

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

```
1  import java.security.MessageDigest;
2  import java.util.Scanner;
3
4
5  /**
6   *
7   */
8  public class MessageDigestSHA1
9  {
10     Run | Debug
11     public static void main(
12         String[] args)
13         throws Exception
14     {
15         Scanner input = new Scanner(System.in);
16         System.out.println("Enter plaintext");
17         String Gettext = input.nextLine();
18
19         MessageDigest hash = MessageDigest.getInstance(algorithm:"SHA1");
20
21         System.out.println("input : " + Gettext);
22
23         hash.update(Utils.toByteArray(Gettext));
24
25         System.out.println("digest : " + Utils.toHexString(hash.digest()));
26
27     }
28 }
```

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: 7126cf6f4058c503da3874f836cf559413c704fc62be0897e2fffeeaeef3a050b9f1d9262c92b0aae93352a64221065ee77f94cb76e10c75879c52ba4e6365b1b8
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.

```

1 import javax.crypto.Cipher;
2 import java.security.Key;
3 import java.security.KeyPair;
4 import java.security.KeyPairGenerator;
5 import java.security.SecureRandom;
6 import java.util.Scanner;
7
8
9 public class Sender {
10     Run | Debug
11     public static void main(String[] args) throws Exception {
12
13         // declare requirements
14         Scanner input = new Scanner(System.in);
15         Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
16         byte[] hashedInput = "".getBytes();
17         SecureRandom random = new SecureRandom();
18         KeyPairGenerator generator = KeyPairGenerator.getInstance(algorithm:"RSA");
19
20         // take the input text
21         System.out.println(x:"Please enter text:");
22         String inputPlainText = input.nextLine();
23
24         // retrieve the hash of the input text or if that fails then exit the program.
25         try {
26             hashedInput = MessageDigestor.messageDigest(inputPlainText);
27         } catch (Exception e) {
28             System.out.println("error: " + e);
29             System.exit(status:1);
30         }
31         System.out.println("sender digest : " + Utils.toHexString(hashedInput));
32
33         // generate the keys
34         generator.initialize(keysize:512, random);

```

Figure 5 - sender program

```

34         // keys
35         KeyPair pair = generator.generateKeyPair();
36         Key pubKey = pair.getPublic();
37         Key privKey = pair.getPrivate();
38
39         // encrypt the hashed input and print out
40         cipher.init(Cipher.ENCRYPT_MODE, privKey);
41         byte[] cipherText = cipher.doFinal(hashedInput);
42         System.out.println("cipher: " + Utils.toHexString(cipherText));
43         System.out.println("Message sent by sender: " + inputPlainText);
44
45
46         // generate the message
47         Message senderMessage = new Message(inputPlainText, cipherText, pubKey);
48
49         // send the message to the verifier
50         Verifier.verifyMessage(senderMessage);
51
52     }
53 }
54
55

```

Figure 6 - sender program

```

1 import javax.crypto.Cipher;
2
3 public class Verifier {
4     Run | Debug
5     public static void main(String[] args) {
6
7         public static void verifyMessage (Message senderMessage) throws Exception{
8             Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
9             byte[] verifierHashedInput = "" .getBytes();
10
11             //Decrypt the senders cipherText to get the hashedMessage using the publicKey
12             cipher.init(Cipher.DECRYPT_MODE, senderMessage.getPubKey());
13             byte[] plainText = cipher.doFinal(senderMessage.getHashedMessage());
14             // Sender digest
15             String senderDigest = Utils.toHexString(plainText);
16
17             //String interceptedMessage = "This is not a sample message";
18
19             System.out.println("message received by verifier: " + senderMessage.getOriginalMessage());
20
21             //calculates own hashed message
22             try{
23                 verifierHashedInput = MessageDigestor.messageDigest(senderMessage.getOriginalMessage());
24             }catch (Exception e){
25                 System.out.println("error: "+ e);
26                 System.exit(status:1);
27             }
28
29             String verifierDigest = Utils.toHexString(verifierHashedInput);
30
31             //compares if the hashed value generated by the verifier using the original message matches the send
32

```

Figure 7 - Verifier Program

```

33     if( verifierDigest.equals(senderDigest)){
34         System.out.println(x:"digest values match");
35         System.out.println("verifier hash: " + verifierDigest);
36         System.out.println("sender hash : " + senderDigest);
37     } else {
38         System.out.println(x:"hashed values dont match this message is not trustworthy.");
39         System.out.println("verifier hash: " + verifierDigest);
40         System.out.println("sender hash : " + senderDigest) ;
41     }
42
43 }
44

