms or combination of cryptographic primitives designed to perform a task guaranteeing security. A typical cryptosystem consists algorithms for key generation, encryption, and decryption. From [Buchmann, 2013], a cryptosystem can be defined as a tuple  $(\mathcal{M}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$  with properties as follows.  $\mathcal{M}$  is the set plaintexts,  $\mathcal{C}$  is the set of ciphertexts,  $\mathcal{K}$  is the key space and  $\mathcal{E}, \mathcal{D}$  are set of encryption and decryption functions.

In mathematical terms, a cryptographic algorithm can be termed as a cipher, a function used for encryption and decryption. If the security of the algorithm depends on the

secrecy how it works, it is called a restricted algorithm. This algorithm is considered as inadequate according to today's standard and cannot be used by a large group of users. For example, if one of the individuals leave the group, remaining group members should switch to a different algorithm. This requires everyone to have their own unique algorithm and they cannot depend on any hardware or software products, as the attacker can buy the same product and learn the algorithm. Considering the drawbacks, even today restricted algorithms are used for low-level security applications. Modern cryptography states that this problem can be solved using a key ( $k$ ). This  $k$  is used in encryption and decryption operations [Schneier, Schneier].

$\therefore \mathcal{E} = \mathcal{E}_k$ , where  $k \in \mathcal{K}$  and  $\mathcal{E}_k(\mathcal{M}) = \mathcal{C}$  for encryption. Similarly,  $\mathcal{D} = \mathcal{D}_k$ , where  $k \in \mathcal{K}$  and  $\mathcal{D}_k(\mathcal{C}) = \mathcal{M}$  for decryption.

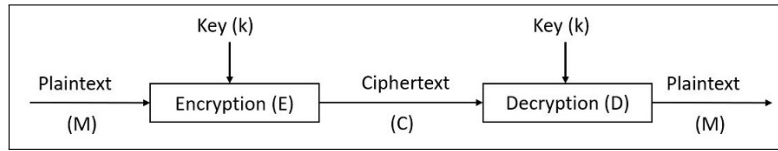


Figure 2.3: Encryption and Decryption with key

From Figure 2.3, we have  $\mathcal{D}_k(\mathcal{E}_k(\mathcal{M})) = \mathcal{M}$ . Some algorithms may use different keys for encryption and decryption.

$\therefore \mathcal{E} = \mathcal{E}_{ke}$  and  $\mathcal{E}_{ke}(\mathcal{M}) = \mathcal{C}$  for encryption,  $\mathcal{D} = \mathcal{D}_{kd}$  and  $\mathcal{D}_{kd}(\mathcal{C}) = \mathcal{M}$  for decryption. Therefore, like Figure 2.3 property, we can define  $\mathcal{D}_{kd}(\mathcal{E}_{ke}(\mathcal{M})) = \mathcal{M}$ .

Now, the security of these algorithms does not depend on the algorithm but on the keys ( $k$  or  $ke, kd$ ).

## 2.7 Symmetric-key cryptography

This algorithm enables a sender and a receiver to share a secret key for communication, see Figure 2.4. This key, in few cases, can be same for encryption and decryption. The main requirement for this algorithm to work successfully is maintaining the secrecy of the key, but this is a major drawback. If an attacker learns about the key and tweaks the input message, then, the communication channel and message are considered as corrupted.

$$\mathcal{E}_k(\mathcal{M}) = \mathcal{C} \text{ and } \mathcal{D}_k(\mathcal{C}) = \mathcal{M}$$

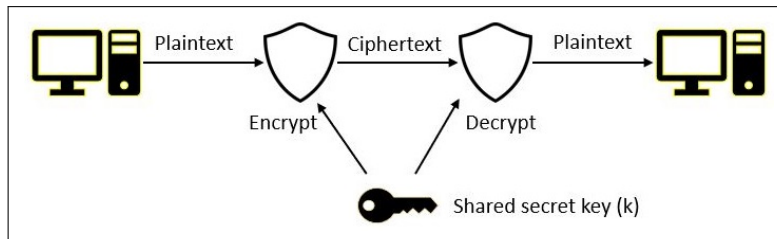


Figure 2.4: Symmetric-key cryptography

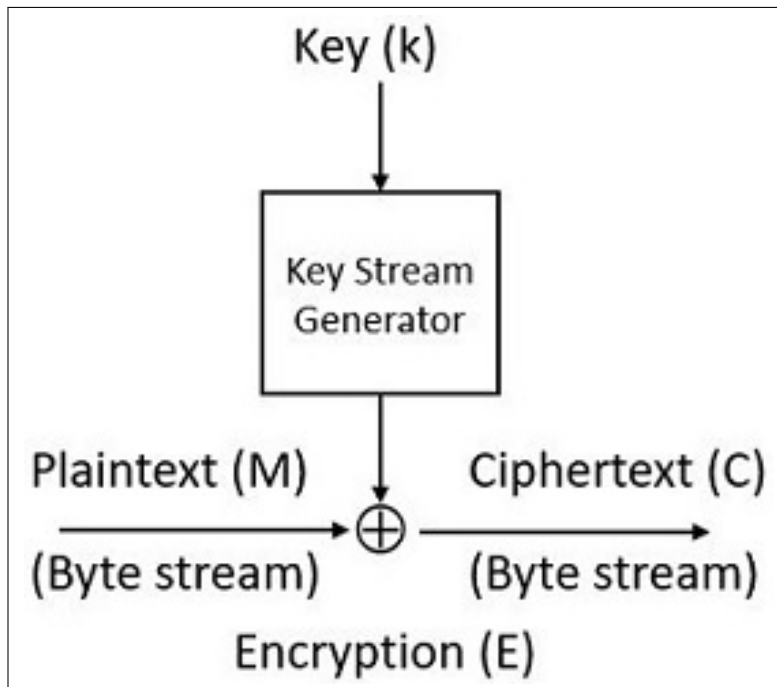


Figure 2.5: Stream cipher encryption

### 2.7.1 Stream ciphers

Symmetric-key cryptography is divided into stream ciphers and block ciphers. Stream ciphers process single bits at a time of a plaintext. Each bit is encrypted with a key which is passed to a key stream generator, which generates a key stream. These message bits and key bits are XORed to produce the ciphertext. The same key must be shared for decryption where ciphertext bits are XORed with key bits to produce plaintext [Schneier, Schneier], see Figures 2.5 and 2.6.

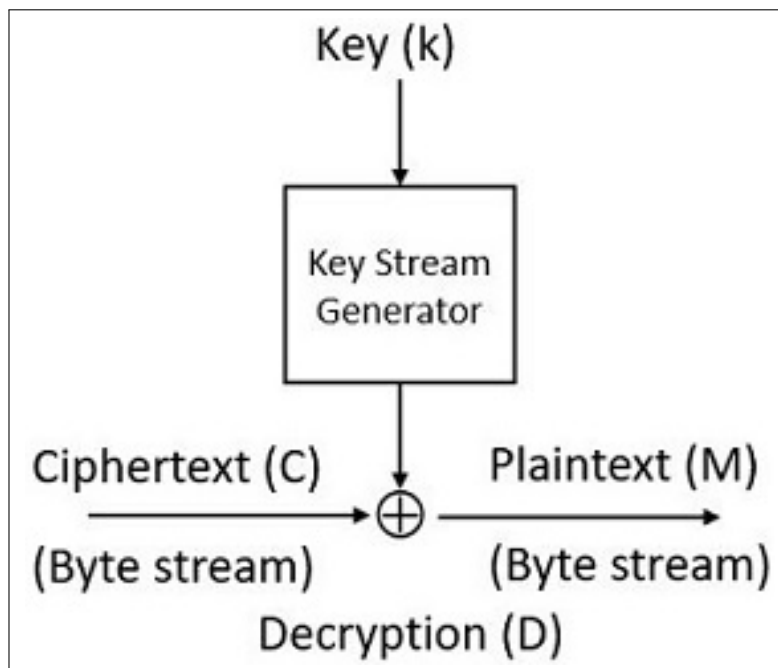


Figure 2.6: Stream cipher decryption

### 2.7.2 Block ciphers

From 2.7.1, stream ciphers consider message bits and key bits together into streams to produce ciphertexts, whereas in block cipher the message is broken down into blocks of fixed length and each block is encrypted using the same key, see Figure 2.7. For example, DES is a block cipher where blocks of 64 bits are encrypted using a 56-bit key.



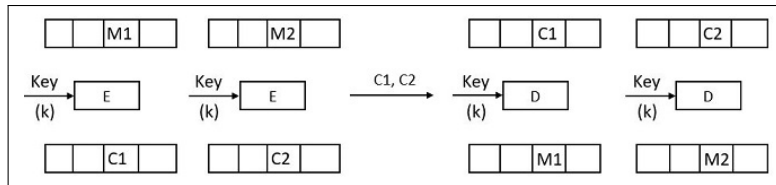


Figure 2.7: Block cipher

## 2.8 Public-key cryptography

In contrast to symmetric-key cryptography, this algorithm uses two keys, a public-key and a private-key. The sender whenever transfers a message, he/she uses the public-key to encrypt the original message. The receiver on the other end of the communication channel uses a private-key to decrypt the message, see figure. The only advantage here is, the sender cannot decrypt the message once encrypted and the decryption key at any given point of time, cannot be calculated from the encryption key. The encryption key ( $\mathcal{E}_k$ ) is called **public-key** and decryption key ( $\mathcal{D}_k$ ) is called **private-key** or **secret-key**. This algorithm is called **public-key** because  $\mathcal{E}_k$  can be made publicly. Any stranger can access this public-key to encrypt the original message, but only the receiver with  $\mathcal{D}_k$  can decrypt and read the messages [Schneier, 2005].  $\mathcal{E}_k(\mathcal{M}) = \mathcal{C}$  represents, encryption using public-key and  $\mathcal{D}_k(\mathcal{C}) = \mathcal{M}$  is decryption using private key. Again, in this process, there may be an ambiguity whether the message was received from the original sender as there are chances for man-in-the-middle to attack. To ensure the security, these messages are digitally signed using digital signature scheme [see section 3.7.1].

## 2.9 Cryptanalysis

The goal of cryptography is to protect the information (keep plaintext or the key secret) from attackers. Cryptanalysis is the science of breaking the cryptosystems (recover the original message).

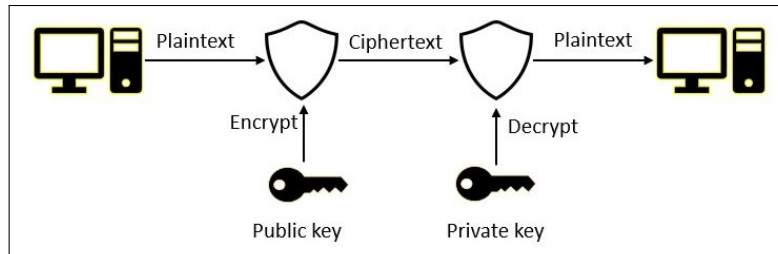


Figure 2.8: Public-key cryptography

A successful attempt of cryptanalysis may recover the key used for encryption. It also helps in finding the weakness of a cryptosystem. It gained importance along with cryptography as it describes whether a system is secure or not. On other hand, cryptanalysts play a major role in highlighting the weakness of (or different ways to compromise) a system.

However, Cryptanalysis is the only way to assure that a system is secure, hence it is considered as an integral part of cryptology.

## Chapter 3

### CRYPTOGRAPHIC HASH FUNCTIONS

In section 2.6 we have seen that encryption system has 5 tuples, whereas in hash function we have only 4 tuples. The reason is we do not have decryption ( $\mathcal{D}$ ). It is supposed to be one-way function [Thomsen and Knudsen, 2005]. The four tuples are set of possible messages ( $\mathcal{X}$ ), finite of set of possible message digests ( $\mathcal{Y}$ ), finite set of possible keys ( $\mathcal{K}$ ) and finite set of functions ( $\mathcal{H}$ ) such that  $h_k : X \rightarrow Y$  for each  $h_k \in \mathcal{H}$  and  $k \in \mathcal{K}$ . Let us assume that there are  $N$  values in  $\mathcal{X}$  and  $M$  values in  $\mathcal{Y}$ . Therefore, there are  $M$  possible mappings, that is, for each possible input  $N$ , there are  $M$  possible outputs, see Figure 3.1.

