Name - Amisha Sherekar
ID- 201771595

COMP522 Assignment 2

1) Comparison of methods for message authentication

1.1) Hash-function

Hash functions are mathematical algorithms that take an input message and produce a fixed-size output called a hash value or digest. The hash function is a one-way function. The use of hash functions is primarily for the verification of message integrity. The receiver can determine whether a message has been tampered with by comparing the received message's hash value to the original message's hash value. In message authentication, hash functions are often paired with secret keys using a process known as HMAC (Hash-based Message Authentication Code), which is a secure and efficient way to authenticate messages. Despite knowing the hash function, an attacker can only forge valid MACs if they know the secret key in HMAC.

The processing of using Hash-function is as follows:

1. A sender transmits an unencrypted message to a receiver along with a hash value created by the sender
2. Upon receiving the message, the receiver performs a verification process to confirm its integrity and authenticity.
3. The receiver generates a new hash value and compares it to the hash value provided by the sender.
4. If there is a disparity between the hash values, the receiver deduces that the message has been altered or corrupted.
5. Conversely, if the hash values match, the receiver can trust that the message has not undergone any unauthorized modifications.

The advantages of Hash function are as follows:-

Hash function ensures data integrity by generating fixed-size hash values, aiding in the quick detection of tampering. Additionally, hash functions optimize data retrieval, enhance password security, and play a pivotal role in cryptographic applications, such as digital signatures. They contribute to efficient and balanced data distribution in distributed systems, support file deduplication, and facilitate quick and reliable data processing in various scenarios. Overall, hash functions are fundamental for secure and efficient computing practices.

The limitations of hash function are as follows -

The fact that hash functions are irreversible makes it impossible to retrieve the original data from the hash value, which can be problematic in some situations. A security risk arises from collision vulnerabilities, which occur when different inputs yield the same hash value, especially when hash functions are weaker. When there are a lot of distinct inputs, the fixed-size output of hash functions can also cause collisions. Furthermore, hash functions could require a lot of resources, particularly in cryptographic applications that use big datasets. With time, new flaws

in hash function algorithms might surface, underscoring the importance of constant security monitoring.
Figure 1 is the outcome of the MessageDigestSHA1 program which is given in Figure 2.

```
This is a sample text
input : This is a sample text
digest : e666e67f66f4038e0c1d2f7c3a2abeaf27b3123f
(base) amishasherekar@Amishas-MacBook-Air old %
```

Figure 1

```java

import java.security.MessageDigest;
import java.util.Scanner;

/**
 *
 */
public class MessageDigestSHA1
{
    Run | Debug
    public static void main(
            String[]    args)
            throws Exception
    {

        Scanner input = new Scanner(System.in);
        System.out.println(x:"Enter plaintext");
        String Gettext = input.nextLine();

        MessageDigest   hash = MessageDigest.getInstance(algorithm:"SHA1");

        System.out.println("input : " + Gettext);

        hash.update(Utils.toByteArray(Gettext));

        System.out.println("digest : " + Utils.toHex(hash.digest()));


    }
}
```

Figure 2

1.2) RSA + SHA1

When RSA and SHA-1 are used together for message authentication, the sender creates a digital signature for the message in order to guarantee its authenticity and integrity.

In order to do this, the sender extracts a fixed-size hash value from the message content using the SHA-1 hash function. The sender then uses their private RSA key to encrypt this hash value, creating a digital signature that is exclusive to that particular message. The recipient uses

the sender's public RSA key to decrypt the signature and extract the hash value after receiving the message and its accompanying signature. Subsequently, the recipient independently calculates the message's SHA-1 hash and contrasts it with the hash that has been decrypted.

A reliable technique for message authentication is provided if the two hashes match, confirming that the message was sent from the intended sender and was not altered during transmission. It's important to remember, though, that SHA-1 is no longer regarded as secure for cryptographic purposes. Instead, more reliable hash functions, like SHA-256, are advised for improved security in modern cryptography systems.

An illustration of how a manipulated message can be identified is demonstrated in Appendices B and C. In the following experiment, I generated a message containing the phrase "This is a sample message," encrypted it, and transmitted it to the receiver class. However, during transit to the verifier, the message was intercepted, and the plaintext was altered to read "This is not a sample message." As depicted in Figure 4 below, the hash no longer corresponds due to the modification, rendering the message untrustworthy. Figure 3, on the other hand, exhibits a successful message that reached the verifier without any interference.

```
This is a sample message
sender digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
cipher: 747e277a0eb719f7e08181dff5ddeae40bbd4844fd34f2bbba0766f1aa8e2d282756a57fdf83d784ee965c57840a59dc8822b56c760d10a0f24507e421494366
Message sent by sender: This is a sample message
message received by verifier: This is a sample message
digest values match
verifier hash: 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
sender hash  : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
```

Figure 3

```
sender digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
cipher: 7126cf6f4058c503da3874f836cf559413c704fc62be0897e2ffeeaef3a050b9f1d9262c92b0aae93352a64221065ee77f94cb76e10c75879c52ba4e6365b1b8
Message sent by sender: This is a sample message
message received by verifier: This is not a sample message
hashed values dont match this message is not trustworthy.
verifier hash: 5f35515a716b5fa74b9cde257badc4b67c21fcce
sender hash  : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
```