```

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 : 6477f0fffeeb28d66717f744e27fe08e2b25b2b44  
DSA signature hash:  
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fb226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49  
sent plainText: This is a sample message  
received plainText: This is a sample message  
received signature:  
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fb226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49  
Original senders signed hash:  
303d021c7552cd7a8e9bd610a70924b08a8c60ba55a858fc95b6c7d047bc14de021d009cca648c597fb226485bc1a76b07f13fb0c25cb1aa44f8edd8e3d49  
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.

```
digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
DSA signature hash:
303c021c47842a58897081fc5cf3e6493635b89190a9e2f0b42db812c6bb51d7021c36e2854d934c5bd92a60cef5a064a27937f8ea2e806f77b3fe5c1349
sent plainText: This is a sample message
received plainText: This is not a sample message
received signature:
303c021c47842a58897081fc5cf3e6493635b89190a9e2f0b42db812c6bb51d7021c36e2854d934c5bd92a60cef5a064a27937f8ea2e806f77b3fe5c1349
Original senders signed hash:
303c021c47842a58897081fc5cf3e6493635b89190a9e2f0b42db812c6bb51d7021c36e2854d934c5bd92a60cef5a064a27937f8ea2e806f77b3fe5c1349
hashed values dont match this message is not trustworthy.
|
```

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.

```
digest : 6477f0ffeeb28d66717f744e27fe08e2b25b2b44
DSA signature hash:
303d021d009fbac2f4307a6c9224e1bc当地8a50123200599d5029a7e754b9e5f45021c039c851c4a28d572e310ece83bc61b6cd fab3c1bbe01d7d12df706e5
sent plainText: This is a sample message
received plainText: This is a sample message
received signature:
333033643032316333373564343332376237636564626530386464373336353965393963323064356430353136336263346163333538616535323666373130
Original senders signed hash:
303d021d009fbac2f4307a6c9224e1bc当地8a50123200599d5029a7e754b9e5f45021c039c851c4a28d572e310ece83bc61b6cd fab3c1bbe01d7d12df706e5
Exception in thread "main" java.security.SignatureException: Invalid encoding for signature
```

Figure 11

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

```
1 import javax.crypto.Cipher;
2 import java.security.*;
3 import java.security.spec.DSAPrivateKeySpec;
4 import java.util.Scanner;
5
6
7 public class Sender {
8     Run | Debug
9     public static void main(String[] args) throws Exception {
10
11         //declare requirements
12         Scanner input = new Scanner(System.in);
13         Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
14         byte[] hashedInput = "".getBytes();
15         SecureRandom random = new SecureRandom();
16
17         //Initiate KeyPairGenerator and Set signature type.
18         KeyPairGenerator keyGen = KeyPairGenerator.getInstance(algorithm:"DSA");
19         Signature dsa = Signature.getInstance(algorithm:"SHA256withDSA");
20
21         // take the input text from user.
22         System.out.println(x:"Please enter text:");
23         String inputPlainText = input.nextLine();
24
25
26         // retrieve the hash of the input text or if that fails then exit the program.
27         try{
28             hashedInput = MessageDigestor.messageDigest(inputPlainText);
29         }catch (Exception e ){
30             System.out.println("error: "+ e);
31             System.exit(status:1);
32         }
33         System.out.println("digest : " + Utils.toHexString(hashedInput));
34
35         //generate the keys
36
37         //keys
38         keyGen.initialize(keysize:2048, random);
39         KeyPair pair = keyGen.generateKeyPair();
```

Figure 12- sender program

```

39     PublicKey pubKey = pair.getPublic();
40     PrivateKey privKey = pair.getPrivate();
41
42     //Sign the hash using the private key
43     dsa.initSign(privKey);
44     dsa.update(hashedInput);
45     byte[] signedHash = dsa.sign();
46
47     System.out.println("DSA signature hash: \n" + Utils.toHex(signedHash));
48
49     //generate the message
50     Message senderMessage = new Message(inputPlainText, signedHash, pubKey);
51
52     //send the message to the verifier
53     Verifier.verifyMessage(senderMessage);
54
55 }
56
57