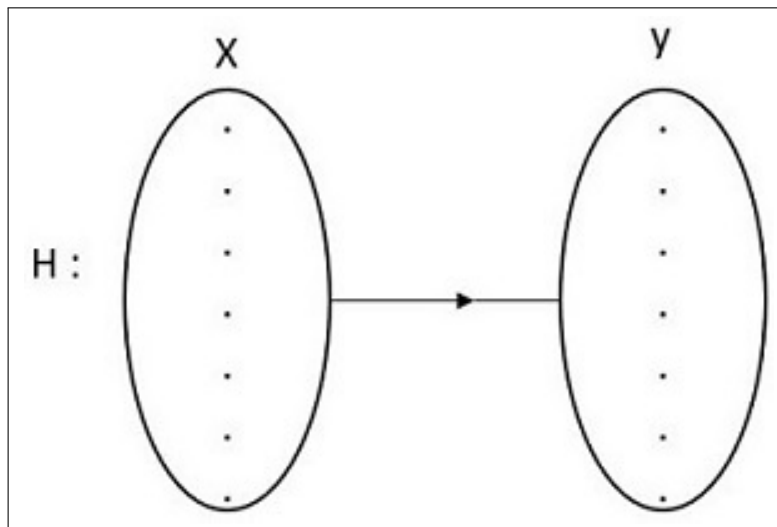


Figure 3.1: Domain  $X$  with  $N$  values and Range  $Y$  with  $M$  values

Hence, there are  $M^N$  possible transformations, from this pool we get to choose any one function and denote as  $\mathcal{H}^{X, Y}$  and the hash function will be one of them. If the hash function  $\mathcal{H} \subseteq \mathcal{H}^{X, Y}$ , this is called as the hash family or  $(N, M)$ -hash family where we chose one function [Stinson, 2005]. The hash functions have three important properties which must be satisfied, and these properties are required for the security of the applications

which we use in real-time. They are Preimage resistant, Secondary preimage resistant and Collision resistant.

### **3.1 Preimage resistant**

Preimage property defines one-wayness. Hash function needs to be one-way. If there is a hash function  $h$  and given a message digest, the problem here is to compute the inverse of  $h(x)$ . For a given  $h(x)$ , if we compute the value of  $x$  which will lead to the same hash value, then  $(x, y)$  is a valid pair. An efficient algorithm should easily solve this. The main idea here is the hash function for which the preimage cannot be efficiently solved is said to be preimage resistant.

### **3.2 Secondary Preimage resistant**

Second preimage defines that, suppose we are provided with hash function  $h$  and an element  $x$ , we are supposed to give another value  $x'$  which is not equal to  $x$  but leads to the same message digest  $y$ . This property is partially different from preimage. The reason is, we are provided with a value of  $x$  and then we must compute the value of  $h(x)$ . Now, we have to find out the value of  $x'$  which is not equal to  $x$  but leads to the same hash value. The real difference here is, suppose an algorithm solves preimage, it is not necessary for it to solve the second preimage, because, in preimage it is just giving the inverse but in second preimage problem, we are using the same algorithm. The problem is that we get  $x$  which is equal to the value of  $x'$ .

### **3.3 Collision resistant**

It the most important and widely studied property. In this property for a given hash function, we must result only two values,  $x$  and  $x'$  which are not same ( $x \neq x'$ ) but leads to same hash value  $h(x) = h(x')$ . Therefore, if this solved, say  $(x, y)$  is a valid pair then  $(x', y)$

is also a valid pair. If this is not efficiently solvable, then the hash function is said to be collision resistant.

### 3.4 Iterated Hash functions

Iterated hash functions deal with a technique called compression which can be used to compute a hash function with infinite domain. Hash function  $h$  constructed by this technique is called Iterated Hash function. Most hash functions like MD4, MD5, Secure Hash Algorithms practiced in real-time are iterated hash functions. Here, we consider hash functions whose inputs and outputs are bit strings. Consider an input message  $x$  with bit strings zeros and ones, the length of this bit string is  $|x|$ . Similarly, if we have bit strings  $x_1$  and  $x_2$ , the concatenation can be denoted as  $x_1 \parallel x_2$  [Stinson, 2005].

#### 3.4.1 Outline of Compression function

Let us suppose that  $\mathcal{F} : \{0, 1\}^{(m+n)} \rightarrow \{0, 1\}^m$  is a compression function that takes  $(m+n)$  bits as input and produces  $m$  bits output, where  $n \geq 1$ . Based on this compression function, we construct an iterated hash function  $h$ . This is achieved in three steps, they are preprocessing stage, processing stage, and output transformation.

Consider an input string  $x$ , which produces output  $y$  is divided into sub-blocks such that each block is divisible by  $n$  or equal to  $n$ . The last block may or may not be equal to  $n$ , therefore we pad it with extra bits by extending to make it equal to  $n$  or multiples of  $n$  denoted as  $y = y_1 \parallel y_2 \parallel y_3 \parallel \dots \parallel y_k$ , where  $|y_t| = n$  for  $1 \leq t \leq k$ . This is called the preprocessing stage [Stinson, 2005].

In the processing stage, we feed  $n$  bits from  $y$  and  $m$  bits to the compression function. These  $m$  bits are called as Initialization vectors or IV, this results in  $m$  bit output say  $z_1$ . Now, the second block of  $n$  and  $z_1$  are fed to the compression function which produces  $z_2$ .

This process continues till  $k^{th}$  block which produces  $z_k$ . Therefore,  $z_k$  is the  $h(x)$ , see Figure 3.2. Sometimes, in the output transformation stage, this  $z_k$  is given an optional transformation  $g(z_k)$ , which is called  $h(x)$ . While constructing a hash function, we should ensure that the preprocessing step is an Injection (one-to-one property). If it is not one-to-one, then it may be possible to find  $x \neq x'$  so that  $y = x'$  then  $h(x) = h(x')$ . Therefore,  $h$  would not be a collision resistant.

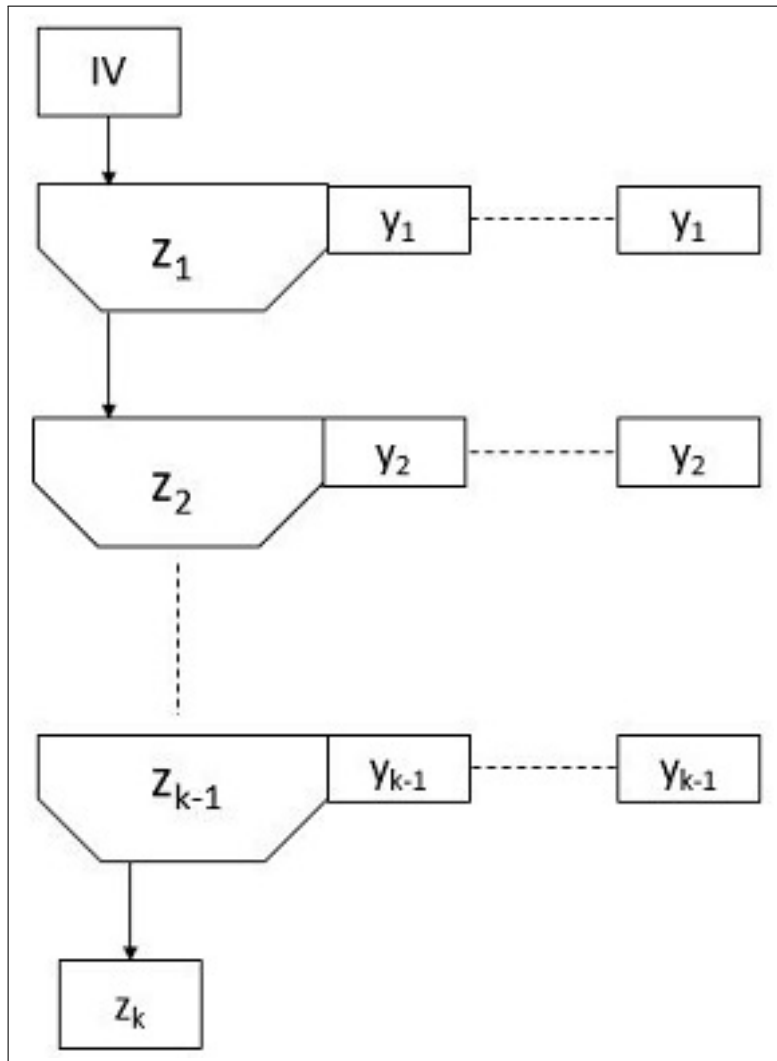


Figure 3.2: Compression function

### 3.5 Merkle—Damgård construction

The MD hash function or construction was first described in [[Merkle et al., 1979], [Damgård, 1989]]. It uses a compression function  $\mathcal{F} : \{0, 1\}^{(m+n)} \rightarrow \{0, 1\}^m$  (from section 3.4.1), which is collision resistant to construct a collision resistant hash function  $h: \{0, 1\}^* \rightarrow \{0, 1\}^m$ . From [Stinson, 2005] we give brief note about Algorithm 1 which describes Merkle—Damgård construction.

---

**Algorithm 1** Merkle—Damgård Algorithm

---

```

1: external compress
2: comment: compression:  $\{0, 1\}^{(m+t)} \rightarrow \{0, 1\}^m$ , where  $t \geq 2$ 
3:  $n \leftarrow |x|$ 
4:  $k \leftarrow \lceil \frac{n}{t-1} \rceil$ 
5:  $d \leftarrow k(t-1) - n$ 
6: for  $i \leftarrow 1$  to  $k-1$  do
|   do  $y_i \leftarrow x_i$ 
|   end
7:  $y_k \leftarrow x_k \| 0^d$ 
8:  $y_{k+1} \leftarrow$  the binary representation of  $d$ 
9:  $z_1 \leftarrow 0^{m+1} \| y_1$ 
10:  $g_1 \leftarrow \text{compress}(z_1)$ 
11: for  $i \leftarrow k$  do
|   do  $\begin{cases} z_{i+1} \leftarrow g_i \| 1 \| y_{i+1} \\ g_{i+1} \leftarrow \text{compress}(z_{i+1}) \end{cases}$ 
|   end
12:  $h(x) \leftarrow g_{k+1}$ 
13: return  $(h(x))$ 

```

---

Let us suppose that  $x$  is the message and divided into  $k$  blocks in the preprocessing stage, say  $x = x_1 \| x_2 \| x_3 \| \dots \| x_k$ . The size of each block  $|x_1| = |x_2| = |x_3| = \dots = |x_{k-1}|$  is equal to  $t-1$ , unlike the compression function described in 3.4.1 and the size of last block  $|x_{k-1}| = t-1-d$ . To make the last block of size  $t-1$ , we add  $d$  0s, where size of  $d$  is  $0 \leq d \leq t-2$ . The value  $k$  depends on the size of  $x$ , denoted as  $n$  and  $d$  0s are padded.

$$\therefore k = \frac{n+d}{t-1} = \lceil \frac{n}{t-1} \rceil$$

Ideally, we assumed that  $t \geq 1$ , but from Algorithm we consider  $t \geq 2$  because if  $t = 1$ , then the value of the  $k$  is undefined. The for-loop  $i$  runs from 1 to  $k-1$ , here we assign each  $x_i$  value to  $y_i$ , this is the preprocessing step. For the  $k^{th}$  block we take  $x_k$  right padded with

$d$ -zeros ( $0^d$ ) and assigned to  $y_k$  so that, the size of  $y_k$  is of length  $t-1$ . In the next step, we assign the binary representation of  $d$  to  $y_{k+1}$  and padded to the left with zeros so that  $|y_{k+1}|$  is equal to  $t-1$ . This is known as MD strengthening, it helps to make the preprocessing step injective.

In the processing stage, we first consider  $y_1$  which has  $t-1$  bits. It is then padded with  $(m+1)$  0s, therefore the size equals to  $(m+t)$  values. These values are fed to the compression function  $g_1$  which gives  $m$  bit value. Therefore, we continue the iterative process, that is,  $g_i$  obtained in the previous step is concatenated with extra bit 1 again concatenated with  $y_{i+1}$ . The size of this string is  $(m+t)$ , which is again fed to the compression function to get  $m$  bit values as output. This can be iterated to obtain values like  $g_1, g_2, \dots, g_{k+1}$ . Therefore, the final value  $g_{k+1}$  is the hash value, that is  $h(x)$ . See Figure 3.3

If the compression function is collision resistant, then the hash function is said to be collision resistant. Hash functions like MD4, MD5, SHA follow this construction method.