Figure 4

The programs used for implementing RSA + SHA1 method is given below in figures 5,6,7,8.

```java

import javax.crypto.Cipher;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.util.Scanner;

public class Sender {
    Run | Debug
    public static void main(String[] args) throws Exception {

        // declare requirements
        Scanner input = new Scanner(System.in);
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] hashedInput = "".getBytes();
        SecureRandom random = new SecureRandom();
        KeyPairGenerator generator = KeyPairGenerator.getInstance(algorithm:"RSA");

        // take the input text
        System.out.println(x:"Please enter text:");
        String inputPlainText = input.nextLine();

        // retrieve the hash of the input text or if that fails then exit the program.
        try {
            hashedInput = MessageDigestor.messageDigest(inputPlainText);
        } catch (Exception e) {
            System.out.println("error: " + e);
            System.exit(status:1);
        }
        System.out.println("sender digest : " + Utils.toHex(hashedInput));

        // generate the keys
        generator.initialize(keysize:512, random);
```

Figure 5 - sender program

```java

        // keys
        KeyPair pair = generator.generateKeyPair();
        Key pubKey = pair.getPublic();
        Key privKey = pair.getPrivate();

        // encrypt the hashed input and print out
        cipher.init(Cipher.ENCRYPT_MODE, privKey);
        byte[] cipherText = cipher.doFinal(hashedInput);
        System.out.println("cipher: " + Utils.toHex(cipherText));
        System.out.println("Message sent by sender: " + inputPlainText);


        // generate the message
        Message senderMessage = new Message(inputPlainText, cipherText, pubKey);

        // send the message to the verifier
        Verifier.verifyMessage(senderMessage);

    }
}

```

Figure 6 - sender program

```java
import javax.crypto.Cipher;

public class Verifier {
    Run | Debug
    public static void main(String[] args) {

    }
    public static void verifyMessage (Message senderMessage) throws Exception{
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] verifierHashedInput = "".getBytes();

        //Decrypt the senders cipherText to get the hashedMessage using the publicKey
        cipher.init(Cipher.DECRYPT_MODE, senderMessage.getPubKey());
        byte[] plainText = cipher.doFinal(senderMessage.getHashedMessage());
        // Sender digest
        String senderDigest = Utils.toHex(plainText);

        //String interceptedMessage = "This is not a sample message";

        System.out.println("message received by verifier: " + senderMessage.getOriginalMessage());

        //calculates own hashed message
        try{
            verifierHashedInput = MessageDigestor.messageDigest(senderMessage.getOriginalMessage());
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(status:1);
        }

        String verifierDigest = Utils.toHex(verifierHashedInput);

        //compares if the hashed value generated by the verifier using the original message matches the senc

```

Figure 7 - Verifier Program

```java
        if( verifierDigest.equals(senderDigest)){
            System.out.println(x:"digest values match");
            System.out.println("verifier hash: " + verifierDigest);
            System.out.println("sender hash  : " + senderDigest);
        } else {
            System.out.println(x:"hashed values dont match this message is not trustworthy.");
            System.out.println("verifier hash: " + verifierDigest);
            System.out.println("sender hash  : " + senderDigest) ;
        }

    }
}
```

Figure 8 - Verifier Program

1.3) DSA

Digital signature algorithms, or DSAs for short, are public-key cryptography based on mathematical principles that have been globally standardised by the National Institute of Standards and Technology. The digital signature for the message is generated by the algorithm using the private key; the recipient can confirm that no modifications have been made to the original message using the signer's public key.

The DSA procedure implemented in JPA involves the following sequence of actions:

- It takes the original message from the sender, generates a hash, and signs the message using a private key to create a unique signature specific to the sent message.
- The message, comprising the original content, the sender's generated signature hash, and the sender's public key, is transmitted to the recipient.
- Upon reception, the verifier can create a hash of the original message.
- The verifier then utilizes their message hash and the sender's public key to confirm that the provided public key corresponds to the private key used for signing the message content.
- If the DSA algorithm's verification function returns "True," it signifies that the received message matches the sent message and is authentic. If either the signature or the message has been altered, the verification function returns "False," indicating that the message cannot be trusted, aligning with other discussed message authentication methods.

Using DSA for message authentication thus ensures not only the integrity of the message but also provides a means for the recipient to verify the sender's identity. While DSA is a well-established algorithm, modern applications might consider other signature schemes like ECDSA for similar purposes, especially when considering key size and computational efficiency. It's important to follow best practices for cryptographic key management and stay informed about advancements in cryptographic algorithms and standards.

These are the results when the method is applied to different scenarios.

Fig. 9 illustrates how the DSA algorithm verifies that the original text and signed hash are original, meaning the original message wasn't intercepted by someone.

```
Please enter text:
This is a sample message
digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
DSA signature hash:
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fbd226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49
sent plainText: This is a sample message
received plainText: This is a sample message
received signature:
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fbd226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49
Original senders signed hash:
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fbd226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49
key values match
```

Figure 9

The figure 10 illustrates how the message can no longer be trusted when the hash has been changed and the keys can't be compared at the verify function.

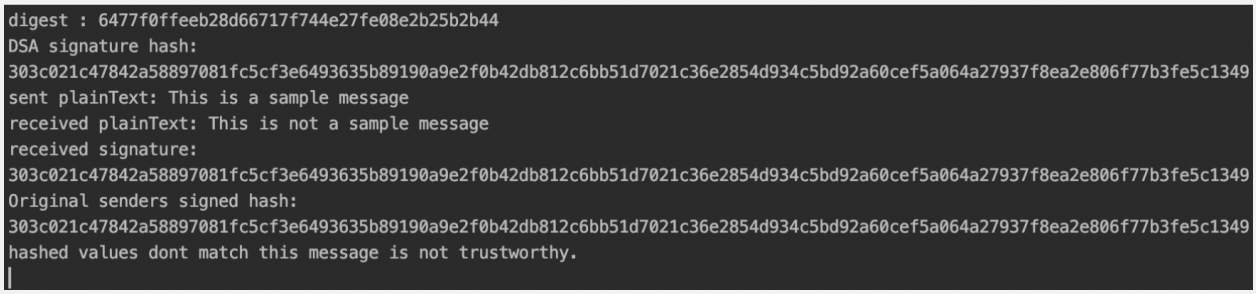


Figure 10

Figure 11 shows how the DSA algorithm throws an exception if the original message is the same but the signed hash has been changed and there has been some sort of manipulation therefore the message is unreliable.