```

Figure 13-sender program

```

1 import javax.crypto.Cipher;
2 import java.security.Signature;
3
4 public class Verifier {
5     Run | Debug
6     public static void main(String[] args) {
7
8     }
9     public static void verifyMessage (Message senderMessage) throws Exception{
10         Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
11         Signature dsa = Signature.getInstance(algorithm:"SHA256withDSA");
12
13         //Initiate verifier hashed input.
14         byte[] verifierHashedInput = """.getBytes();
15
16         //If the message has been intercepted and changed then again the signatures will not match when they
17         String interceptedMessage = "This is not a sample message";
18
19         //assign received plain text to verifier variable
20         String verifierPlainText = senderMessage.getOriginalMessage();
21
22         System.out.println("sent plainText: " + senderMessage.getOriginalMessage());
23         System.out.println("received plainText: " + verifierPlainText);
24         System.out.println("received signature: \n" + Utils.toHex(senderMessage.getSignedHash()));
25         System.out.println("Original senders signed hash: \n" + Utils.toHex(senderMessage.getSignedHash()));
26
27         //calculates own hashed message using the input that you have provided in the message being received
28         try{
29             verifierHashedInput = MessageDigestor.messageDigest(verifierPlainText);
30         }catch (Exception e ){
31             System.out.println("error: "+ e);
32             System.exit(status:1);
33         }

```

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 key
44     if(verifies){
45         System.out.println("key values match");
46     } else {
47         System.out.println("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

```
1 import java.security.*;
2 import javax.crypto.*;
3 |
4 public class HMAC_SHA256 {
5
6     Run | Debug
7     public static void main(String[] args) throws Exception {
8
9         // Generate secret key for HmacSHA256
10        KeyGenerator kg = KeyGenerator.getInstance("HmacSHA256");
11        SecretKey sk = kg.generateKey();
12
13        // Get instance of Mac object implementing HmacSHA256, and
14        // initialize it with the above secret key
15        Mac mac = Mac.getInstance("HmacSHA256");
16        mac.init(sk);
17        byte[] result = mac.doFinal("Hi".getBytes());
18        System.out.println(toHexString(result));
19    }
20
21    /*
22     * Converts a byte to hex digit and writes to the supplied buffer
23     */
24    private static void byte2hex(byte b, StringBuffer buf) {
25        char[] hexChars = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
26                           '9', 'A', 'B', 'C', 'D', 'E', 'F' };
27        int high = ((b & 0xf0) >> 4);
28        int low = (b & 0x0f);
29        buf.append(hexChars[high]);
30        buf.append(hexChars[low]);
31    }
32}
```

Figure - 17

```
34     * Converts a byte array to hex string
35     */
36     private static String toHexString(byte[] block) {
37         StringBuffer buf = new StringBuffer();
38         int len = block.length;
39         for (int i = 0; i < len; i++) {
40             byte2hex(block[i], buf);
41             if (i < len-1) {
42                 buf.append(":");
43             }
44         }
45         return buf.toString();
46     }
47 }
```

Figure - 18

2) Generating Two Different Files with the Same MD5 hash

Introduction:-

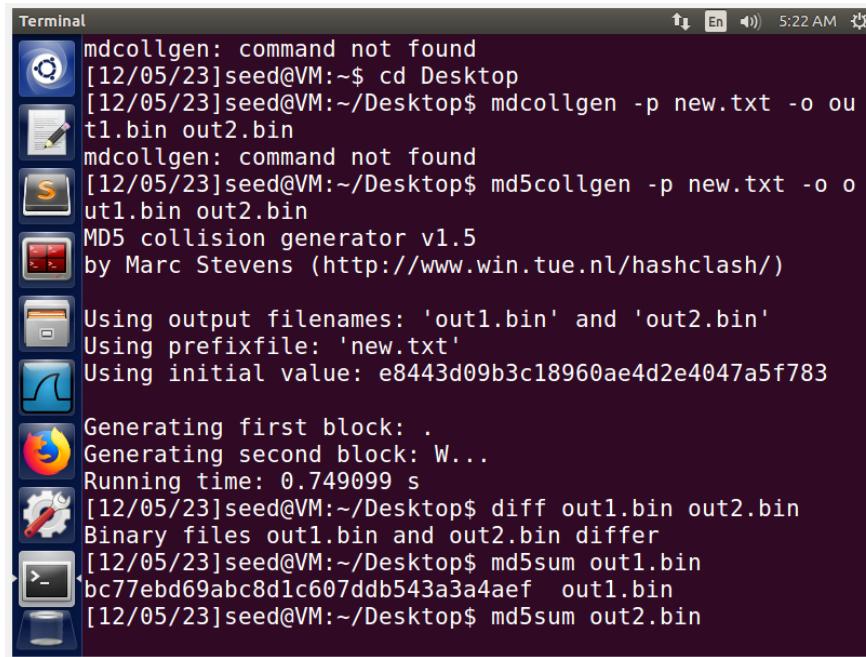
In this task two different files with the same MD5 hash values are to be created. Using following command two output files are generated(out1.bin and out2.bin).

```
$ 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?



The screenshot shows a terminal window titled "Terminal". The session starts with the user trying to run "md5collgen" without specifying the prefix file, which results in an error message: "md5collgen: command not found". The user then navigates to the Desktop directory and runs "md5collgen -p new.txt -o out1.bin out2.bin". This command generates two files, "out1.bin" and "out2.bin", using a specified initial value and prefix file. The process takes approximately 0.749 seconds. After generating the files, the user runs "diff out1.bin out2.bin", which outputs "Binary files out1.bin and out2.bin differ", indicating that the files are not identical. Finally, the user runs "md5sum" on both files, showing that they have different MD5 hashes: "bc77ebd69abc8d1c607ddb543a3a4aef" for "out1.bin" and "bc77ebd69abc8d1c607ddb543a3a4aef" for "out2.bin".