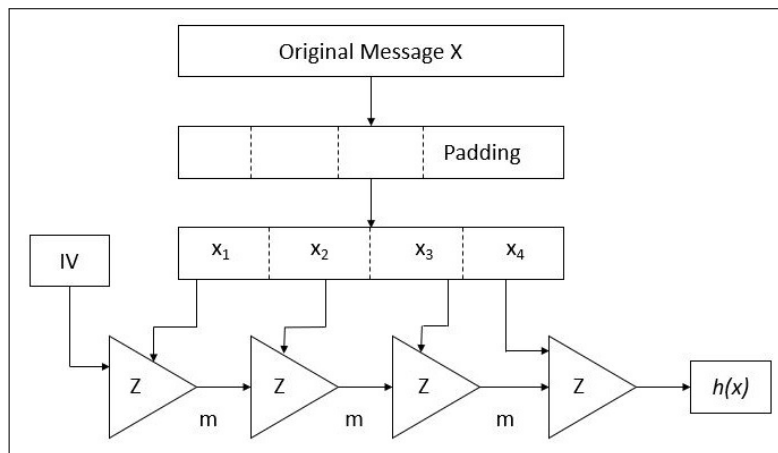


Figure 3.3: Merkle—Damgård construction



## 3.6 Description of SHA-1

### 3.6.1 SHA-1 overview

SHA-1 (Secure Hash Algorithm) is an iterated hash function and successor of MD-SHA family designed by NIST and first published in 1995. SHA-1 is minor variations of SHA-0 which was proposed by NIST in 1993. The only difference between SHA-0 and SHA-1 is the 1-bit rotations in the constructions of word from  $W_{16}$  to  $W_{79}$  [Stinson, 2005].

SHA-1 takes an arbitrary length of input less than  $2^{64}$  bits and produces a 160-bit message digest or hash value, typically hexadecimal digits. This message digest can be used as an input for signature algorithms for signing documents or verify the integrity of a message. It is always preferred to sign the message digest instead of original message (because the size of the input message is arbitrary, and length of the message digest is small) to improve the efficiency of the algorithm. The same signature algorithm can be used to verify the digital signature. Any changes to the message, with high probability the hash algorithm results a different message digest. SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest and difficult to find two different messages that results same message digest [FIPS, 1995].

### 3.6.2 SHA-1 Padding scheme

The purpose of padding scheme is to make the total length of message block 512-bits which will be fed to the compression function in blocks for computing the message digest. SHA-1 requires input message of length (typically less than or equal to)  $2^{64} - 1$  bits and binary representation of the original message, denoted by  $l$  of length 64 bits. If  $l$  is less than 64 bits, then it is padded on the left side with zeros so that its length  $l$  is exactly 64





$X \wedge Y$  = bitwise logical **AND** of X and Y

$X \vee Y$  = bitwise logical **OR** of X and Y

$X \oplus Y$  = bitwise logical **XOR** of X and Y

$\neg X$  = bitwise logical **complement** of X

- Integer modulo addition:  $X + Y$  = words X and Y represent integers modulo addition. Here, X and Y are integers where  $0 \leq X < 2^{32}, 0 \leq Y < 2^{32}$  and denoted as  $(X + Y) \bmod 2^{32}$ .

- Circular left shift:  $ROTL^s(X)$ , left shift X by s positions ( $0 \leq s < 31$ ).

- For 80 rounds computations, we define functions from  $f_0$  to  $f_{79}$  which operate on three 32-bit words B, C, D, and results one 32-bit word as output.

$$f_t(B, C, D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq t \leq 19 \\ B \oplus C \oplus D & \text{if } 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq t \leq 59 \\ B \oplus C \oplus D & \text{if } 60 \leq t \leq 79 \end{cases}$$

- Five-word buffers are used in process, these are initialized before processing the message blocks. They are defined as follows:

1. H0 = 67452301

2. H1 = EFCDAB89

3. H2 = 98BADCFE

4. H3 = 10325476

5. H4 = C3D2E1F0

- The first four values are taken from MD4 and MD5 and are represented in big-endian format [[Maibaum, 2015], [Rivest, 1990]].

- We define a sequence of word constants from  $K_0$  to  $K_{79}$  for 80 rounds computation, defined as follows:

$$K_t = \begin{cases} 5A827999 & \text{if } 0 \leq t \leq 19 \\ 6ED9EBA1 & \text{if } 20 \leq t \leq 39 \\ 8F1BBCDC & \text{if } 40 \leq t \leq 59 \\ CA62C1D6 & \text{if } 60 \leq t \leq 79 \end{cases}$$

These constants are represented in hex. It is believed that these round constants are square roots of  $\sqrt{2}$ ,  $\sqrt{3}$ ,  $\sqrt{5}$  and  $\sqrt{10}$  respectively [Maibaum, 2015].

### 3.6.4 SHA-1 Message Digest computation

The hash value or message digest is computed using the padded message described in section 3.6.2 and involves 80 steps in processing. The first 5-word buffers are denoted as A, B, C, D, E, and the 80-word sequence are denoted as  $W(t)$ , where  $0 \leq t \leq 79$ . From 3.6.2, the padded message is considered as sequence of  $n$  blocks containing  $16 * n$  words ( $M(0), M(1), \dots, M(n)$ ) where  $n > 0$  [FIPS, 1995]. The processing is described as follows:

- $M(i)$  is divided into 16 words ( $W(0)$  to  $W(15)$ ), where  $i$  runs from 1 to  $n$ .

$$M_i = W_0 || W_1 || W_2 || W_3 || \dots || W_{15}, W_i \text{ is a word [Stinson, 2005].}$$

- The words from 16 to 79 are circular left shifted.

$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$$

- The initialized 5-buffer words (H0 to H4) are assigned to second 5-word buffers A, B, C, D, E.

$$A = H_0, B = H_1, C = H_2, D = H_3 \text{ and } E = H_4$$

- These second 5-word buffers are fed to the 80-round computation ( $t = 0$  to  $79$ ) and we employ a temporary word buffer denoted as *temp*.

- After 80 rounds of computation, the values of second 5-word buffer are added to the first 5-word buffers.

$$\begin{aligned} \therefore H_0 \leftarrow H_0 + A, H_1 \leftarrow H_1 + B, H_2 \leftarrow H_2 + C, H_3 \leftarrow H_3 + D, H_4 \leftarrow H_4 + E \\ \text{80 rounds computation} = \left\{ \begin{array}{l} temp \leftarrow ROTL^5(A) + f_t(B, C, D) + E + W_t + K_t \\ E \leftarrow D \\ D \leftarrow C \\ C \leftarrow ROTL^{30}(B) \\ B \leftarrow A \\ A \leftarrow temp \end{array} \right. \end{aligned}$$

- In the last step, the final output of the first 5-buffers are concatenated. This is the final 160-bit message digest or hash value.

$$SHA-1(x) = H_0 \parallel H_1 \parallel H_2 \parallel H_3 \parallel H_4$$

Therefore, the message digest of the original message  $x$  is given below:

$$\therefore SHA-1(hello) = AA \ F4 \ C6 \ 1D \ DC \ C5 \ E8 \ A2 \ DA \ BE \ DE \ 0F \ 3B \ 48 \ 2C \ D9 \ AE \\ A9 \ 43 \ 4D$$

### 3.7 Applications of SHA-1

#### 3.7.1 Data Integrity

Data integrity is one the most important applications of cryptographic hash functions. It verifies the consistency and accuracy of the data stored. Let us suppose that  $h$  is hash function and  $x$  is the message. The hash function creates a fingerprint (message digest) of the data, denoted by  $y$ .

$$\therefore y = h(x) \text{ (unkeyed hash function)}$$

Assume that  $y$  is stored in some secure place. If  $x$  is altered to  $x'$  and if we assume that  $h(x) \neq h(x')$ , then the alteration of the data is identified by verifying whether  $y$  and  $y'$  are same or not, that is,  $y \neq y'$ , where  $y' = h(x')$ .

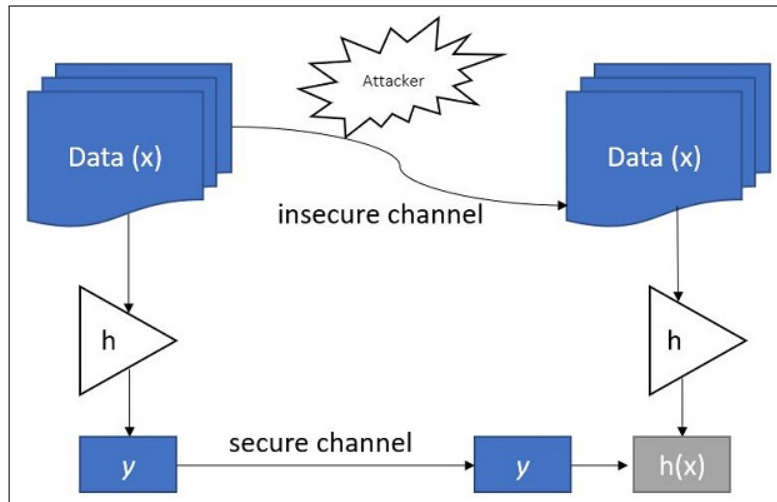


Figure 3.4: Data Integrity

From Figure 3.4, Assume that the data is transferred through an insecure channel and the message digest  $y$  of the data is passed through a secure channel. If the attacker tries to modify the data, the user computes the hash value and checks whether it is equal to

actual hash value. This verification can be done based on the assurance that the message digest is not changed.

### 3.7.2 Digital Signature

From 3.7.1, if the sender transfers the data confidently through an insecure channel to the receiver, we can assume that both the end users shared a secret key  $k$ . Therefore, instead of  $h(x)$  we assume,  $h_k(x)$ .

$$\therefore y = h_k(x) \text{ (Keyed hash function)}$$

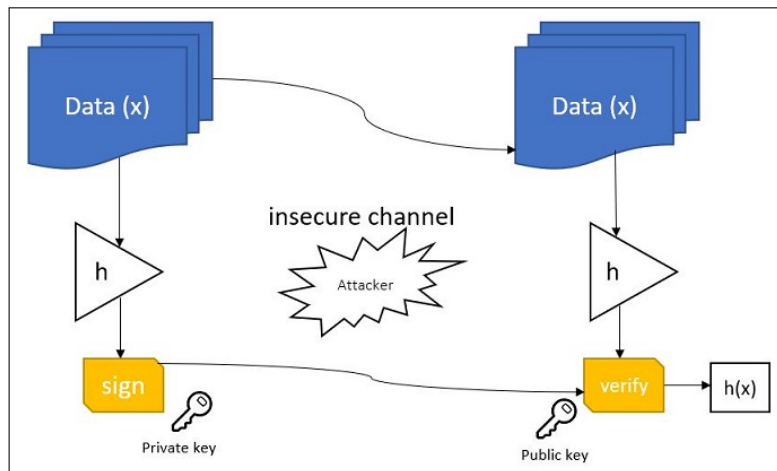


Figure 3.5: Digital Signature

From Figure 3.5, it is assumed that the data is being transferred to receivers end through an insecure channel. First the data is passed to hash function  $h$ , at the same time, it is signed using senders private-key and then it is transferred through the insecure channel. Now, the attacker cannot compute the signature because they do not have private-key. To verify the data, the receiver on other end of the system first applies hash function to the data received and then verify by decrypting using the public-key. If the hash values match, then the data is unaltered, otherwise the we can assume that the attacker altered the data.



## Chapter 4

### HUFFMAN COMPRESSION ALGORITHM

#### 4.1 File compression

A text file is usually stored as bits in our system by converting each character to 8-bit ASCII code. The ASCII encoding is called as fixed-length encoding, where each character is converted to a fixed number (same number) of bits. If each character uses 8-bits, then the total number of bits required is 8 times the number of characters. But the main idea is to reduce the space the required to store the text file [Mackenzie, 1980]. Hence, we need a compression function which results same output with less number of bits. This achieved by variable-length encoding, which uses binary codes of different characters with different lengths. If a character occurs more frequently, we represent it with fewer bits and more number of bits to characters with low frequency.

Let us consider an example, say  $x$  is a text we want to encode.

$$\therefore x = bala$$

Here, we have only three characters  $a, b, l$  for encoding. if we do a fixed-length encoding, we need at least two bits for each character. Therefore, we need 8-bits to represent. Suppose, if we decide to use the following bits for  $a, b$ , and  $l$

$$a = 0, b = 11, l = 10$$

Then  $x$  is encoded as 110100. It takes only 6 bits which is less when compared to fixed-length encoding. The reason we choose these values for  $a, b, l$  is because of the ambiguity faced by variable length encoding while decoding the text. In any given case, we want to decode a text uniquely [Hopcroft and Ullman, 1983].

Suppose that, we want to decode 110100, we get our original text  $x$  as  $bala$ . Assume

that,  $a = 0$ ,  $b = 01$ ,  $l = 00$ . Still the encoding text should result 6 bits only and the encoding text is 010000. Now, there is an ambiguity for decoding the encoded text. With 01 it could either be  $a$  or  $b$ . But it must be  $b$ , because we are using 01. The remaining zeros (0000) could be  $bala$ ,  $bll$  or  $baaaa$ . Therefore, proper care should be taken while working with variable length coding.

To prevent the ambiguities while decoding, we need to make sure that encoding satisfies the prefix rule, that is, no code should be prefix of the other.

