



COMP 522 Privacy and Security

Assignment 2 Report

Submitted To: Alexei Lisitsa, Emmanouil Pitsikalis
Submitted By: Balkrishna Bhatt (201673048)

Abstract:

This report compares four ways for message authentication, including hash functions, the RSA + SHA1 method, the HMAC-SHA256 method, and the DSA method. I have also highlighted the practical elements of Diffie-Hellman key exchange algorithms. In addition, I create and implement a Diffie-Hellman Key exchange protocol variation that enables four parties to communicate secret keys in a network.

I. Comparison of methods for message authentication

- **Hash functions:** Hash functions are mathematical operations that "map" or convert a collection of data into a bit string with a certain length, commonly referred to as the "hash value." Examples of such functions are SHA-1, SHA-224, SHA-256, SHA3-256, SHA-512 and many more [1]. It is a one-way function for which it is practically infeasible to invert or reverse the computation [2]. It's also a process that takes plaintext data of any size and converts it into a unique ciphertext of a specific length.

Message authentication is not provided by a hash function on its own. To achieve authentication, a secret key must be combined in some way with the hash function. By definition, a MAC algorithm generates an integrity check code (MAC) that serves as data authentication using a secret key.

Hash values are also not useful for verifying the non-repudiation of the data. Because it calculates and converts the data into respective hash blocks (refer to the below-illustrated figure 2.1). So, non-repudiation ensures that no party can deny that it sent or received a message via encryption and/or digital signatures or approved some information [3].

But **Hash values are also useful for verifying the integrity of data sent through insecure channels.** With a good hash function, even a 1-bit change in a message will produce a different hash (on average, half of the bits change). With digital signatures, a message is hashed and then the hash itself is signed. [4]

Diagram:

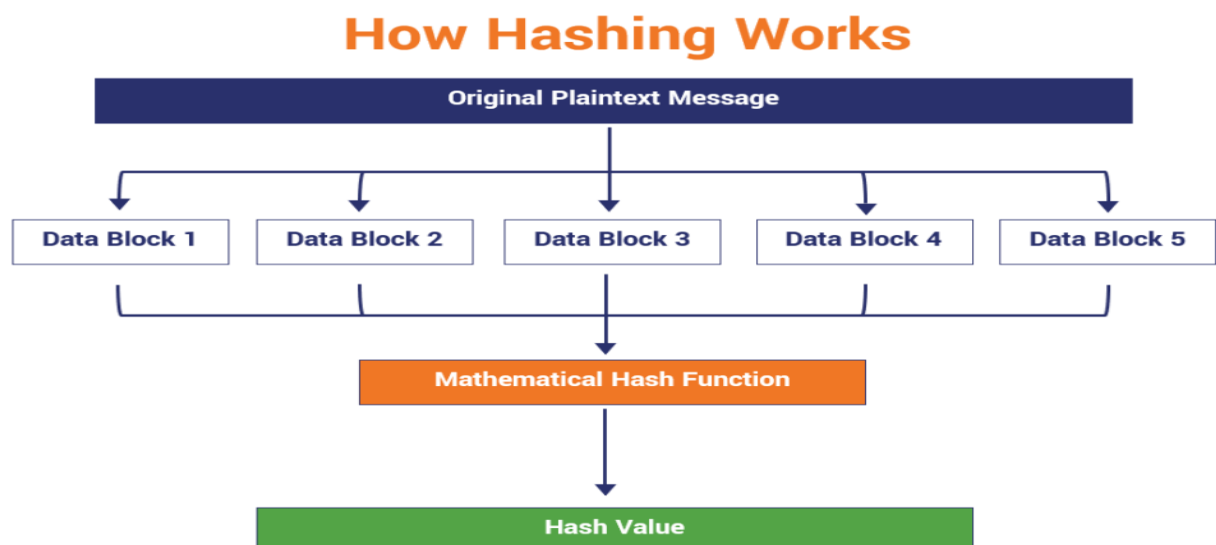


Figure 2.1: Hash function's working model

Code Snippet:

```
// Java program to calculate SHA-512 hash value

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class SHABK {
    public static String encryptThisString(String input)
    {
        try {
            // getInstance() method is called with algorithm SHA-512
            MessageDigest md = MessageDigest.getInstance("SHA-512");

            // digest() method is called
            // to calculate message digest of the input string
            // returned as array of byte
            byte[] messageDigest = md.digest(input.getBytes());

            // Convert byte array into signum representation
            BigInteger no = new BigInteger(1, messageDigest);

            // Convert message digest into hex value
            String hashtext = no.toString(16);

            // Add preceding 0s to make it 32 bit
            while (hashtext.length() < 32) {
                hashtext = "0" + hashtext;
            }

            // return the HashText
            return hashtext;
        }

        // For specifying wrong message digest algorithms
        catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    // Driver code
    public static void main(String args[]) throws NoSuchAlgorithmException
    {
        System.out.println("HashCode Generated by SHA-512 for: ");

        String s1 = "Balkrishna Bhatt";
        System.out.println("\n" + s1 + " : " + encryptThisString(s1));

        String s2 = "John Wick";
        System.out.println("\n" + s2 + " : " + encryptThisString(s2));
    }
}
```

Result:

```
HashCode Generated by SHA-512 for:
```

```
Balkrishna Bhatt :
```

```
4ec7947d0aca9eddd851d0b3d17d998dcff8c9068d7f0f81f28b1d7911dce59932e4414d235b6b8ae93656874a1620aa8bb350e0ea10ed846e25fc9bc1588534
```

```
John Wick : 160f8fffd3724a910e163bcc2c273a04bcd3eeb8037f40d658392c039c3fe6f24bd83f97b0a91f1869d54e9e1e327773ebd90ec4f60160139ed45e5243ee336
```

```
** Process exited - Return Code: 0 **
```

- **DSA method:** Digital signatures are used for this identification. Authentication of the documents means being aware of who created them and that they did not interfere during their transmission. These signatures are created using certain algorithms. Digital Signature is a technique that binds a person or entity to digital data. This binding can be independently verified by the receiver as well as any third party. A digital signature is a cryptographic value that is calculated from the data and a secret key is known only to the signer [5].