```
Terminal  
[12/05/23]seed@VM:~$ cd Desktop  
[12/05/23]seed@VM:~/Desktop$ md5collgen -p new.txt -o out1.bin out2.bin  
md5collgen: command not found  
[12/05/23]seed@VM:~/Desktop$ md5collgen -p new.txt -o out1.bin out2.bin  
MD5 collision generator v1.5  
by Marc Stevens (http://www.win.tue.nl/hashclash/)  
  
Using output filenames: 'out1.bin' and 'out2.bin'  
Using prefixfile: 'new.txt'  
Using initial value: e8443d09b3c18960ae4d2e4047a5f783  
  
Generating first block: .  
Generating second block: W...  
Running time: 0.749099 s  
[12/05/23]seed@VM:~/Desktop$ diff out1.bin out2.bin  
Binary files out1.bin and out2.bin differ  
[12/05/23]seed@VM:~/Desktop$ md5sum out1.bin  
bc77ebd69abc8d1c607ddb543a3a4aef out1.bin  
[12/05/23]seed@VM:~/Desktop$ md5sum out2.bin
```

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.

Text Editor	
00000000	68 79 79 0A 00 hyy.....
00000012	00
00000024	00
00000036	00
00000048	FC AF D1 3E 28 06 1B 91 52 B3 25 96 47 49 07 CC 0C 06 ...>(..R.%
0000005a	84 BD 0E 36 9D E3 85 B2 ED D9 6E E9 49 79 E7 EB C6 51 ...6.....I
0000006c	ED 41 0B 57 9B 22 40 EA D6 62 6D 0A 67 3F 9A 74 7C 20 .A.W."@..br
0000007e	B8 7F A0 70 DC F9 D8 B1 03 DF 4F 3C EC 0D F5 11 EA 53 ...p.....C
00000090	23 86 B7 2F 44 9C 12 18 AD B5 F0 C6 2A 60 68 C7 D7 33 #./D.....
000000a2	D8 67 A7 08 EE 1B F7 3F 86 92 C2 5B 1D 47 8E 94 8C 5F .g.....?...
000000b4	7C F1 6A 2C 4F 53 A4 D8 98 A3 DC 4A .j,OS.....

Figure 20

Figure 21

out1.bin		out2.bin	
00000000	68 79 20 74 68 65 72 65 20 69 20 61 6D 20 77 6F 72 6B	hy there i	00000000 68 79 20 74 68 65 72 65 20 69 20 61 6D 20 77 6F 72 6B
00000012	69 6E 67 20 6F 6E 20 74 68 69 73 20 61 73 73 69 67 6E	ing on thi	00000012 69 6E 67 20 6F 6E 20 74 68 69 73 20 61 73 73 69 67 6E