$\therefore a = 0, b = 11, l = 10$  satisfies the prefix rule

The code or bits which satisfy this rule are represented using tree structure. The characters are stored at the external nodes, 0 is assigned to left child and 1 is assigned to right child [Mackenzie, 1980].

## 4.2 Optimal compression

We need to ensure that the encoded text is short and requires less number of bits.

for example, let us consider a text message:

$x = \text{malayalam}$

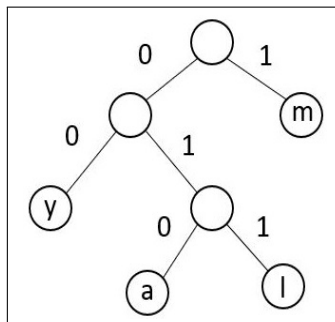


Figure 4.1: Encoded Tree structure

From Figure 4.1 and table 4.1, the total number of bits required is 22 and the text *malayalam* is represented as following:

Characters	Frequency	Code	Bits
a	4	010	3
l	2	011	3
m	2	1	1
Y	1	00	2

Table 4.1: Character frequency table for Encoded Tree Structure

$$\text{encodedText}(x) = 1010011010000100110101$$

Let us suppose that we have a different tree for the same word, see Figure 4.2. From table 4.2, the total number of bits required is only 19 and the word malayalam is represented as following:

$$\text{encodedText}(x) = 1000110001000011001$$

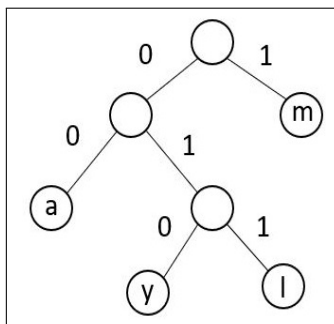


Figure 4.2: Encoded Tree structure – 2

Characters	Frequency	Code	Bits
a	4	00	2
l	2	011	3
m	2	1	1
Y	1	010	3

Table 4.2: Character frequency table for Encoded Tree Structure – 2

Hence, we need a compression function that results less number of bits.

### 4.3 Huffman Data compression

Huffman compression algorithm is an optimal compression or prefix algorithm where the frequencies of the letters are used for lossless compression of data. This method

uses a special technique for representing symbols for each word, resulting in bit strings representation [Huffman, 1952].

#### 4.3.1 Construction of algorithm

Suppose, for a given text, we need to count the frequency of characters and compute a tree so that the length of the encoding text is minimum, each character is a node in the tree. The root is always zero and level numbers are represented using number of bits to encode a character. If  $f$  is the frequency, then  $f_k$  is the frequency of the  $k^{th}$  character. Here,  $l$  is the level and  $l_k$  is the level of the node of  $k^{th}$  character. Therefore, we need to find a tree which minimizes  $\sum_k f_k l_k$  which is known as **the total external weighted path length of a tree**.

We consider each node having weight equal to the frequency of the characters. If there are  $n$  number of weights, the frequencies are represented as  $f_1, f_2, f_3, \dots, f_n$ . For these frequencies, we can build a tree whose external weighted path length is minimum, it is denoted by  $WEPL(f_1, f_2, f_3, \dots, f_n)$  [Hopcroft and Ullman, 1983].

#### 4.3.2 Huffman Tree

Let us suppose that, an input  $x$  is to be compressed. Huffman algorithm calculates the weights of the tuples in the input. Once this step is achieved, it begins to sum the least weights of the order at each level. Once the last node or symbol weight is calculated, it forms a tree and considers the sum of weights at each level as nodes. This tree is assigned with a bit strings for presentation, left-side of the tree with 0s and right-side with 1s.

$x = \text{mississippi}$

The frequencies of  $x$  are calculated as follows:

$m = 1, p = 2, i = 4, s = 4$

Now, the least weights are calculated as follows:

$$m + p = (1+2) = 3, \text{ at level 1. See Figure 4.3.}$$

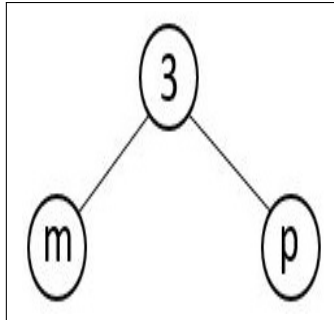


Figure 4.3: Huffman level 1

Again, it compares the weights with next character in queue and calculates the sum as follows:

$$(\text{level 1} + i) = (3 + 4) = 7, \text{ at level 2}$$

$$(\text{level 2} + s) = (7 + 4) = 11, \text{ at level 3}$$

Bit strings of 0s and 1s are assigned as follows:

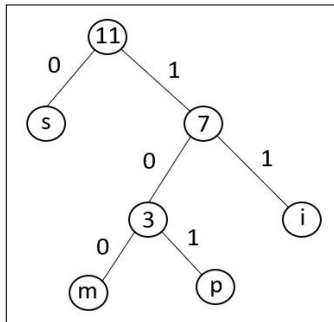


Figure 4.4: Huffman Tree

The number of bits required by the word *mississippi* is calculated from table 4.3.

The total number of bits required is 21.

$$\therefore \text{Huffman}(\text{mississippi}) = 100110011001110110111$$

Using the fixed-variable length encoding scheme, the word *mississippi* requires 88

Characters	Frequency	Code	Bits
m	1	100	3
p	2	101	3
i	4	11	2
s	4	0	1

Table 4.3: Character frequency table of Huffman Tree

bits. From Table 4.3, the Huffman codes require only 21 bits. Hence, it is reduced from 88 bits to 21 bits.

## Chapter 5

### NEW SEED CONSTRUCTION

We assume this title is apt as we are going to explain our approach in constructing message block for the preprocessing stage.

#### 5.1 Base

Before proceeding with the research work, we understood the basic definitions and requirements of a cryptosystem to produce a cryptographic hash function.

##### 5.1.1 SHA-1 as base

After the practical freestart collision in 2015 [Stevens, Karpman, and Peyrin, Stevens et al.], we decided to work towards find a message that results same message digest and prove the preprocessing stage of SHA-1. Many brute force words were used to test the SHA-1 collisions. While the research was on the go, SHA-1 was broken by Dr. Marc Stevens in collaboration with Google [Stevens et al., 2017]. We had to reiterate of study and proceeded for constructing a seed (message block) which results a different message digest when tested with the sample message that produced a collision in SHA-1.

#### 5.2 New Approach

##### 5.2.1 Why Huffman compressed code

It is clearly visible that, most of the cryptanalytic attacks either theoretical or practical on [[Rivest, 1992], [Stevens, Karpman, and Peyrin, Stevens et al.], [Stevens et al., 2017]] followed chosen-prefix collision attack to break the compression algorithm. In [Stevens et al., 2017] describes that the researchers has followed Identical-prefix collision attack, that is, a prefix is extended with a pair of messages which has close calls to collide for any suffix. With  $2^{63.1}$  SHA-1 calls the messages collided and resulted same hash value.

Understanding this theory, we first introduced the concept of *salt*, it is a random data used as an additional input for defending dictionary attacks or any pre-computed rainbow attacks [Alexander, 2012]. But we realized that salt cannot protect the data against an attacker who is after only one data file. Furthermore, with a correctly chosen salt, the adversary can attack a huge amount of data [Gillies, 2012]. Observing [Stevens et al., 2017] we decided not to proceed with Salt technique and introduced Huffman compressed codes [see section 4].

With the idea of chosen-prefix attack, we assumed that introducing the bit strings (0s and 1s) generated by Huffman compression algorithm would be ideal and it will be harder for the attacker to break the algorithm.

### 5.3 New Seed Construction

In this section, we explain how the 512-bit message block for preprocessing stage is constructed. As mentioned in section 5.1, we reused SHA-1 algorithm. For constructing a message block  $y$  in SHA-1 [section 3.6.2], the length is 512-bits or multiples of 512 and we pad the original message with 1 extra bit, 447 0s and length of the original message (64 bits).

$$y = \text{message} \parallel 1 \parallel 0^{447} \parallel 64$$

In our new seed construction, we removed 1 extra bit and replaced 447 0s with 288 0s and 160-bit message digest of the Huffman codes.

$$y_{seed} = \text{message} \parallel \text{SHA-1}(\text{Huffman}) \parallel 0^{288} \parallel 64$$

If we observe  $y_{seed}$ , the total length of the message block is 512 bits or multiples of 512 bits.



### 5.3.1 Steps involved in construction

- For any given message  $x$ , we calculate the frequencies of the characters and generate the Huffman compressed codes, see table 5.1 and Figure 5.1

$x = \text{abracadabra}$

$\text{Huffman}(x) = 0\ 110\ 111\ 0\ 100\ 0\ 101\ 0\ 110\ 111\ 0$

Characters	Frequency	Code	Bits
a	5	0	1
c	1	100	3
d	1	101	3
b	2	110	3
r	2	111	3

Table 5.1: Character frequency table of Huffman Tree – seed construction

- Now, we hash the Huffman compressed codes, which results a 160-bit message digest.

$\text{SHA-1}(\text{Huffman}(x)) = 5A\ 08\ 07\ 94\ A9\ D6\ 40\ 62\ 94\ 1E\ F2\ 59\ B8\ F3\ C7\ 79\ 52\ 9E$

17 61

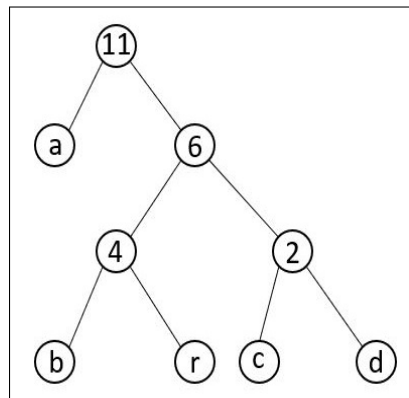


Figure 5.1: Huffman Tree – seed construction

- We feed the message block with original message  $x$ , SHA-1 of Huffman compressed codes. The padding scheme is the same as SHA-1.

1.  $x = \text{abracadabra}$
2.  $\text{SHA-1}(\text{Huffman}(x)) = 5A\ 08\ 07\ 94\ A9\ D6\ 40\ 62\ 94\ 1E\ F2\ 59\ B8\ F3\ C7\ 79\ 52\ 9E\ 17\ 61$
3.  $d = (288 - 11) \bmod 512 \sim 277\ 0\text{'s}$
4. length  $l = 64$
5.  $y_{seed} = x \parallel \text{SHA-1}(\text{Huffman}(x)) \parallel 0^{277} \parallel 64$  (Preprocessing Stage)

- The total length of the message block or seed  $y_{seed} = 512$ -bits.
- Now, we feed this seed to SHA-1 compression function, which results a 160-bit message digest.

- Our observation

$$h(x)_{seed} = B0\ 07\ 94\ C1\ 80\ BA\ E5\ DB\ 44\ 8F\ 82\ CC\ DE\ 6C\ CA\ 76\ A4\ D4\ 2E\ 93$$

Universally, if we consider any platform and compute the SHA-1 of *abracadabra*, it produces the following as its message digest:

$$h(x) = 0B\ 8C\ 31\ DD\ 3A\ 4C\ 1E\ 74\ B0\ 76\ 4D\ 5B\ 51\ 0F\ D5\ EA\ AC\ 00\ 42\ 6C$$

But, our seed construction resulted a different message digest for the same input, see Figures 5.2 and 17.

Our observations for different words are given in [ section 5.4]

## 5.4 Observations

We can categorize the above observation into two cases:

1. Case 1: missisppi, nississippi and massassappa have characters with same frequency and share same Huffman codes, SHA-1 message digest. But, when these outputs are



```
C:\Users\Lakshmi Devulapalli\source\repos\Compression\Debug\Compression.exe
Huffman Codes are :
a 0
c 100
d 101
r 111
b 110

Original string was :
abracadabra

Encoded string is :
01101110100010101101110

SHA-1 of Huffman Encoded String :
5A 08 07 94 A9 D6 40 62 94 1E F2 59 B8 F3 C7 79 52 9E 17 61
```

Figure 5.2: Compression function output



```
C:\Users\Lakshmi Devulapalli\source\repos\SHA1\Debug\SHA1.exe
Our Observation - B0 07 94 C1 80 BA E5 DB 44 8F 82 CC DE 6C CA 76 A4 D4 2E 93
```

Figure 5.3: New Seed construction output – our observation

fed to our seed, it resulted a message digest which is different from respective SHA-1 of these words.