The DSA technique, which is the industry standard for digital signatures, is based on the public-key cryptosystems principle and the algebraic features of discrete logarithm problems and modular exponentiations. Digital signatures work on the principle of two mutually authenticating cryptographic keys. Signatures are based on public/private key pairs.

Algorithm:**DSA Signature Generation -**

INPUT: Domain parameters (p,q,g); signer's private key a; message-to-be-signed, M; a secure hash function Hash() with output of length |q|.

OUTPUT: Signature (r,s).

Choose a random k in the range [1,q-1].

Compute $X = g^k \text{ mod } p$ and $r = X \text{ mod } q$. If $r=0$ (unlikely) then go to step 1.

Compute $k^{-1} \text{ mod } q$.

Compute $h = \text{Hash}(M)$ interpreted as an integer in the range $0 \leq h < q$.

Compute $s = k^{-1}(h + ar) \text{ mod } q$. If $s=0$ (unlikely) then go to step 1.

Return (r,s).

DSA Signature Verification –

INPUT: Domain parameters (p, q, g) ; signer's public key A ; signed-message, M ; a secure hash function $\text{Hash}()$ with output of length $|q|$; signature (r, s) to be verified.

OUTPUT: "Accept" or "Reject".

Verify that r and s are in the range $[1, q-1]$. If not then return "Reject" and stop.

Compute $w = s^{-1} \bmod q$.

Compute $h = \text{Hash}(M)$ interpreted as an integer in the range $0 \leq h < q$.

Compute $u_1 = hw \bmod q$ and $u_2 = rw \bmod q$.

Compute $X = gu_1Au_2 \bmod p$ and $v = X \bmod q$.

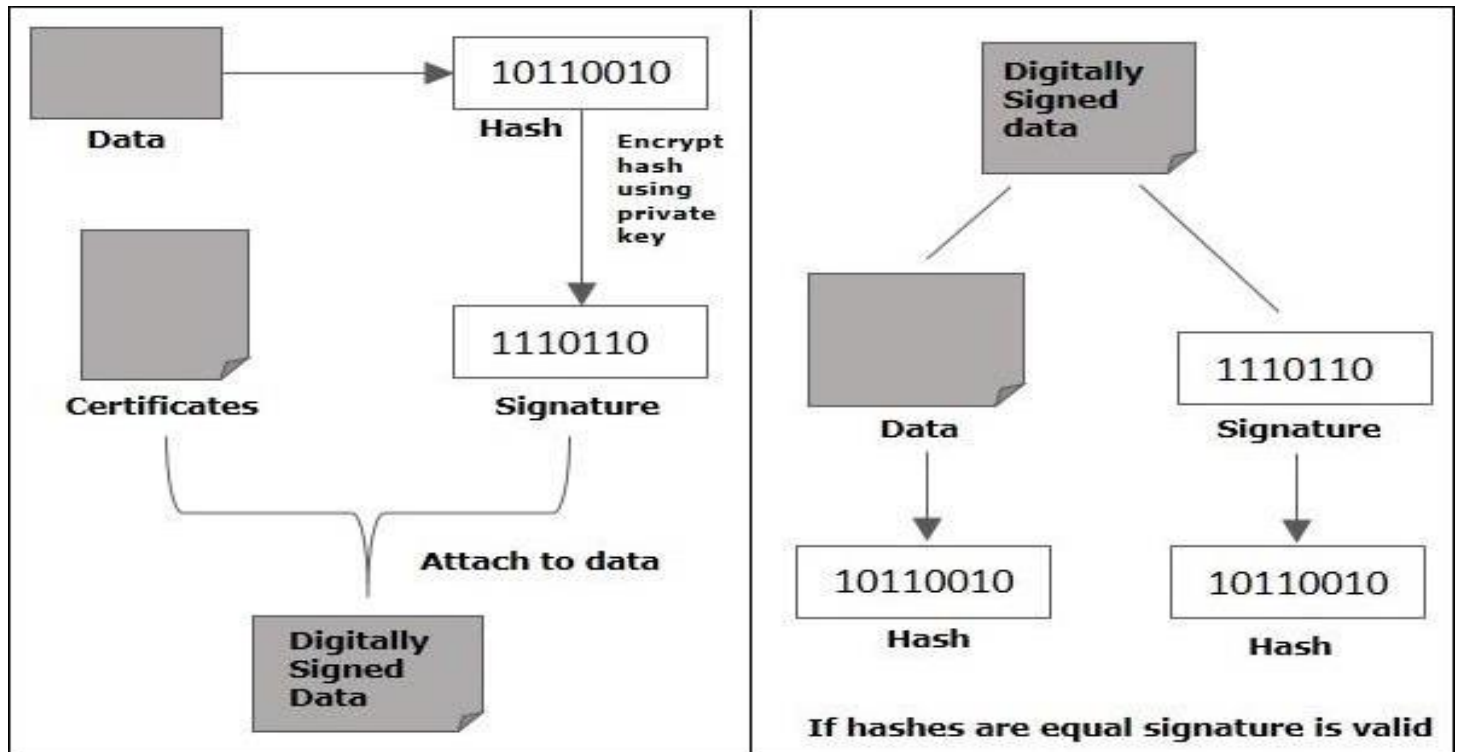
If $v = r$ then return "Accept" otherwise return "Reject".

Out of all cryptographic primitives, the digital signature using public key cryptography is considered as very important and useful tool to achieve information security. **The digital signature also offers data integrity and message authentication in addition to non-repudiation of the message** [6]. Let's take a quick look at how the digital signature does this.

Message authentication: When the verifier validates the digital signature using the public key of a sender, he is assured that the signature has been created only by a sender who possesses the corresponding secret private key and no one else.

Data Integrity: In the event that an attacker has access to the data and modifies it, the digital signature verification at the receiver end fails. The hash of modified data and the output provided by the verification algorithm will not match. Hence, the receiver can safely deny the message, assuming that data integrity has been breached.

Non-repudiation: Since it is assumed that only the signer has knowledge of the signature key, he can only create a unique signature on given data. Thus, the receiver can present the data and the digital signature to a third party as evidence if any dispute arises in the future.