00000024	65 6D 65 6E 20 6F 66 20 70 72 69 76 61 63 79 20 61 6E	emen of pr	00000024 65 6D 65 6E 20 6F 66 20 70 72 69 76 61 63 79 20 61 6E
00000036	64 20 73 65 63 75 72 69 74 0A 0B 2B 21 C0 25 93 0C 20	d securit.	00000036 64 20 73 65 63 75 72 69 74 0A 0B 2B 21 C0 25 93 0C 20
00000048	5F D6 5D 91 D2 E2 0E 6B 97 D4 43 FE CE 8F 4C D4 15 CA	_...]....k..	00000048 5F D6 5D 91 D2 E2 0E 6B 97 D4 43 FE CE 8F 4C D4 15 CA
0000005a	77 09 2C 2A 17 C0 5D C5 10 C8 EE 81 09 72 EB 2B D7 24	w.,*...]....	0000005a 77 09 2C 2A 17 C0 5D C5 10 C8 EE 81 09 72 EB 2B D7 24
0000006c	8E 95 08 62 80 95 DF F5 87 CE 76 70 E4 1C 4E 87 99 A4	...b.....	0000006c 8E 15 09 62 80 95 DF F5 87 CE 76 70 E4 1C 4E 87 99 A4
0000007e	94 AA C0 11 F4 A5 79 56 93 B7 FD C6 F3 30 23 43 C1 C3yV..	0000007e 94 AA C0 11 F4 A5 79 56 93 B7 FD C6 F3 30 23 43 C1 C3
00000090	A4 9E 63 07 F7 65 46 E6 CB 1A 76 4D EC 46 FE 6D 06 45	...c..eF...	00000090 A4 9E 63 87 F7 65 46 E6 CB 1A 76 4D EC 46 FE 6D 06 45
000000a2	9E D7 F4 0B FC 12 C8 83 40 F7 D5 98 61 EA 37 49 16 E8@.	000000a2 9E D7 F4 0B FC 12 C8 83 40 F7 D5 18 61 EA 37 49 16 E8
000000b4	8C D1 98 6F 74 9C 6D 56 72 B4 99 A4	...ot.mVr.	000000b4 8C D1 98 6F 74 9C 6D 56 72 B4 99 A4

Figure 22

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.

out1.bin	
00000000	68 79 20 74 68 65 72 65 20 69 20 61 6D 20 77 6F 72 6B
00000012	69 6E 67 20 6F 6E 20 74 68 69 73 20 61 73 73 69 67 6E
00000024	65 6D 65 6E 20 6F 66 20 70 72 69 76 61 63 79 20 61 6E
00000036	64 20 73 65 63 75 72 69 74 0A 0B 2B 21 C0 25 93 0C 20
00000048	5F D6 5D 91 D2 E2 0E 6B 97 D4 43 FE CE 8F 4C D4 15 CA
0000005a	77 09 2C 2A 17 C0 5D C5 10 C8 EE 81 09 72 EB 2B D7 24
0000006c	8E 95 08 62 80 95 DF F5 87 CE 76 70 E4 1C 4E 87 99 A4