2. Case 2: our aim was to prove that, two different messages with same message digest when fed to our seed should produce different message digests. For this, we considered the sample message created by Dr. Marc Stevens team in collaboration with Google see Figures 5.4 and 5.5 (We took screenshot of these observation and updated them as figures due to space management). The messages look similar, but they are different and share same message digest. When fed to our seed, it successfully resulted two different message digests

Word	mississippi
SHA-1 of Word	FE 64 8F C4 59 A6 F6 EF 6C D3 47 BE E3 D4 94 76 62 39 BB B5
Huffman(Word)	100011110111101011010
SHA-1(Huffman)	45 30 A9 14 B9 B9 37 73 59 89 A3 27 71 A4 92 33 0E BB DB 7F
Our Observation	27 D1 83 19 08 15 0E E8 AF D8 84 55 7C 8F 0B 1D E9 96 43 8C

Table 5.2: Observation Table 1

Word	nississippi
SHA-1 of Word	E9 52 9D 02 B8 13 A9 26 1B 8B CE B0 6D 4F 4B 94 DC 63 44 99
Huffman(Word)	100011110111101011010
SHA-1(Huffman)	45 30 A9 14 B9 B9 37 73 59 89 A3 27 71 A4 92 33 0E BB DB 7F
Our Observation	CB F5 B7 9C 15 6D C1 C0 1F A6 8F 92 0F 01 C2 E4 2D A8 D1 DE

Table 5.3: Observation Table 2

Word	massassappa
SHA-1 of Word	C7 C9 B7 3C 2E 36 E0 52 8E 48 19 2C 1A 14 94 FE F3 D7 3A 7B
Huffman(Word)	100011110111101011010
SHA-1(Huffman)	45 30 A9 14 B9 B9 37 73 59 89 A3 27 71 A4 92 33 0E BB DB 7F
Our Observation	B5 F2 79 7E 73 F1 8D C4 7D E1 03 0F 39 56 67 9D A8 AD BE 5C

Table 5.4: Observation Table 3

Word	7f46dc93a6b67e013b029aaa1db2560b45ca67d688c7f84b8c4c791fe02b3df6 14f86db1690901c56b45c1530afedfb76038e972722fe7ad728f0e4904e046c2 30570fe9d41398abe12ef5bc942be33542a4802d98b5d70f2a332ec37fac3514 e74ddc0f2cc1a874cd0c78305a21566461309789606bd0bf3f98cda8044629a1
SHA-1 of Word	38 76 2C F7 F5 59 34 B3 4D 17 9A E6 A4 C8 0C AD CC BB 7F 0A
Huffman(Word)	1011100111101101001110100110100001001101010111011011000111110010 1000010111111100011001000100010000100011010111000000110111110101 1110000010100100110110110011110101110111101010111001011111100101 0111101011101010101101100010100100011111110001011000001110011101 0010111010010111110100110101001011010110111101101111001010100000 1101010111100000101000100000100011110100100100010011100101011011 1101111110000111000101101011110010111100110010010001101101000011 10111100011110011111000111100110111111000011111110110110101100 1000111100001011111110010001011000111110001010000110011101000101 0001001011000001100100000101101001101110110001010001100010000000 1110110001001110011111111000011011001110101000000111011111110011 10001001000100011000001101010001011100101001010100000000010111000 01101111100011001110101111100111001010101000100100011110111110101 00011111110101011011110001111000001001100001000001101110111101101 00101000111101101011011101101101111111010101001111110101100110001 0010110011110100011010001111111110111011011100011001000010
SHA-1 (Huffman)	99 FA CF 6D F3 89 CE 96 8B A0 2E A5 8A 14 E6 63 08 26 87 AD
Our observation	A6 E0 B8 76 98 4E C9 DE 53 63 E9 69 65 99 8E E7 86 B1 EC 7D

Figure 5.4: Google’s sample message 1

Word	7346dc9166b67e118f029ab621b2560ff9ca67cca8c7f85ba84c79030c2b3de2 18f86db3a90901d5df45c14f26fedfb3dc38e96ac22fe7bd728f0e45bce046d2 3c570feb141398bb552ef5a0a82be331fea48037b8b5d71f0e332edf93ac3500 eb4ddc0decc1a864790c782c76215660dd309791d06bd0af3f98cda4bc4629b1
SHA-1 of Word	38 76 2C F7 F5 59 34 B3 4D 17 9A E6 A4 C8 0C AD CC BB 7F 0A
Huffman(Word)	001110100000100111001111010001011001100111011001001101100101010101 111011111010000100001011011001100001011101100000011001111010111011 0100111100101001001111111110010011111100111011011100011101001001 11000011110011010011101010111011110001101101011000110100001010111 101101111001110011011010001001001110010011100101110000011100101100 000001111101010000101110001001101101101100101111011010110011111010 011101100100100100101111100010001011011000111101110000111000011110 111110011000000001110111110110111000001001110010001010111100010011 111010110110110101010000010110100100011111011101000100011000011010 110001001011100010011110001101011010101010010110110110001000000111 111010100011110101111101000111000011010110111110011010101010100001 101100101101001010001011111010000111101110011011010000110011001111 111011000110111111110101001001111001000000110100111011110011011110 001111001110011000010100011001100111101100110010101110010000110100 0101110011101001110111001110001010111010101101000
SHA-1 (Huffman)	8F 58 20 38 C4 0B DC B8 DC 0A B0 30 66 BC 0B 50 57 58 82 9B
Our observation	71 20 47 7F 7D 55 83 8D FF 01 C8 30 01 22 AC 19 52 CD 2C 51

Figure 5.5: Google’s sample message 2

## Chapter 6

### CONCLUSION

There are always chances for cryptanalytic attacks due to over growing technological revolution and can compromise the security of the systems. This study aims to design a hash algorithm by adding an extra layer, that is, by adding the hash value of Huffman compressed codes of the original message to the preprocessing stage of SHA-1. We tested our algorithm for different bruteforce words and Google's sample messages which broke SHA-1. The results show that, any message with same original SHA-1 *message digest* and messages with same *Huffman compressed codes* did not collide and resulted different message digests. SHA-1 algorithm depends on the length of the original message, whereas, Our algorithm depends the frequency of the characters of original message as well as the length of the original message. Hence, at this point of time we conclude that our algorithm is collision resistant.

## REFERENCES

- Alexander, S. (2012, June). The bug charmer: Passwords matter. <http://bugcharmer.blogspot.com/2012/06/passwords-matter.html>.
- Barker, E. B. (1993). Secure hash standard (shs). Technical report.
- Boesche, R. (2002). *The first great political realist: Kautilya and his Arthashastra*. Lexington Books.
- Brooks, R. R. (2013). Introduction to computer and network security: Navigating shades of gray.
- Buchmann, J. (2013). *Introduction to cryptography*. Springer Science & Business Media.
- Damgård, I. B. (1989). A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*, pp. 416–427. Springer.
- David, K. (1999). Crises of the union. *The Codebreakers: The Story of Secret Writing*, 217–221.
- Diffie, W. and M. E. Hellman (1976). Multiuser cryptographic techniques. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pp. 109–112. ACM.
- Eastlake 3rd, D. and P. Jones (2001). Us secure hash algorithm 1 (sha1). Technical report.
- FIPS, P. (1995). 180-1. secure hash standard. *National Institute of Standards and Technology 17*, 45.
- Gillies (2012, April). passwords - why is using salt more secure? - information security stack exchange. <https://security.stackexchange.com/questions/14025/why-is-using-salt-more-secure>.
- Goodin, D. (2017, February). At death's door for years, widely used sha1 function is now dead. <https://arstechnica.com/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/>. (Accessed on 03/20/2017).
- Hat, R. (2013, August). A brief history of cryptography - red hat customer portal. <https://access.redhat.com/blogs/766093/posts/1976023>. (Accessed on 02/05/2017).



- Hopcroft, J. E. and J. D. Ullman (1983). *Data structures and algorithms*.
- Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40(9), 1098–1101.
- Joux, A. and T. Peyrin (2007). Hash functions and the (amplified) boomerang attack. In *Annual International Cryptology Conference*, pp. 244–263. Springer.
- Kaliski, B. (1992). The md2 message-digest algorithm.
- Karpman, P., T. Peyrin, and M. Stevens (2015). Practical free-start collision attacks on 76-step sha-1. In *Annual Cryptology Conference*, pp. 623–642. Springer.
- Kessler, G. C. (1998). An overview of cryptography. *published by Auerbach* 22.
- Leeuwen, J. (1990). *Handbook of theoretical computer science*, Volume 1. Elsevier.
- Mackenzie, C. E. (1980). *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc.
- Maibaum, J. (2015, June). sha 1 - why initialize sha1 with specific buffer? - cryptography stack exchange. <https://crypto.stackexchange.com/questions/10829/why-initialize-sha1-with-specific-buffer/10857>.
- Mao, W. (2003). *Modern cryptography: theory and practice*. Prentice Hall Professional Technical Reference.
- Menezes, A. J., P. C. Van Oorschot, and S. A. Vanstone (1996). *Handbook of applied cryptography*. CRC press.
- Merkle, R. C. (1989). One way hash functions and des. In *Conference on the Theory and Application of Cryptology*, pp. 428–446. Springer.
- Merkle, R. C., R. Charles, et al. (1979). Secrecy, authentication, and public key systems. networks, X. Q99: What are md2, md4, and md5? <http://x5.net/faqs/crypto/q99.html>. (Accessed on 02/02/2018).
- Paar, C. and J. Pelzl (2009). *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media.
- Ramsinghani, M. (2016, feb). Innovations in cybersecurity at rsa 2016 [techcrunch. https://techcrunch.com/2016/02/29/](https://techcrunch.com/2016/02/29/)

innovations-in-cybersecurity-at-rsa-2016/. (Accessed on 02/05/2018).

Rivest, R. (1992). The md5 message-digest algorithm.

Rivest, R. L. (1990). Md4 message digest algorithm.

Rogaway, P. and T. Shrimpton (2004). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International Workshop on Fast Software Encryption*, pp. 371–388. Springer.

Schneier, B. Foundations. *Applied Cryptography, Second Edition, 20th Anniversary Edition*, 1–18.

Schneier, B. (2005, February). Cryptanalysis of sha-1 - schneier on security. [https://www.schneier.com/blog/archives/2005/02/cryptanalysis\\_o.html](https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html). (Accessed on 02/04/2017).

Simpson, S. (1997). Cryptography defined/brief history. <http://www.laits.utexas.edu/~anorman/BUS.FOR/course.mat/SSim/history.html>. (Accessed on 02/02/2018).

Stevens, M. (2013). New collision attacks on sha-1 based on optimal joint local-collision analysis. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 245–261. Springer.

Stevens, M., E. Bursztein, P. Karpman, A. Albertini, and Y. Markov (2017). The first collision for full sha-1. *IACR Cryptology ePrint Archive 2017*, 190.

Stevens, M., P. Karpman, and T. Peyrin. Freestart collision on full sha-1.

Stinson, D. R. (2005). *Cryptography: theory and practice*. CRC press.

tbs internet (2018, february). Sha1: Depreciation of sha1 algorithm scheduled for 2015, 2016, 2017? [https://www.tbs-certificates.co.uk/FAQ/en/microsoft\\_depreciation\\_sha1.html](https://www.tbs-certificates.co.uk/FAQ/en/microsoft_depreciation_sha1.html). (Accessed on 02/03/2018).

Thomsen, S. S. and L. R. Knudsen (2005). *Cryptographic hash functions*. Ph. D. thesis, Technical University of Denmark Danmarks Tekniske Universitet, Department of Applied Mathematics and Computer Science Institut for Matematik og Computer Science.

Wang, X., Y. L. Yin, and H. Yu (2005). Finding collisions in the full sha-1. In *Annual international cryptology conference*, pp. 17–36. Springer.

Wang, X. and H. Yu (2005). How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pp. 19–35. Springer.

# Appendices

## Appendix A

### APPENDIX A: SETTING UP VISUAL STUDIO

#### A.1 Installation guide for Visual Studio Community 2017

The following steps are involved in Installing Visual Studio.

1. Go to <https://www.visualstudio.com/downloads/>
2. Once you're on the Microsoft website, click the Blue button to download the Visual Studio Community 2017, see figure A.1.

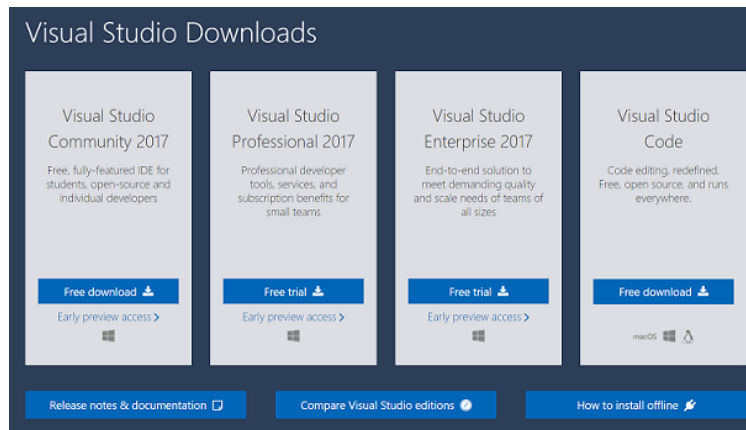


Figure A.1: Visual Studio downloads page

3. Once the installer has downloaded, double-click it to install Visual Studio Community 2017.
4. Accept the policy and continue to run the installer, see figure A.2.