Diagram:**Figure 2.2: Digital Signature Algorithm****Code Snippet:**

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.Signature;
import java.util.Scanner;

public class CreatingDigitalSignature {
    public static void main(String args[]) throws Exception {
        //Accepting text from user
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter some text");
        String msg = sc.nextLine();

        //Creating KeyPair generator object
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");

        //Initializing the key pair generator
        keyPairGen.initialize(2048);

        //Generate the pair of keys
        KeyPair pair = keyPairGen.generateKeyPair();

        //Getting the private key from the key pair
```

```

PrivateKey privKey = pair.getPrivate();

//Creating a Signature object
Signature sign = Signature.getInstance("SHA256withDSA");

//Initialize the signature
sign.initSign(privKey);
byte[] bytes = "msg".getBytes();

//Adding data to the signature
sign.update(bytes);

//Calculating the signature
byte[] signature = sign.sign();

//Printing the signature
System.out.println("Digital signature for given text: "+new String(signature,
"UTF8"));
}
}

```

Result:

```

Enter some text
Balkrishna Bhatt
Digital signature for given text: 0<[##]#3[##V##]>[J]E&[##]C[##]/\['[0]1-[##]  [##]a?[$O[##]9[##]([##]YD

** Process exited - Return Code: 0 **

```

- HMAC-SHA256:** As a Hash-based Message Authentication Code, the SHA-256 hash function is used to create the kind of keyed hash algorithm known as HMACSHA256 (HMAC). The HMAC procedure combines a secret key with the message data, performs a hash operation on the result, combines that hash value with the secret key once again, and then repeats the hash operation. The output hash has a length of 256 bits [7].

HMAC-SHA256 accepts keys of any size and produces a hash sequence 256 bits in length. An HMAC can be used to determine whether a message sent over an insecure channel has been tampered with, provided that the sender and receiver share a secret key. The sender computes the hash value for the original data and sends both the original data and hash value as a single message. The receiver recalculates the hash value on the received message and checks that the computed HMAC matches the transmitted HMAC.

HMAC is used to provide data integrity and authentication. **It doesn't provide non-repudiation,** because it involves using the key, which is shared by communicating entities.

HMAC (Hash-based Message Authentication Code) is a type of a message authentication code (MAC) **which is used for both data integrity and authentication.** It's a message authentication code obtained by running a cryptographic hash function (like MD5, SHA1, and SHA256) over the data (to be authenticated) and a shared secret key.

Formulation:

$$\text{HMAC}(K, m) = H((K' \oplus \text{opad}) \parallel H((K' \oplus \text{ipad}) \parallel m))$$

$$K' = \begin{cases} H(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

Diagram:

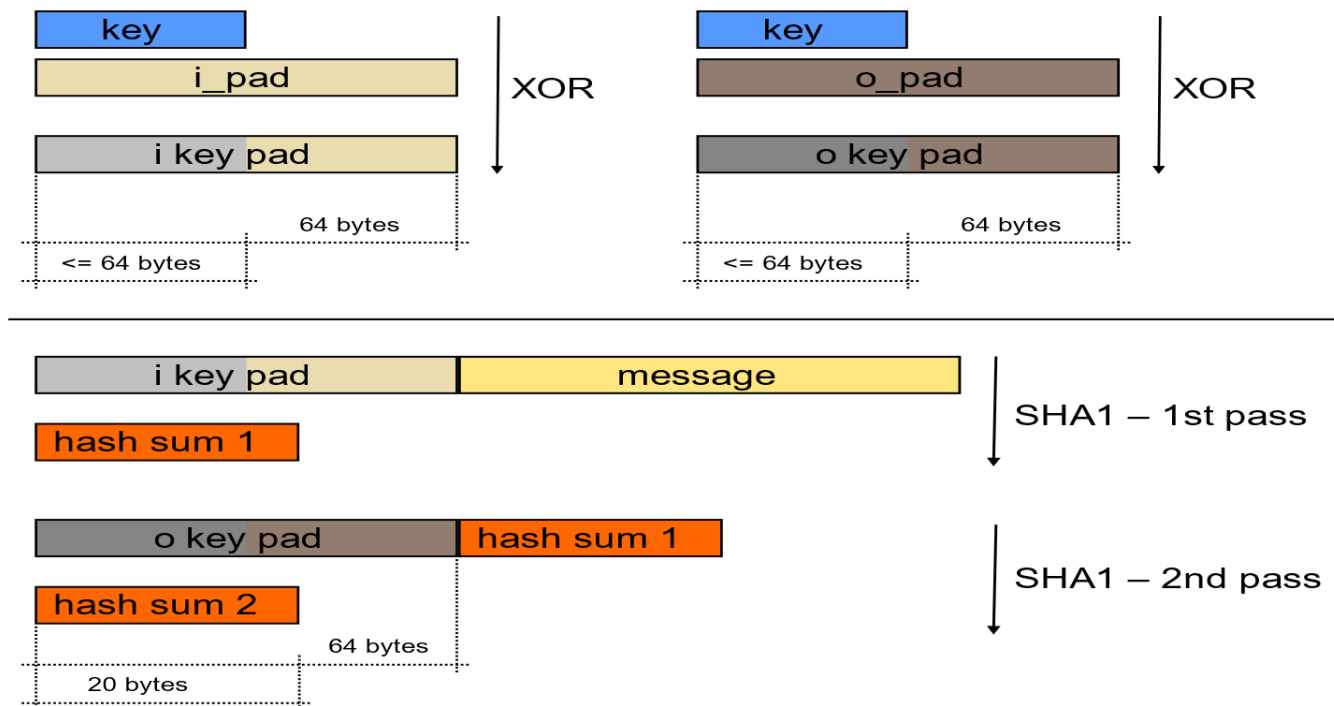


Figure 2.3: HMAC-SHA256 explanation

Code Snippet:

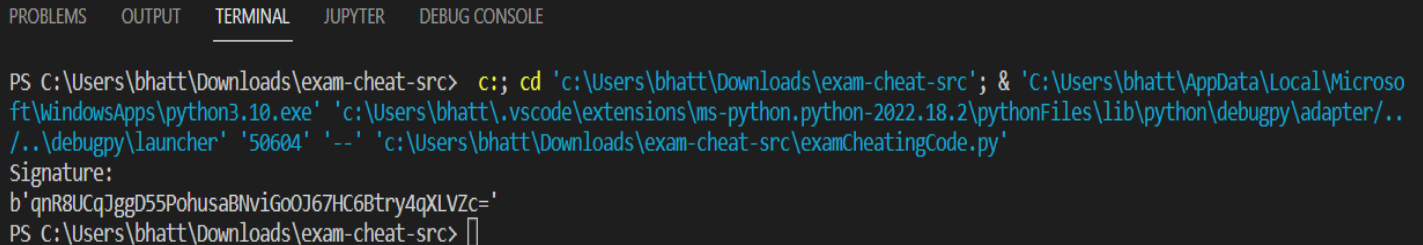
```
import hashlib
import hmac
import base64