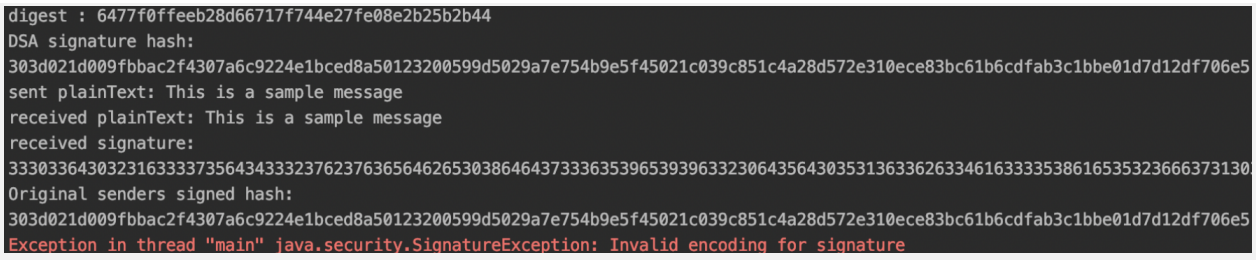
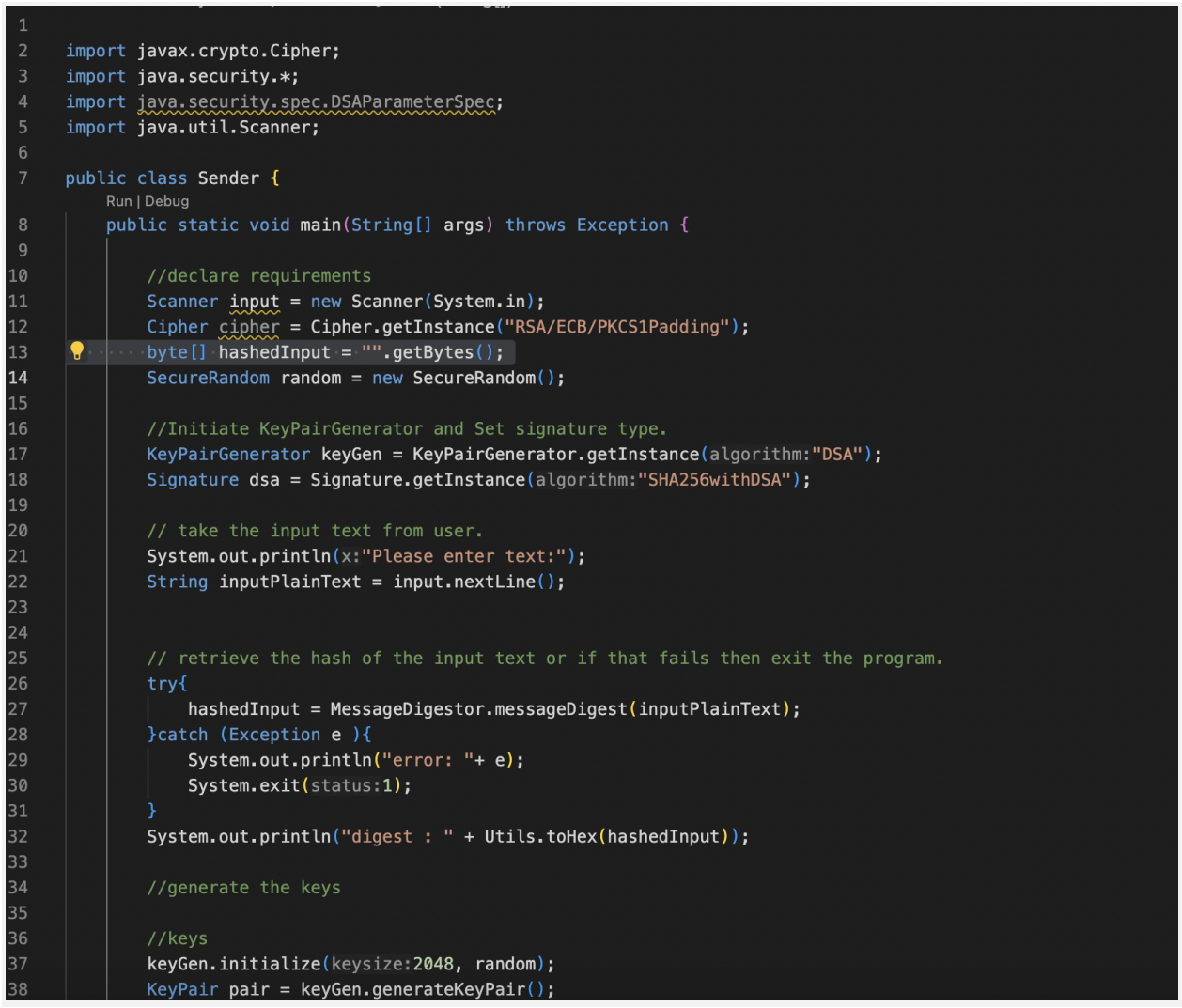