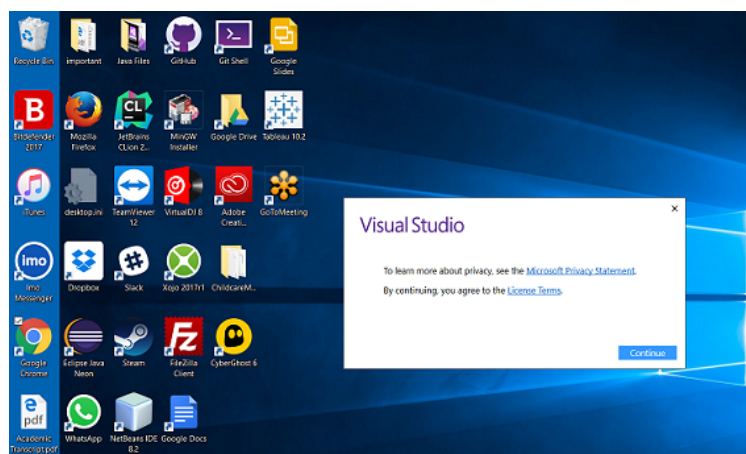


Figure A.2: Installation policy

5. Select the Packages to be installed on your system, see figure A.3.

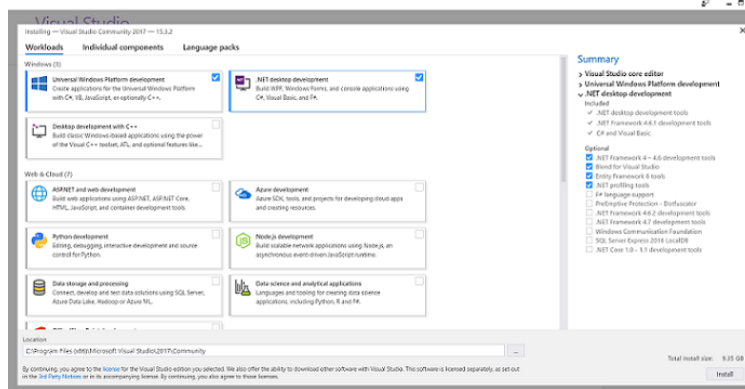


Figure A.3: Installation packages

6. Once the Installation is done, Run the Visual Studio Community 2017.

## Appendix B

### APPENDIX B: SHA-1 OF HUFFMAN COMPRESSED CODES

```
1 #pragma once
2 /*
3  * Header file for sha code
4  * reference: US Secure Hash Algorithm 1 by Network working group
5  */
6
7 /* This Header will decalre set of integer types having specified
8  * widths. Width of an integer type is the numnber of bits used to
9  * store its value in a pure binary system.
10 */
11 * The actual type may use more bits that that. for example, a 28-bit
12 * type could store in 32-bit of actual storage.
13 */
14 #include <stdint.h>
15
16 /*
17 * enum is a user-defined data type that consists of integral constants.
18 */
19 enum
20 {
21     shaOk = 0,
22     shaNull, /* Null pointer parameter */
23     shaInputLong, /* long input data */
24     shaError /* called after result */
25 };
26
27 #define Sha1HashSize 20
28
29 /*
30 * This structure will hold the context information for the SHA-1 hash
31 * operation typedef is a reserved keyword. By defining the typedef,
32 * it is assured that all the variables are structure pointer types,
33 * or each variable is a pointer type pointing to a structure type.
34 */
35 /*
36 * uint32_t ----> unsigned 32 bit integer
37 * uint8_t ----> unsigned 8 bit integer
38 */
39 typedef struct SHA1Context
40 {
41     uint32_t Intermediate_Hash[Sha1HashSize / 4]; /* Message Digest */
42
43     uint32_t Length_Low; /* Message length in bits */
44     uint32_t Length_High; /* Message length in bits */
45     int_least16_t Message_Block_Index; /* Index into message block array */
46     uint8_t Message_Block[64]; /* 512-bit message blocks */
47
48     int Computed; /* Is the digest computed? */
49     int Corrupted; /* Is the message digest corrupted? */
50 } SHA1Context; /*is the message digest corrupted ? */
51
52 /*
53  * Function Prototypes
54  */
55 int SHA1Reset(SHA1Context *);
56 int SHA1Input(SHA1Context *, const uint8_t *, unsigned int);
57 int SHA1Result(SHA1Context *,
58     uint8_t Message_Digest[Sha1HashSize]);
```

Figure B.1: *SHA1-Header.h*.

```

1/*
2* Description: this file implements the SHA1
3* SHA1 produces 1 160-bit message digest for a given input
4* SHA1 is defined in terms of 32-bit words. This code uses Header.h file
5* to define 32-bit and 8-bit unsigned integer types.
6* This code only works with message with length that is a multiple of the size of an
7* 8-bit character
8*/
9#include "stdafx.h"
10#include "SHA_1_Header.h"
11
12/*
13* Define the SHA1 circular left shift
14* The circular left shift operation S^n(X), where X is a word and n is an integer
15* with 0 <= n < 32
16* X << n is obtained as follows - discard the left-most n bits of X and then pad the
17* result with n zeros on the right( the result will still be 32-bits).
18* X >> 32- n is obtained by discarding the right-most n bits of X and then padding
19* the result with n zeros on the left. Thus S^n(X) is equivalent to a circular shift
20* of X by n positions to the left.
21*/
22#define SHA1CircularShift(bits ,word) \
23    (((word) << (bits)) | ((word) >> (32-(bits))))
24
25/* Local Function Prototypes */
26void SHA1PadMessage(SHA1Context *);
27void SHA1ProcessMessageBlock(SHA1Context *);
28
29/*
30* SHA1 Reset
31* This function will initialize the SHA1Context in preparation for computing
32* a new SHA1 message digest.
33* It initializes Length_Low, Length_High and Message_block_index to zero
34* Returns SHA1 Error code
35*/
36int SHA1Reset(SHA1Context *context) {
37    if (!context) {
38        return shaNull;
39    }
40    context->Length_Low = 0;
41    context->Length_High = 0;
42    context->Message_Block_Index = 0;
43
44    context->Intermediate_Hash[0] = 0x67452301;
45    context->Intermediate_Hash[1] = 0xEFCDAB89;
46    context->Intermediate_Hash[2] = 0x98BADCFE;
47    context->Intermediate_Hash[3] = 0x10325476;
48    context->Intermediate_Hash[4] = 0xC3D2E1F0;
49
50    context->Computed = 0;
51    context->Corrupted = 0;
52
53    return shaOk;
54}
55
56/*
57* SHA1Result
58*
59* Description:
60*     This function will return the 160-bit message digest into the
61*     Message_Digest array provided by the caller.
62*     NOTE: The first octet of hash is stored in the 0th element,
63*     the last octet of hash in the 19th element.
64*
65* Parameters:
66*     context: [in/out]

```

Figure B.2: *SHA-1 Source.cpp*.



```

1* The context to use to calculate the SHA-1 hash.
2* Message_Digest: [out]
3*     Where the digest is returned.
4*
5* Returns:
6*     sha Error Code.
7*
8*/
9 int SHA1Result(SHA1Context *context ,
10 uint8_t Message_Digest[ Sha1HashSize ])
11 {
12     int i;
13
14     if (!context || !Message_Digest)
15     {
16         return shaNull;
17     }
18
19     if (context->Corrupted)
20     {
21         return context->Corrupted;
22     }
23
24     if (!context->Computed)
25     {
26         SHA1PadMessage(context);
27         for (i = 0; i < 64; ++i)
28         {
29             /* message may be sensitive, clear it out */
30             context->Message_Block[i] = 0;
31         }
32         context->Length_Low = 0;     /* and clear length */
33         context->Length_High = 0;
34         context->Computed = 1;
35     }
36
37     for (i = 0; i < Sha1HashSize; ++i)
38     {
39         Message_Digest[i] = context->Intermediate_Hash[i >> 2]
40             >> 8 * (3 - (i & 0x03));
41     }
42
43     return shaOk;
44 }
45
46
47 /*
48 * SHA1 Input
49 * Description: This function accepts an array of octects as the next portion of the
50 * message
51 * message_array is a parameter : An array of Characters representing the next
52 * portion of the message
53 * length: length of the message in message_array
54 * Returns SHA Error code
55 * If the number of bits in a message is a multiple of 8, for compactness we can represent
56 * the message in hex.
57 * The padded message will contain 16 * n words for some n > 0. The padded message is regarded
58 * as a sequence of n blocks M(1) , M(2), first characters (or bits) of the message.
59 */
60 int SHA1Input(SHA1Context *context , const uint8_t *message_array , unsigned length) {
61     if (!length) {           //condition for length
62         return shaOk;
63     }
64     if (!context || !message_array) { //condition for no message array
65         return shaNull;
66     }

```

Figure B.3: *SHA-1 Source.cpp continuation-1.*

```

1 if (context->Computed) {
2     context->Corrupted = shaError; // condition to check corrupted bits
3     return shaError;
4 }
5
6 if (context->Corrupted) {
7     return context->Corrupted; // condition to check corrupted bits
8 }
9 while (length — && !context->Corrupted)
10 {
11     context->Message_Block[context->Message_Block_Index++] = (*message_array & 0xFF);
12     context->Length_Low += 8;
13     if (context->Length_Low == 0) {
14         context->Length_High++;
15         if (context->Length_High == 0) {
16             /* Message is too long*/
17             context->Corrupted = 1;
18         }
19     }
20     if (context->Message_Block_Index == 64) {
21         SHA1ProcessMessageBlock(context);
22     }
23     message_array++;
24 }
25 return shaOk;
26 }
27
28 /*
29 * SHA1ProcessMessageBlock
30 * Description: This function will process the next 512 bits of the message stored
31 * in the Message_block array
32 * this function has no parameters and returns nothing
33 */
34 void SHA1ProcessMessageBlock(SHA1Context *context) {
35     const uint32_t K[] = { /* Constants defined in SHA-1 */
36         0x5A827999,
37         0x6ED9EBA1,
38         0x8F1BBCDC,
39         0xCA62C1D6
40     };
41
42     int         t; /* Loop counter */
43     uint32_t    temp; /* Temporary word value */
44     uint32_t    W[80]; /* Word sequence */
45     uint32_t    A, B, C, D, E; /* Word buffers */
46
47     /*
48      * Initialize the first 16 words in the array W
49      * | is a bitwise or, example x |= 8 means x = x | 8
50      */
51     for (t = 0; t < 16; t++)
52     {
53         W[t] = context->Message_Block[t * 4] << 24;
54         W[t] |= context->Message_Block[t * 4 + 1] << 16;
55         W[t] |= context->Message_Block[t * 4 + 2] << 8;
56         W[t] |= context->Message_Block[t * 4 + 3];
57     }
58
59     for (t = 16; t < 80; t++)
60     {
61         W[t] = SHA1CircularShift(1, W[t - 3] ^ W[t - 8] ^ W[t - 14] ^ W[t - 16]);
62     }
63
64     A = context->Intermediate_Hash[0];
65     B = context->Intermediate_Hash[1];

```

Figure B.4: *SHA-1 Source.cpp continuation-2.*

```

1
2 C = context->Intermediate_Hash[2];
3 D = context->Intermediate_Hash[3];
4 E = context->Intermediate_Hash[4];
5
6 for (t = 0; t < 20; t++)
7 {
8     temp = SHA1CircularShift(5, A) +
9         ((B & C) | ((~B) & D)) + E + W[t] + K[0];
10    E = D;
11    D = C;
12    C = SHA1CircularShift(30, B);
13    B = A;
14    A = temp;
15 }
16
17 for (t = 20; t < 40; t++)
18 {
19     temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[1];
20     E = D;
21     D = C;
22     C = SHA1CircularShift(30, B);
23     B = A;
24     A = temp;
25 }
26
27 for (t = 40; t < 60; t++)
28 {
29     temp = SHA1CircularShift(5, A) +
30         ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
31     E = D;
32     D = C;
33     C = SHA1CircularShift(30, B);
34     B = A;
35     A = temp;
36 }
37
38 for (t = 60; t < 80; t++)
39 {
40     temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[3];
41     E = D;
42     D = C;
43     C = SHA1CircularShift(30, B);
44     B = A;
45     A = temp;
46 }
47
48 context->Intermediate_Hash[0] += A;
49 context->Intermediate_Hash[1] += B;
50 context->Intermediate_Hash[2] += C;
51 context->Intermediate_Hash[3] += D;
52 context->Intermediate_Hash[4] += E;
53
54 context->Message_Block_Index = 0;
55 }
56
57 /*
58 * SHA1PadMessage
59 * Description: According to SHA1 standard, the message must be padded to an even
60 * 512 bits. The first paddign bit must be a '1'. the last 64 bits represent the
61 * length of the original message. All bits in between should be 0.This function
62 * will pad the message according to those rules by filling the Message_Block array
63 * accordingly. It will also call the ProcessMessageBlock function provided appropriately.
64 * When it returns, it can be assumed that the message digest has been computed.
65 * Parameters are context and the ProcessMessageBlock function