message = bytes('Message', 'utf-8')
secret = bytes('secret', 'utf-8')
```



```
signature = base64.b64encode(hmac.new(secret, message,
digestmod=hashlib.sha256).digest())
print("Signature: ")
print(signature)
```

Result:



The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, TERMINAL, JUPYTER, and DEBUG CONSOLE. The TERMINAL tab is active, displaying the following commands and output:

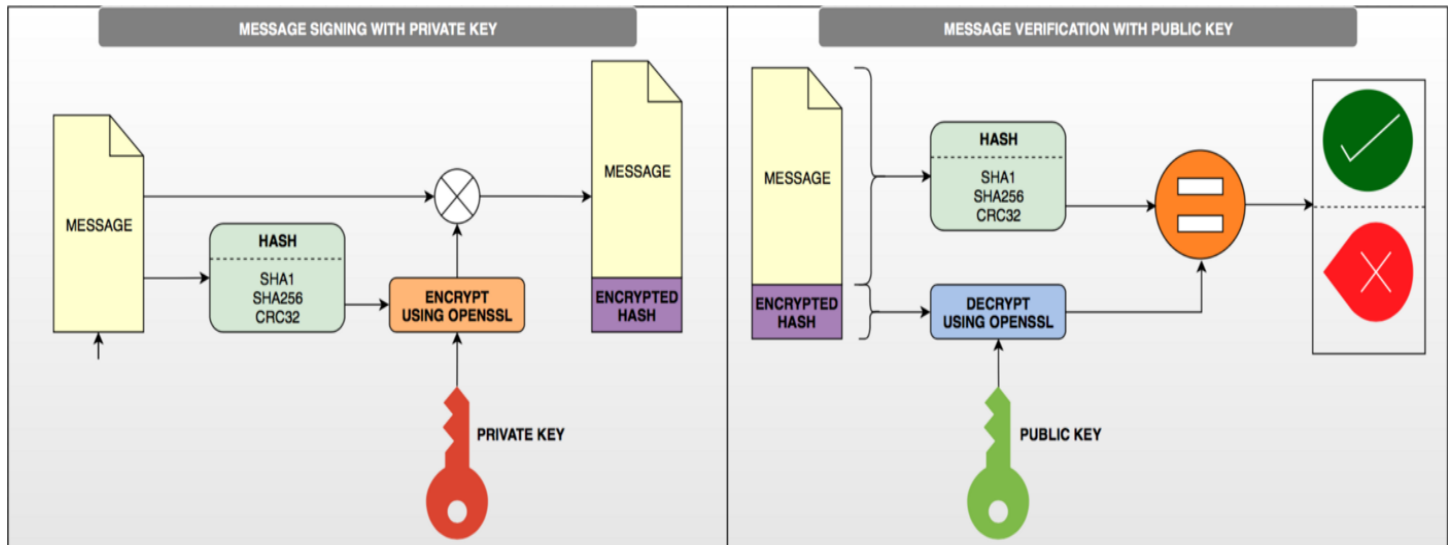
```
PS C:\Users\bhatt\Downloads\exam-cheat-src> c:: cd 'c:\Users\bhatt\Downloads\exam-cheat-src'; & 'C:\Users\bhatt\AppData\Local\Microsoft\WindowsApps\python3.10.exe' 'c:\Users\bhatt\.vscode\extensions\ms-python.python-2022.18.2\pythonFiles\lib\python\debugpy\adapter\..\debugpy\launcher' '50604' '--' 'c:\Users\bhatt\Downloads\exam-cheat-src\examCheatingCode.py'
Signature:
b'qnR8UCqJggD55PohusaBNviGo0J67HC6Btry4qXLVZc='
PS C:\Users\bhatt\Downloads\exam-cheat-src> 
```

- **RSA + SHA1 method:** The RSA algorithm (Rivest-Shamir-Adleman) is the basis of a cryptosystem -- a suite of cryptographic algorithms that are used for specific security services or purposes -- which enables public key encryption and is widely used to secure sensitive data, particularly when it is being sent over an insecure network such as the internet. RSA was first publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman of the Massachusetts Institute of Technology [8].

RSA is the signing (not encrypting, despite what the text says) algorithm, and it operates on a hash of the content to be signed. SHA1 is the hashing algorithm (it produces a short, one-way non-reversible version of the full certificate) that is used to produce the string that RSA then signs [9].

Authentication and non-repudiation can be handled by the RSA algorithm. When the exchange of keys is used AES algorithm, the key would be encrypted and decrypted by the RSA algorithm for authentication and non-repudiation. And SHA-1 is hashing algorithm which validates the message integrity. **So the combination of RSA and SHA can also use to verify message integrity.**

RSA derives its security from the difficulty of factoring large integers that are the product of two large prime numbers. Multiplying these two numbers is easy but determining the original prime numbers from the total -- or factoring -- is considered infeasible due to the time it would take using even today's supercomputers.

Diagram:**Figure 2.4: RSA + SHA1 explanation****Code Snippet:**

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

public class initMac {

    // Generate new key
    KeyPair keyPair = KeyPairGenerator.getInstance("RSA").generateKeyPair();
    PrivateKey privateKey = keyPair.getPrivate();
    String plaintext = "This is the message being signed";

    // Compute signature
    Signature instance = Signature.getInstance("SHA1withRSA");
    instance.initSign(privateKey);
    instance.update(plaintext.getBytes());
    byte[] signature = instance.sign();

    // Compute digest
    MessageDigest sha1 = MessageDigest.getInstance("SHA1");
    byte[] digest = sha1.digest(plaintext.getBytes());

    // Encrypt digest
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, privateKey);
    byte[] cipherText = cipher.doFinal(digest);

    // Display results
    System.out.println("Input data: " + plaintext);
    System.out.println("Digest: " + bytes2String(digest));
    System.out.println("Cipher text: " + bytes2String(cipherText));
    System.out.println("Signature: " + bytes2String(signature));
}
```

```
/*
 * 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(":");
        }
    }
    return buf.toString();
}
```

Result:

Input data: This is the message being signed

Digest: 62b0a9ef15461c82766fb5bdaae9edbe4ac2e067

Cipher text: 057dc0d2f7f54acc95d3cf5cba9f944619394711003bdd12...