Figure 11

The program used for this method are given below in the figures 12,13,14,15



```java
import javax.crypto.Cipher;
import java.security.*;
import java.security.spec.DSAParameterSpec;
import java.util.Scanner;

public class Sender {
    Run | Debug
    public static void main(String[] args) throws Exception {

        //declare requirements
        Scanner input = new Scanner(System.in);
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        byte[] hashedInput = "".getBytes();
        SecureRandom random = new SecureRandom();

        //Initiate KeyPairGenerator and Set signature type.
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance(algorithm:"DSA");
        Signature dsa = Signature.getInstance(algorithm:"SHA256withDSA");

        // take the input text from user.
        System.out.println(x:"Please enter text:");
        String inputPlainText = input.nextLine();


        // retrieve the hash of the input text or if that fails then exit the program.
        try{
            hashedInput = MessageDigestor.messageDigest(inputPlainText);
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(status:1);
        }
        System.out.println("digest : " + Utils.toHex(hashedInput));

        //generate the keys

        //keys
        keyGen.initialize(keysize:2048, random);
        KeyPair pair = keyGen.generateKeyPair();
```

Figure 12- sender program

```
        PublicKey pubKey = pair.getPublic();
        PrivateKey privKey = pair.getPrivate();

        //Sign the hash using the private key
        dsa.initSign(privKey);
        dsa.update(hashedInput);
        byte[] signedHash = dsa.sign();

        System.out.println("DSA signature hash: \n" + Utils.toHex(signedHash));

        //generate the message
        Message senderMessage = new Message(inputPlainText, signedHash, pubKey);

        //send the message to the verifier
        Verifier.verifyMessage(senderMessage);

    }
}
```

Figure 13-sender program

```
import javax.crypto.Cipher;
import java.security.Signature;

public class Verifier {
    Run | Debug
    public static void main(String[] args) {

    }
    public static void verifyMessage (Message senderMessage) throws Exception{
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        Signature dsa = Signature.getInstance(algorithm:"SHA256withDSA");

        //Initiate verifier hashed input.
        byte[] verifierHashedInput = "".getBytes();

        //If the message has been intercepted and changed then again the signatures will not match when they
        String interceptedMessage = "This is not a sample message";

        //assign received plain text to verifier variable
        String verifierPlainText = senderMessage.getOriginalMessage();

        System.out.println("sent plainText: " + senderMessage.getOriginalMessage());
        System.out.println("received plainText: " + verifierPlainText);
        System.out.println("received signature: \n" + Utils.toHex(senderMessage.getSignedHash()));
        System.out.println("Original senders signed hash: \n" + Utils.toHex(senderMessage.getSignedHash()));

        //calculates own hashed message using the input that you have provided in the message being received
        try{
            verifierHashedInput = MessageDigestor.messageDigest(verifierPlainText);
        }catch (Exception e ){
            System.out.println("error: "+ e);
            System.exit(status:1);
        }

```

Figure 14-Verifier Program

```
34          //start verify password using the public key from the message
35          dsa.initVerify(senderMessage.getPubKey());
36          //run through the DSA using the hashed value that verifier has generated
37          dsa.update(verifierHashedInput);
38
39          //then verify the updated dsa against the key that was passed with the sender.
40          //If the values verify then they will return true otherwise they will return false.
41          boolean verifies = dsa.verify(senderMessage.getSignedHash());
42
43          //verifier using the original message matches the sender hash which was decrypted using the public ke
44          if(verifies){
45              System.out.println(x:"key values match");
46          } else {
47              System.out.println(x:"hashed values dont match this message is not trustworthy.");
48          }
49
50
51
52
53      }
54  }
55
```

Figure 15- Verifier Program

1.4) HMAC-SHA256

MAC (Message Authentication Code) serves as an additional approach to message authentication, with the HMAC-SHA256 method closely resembling the hash functions. The fundamental goal of MAC, like other authentication methods, is to guarantee that messages exchanged between two parties remain unaltered and free from interference. In MAC operations, it is assumed that both the sender and receiver have previously established and shared a common key. The sender employs their plaintext message to generate a MAC using the shared private key. This MAC functions as a tag, akin to those in previously discussed hashing processes, and is sent alongside the original message to the receiver. Upon receiving the message, the receiver utilizes the received plaintext and the previously shared private key to independently generate a MAC. Subsequently, the receiver compares the received MAC with the one they generated to detect any interference. If the MACs do not align or if the message content has been altered, the integrity of the message is compromised. Similar to hash functions, MACs are considered one-way functions since they are not reversible. However, a drawback lies in the necessity for the sender and receiver to share a secret key, introducing potential challenges if the key is compromised and requiring the establishment of a new secret key, thereby incurring additional overhead.

Figure 16 shows the output when HMAC_SHA256 program implemented. The program used is given in figure 17 and 18.

```
(base) amishasherekar@Amishas-MacBook-Air old % javac HMAC_SHA256.java
(base) amishasherekar@Amishas-MacBook-Air old % java HMAC_SHA256
0B:D0:2F:F3:FD:E8:96:1B:18:66:62:F0:51:85:FC:87:43:49:11:29:D2:A1:9E:9A:55:CA:BB
:88:06:23:9F:A1
(base) amishasherekar@Amishas-MacBook-Air old %
```

Figure - 16

```java
import java.security.*;
import javax.crypto.*;

public class HMAC_SHA256 {

    Run | Debug
    public static void main(String[] args) throws Exception {

        // Generate secret key for HmacSHA256
        KeyGenerator kg = KeyGenerator.getInstance("HmacSHA256");
        SecretKey sk = kg.generateKey();

        // Get instance of Mac object implementing HmacSHA256, and
        // initialize it with the above secret key
        Mac mac = Mac.getInstance("HmacSHA256");
        mac.init(sk);
        byte[] result = mac.doFinal("Hi".getBytes());
        System.out.println(toHexString(result));

    }

    /*
     * Converts a byte to hex digit and writes to the supplied buffer
     */
    private static void byte2hex(byte b, StringBuffer buf) {
        char[] hexChars = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                '9', 'A', 'B', 'C', 'D', 'E', 'F' };
        int high = ((b & 0xf0) >> 4);
        int low = (b & 0x0f);
        buf.append(hexChars[high]);
        buf.append(hexChars[low]);
    }

```

Figure - 17

```java
    /*
     * Converts a byte array to hex string
     */
    private static String toHexString(byte[] block) {
        StringBuffer buf = new StringBuffer();
        int len = block.length;
        for (int i = 0; i < len; i++) {
            byte2hex(block[i], buf);
            if (i < len-1) {
                buf.append(str:":");
            }
        }
        return buf.toString();
    }
}
```

Figure - 18

2) Generating Two Different Files with the Same MD5 hash

Introduction:-

In this task two different files with the same MD5 hash values are to be created. Using following command two output files are generated(out1.bin and out2.bin).