```

Figure B.5: *SHA-1 Source.cpp continuation-3.*

```

1* returns nothing
2* If the number of bits in a message is a multiple of 8, for compactness we can represent
3* the message in hex
4*/
5 void SHA1PadMessage(SHA1Context *context) {
6  /*
7  * Check to see if the current message block is too small to hold
8  * the initial padding bits and length. If so, we will pad the
9  * block, process it, and then continue padding into a second
10 * block.
11 */
12 if (context->Message_Block_Index > 55)
13 {
14     context->Message_Block[context->Message_Block_Index++] = 0x80;
15     while (context->Message_Block_Index < 64)
16     {
17         context->Message_Block[context->Message_Block_Index++] = 0;
18     }
19     SHA1ProcessMessageBlock(context);
20
21     while (context->Message_Block_Index < 56)
22     {
23         context->Message_Block[context->Message_Block_Index++] = 0;
24     }
25 }
26 else
27 {
28     context->Message_Block[context->Message_Block_Index++] = 0x80;
29     while (context->Message_Block_Index < 56)
30     {
31         context->Message_Block[context->Message_Block_Index++] = 0;
32     }
33 }
34 }
35 /*
36 * Store the message length as the last 8 octets
37 */
38 context->Message_Block[56] = context->Length_High >> 24;
39 context->Message_Block[57] = context->Length_High >> 16;
40 context->Message_Block[58] = context->Length_High >> 8;
41 context->Message_Block[59] = context->Length_High;
42 context->Message_Block[60] = context->Length_Low >> 24;
43 context->Message_Block[61] = context->Length_Low >> 16;
44 context->Message_Block[62] = context->Length_Low >> 8;
45 context->Message_Block[63] = context->Length_Low;
46
47 SHA1ProcessMessageBlock(context);
48 }

```

Figure B.6: *SHA-1 Source.cpp continuation-4.*

```

1/*
2* Huffman Data Compression Algorithm
3* Reference of code and notes taken from Techie-Delight - Aditya Goel and
4* Dr.Naveen garg's (IIT-Delhi) lecture.
5*/
6
7#include <iostream>
8#include <fstream>
9#include <cstring>
10#include <chrono>
11#include <thread>
12#include <string>
13#include <queue>
14#include <unordered_map>
15using namespace std;
16
17/* Declaring Tree node */
18struct treeNode
19{
20    char ch;
21    int freq;
22    treeNode *left , *right;
23};
24
25/* getNode function to allocate a new tree node */
26treeNode* getNode(char ch, int freq, treeNode* left, treeNode* right)
27{
28    treeNode* node = new treeNode();
29
30    node->ch = ch;
31    node->freq = freq;
32    node->left = left;
33    node->right = right;
34
35    return node;
36}
37
38/* Comparing characters of left and right nodes of tree */
39struct comp
40{
41    bool operator()(treeNode* l, treeNode* r)
42    {
43        // Characters with lowest frequency
44        return l->freq > r->freq;
45    }
46};
47
48/* After Calculating the frequencies of the characters, weights are calculated and
49* values are assigned to left and right side of the huffman tree from root node.
50*/
51void encodeTree(treeNode* rootNode, string str,
52    unordered_map<char, string> &huffmanCode)
53{
54    if (rootNode == nullptr)
55        return;
56
57    // found a leaf node
58    if (!rootNode->left && !rootNode->right) {
59        huffmanCode[rootNode->ch] = str;
60    }
61    /* Assign 0' to leftside of the root node and 1's to rightside of the root node */
62    encodeTree(rootNode->left, str + "0", huffmanCode);
63    encodeTree(rootNode->right, str + "1", huffmanCode);
64}
65
66/* Frequencies of characters are calculated.
67* Based on weights they are separated */

```

Figure B.7: *Huffman.h*.

```

1 void buildHuffmanTree(string text)
2 {
3     /* Count frequency of appearance of each character
4     * and store it in a map
5     */
6     unordered_map<char, int> freq;
7     for (char ch : text) {
8         freq[ch]++;
9     }
10
11     priority_queue<treeNode*, vector<treeNode*>, comp> pq;
12
13     for (auto pair : freq) {
14         pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
15     }
16
17     while (pq.size() != 1)
18     {
19         treeNode *left = pq.top(); pq.pop();
20         treeNode *right = pq.top(); pq.pop();
21
22         int sum = left->freq + right->freq;
23         pq.push(getNode('\0', sum, left, right));
24     }
25
26     treeNode* rootNode = pq.top();
27
28     unordered_map<char, string> huffmanCode;
29     encodeTree(rootNode, "", huffmanCode); // after assigning 0's and 1's
30
31     cout << "Huffman Codes of each character :\n" << '\n';
32     for (auto pair : huffmanCode) {
33         cout << pair.first << " " << pair.second << '\n';
34     }
35
36     cout << "\nOriginal message string is :\n" << text << '\n';
37     string str = "";
38     for (char ch : text) {
39         str += huffmanCode[ch];
40     }
41
42     /*write the string into output text file*/
43     ofstream outfile;
44     outfile.open("Huffman_Codes.txt", ios::out);
45     outfile << str;
46     cout << "\nHuffman Compressed Codes of original string:\n" << str << '\n';
47 }
48 }

```

Figure B.8: *Huffman.h continuation – 1.*

```

1/**
2* Description: This program generates SHA-1 of Huffman compressed codes.
3*/
4
5#include "stdafx.h"
6#include <stdio.h>
7#include <string>
8#include "SHA_1_Header.h"
9#include "Huffman.h"
10#include <iostream>
11#include <fstream>
12#include <cstring>
13#include <chrono>
14#include <thread>
15
16using namespace std;
17using namespace std::this_thread;
18using namespace std::chrono;
19
20long int repeatcount[1] = { 1 };
21
22/* Main function */
23int main() {
24
25    /* Takes input from the text file */
26    string input, lines;
27    ifstream inputfile("Input.txt", ios::out);
28    while (getline(inputfile, lines)) {
29        input = lines;
30    }
31    const char* message = input.c_str();
32
33    /**
34     * Input is passed to the buildHuffmanTree method to generate
35     * Huffman Tree and produces Huffman compressed codes
36     */
37    buildHuffmanTree(message);
38
39    /**
40     * Huffman compressed codes are now considered as input for
41     * generating SHA-1 of the compressed codes.
42     */
43    string text, line;
44    ifstream infile("Huffman_Codes.txt", ios::in);
45    while (getline(infile, line)) {
46        text = line;
47    }
48    const char* testarray[] = { text.c_str() };
49
50    SHA1Context sha; // Holds the SHA-1 information
51    int i, j, err;
52    uint8_t Message_Digest[20]; // Message Digest
53
54    /* Perform SHA1 operation */
55    for (j = 0; j < 1; j++) {
56        err = SHA1Reset(&sha);
57        if (err)
58        {
59            fprintf(stderr, "Error Resetting SHA-1 %d.\n", err);
60            break; /* out of for j loop */
61        }
62
63        for (i = 0; i < repeatcount[j]; ++i)
64        {

```

Figure B.9: *Compression.cpp*.

```

1  err = SHA1Input(&sha ,
2      (const unsigned char *)testarray[j],
3      strlen(testarray[j]));
4      if (err)
5          {
6              fprintf(stderr, "\n SHA-1 input error %d.\n", err);
7              break;    /* out of for i loop */
8          }
9      }
10 err = SHA1Result(&sha, Message_Digest);
11 if (err)
12 {
13     fprintf(stderr ,
14         "SHA-1 output Error %d, could not compute message digest.\n",
15         err);
16 }
17 else
18 {
19     cout << "\nSHA-1 of Huffman Encoded String :\n";
20     for (i = 0; i < 20; ++i)
21     {
22         printf("%02X ", Message_Digest[i]); /* prints the message digest */
23     }
24     printf("\n");
25 }
26 }
27 getchar();
28 return 0;
29 }

```

Figure B.10: *Compression.cpp continuation – 1.*

```

1  mississippi

```

Figure B.11: *Input.txt.*

```

1  100011110111101011010

```

Figure B.12: *HuffmanCodes.txt.*



## Appendix C

### APPENDIX C: NEW SEED CONSTRUCTION SOURCE

```
1 #pragma once
2 /*
3 * Header file for sha code
4 * reference: US Secure Hash Algorithm 1 by Network working group
5 */
6
7 /* This Header will decalre set of integer types having specified
8 * widths. Width of an integer type is the numnber of bits used to
9 * store its value in a pure binary system.
10 *
11 * The actual type may use more bits that that. for example, a 28-bit
12 * type could store in 32-bit of actual storage.
13 */
14 #include <stdint.h>
15
16 /*
17 * enum is a user-defined data type that consists of integral constants.
18 */
19 enum
20 {
21     shaOk = 0,
22     shaNull, /* Null pointer parameter */
23     shaInputLong, /* long input data */
24     shaError /* called after result */
25 };
26
27 #define Sha1HashSize 20
28
29 /*
30 * This structure will hold the context information for the SHA-1 hash
31 * operation typedef is a reserved keyword. By defining the typedef,
32 * it is assured that all the variables are structure pointer types,
33 * or each variable is a pointer type pointing to a structure type.
34 */
35 /*
36 * uint32_t ----> unsigned 32 bit integer
37 * uint8_t ----> unsigned 8 bit integer
38 */
39 typedef struct SHA1Context
40 {
41     uint32_t Intermediate_Hash[Sha1HashSize / 4]; /* Message Digest */
42
43     uint32_t Length_Low; /* Message length in bits */
44     uint32_t Length_High; /* Message length in bits */
45     int_least16_t Message_Block_Index; /* Index into message block array */
46     uint8_t Message_Block[64]; /* 512-bit message blocks */
47
48     int Computed; /* Is the digest computed? */
49     int Corrupted; /* Is the message digest corrupted? */
50 } SHA1Context; /*is the message digest corrupted ? */
51
52 /*
53 * Function Prototypes
54 */
55 int SHA1Reset(SHA1Context *);
56 int SHA1Input(SHA1Context *, const uint8_t *, unsigned int);
57 int SHA1Result(SHA1Context *,
58     uint8_t Message_Digest[Sha1HashSize]);
```

Figure C.1: *NewSeedHeader.cpp*.

```

1  /*
2 * Description: this file implements the SHA1
3 * SHA1 produces 1 160-bit message digest for a given input
4 * SHA1 is defined in terms of 32-bit words. This code uses Header.h file
5 * to define 32-bit and 8-bit unsigned integer types.
6 * This code only works with message with length that is a multiple of the size of an
7 * 8-bit character
8 */
9
10 #include "New_Seed_Header.h"
11 #include <iostream>
12 #include <vector>
13 #include <string>
14 #include <sstream>
15 #include <fstream>
16 #include <cstring>
17
18 using namespace std;
19
20 /* Method to assign message block numbers from 36 to 55*/
21 int assignBlockNumber = 0;
22 int text(int value) {
23
24     for (int a = value + 1; a < 56; ) {
25         assignBlockNumber = a;
26         a++;
27         break;
28     }
29     return(assignBlockNumber);
30 }
31 }
32
33 /* Dynamic vector holds the string values and
34 * Identifies the values by whitespaces in between them.
35 * returns the hash values after delimitng.
36 */
37 vector<string> split(string str, char delimiter) {
38     vector<string> inputHashValue;
39     stringstream ss(str); // Turn the string into a stream.
40     string tok;
41
42     while (getline(ss, tok, delimiter)) {
43         inputHashValue.push_back(tok);
44     }
45
46     return inputHashValue;
47 }
48
49
50 /*
51 * Define the SHA1 circular left shift
52 *
53 * The circular left shift operation  $S^n(X)$ , where X is a word and n is an integer
54 * with  $0 \leq n < 32$ 
55 *
56 *  $X \ll n$  is obtained as follows - discard the left-most n bits of X and then pad the
57 * result with n zeros on the right( the result will still be 32-bits).
58 *
59 *  $X \gg 32- n$  is obtained by discarding the right-most n bits of X and then padding
60 * the result with n zeros on the left. Thus  $S^n(X)$  is equivalent to a circular shift
61 * of X by n positions to the left.
62 */
63 #define SHA1CircularShift(bits, word) \
64     (((word) << (bits)) | ((word) >> (32-(bits))))