Signature: 7177c74bbbb871cc0af92e30d2808ebae146f25d3fd8ba1622...

II. Diffie-Hellman Key Exchange for Four parties

One of the most significant advancements in public-key cryptography was the Diffie-Hellman key exchange, which is still extensively used in a variety of modern security protocols. It is a mathematical approach for safely transferring cryptographic keys over a public channel and was one of the earliest public-key protocols [10].

The Diffie-Hellman key exchange's primary goal is the secure development of shared secrets that may be used to generate keys. Then, information may be sent safely using symmetric-key methods using these keys. The majority of the data is often encrypted using symmetric algorithms since they are more effective than public key methods [11]. The Diffie-Hellman scheme actually makes calculations based on exceptionally large prime numbers, then sends them across. Security protocols including Transport Layer Security (TLS), Secure Shell (SSH), and IP Security frequently use Diffie-Hellman key exchange (IPsec). For instance, the encryption technique is applied to key creation and key rotation in IPsec [12].

Security Vulnerability:

Despite the fact that Diffie-Hellman has proven to be extremely resistant to attack when used correctly (not bad for an algorithm developed more than 50 years ago), there is one emerging technology that may eventually

overthrow it: quantum computers. The users of Diffie-Hellman could be vulnerable to a handful of attacks that depend on exploiting the computing environment, known as "side-channel attacks."

Design and Implementation of The Diffie-Hellman key exchanges for four Parties:

The multiplicative group of integers modulo p , where p is prime, and g is a primitive root modulo p are used in the protocol's initial and most basic implementation. In order to guarantee that the resultant shared secret can take on any value between 1 and $p-1$, these two values were selected in this manner.

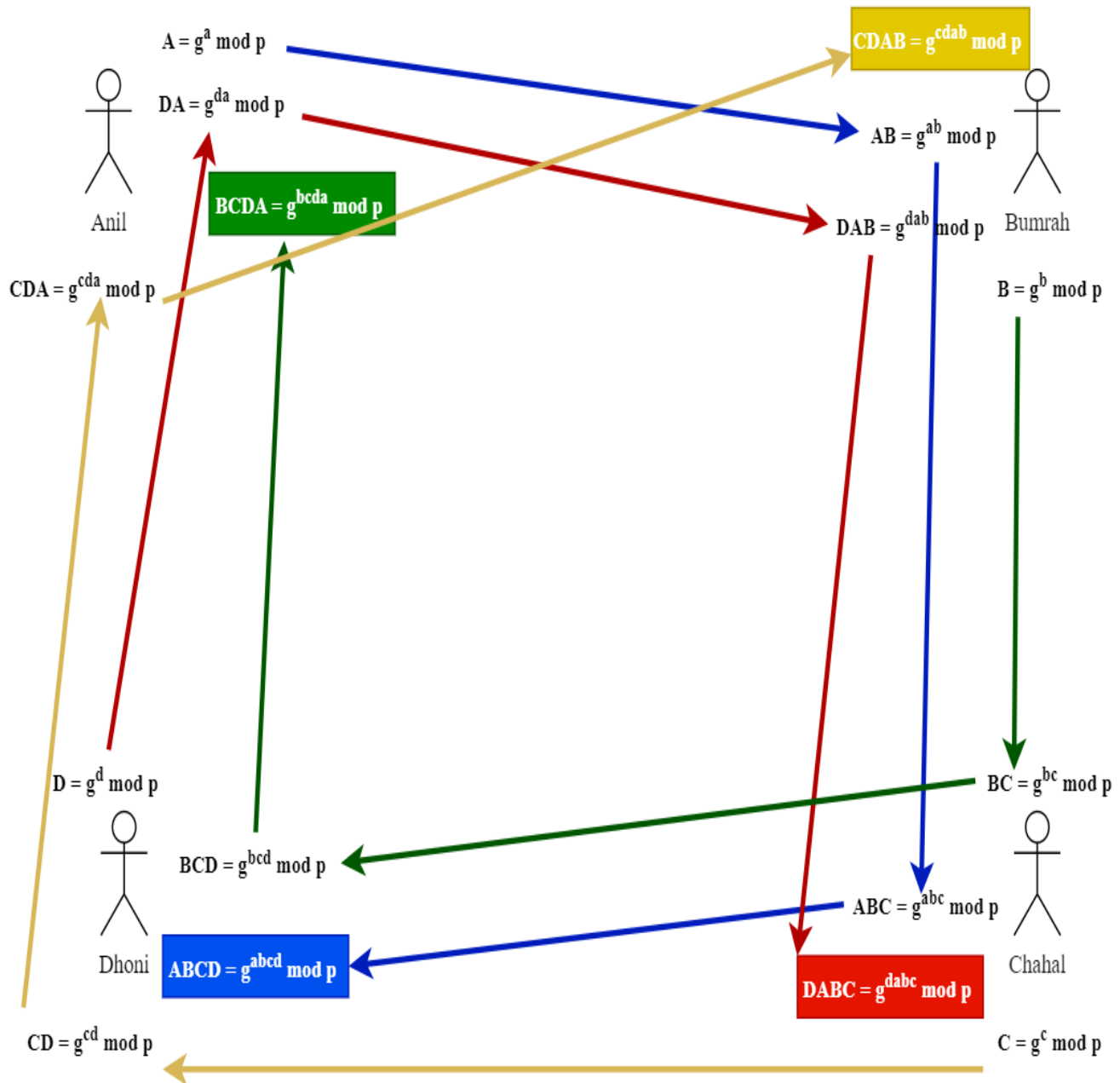
Terminology:

- g = public (primitive root) base, known to Anil, Bumrah, Chahal and Dhoni.
- p = public (prime) modulus, known to Anil, Bumrah, Chahal and Dhoni.
- a = Anil's private key, known only to Anil.
- b = Bumrah's private key known only to Bumrah.
- c = Chahal's private key known only to Chahal.
- d = Dhoni's private key known only to Dhoni.
- A = Anil's public key, known to everyone.
- B = Bumrah's public key, known to everyone.
- C = Chahal's public key, known to everyone.
- D = Dhoni's public key, known to everyone.