0000007e	94 AA C0 11 F4 A5 79 56 93 B7 FD C6 F3 30 23 43 C1 C3
00000090	A4 9E 63 07 F7 65 46 E6 CB 1A 76 4D EC 46 FE 6D 06 45
000000a2	9E D7 F4 0B FC 12 C8 83 40 F7 D5 98 61 EA 37 49 16 E8
000000b4	8C D1 98 6F 74 9C 6D 56 72 B4 99 A4

out1.bin	
Signed 8 bit:	104
Unsigned 8 bit:	104
Signed 16 bit:	26745
Unsigned 16 bit:	26745
<input type="checkbox"/> Show little endian decoding	<input type="checkbox"/> Show unsigned as hexadecimal
Offset: 0x0 / 0xbff	Selection: None

Figure 24

Signed 8 bit:	104	Signed 32 bit:	1752768628	Hexadecimal:	68 79 20 74
Unsigned 8 bit:	104	Unsigned 32 bit:	1752768628	Decimal:	104 121 032 11
Signed 16 bit:	26745	Float 32 bit:	4.705872E+24	Octal:	150 171 040 16
Unsigned 16 bit:	26745	Float 64 bit:	1.83423079226498E+195	Binary:	01101000 0111
<input type="checkbox"/> Show little endian decoding	<input type="checkbox"/> Show unsigned as hexadecimal	Offset: 0x0 / 0xbff	Selection: None	ASCII Text:	hyt

Figure 23

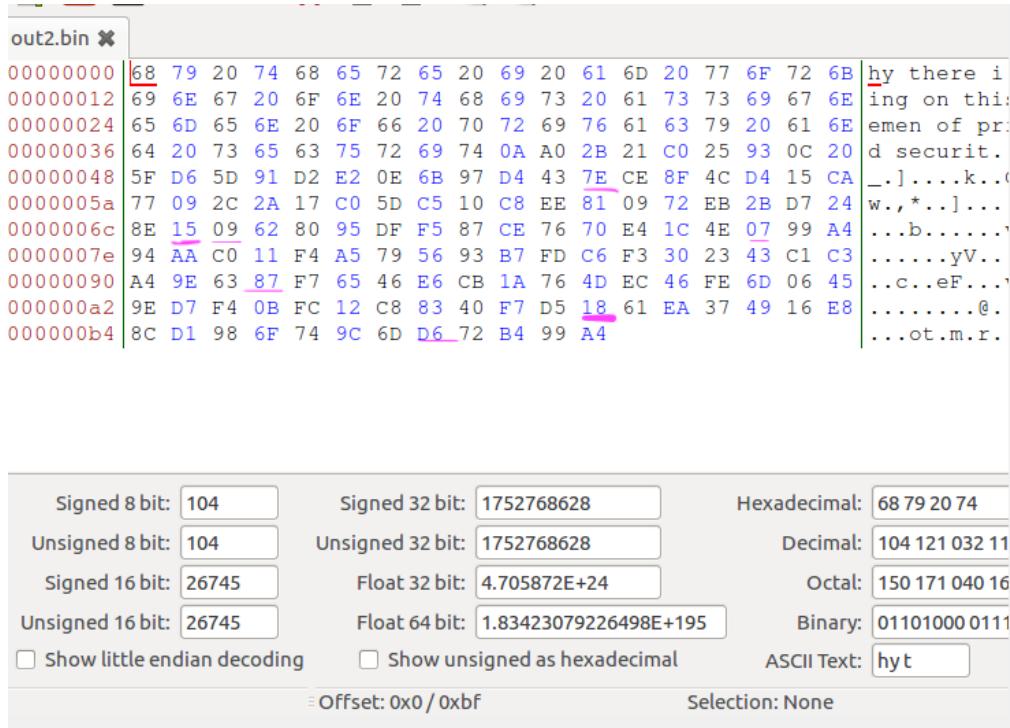


Figure 25

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 (α) mod q . Let A, B, C and D wish to exchange a key, the following operation:

$$Y_A = \alpha^{XA} \text{ mod } q$$

$$Y_B = \alpha^{XB} \text{ mod } q$$

$$Y_C = \alpha^{XC} \text{ mod } q$$

$$Y_D = \alpha^{XD} \text{ mod } q$$

Then, they are now able to calculate the common secret key:

A calculates $K = (Y_{BCD})^{XA} \text{ mod } q$

B calculates $K = (Y_{ACD})^{XB} \text{ mod } q$

C calculates $K = (Y_{ABD})^{XC} \text{ mod } q$

D calculates $K = (Y_{ABC})^{XD} \text{ mod } q$

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

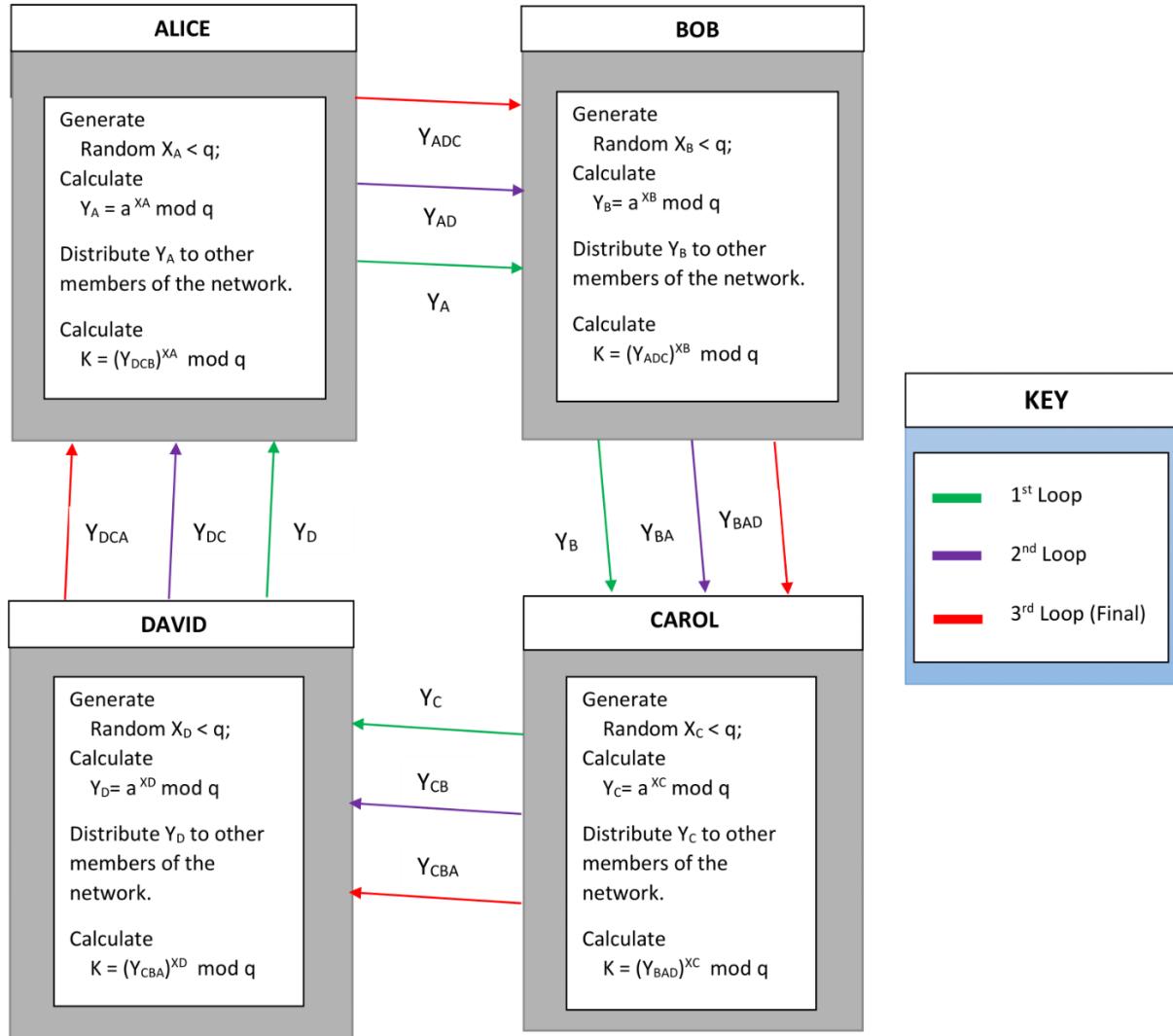


Figure 26

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"