$ md5collgen -p prefix.txt -o out1.bin out2.bin

With the diff command, we can determine if the output files are distinct or not. We can also verify each output file's MD5 hash using the md5sum command.

$ diff out1.bin out2.bin
$ md5sum out1.bin
$ md5sum out2.bin

2.1) Question 1. If the length of your prefix file is not multiple of 64, what is going to happen?

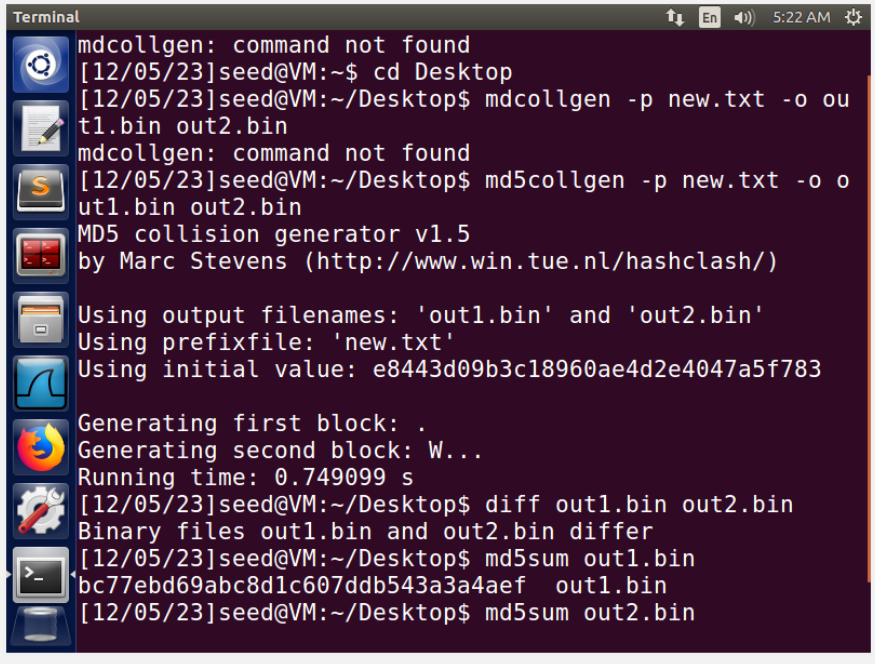


Figure 19

I made a file called new.txt, which was not a multiple of 64. Upon executing md5collgen -p hi.txt -o hi1 hi2, we can observe in figure 20,21, that the output has been padded with zeros by utilising bless hi1.Figure 19 shows the commands that were used in terminal to execute this task.
.