```

Figure C.2: *NewSeedSource.cpp*.

```

1/* Local Function Prototypes */
2void SHA1PadMessage(SHA1Context *);
3void SHA1ProcessMessageBlock(SHA1Context *);
4/*
5* SHA1 Reset
6*
7* This function will initialize the SHA1Context in preparation for computing
8* a new SHA1 message digest.
9*
10* It initializes Length_Low, Length_High and Message_block_index to zero
11*
12* Returns SHA1 Error code
13*/
14int SHA1Reset(SHA1Context *context) {
15    if (!context) {
16        return shaNull;
17    }
18    context->Length_Low = 0;
19    context->Length_High = 0;
20    context->Message_Block_Index = 0;
21
22    context->Intermediate_Hash[0] = 0x67452301;
23    context->Intermediate_Hash[1] = 0xEFCDAB89;
24    context->Intermediate_Hash[2] = 0x98BADCFE;
25    context->Intermediate_Hash[3] = 0x10325476;
26    context->Intermediate_Hash[4] = 0xC3D2E1F0;
27
28    context->Computed = 0;
29    context->Corrupted = 0;
30
31    return shaOk;
32}
33
34/*
35* SHA1Result
36*
37* Description:
38*     This function will return the 160-bit message digest into the
39*     Message_Digest array provided by the caller.
40*     NOTE: The first octet of hash is stored in the 0th element,
41*           the last octet of hash in the 19th element.
42*
43* Parameters:
44*     context: [in/out]
45*         The context to use to calculate the SHA-1 hash.
46*     Message_Digest: [out]
47*         Where the digest is returned.
48*
49* Returns:
50*     sha Error Code.
51*
52*/
53int SHA1Result(SHA1Context *context,
54    uint8_t Message_Digest[Sha1HashSize])
55{
56    int i;
57
58    if (!context || !Message_Digest)
59    {
60        return shaNull;
61    }
62
63    if (context->Corrupted)
64    {
65        return context->Corrupted;
66    }

```

Figure C.3: *NewSeedSource.cpp* continuation-1.

```

1
2  if (!context->Computed)
3  {
4      SHA1PadMessage(context);
5      for (i = 0; i < 64; ++i)
6      {
7
8          /* message may be sensitive, clear it out */
9          context->Message_Block[i] = 0;
10     }
11     context->Length_Low = 0;    /* and clear length */
12     context->Length_High = 0;
13     context->Computed = 1;
14 }
15
16 for (i = 0; i < Sha1HashSize; ++i)
17 {
18     Message_Digest[i] = context->Intermediate_Hash[i >> 2]
19         >> 8 * (3 - (i & 0x03));
20 }
21
22 return shaOk;
23 }
24
25
26 /*
27 * SHA1 Input
28 * Description: This function accepts an array of octects as the next portion of the
29 * message
30
31 * message_array is a parameter : An array of Characters representing the next
32 * portion of the message
33
34 * length: length of the message in message_array
35
36 * Returns SHA Error code
37 * If the number of bits in a message is a multiple of 8, for compactness we can represent the
38 *   ↳ message in hex.
39 * The padded message will contain 16 * n words for some n > 0. The padded message is regarded as
40 *   ↳ a sequence of n blocks M(1) , M(2), first characters (or bits) of the message.
41 */
42 int SHA1Input(SHA1Context *context, const uint8_t *message_array, unsigned length) {
43     if (!length) { //condition for length
44         return shaOk;
45     }
46     if (!context || !message_array) { //condition for no message array
47         return shaNull;
48     }
49     if (context->Computed) {
50         context->Corrupted = shaError; // condition to check corrupted bits
51         return shaError;
52     }
53     if (context->Corrupted) {
54         return context->Corrupted; // condition to check corrupted bits
55     }
56     while (length -- && !context->Corrupted)
57     {
58         context->Message_Block[context->Message_Block_Index++] = (*message_array & 0xFF);
59         context->Length_Low += 8;
60         if (context->Length_Low == 0) {
61             context->Length_High++;
62             if (context->Length_High == 0) {
63                 /* Message is too long*/
64                 context->Corrupted = 1;
65             }
66         }
67     }
68     if (context->Message_Block_Index == 64) {

```

Figure C.4: *NewSeedSource.cpp Continuation-2.*

```

1  SHA1ProcessMessageBlock( context );
2  }
3  message_array++;
4  }
5  return shaOk;
6 }
7 /*
8 * SHA1ProcessMessageBlock
9 *
10 * Description: This function will process the next 512 bits of the message stored
11 * in the Message_block array
12 *
13 * this function has no parameters and returns nothing
14 */
15 void SHA1ProcessMessageBlock( SHA1Context *context ) {
16     const uint32_t K[] = {          /* Constants defined in SHA-1 */
17         0x5A827999,
18         0x6ED9EBA1,
19         0x8F1BBCDC,
20         0xCA62C1D6
21     };
22
23     int         t;                /* Loop counter */
24     uint32_t    temp;            /* Temporary word value */
25     uint32_t    W[80];          /* Word sequence */
26     uint32_t    A, B, C, D, E;  /* Word buffers */
27
28     /*
29      * Initialize the first 16 words in the array W
30      * | is a bitwise or, example x |= 8 means x = x | 8
31      */
32     for ( t = 0; t < 16; t++)
33     {
34         W[t] = context->Message_Block[t * 4] << 24;
35         W[t] |= context->Message_Block[t * 4 + 1] << 16;
36         W[t] |= context->Message_Block[t * 4 + 2] << 8;
37         W[t] |= context->Message_Block[t * 4 + 3];
38     }
39
40     for ( t = 16; t < 80; t++)
41     {
42         W[t] = SHA1CircularShift(1, W[t - 3] ^ W[t - 8] ^ W[t - 14] ^ W[t - 16]);
43     }
44
45     A = context->Intermediate_Hash[0];
46     B = context->Intermediate_Hash[1];
47     C = context->Intermediate_Hash[2];
48     D = context->Intermediate_Hash[3];
49     E = context->Intermediate_Hash[4];
50
51     for ( t = 0; t < 20; t++)
52     {
53         temp = SHA1CircularShift(5, A) +
54             ((B & C) | ((~B) & D)) + E + W[t] + K[0];
55         E = D;
56         D = C;
57         C = SHA1CircularShift(30, B);
58         B = A;
59         A = temp;
60     }
61
62     for ( t = 20; t < 40; t++)
63     {
64         temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[1];

```

Figure C.5: *NewSeedSource.cpp Continuation-3.*

```

1     E = D;
2     D = C;
3     C = SHA1CircularShift(30, B);
4     B = A;
5     A = temp;
6 }
7
8 for (t = 40; t < 60; t++)
9 {
10    temp = SHA1CircularShift(5, A) +
11        ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
12    E = D;
13    D = C;
14    C = SHA1CircularShift(30, B);
15    B = A;
16    A = temp;
17 }
18
19 for (t = 60; t < 80; t++)
20 {
21    temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[3];
22    E = D;
23    D = C;
24    C = SHA1CircularShift(30, B);
25    B = A;
26    A = temp;
27 }
28
29 context->Intermediate_Hash[0] += A;
30 context->Intermediate_Hash[1] += B;
31 context->Intermediate_Hash[2] += C;
32 context->Intermediate_Hash[3] += D;
33 context->Intermediate_Hash[4] += E;
34
35 context->Message_Block_Index = 0;
36 }
37
38 /*
39 * SHA1PadMessage
40
41 * Description: According to SHA1 standard, the message must be padded to an even 512 bits.
42 * 160-bit hash value of Huffman compressed codes and 288 0's appended.
43 * The last 64 bits represent the length of the original message.
44 * This function will pad the message according to those rules by filling the Message_Block array
45 *   ↳ accordingly.
46 *
47 * It will also call the ProcessMessageBlock function provided appropriately.
48 * When it returns, it can be assumed that the message digest has been computed.
49 *
50 * Parameters are context and the ProcessMessageBlock function
51 *
52 * returns nothing
53 *
54 * If the number of bits in a message is a multiple of 8, for compactness we can represent the
55 *   ↳ message in hex
56 */
57 void SHA1PadMessage(SHA1Context *context) {
58     /*
59     * Check to see if the current message block is too small to hold
60     * the initial padding bits and length. If so, we will pad the
61     * block, process it, and then continue padding into a second
62     * block.
63     */
64     if (context->Message_Block_Index > 35)
65     {

```

Figure C.6: *NewSeedSource.cpp Continuation-4.*

```

1  while (context->Message_Block_Index < 64)
2  {
3      context->Message_Block[context->Message_Block_Index++] = 0;
4  }
5
6  SHA1ProcessMessageBlock(context);
7
8  while (context->Message_Block_Index < 36)
9  {
10     context->Message_Block[context->Message_Block_Index++] = 0;
11 }
12 }
13 else
14 {
15     while (context->Message_Block_Index < 36)
16     {
17         context->Message_Block[context->Message_Block_Index++] = 0;
18     }
19 }
20 /* SHA-1 of Huffman compressed codes are taken as input*/
21 string line;
22 ifstream inputfile("Huffman_SHA1_input.txt", ios::out);
23 if (inputfile.is_open())
24 {
25     int index = 0;
26     while (std::getline(inputfile, line))
27     {
28         istream& getline(inputfile >> line);
29     }
30 }
31
32 int blockNumber = 35;
33 string startingNumber = "";
34 vector<string> sep = split(line, ' '); //Dynamic vector holds the Hash values from text file.
35 for (unsigned int i = 0; i < sep.size(); ++i) {
36     startingNumber = sep[i];
37     text(blockNumber); // Call Text function
38     blockNumber = assignBlockNumber;
39     int num = stoi(startingNumber, 0, 16); //Convert String to Hexadecimal
40     context->Message_Block[assignBlockNumber] = num; //Assign hash values values to Blocks 36 to
41     ↪55.
42 }
43
44 /*
45 * Store the message length as the last 8 octets
46 */
47 context->Message_Block[56] = context->Length_High >> 24;
48 context->Message_Block[57] = context->Length_High >> 16;
49 context->Message_Block[58] = context->Length_High >> 8;
50 context->Message_Block[59] = context->Length_High;
51 context->Message_Block[60] = context->Length_Low >> 24;
52 context->Message_Block[61] = context->Length_Low >> 16;
53 context->Message_Block[62] = context->Length_Low >> 8;
54 context->Message_Block[63] = context->Length_Low;
55
56 SHA1ProcessMessageBlock(context);
57 }

```

Figure C.7: *NewSeedSource.cpp Continuation-5.*

```

1/*
2* This Program will result our observation
3*/
4#include <stdio.h>
5#include <string.h>
6#include "New_Seed_Header.h"
7/*
8* Define test patterns
9*/
10#define TEST1 "mississippi"
11//huffman-D5 8B 85 9A 2A C4 44 30 8D 5C 18 19 ED DC A0 77 23 3B B8 2D
12/* an exact multiple of 512 bits */
13char *testarray[1] =
14{
15    TEST1
16};
17long int repeatcount[1] = { 1 };
18
19/* Main function */
20int main() {
21    SHA1Context sha;
22    int i, j, err;
23    uint8_t Message_Digest[20];
24
25    /* Perform NewSeed test */
26    for (j = 0; j < 1; ++j) {
27        err = SHA1Reset(&sha);
28        if (err)
29            {
30                fprintf(stderr, "SHA1Reset Error %d.\n", err);
31                break; /* out of for j loop */
32            }
33        for (i = 0; i < repeatcount[j]; ++i)
34            {
35                err = SHA1Input(&sha,
36                    (const unsigned char *)testarray[j],
37                    strlen(testarray[j]));
38                if (err)
39                    {
40                        fprintf(stderr, "SHA1Input Error %d.\n", err);
41                        break; /* out of for i loop */
42                    }
43            }
44        err = SHA1Result(&sha, Message_Digest);
45        if (err)
46            {
47                fprintf(stderr,
48                    "SHA1Result Error %d, could not compute message digest.\n",
49                    err);
50            }
51        else
52            {
53                printf("Our Observation - ");
54                for (i = 0; i < 20; ++i)
55                    {
56                        printf("%02X ", Message_Digest[i]);
57                    }
58                printf("\n");
59            }
60    }
61    getchar();
62    return 0;
63}

```

Figure C.8: *NewSeedMain.cpp*.



1 45 30 A9 14 B9 B9 37 73 59 89 A3 27 71 A4 92 33 0E BB DB 7F

Figure C.9: *HuffmanSHA1input.txt*.