```
(base) amishasherekar@Amishas-MacBook-Air pri2 % java FourPartyDiffieHellman
ALICE: Generate DH keypair ...
BOB: Generate DH keypair ...
CAROL: Generate DH keypair ...
DAVID: Generate DH keypair ...
ALICE: Initialize ...
BOB: Initialize ...
CAROL: Initialize ...
DAVID: Initialize ...
Alice secret: 04:6A:4B:1C:EA:F2:A6:A4:5B:CD:1B:32:8A:D6:8F:2B:92:A5:BF:C3:9F:F5:E4:73:2A:0C:0D:6B:DB:62:C1:A1:1D:92:8B:6B:8C:05:06:DD:65:B4:3E:84:EC:01:2C:92:27:2B:FE:21:09:3E:F7:59:34:D3:B9:2A:93:19:77:BB:CD:D5:BB:D7:F9:32:DB:52:45:5D:9C:17:1A:C7:1C:5D:CB:6C:C2:FB:2B:E6:47:0B:79:C6:E4:B7:E9:8F:78:A7:D8:71:4A:62:D5:12:7C:54:67:3A:FF:1E:C9:AD:5E:ED:57:E2:E1:0C:78:59:7D:82:52:2B:43:48:A0:84:8D:14:24:2A:DA:52:2A:BE:21:CC:9F:F3:09:6B:2E:3E:0A:D4:E9:A3:FD:C9:45:7C:CC:C0:47:F3:0E:04:FE:30:5E:6B:F3:2A:EE:FE:64:B9:37:DF:43:7D:D5:47:9C:02:AD:F0:76:C0:53:F2:F6:04:5A:20:C7:A5:71:5D:ED:36:2B:BC:AA:CE:BC:5D:2E:6E:DF:81:B4:55:0C:A9:7E:43:8F:9C:F8:D9:6B:14:0D:5F:F1:CB:8D:74:A5:27:7D:A5:E4:9E:E7:0F:1D:B5:75:D1:5A:C3:5F:4C:D2:7A:D2:F5:01:D2:2C:0F:F8:1F:E0:BE:3F:0B:80:13:D2:CC:BF:76:4E:D1
Bob secret: 04:6A:4B:1C:EA:F2:A6:A4:5B:CD:1B:32:8A:D6:8F:2B:92:A5:BF:C3:9F:F5:E4:73:2A:0C:0D:6B:DB:62:C1:A1:1D:92:8B:6B:8C:05:06:DD:65:B4:3E:84:EC:01:2C:92:27:2B:FE:21:09:3E:F7:59:34:D3:B9:2A:93:19:77:BB:CD:D5:BB:D7:F9:32:DB:52:45:5D:9C:17:1A:C7:1C:5D:CB:6C:C2:FB:2B:E6:47:0B:79:C6:E4:B7:E9:8F:78:A7:D8:71:4A:62:D5:12:7C:54:67:3A:FF:1E:C9:AD:5E:ED:57:E2:E1:0C:78:59:7D:82:52:2B:43:48:A0:84:8D:14:24:2A:DA:52:2A:BE:21:CC:9F:F3:09:6B:2E:3E:0A:D4:E9:A3:FD:C9:45:7C:CC:C0:47:F3:0E:04:FE:30:5E:6B:F3:2A:EE:FE:64:B9:37:DF:43:7D:D5:47:9C:02:AD:F0:76:C0:53:F2:F6:04:5A:20:C7:A5:71:5D:ED:36:2B:BC:AA:CE:BC:5D:2E:6E:DF:81:B4:55:0C:A9:7E:43:8F:9C:F8:D9:6B:14:0D:5F:F1:CB:8D:74:A5:27:7D:A5:E4:9E:E7:0F:1D:B5:75:D1:5A:C3:5F:4C:D2:7A:D2:F5:01:D2:2C:0F:F8:1F:E0:BE:3F:0B:80:13:D2:CC:BF:76:4E:D1