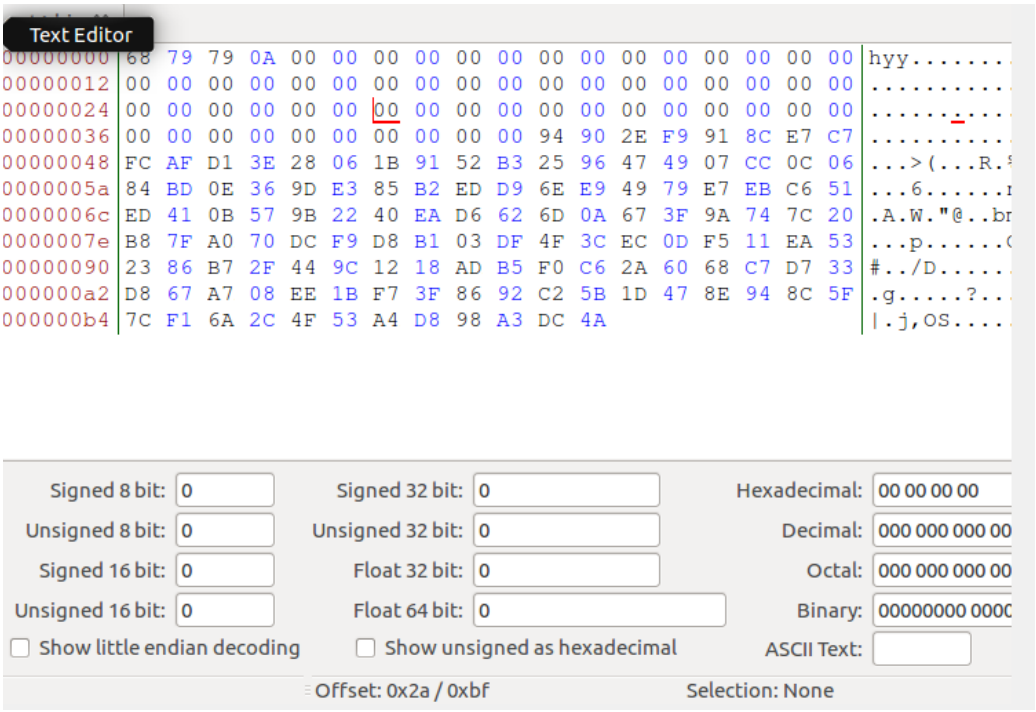


Figure 20

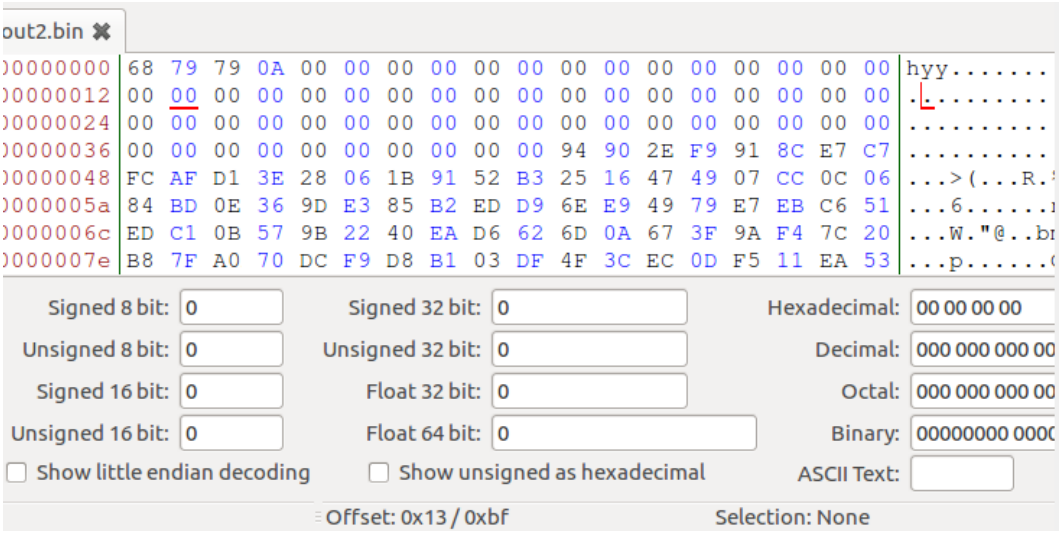


Figure 21

2.2) Question 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

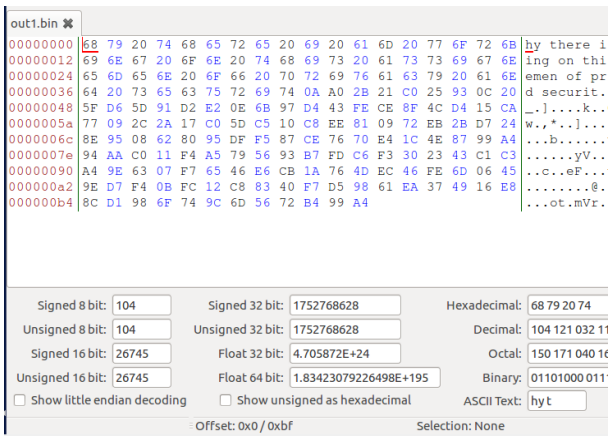


Figure 22

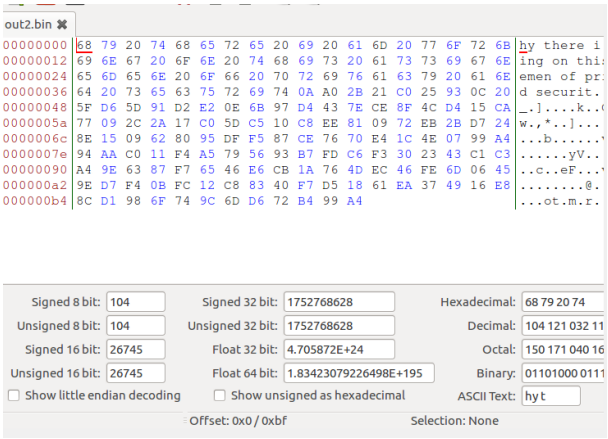


Figure 23

No extra zero-bytes are added for padding when using the 64-byte prefix. The data that was generated for the collision looks random and comes right after the prefix file.This is shown in figure 22 and 23.

2.3) Question 3. Are the data (128 bytes) generated by md5collgen completely different for the two files? Please identify all the bytes that are different.

There around 8 bytes that are different. Those are highlighted in pink in the figures 24 and 25 below.