Calculation:

So, the process begins by having the 4 parties, Anil, Bumrah, Chahal and Dhoni, publicly agree on an arbitrary starting key that does not need to be kept secret.

1. Anil computes $g^a \bmod p$ and sends it to Bumrah.
2. Bumrah computes $(g^a)^b \bmod p = g^{ab} \bmod p$ and sends it to Chahal.
3. Chahal computes $(g^{ab})^c \bmod p = g^{abc} \bmod p$ and sends it to Dhoni.
4. Dhoni computes $(g^{abc})^d \bmod p = g^{abcd} \bmod p$ and uses it as his secret.
5. Bumrah computes $g^b \bmod p$ and sends it to Chahal.
6. Chahal computes $(g^b)^c \bmod p = g^{bc} \bmod p$ and sends it to Dhoni.
7. Dhoni computes $(g^{bc})^d \bmod p = g^{bcd} \bmod p$ and sends it to Anil.
8. Anil computed $(g^{bcd})^a \bmod p = g^{bcda} \bmod p = g^{abcd} \bmod p$ uses it as his secret.
9. Chahal computes $g^c \bmod p$ and sends it to Dhoni.
10. Dhoni computes $(g^c)^d \bmod p = g^{cd} \bmod p$ sends it to Anil.
11. Anil computes $(g^{cd})^a \bmod p = g^{cda} \bmod p$ and sends it to Bumrah.
12. Bumrah computes $(g^{cda})^b \bmod p = g^{cdab} \bmod p = g^{abcd} \bmod p$ and uses it as his secret.
13. Dhoni computes $g^d \bmod p$ and sends it to Anil.
14. Anil computes $(g^d)^a \bmod p = g^{da} \bmod p$ and sends it to Bumrah.
15. Bumrah computes $(g^{da})^b \bmod p = g^{dab} \bmod p$ and sends it to Chahal.
16. Chahal computes $(g^{dab})^c \bmod p = g^{dabc} \bmod p = g^{abcd} \bmod p$ and uses it as his secret.

Diagram:**Figure 2.5: Diffie-Hellman key exchanges for four Parties****Code Snippet:**

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

```
import javax.crypto.interfaces.*;
/*
 * This program executes the Diffie-Hellman key agreement protocol between
 * 4 parties: Anil, Bumrah, Chahal and Dhoni using a shared 2048-bit DH parameter.
 */
public class DHKeyAgreement4 {
    private DHKeyAgreement4() {}
    public static void main(String argv[]) throws Exception {

        // Anil creates her own DH key pair with 2048-bit key size
        KeyPairGenerator AnilKpairGen = KeyPairGenerator.getInstance("DH");
        AnilKpairGen.initialize(2048);
        KeyPair AnilKpair = AnilKpairGen.generateKeyPair();
        // This DH parameters can also be constructed by creating a

        // DHParameterSpec object using agreed-upon values
        DHParameterSpec dhParamShared =
((DHPublicKey)AnilKpair.getPublic()).getParams();
        // Bumrah creates his own DH key pair using the same params
        KeyPairGenerator BumrahKpairGen = KeyPairGenerator.getInstance("DH");
        BumrahKpairGen.initialize(dhParamShared);
        KeyPair BumrahKpair = BumrahKpairGen.generateKeyPair();
        // Chahal creates her own DH key pair using the same params
        KeyPairGenerator ChahalKpairGen = KeyPairGenerator.getInstance("DH");
        ChahalKpairGen.initialize(dhParamShared);
        KeyPair ChahalKpair = ChahalKpairGen.generateKeyPair();
        // Chahal creates her own DH key pair using the same params
        KeyPairGenerator DhoniKpairGen = KeyPairGenerator.getInstance("DH");
        DhoniKpairGen.initialize(dhParamShared);
        KeyPair DhoniKpair = DhoniKpairGen.generateKeyPair();

        //Anil initialize
        KeyAgreement AnilKeyAgree = KeyAgreement.getInstance("DH");
        //Anil computes gA
        AnilKeyAgree.init(AnilKpair.getPrivate());

        //Bumrah initialize
        KeyAgreement BumrahKeyAgree = KeyAgreement.getInstance("DH");
        //Bumrah computes gB
        BumrahKeyAgree.init(BumrahKpair.getPrivate());

        //Chahal initialize
        KeyAgreement ChahalKeyAgree = KeyAgreement.getInstance("DH");
        //Chahal computes gC
        ChahalKeyAgree.init(ChahalKpair.getPrivate());

        //Dhoni initialize
        KeyAgreement DhoniKeyAgree = KeyAgreement.getInstance("DH");
        //Dhoni computes gD
        DhoniKeyAgree.init(DhoniKpair.getPrivate());

        //First Pass

        //Anil computes gDA
        Key gDA = AnilKeyAgree.doPhase(DhoniKpair.getPublic(), false);

        //Bumrah computes gAB
```

```
Key gAB = BumrahKeyAgree.doPhase(AnilKpair.getPublic(), false);

//Chahal computes gBC
Key gBC = ChahalKeyAgree.doPhase(BumrahKpair.getPublic(), false);

//Dhoni computes gCD
Key gCD = DhoniKeyAgree.doPhase(ChahalKpair.getPublic(), false);

//Second Pass

//Anil computes gCDA
Key gCDA = AnilKeyAgree.doPhase(gCD, false);

//Bumrah computes gDAB
Key gDAB = BumrahKeyAgree.doPhase(gDA, false);

//Chahal computes gABC
Key gABC = ChahalKeyAgree.doPhase(gAB, false);

//Dhoni computes gBCD
Key gBCD = DhoniKeyAgree.doPhase(gBC, false);

//Third Pass

//Anil computes gBCDA
Key gBCDA = AnilKeyAgree.doPhase(gBCD, true); //This is Anil's secret

//Bumrah computes gCDAB
Key gCDAB = BumrahKeyAgree.doPhase(gCDA, true); //This is Bumrah's secret

//Dhoni Computes gABCD
Key gABCD = DhoniKeyAgree.doPhase(gABC, true); //This is Dhoni's secret

//Chahal computes gDABC
Key gDABC = ChahalKeyAgree.doPhase(gDAB, true); //This is Chahal's secret

// Anil, Bumrah, Chahal and Dhoni compute their secrets
byte[] AnilSharedSecret = AnilKeyAgree.generateSecret();
System.out.println("Anil secret: " + toHexString(AnilSharedSecret));

byte[] BumrahSharedSecret = BumrahKeyAgree.generateSecret();
System.out.println("Bumrah secret: " + toHexString(BumrahSharedSecret));

byte[] ChahalSharedSecret = ChahalKeyAgree.generateSecret();
System.out.println("Chahal secret: " + toHexString(ChahalSharedSecret));

byte[] DhoniSharedSecret = DhoniKeyAgree.generateSecret();
System.out.println("Dhoni secret: " + toHexString(DhoniSharedSecret));

// Compare Anil and Bumrah
if (!java.util.Arrays.equals(AnilSharedSecret, BumrahSharedSecret))
    System.out.println("Anil and Bumrah differ"); //throw new
Exception("Anil and Bumrah differ");
else
```

```

        System.out.println("Anil and Bumrah are the same");
        // Compare Bumrah and Chahal
        if (!java.util.Arrays.equals(BumrahSharedSecret, ChahalSharedSecret))
            System.out.println("Bumrah and Chahal differ"); //throw new
Exception("Bumrah and Chahal differ");
        else
            System.out.println("Bumrah and Chahal are the same");
        //Compare Chahal and Dhoni
        if (!java.util.Arrays.equals(ChahalSharedSecret, DhoniSharedSecret))
            System.out.println("Chahal and Dhoni differ"); //throw new
Exception("Chahal and Dhoni differ");
        else
            System.out.println("Chahal and Dhoni are the same");
        //Compare Dhoni and Anil
        if (!java.util.Arrays.equals(DhoniSharedSecret, AnilSharedSecret))
            System.out.println("Dhoni and Anil differ"); //throw new
Exception("Dhoni and Anil differ");
        else
            System.out.println("Dhoni and Anil are the same");

    }

    /*
    * 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]);
    }

    /*
    * 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(":");
            }
        }
        return buf.toString();
    }
}