Carol secret: 04:6A:4B:1C:EA:F2:A6:A4:5B:CD:1B:32:8A:D6:8F:2B:92:A5:BF:C3:9F:F5:E4:73:2A:0C:0D:6B:DB:62:C1:A1:1D:92:8B:6B:8C:05:06:DD:65:B4:3E:84:EC:01:2C:92:27:2B:FE:21:09:3E:F7:59:34:D3:B9:2A:93:19:77:BB:CD:D5:BB:D7:F9:32:DB:52:45:5D:9C:17:1A:C7:1C:5D:CB:6C:C2:FB:2B:E6:47:0B:79:C6:E4:B7:E9:8F:78:A7:D8:71:4A:62:D5:12:7C:54:67:3A:FF:1E:C9:AD:5E:ED:57:E2:E1:0C:78:59:7D:82:52:2B:43:48:A0:84:8D:14:24:2A:DA:52:2A:BE:21:CC:9F:F3:09:6B:2E:3E:0A:D4:E9:A3:FD:C9:45:7C:CC:C0:47:F3:0E:04:FE:30:5E:6B:F3:2A:EE:FE:64:B9:37:DF:43:7D:D5:47:9C:02:AD:F0:76:C0:53:F2:F6:04:5A:20:C7:A5:71:5D:ED:36:2B:BC:AA:CE:BC:5D:2E:6E:DF:81:B4:55:0C:A9:7E:43:8F:9C:F8:D9:6B:14:0D:5F:F1:CB:8D:74:A5:27:7D:A5:E4:9E:E7:0F:1D:B5:75:D1:5A:C3:5F:4C:D2:7A:D2:F5:01:D2:2C:0F:F8:1F:E0:BE:3F:0B:80:13:D2:CC:BF:76:4E:D1
David secret: 04:6A:4B:1C:EA:F2:A6:A4:5B:CD:1B:32:8A:D6:8F:2B:92:A5:BF:C3:9F:F5:E4:73:2A:0C:0D:6B:DB:62:C1:A1:1D:92:8B:6B:8C:05:06:DD:65:B4:3E:84:EC:01:2C:92:27:2B:FE:21:09:3E:F7:59:34:D3:B9:2A:93:19:77:BB:CD:D5:BB:D7:F9:32:DB:52:45:5D:9C:17:1A:C7:1C:5D:CB:6C:C2:FB:2B:E6:47:0B:79:C6:E4:B7:E9:8F:78:A7:D8:71:4A:62:D5:12:7C:54:67:3A:FF:1E:C9:AD:5E:ED:57:E2:E1:0C:78:59:7D:82:52:2B:43:48:A0:84:8D:14:24:2A:DA:52:2A:BE:21:CC:9F:F3:09:6B:2E:3E:0A:D4:E9:A3:FD:C9:45:7C:CC:C0:47:F3:0E:04:FE:30:5E:6B:F3:2A:EE:FE:64:B9:37:DF:43:7D:D5:47:9C:02:AD:F0:76:C0:53:F2:F6:04:5A:20:C7:A5:71:5D:ED:36:2B:BC:AA:CE:BC:5D:2E:6E:DF:81:B4:55:0C:A9:7E:43:8F:9C:F8:D9:6B:14:0D:5F:F1:CB:8D:74:A5:27:7D:A5:E4:9E:E7:0F:1D:B5:75:D1:5A:C3:5F:4C:D2:7A:D2:F5:01:D2:2C:0F:F8:1F:E0:BE:3F:0B:80:13:D2:CC:BF:76:4E:D1
Alice and Bob are the same
Bob and Carol are the same
Carol and David are the same
(base) amishasherekar@Amishas-MacBook-Air pri2 %
```

Figure 27