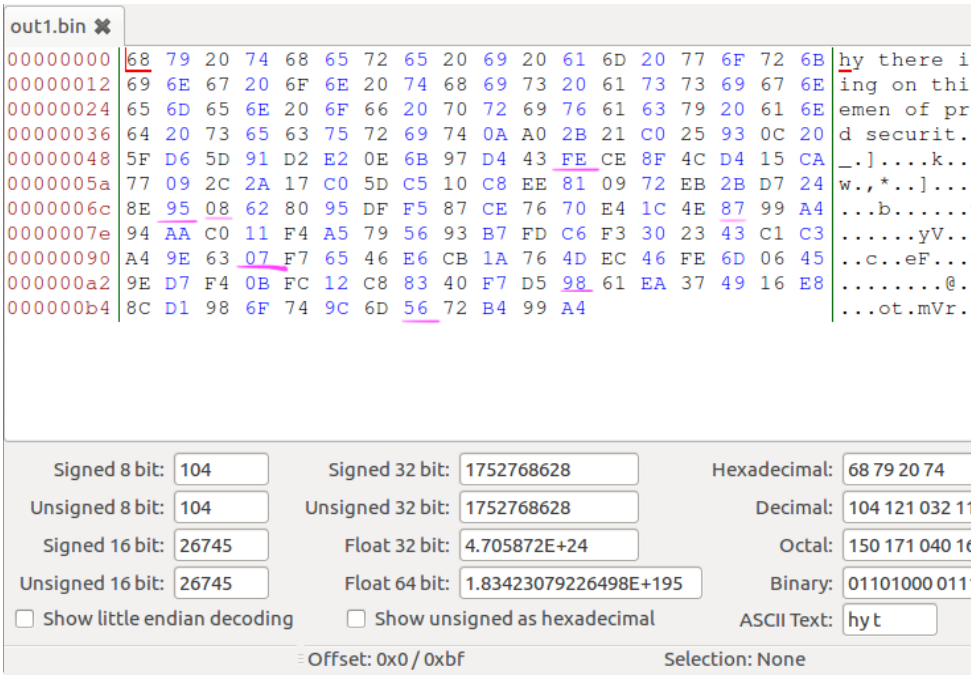


Figure 24

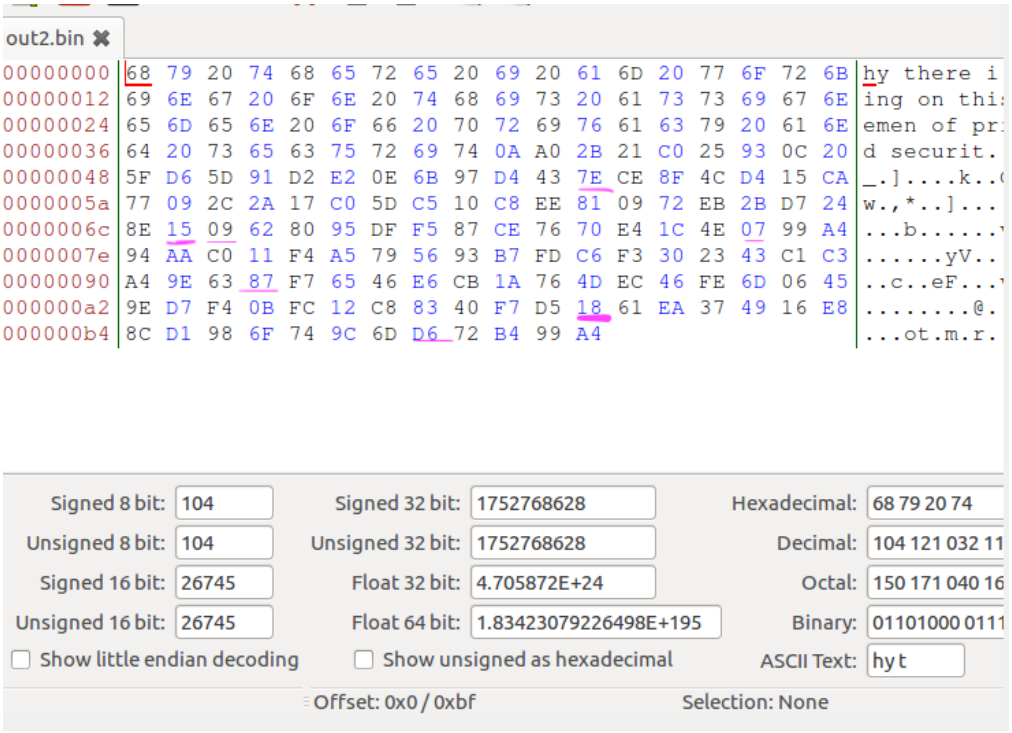


Figure 25

3) Key Exchange for Four parties

In a Diffie-Hellman key exchange involving four parties, each participant generates a private key and agrees upon a shared prime number (q) and a primitive root ($\propto$) q. Let
A, B, C and D wish to exchange a key, the following operation:

$Y_A = a^{XA} \ mod \ q$
$Y_B = a^{XB} \ mod \ q$
$Y_C = a^{XC} \ mod \ q$
$Y_D = a^{XD} \ mod \ q$
Then, they are now able to calculate the common secret key:
A calculates $K = (Y_{BCD})^{XA} \ mod \ q$
B calculates $K = (Y_{ACD})^{XB} \ mod \ q$
C calculates $K = (Y_{ABD})^{XC} \ mod \ q$
D calculates $K = (Y_{ABC})^{XD} \ mod \ q$