```


Result:

Anil secret:

E8:D8:71:18:D0:C9:84:F8:AD:DC:7D:CB:1C:E1:77:7C:12:EF:B0:9B:F0:ED:49:81:5A:8F:EB:42:16:01:E4:39:AB:4D:D2:07:28:C9:3F:AA:90:3C:23:BC:98:3F:A4:B6:2C:2B:D2:1C:BF:0A:CC:7C:9B:66:7D:3A:02:D1:CA:B9:16:4D:79:A7:A6:C7:62:C0:00:8A:B1:5A:2A:C6:87:6F:81:3D:C9:84:1D:DF:32:3B:29:45:07:FF:6C:7A:91:C8:D5:7E:31:E9:3E:E6:5F:43:60:D0:9B:1F:DD:36:A8:CB:ED:13:4C:B1:8A:F4:1E:F3:6C:0C:C7:BD:57:7A:45:17:8E:56:C7:7E:99:C2:DF:63:D9:C8:44:73:82:0E:97:BD:B0:71:59:F1:70:D7:6A:E3:84:95:90:B9:44:AE:EE:25:3B:F6:1E:B2:CC:17:8C:5E:E8:48:13:C9:00:88:17:D4:53:0D:20:F0:E4:DE:FF:FC:73:D9:72:C2:1B:4B:D1:CF:3B:35:AE:BB:FD:44:8B:06:7D:5F:2A:1A:59:BB:67:69:BF:63:46:7F:0B:C4:69:13:37:86:BB:D6:78:88:36:0F:B4:AA:4B:C3:A3:3D:EE:FE:D2:59:2A:BF:B3:27:9F:49:46:C8:D9:E2:0E:32:90:AC:55:9B:73:12:E8:32:92:EE

Bumrah secret:

E8:D8:71:18:D0:C9:84:F8:AD:DC:7D:CB:1C:E1:77:7C:12:EF:B0:9B:F0:ED:49:81:5A:8F:EB:42:16:01:E4:39:AB:4D:D2:07:28:C9:3F:AA:90:3C:23:BC:98:3F:A4:B6:2C:2B:D2:1C:BF:0A:CC:7C:9B:66:7D:3A:02:D1:CA:B9:16:4D:79:A7:A6:C7:62:C0:00:8A:B1:5A:2A:C6:87:6F:81:3D:C9:84:1D:DF:32:3B:29:45:07:FF:6C:7A:91:C8:D5:7E:31:E9:3E:E6:5F:43:60:D0:9B:1F:DD:36:A8:CB:ED:13:4C:B1:8A:F4:1E:F3:6C:0C:C7:BD:57:7A:45:17:8E:56:C7:7E:99:C2:DF:63:D9:C8:44:73:82:0E:97:BD:B0:71:59:F1:70:D7:6A:E3:84:95:90:B9:44:AE:EE:25:3B:F6:1E:B2:CC:17:8C:5E:E8:48:13:C9:00:88:17:D4:53:0D:20:F0:E4:DE:FF:FC:73:D9:72:C2:1B:4B:D1:CF:3B:35:AE:BB:FD:44:8B:06:7D:5F:2A:1A:59:BB:67:69:BF:63:46:7F:0B:C4:69:13:37:86:BB:D6:78:88:36:0F:B4:AA:4B:C3:A3:3D:EE:FE:D2:59:2A:BF:B3:27:9F:49:46:C8:D9:E2:0E:32:90:AC:55:9B:73:12:E8:32:92:EE

Chahal secret:

E8:D8:71:18:D0:C9:84:F8:AD:DC:7D:CB:1C:E1:77:7C:12:EF:B0:9B:F0:ED:49:81:5A:8F:EB:42:16:01:E4:39:AB:4D:D2:07:28:C9:3F:AA:90:3C:23:BC:98:3F:A4:B6:2C:2B:D2:1C:BF:0A:CC:7C:9B:66:7D:3A:02:D1:CA:B9:16:4D:79:A7:A6:C7:62:C0:00:8A:B1:5A:2A:C6:87:6F:81:3D:C9:84:1D:DF:32:3B:29:45:07:FF:6C:7A:91:C8:D5:7E:31:E9:3E:E6:5F:43:60:D0:9B:1F:DD:36:A8:CB:ED:13:4C:B1:8A:F4:1E:F3:6C:0C:C7:BD:57:7A:45:17:8E:56:C7:7E:99:C2:DF:63:D9:C8:44:73:82:0E:97:BD:B0:71:59:F1:70:D7:6A:E3:84:95:90:B9:44:AE:EE:25:3B:F6:1E:B2:CC:17:8C:5E:E8:48:13:C9:00:88:17:D4:53:0D:20:F0:E4:DE:FF:FC:73:D9:72:C2:1B:4B:D1:CF:3B:35:AE:BB:FD:44:8B:06:7D:5F:2A:1A:59:BB:67:69:BF:63:46:7F:0B:C4:69:13:37:86:BB:D6:78:88:36:0F:B4:AA:4B:C3:A3:3D:EE:FE:D2:59:2A:BF:B3:27:9F:49:46:C8:D9:E2:0E:32:90:AC:55:9B:73:12:E8:32:92:EE

Dhoni secret:

E8:D8:71:18:D0:C9:84:F8:AD:DC:7D:CB:1C:E1:77:7C:12:EF:B0:9B:F0:ED:49:81:5A:8F:EB:42:16:01:E4:39:AB:4D:D2:07:28:C9:3F:AA:90:3C:23:BC:98:3F:A4:B6:2C:2B:D2:1C:BF:0A:CC:7C:9B:66:7D:3A:02:D1:CA:B9:16:4D:79:A7:A6:C7:62:C0:00:8A:B1:5A:2A:C6:87:6F:81:3D:C9:84:1D:DF:32:3B:29:45:07:FF:6C:7A:91:C8:D5:7E:31:E9:3E:E6:5F:43:60:D0:9B:1F:DD:36:A8:CB:ED:13:4C:B1:8A:F4:1E:F3:6C:0C:C7:BD:57:7A:45:17:8E:56:C7:7E:99:C2:DF:63:D9:C8:44:73:82:0E:97:BD:B0:71:59:F1:70:D7:6A:E3:84:95:90:B9:44:AE:EE:25:3B:F6:1E:B2:CC:17:8C:5E:E8:48:13:C9:00:88:17:D4:53:0D:20:F0:E4:DE:FF:FC:73:D9:72:C2:1B:4B:D1:CF:3B:35:AE:BB:FD:44:8B:06:7D:5F:2A:1A:59:BB:67:69:BF:63:46:7F:0B:C4:69:13:37:86:BB:D6:78:88:36:0F:B4:AA:4B:C3:A3:3D:EE:FE:D2:59:2A:BF:B3:27:9F:49:46:C8:D9:E2:0E:32:90:AC:55:9B:73:12:E8:32:92:EE

Anil and Bumrah are the same

Bumrah and Chahal are the same

Chahal and Dhoni are the same

Dhoni and Anil are the same

** Process exited - Return Code: 0 **

Conclusion:

So, from the particle implementation, definitions, and diagrams for the comparison of methods for message authentication, we have derived which methods are useful for message integrity, authentication, and non-repudiation. Which is mentioned in the below table (Table 2.1). By analyzing and implementing the Diffie-Hellman protocol, this report presents its practical use with an example and graphical illustration, which would allow the exchanging of secret keys between four parties.

Methods	Message integrity	Auth	Non-repudiation
Hash	Yes	No	No
RSA + SHA1	Yes	Yes	Yes
DSA	Yes	Yes	Yes
Hmac-sha-256	Yes	Yes	No

Table 2.1: Conclusion Table

Bibliography

- [1] "Cryptographic hash function," Wikipedia, 30 11 2022. [Online]. Available: https://en.wikipedia.org/wiki/Cryptographic_hash_function.
- [2] S. Halevi and H. Krawczyk, "Randomized Hashing and Digital Signatures," *Faculty of Electrical & Computer Engineering, Technion Institute of Technology of Israel*, p. 8, 30 1 2007.
- [3] R. Awati, "Authentication and access control : nonrepudiation," *techtargert*, August 2021. [Online]. Available: <https://www.techtargert.com/searchsecurity/definition/nonrepudiation>.
- [4] A. J. Menezes, v. Oorschot, P. C. Vanstone and S. A, *Handbook of Applied Cryptography*, Boca Raton: CRC Press, 1997.
- [5] S. Bruce, *Applied cryptography : protocols, algorithms, and source code in C*, New York: Wiley, 1996.
- [6] "Java Cryptography Architecture (JCA) Reference Guide," *oracle*, 2020. [Online]. Available: <https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SigEx>.
- [7] . O. Daisuke, Y. Masao and T. Nozomu , "A robust scan-based side-channel attack method against HMAC-SHA-256 circuits," *IEEE*, pp. 79-84, 2017.
- [8] R. L. Rivest, S. A and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *The ACM Digital Library*, p. 120–126, 1 2 1978.
- [9] Philip A. DesAutels and P. Lipp , "RSA-SHA1 Signature Suite - Version 1.0," 9 10 1997. [Online]. Available: https://www.w3.org/PICS/DSig/RSA-SHA1_1_0.html. [Accessed 2 1998].
- [10] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *ACM DL Digital Library*, pp. 365-390, 2022.

- [11] N. Li, "Research on Diffie-Hellman key exchange protocol," *Institute of Electrical and Electronics Engineers*, vol. 4, pp. V4-634-V4-637, 2010.
- [12] Alexander S. Gillis, "Diffie-Hellman key exchange (exponential key exchange)," techtarget, 2022.