Figure 26 is an example to explain how to reach the shared secret key

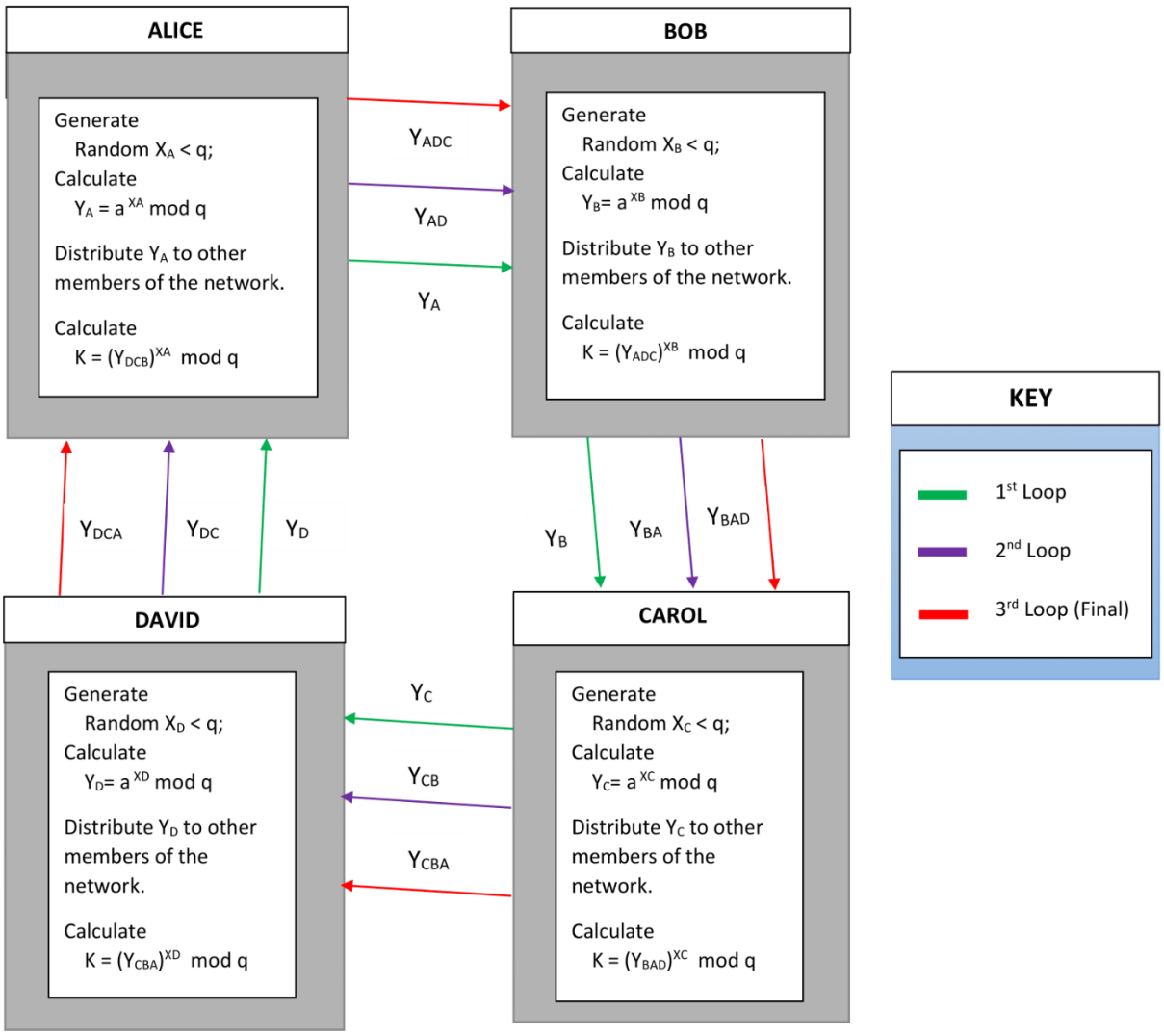


Figure 26

Table 1: Table outlining the order in which the keys are gathered in the Diffie-Hellman exchange. Public Keys held by network members

N	Alice	Bob	Carol	David
0	A	B	C	D
1	AD	BA	CB	DC
2	ADC	BAD	CBA	DCB
3	ADCB	BADC	CBAD	DCBA

Table 1

The process of gathering the keys within the programme is described in Table 1. Each member only has their own keys in iteration 0, but as we loop through the network, more keys are collected until each member has access to the public keys of every other member. Every member can determine the shared secret needed to decode every message once they can access everyone else's public keys. The output should be the same as the figure, demonstrating that all programme participants share the same secretkey and can thus communicate with one another if all the keys match. If not, the program will throw an error.

Figure 27 shows the output for Diffie-Hellman key exchange involving four parties.The code is given in a different file named “FourPartyDiffieHellman.java”

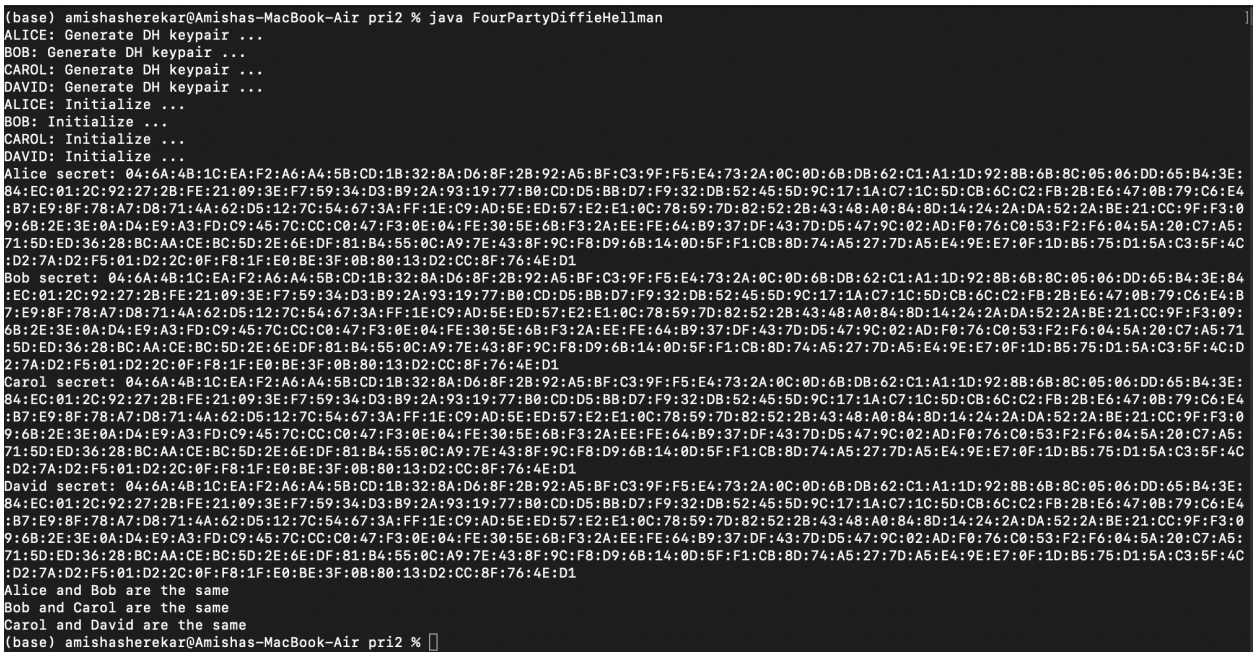


Figure 27