COMP522 Assignment 2

Comparison of methods for message authentication &

Diffie-Hellman Key exchange

Chunyu Gong
Student No: 201608534

Abstract

This report is for COMP522 Assignment 2 about comparing the method for message authentication and Diffie-Hellman Key exchange. It discusses the advantage and disadvantages of using the method for message authentication and designing and implementing a variant of the Diffie-Hellman key exchange protocol, which allows the exchange of secrets between four parties.

1. Introduction

The purpose of this report is divided into two parts. The first part is a practical overview of four encryption methods, hash function, RSA+SHA1 method, DSA method and HMAC-SHA256. The other is a variant of the Diffie-Hellman key exchange protocol.

1.1 Hash-function

The hash function is a one-way function. It can compute any given message to a short fixed-length output string, the message digest cannot reverse the computation, the same message always results in the same hash, and different messages cannot result in the same value. (Diffie et al., 1976) Hash functions can be used in many applications: to detect duplicates message or unique message and check if the data has been manipulated or corrupted (e.g., message digest). (Diffie et al., 1976)

1.2 RSA + SHA1 method

RSA(Rivest-Shamir-Adleman) is a method that implements public-key cryptography, an elegant idea by Diffie and Hellman. In public-key encryption, the encryption key is public, and the decryption key is private is the novelty feature. It can avoid the messenger and another secure way to transmit the key since the message can be encrypted using the key revealed by the target receiver. (Rivest et al., 1978)

1.3 DSA method

DSA (Digital Signature Algorithm) is a public-key cryptography for digital signatures based on the mathematical concept and globally standardised by the National Institute of Standards and Technology. The algorithm requires the private key to generate the digital signature for the message, and the receiver can verify by the public key from the signer to make sure the original message has not been modified after the signer. If the receiver verifies the signature, the sender cannot claim sender never sent the message. (Jena, 2022)

1.4 HMAC-SHA256

HMAC (Hash-based message authentication code) is a specific type of MAC (Message authentication code). It is similar to the Hash function discussed in Section 1.1. Like all authentication methods, ensure the original message has not been modified. Any cryptographic hash function can be used in HMAC, and the result is termed HMAC-X; HMAC-SHA256 means the hash function use SHA256. It directly decides the key's quality and the cryptographic's strength. (Bellare et al., 1996)

1.5 Diffie-Hellman Key Exchange Protocol.

Diffie-Hellman key exchange is a mathematical method in the members who do not know each other in advance build shared keys together in an unsecured channel. Using this key cipher, further communications. (Diffie et al., 1976) Although it is a non-authenticated key-agreement protocol, it still provides the basis for various authentication protocols, such as RSA, which discussion in section 1.2.

2. Methods

All programs are copied and modified from the COMP522 Lab session, in this section only provides the method to implement.

2.1 Hash-function

The processing of using Hash-function is as follows:

An unencrypted message sends from a sender to a receiver with a hash value generated by the sender. When the receiver receives the message, they verify it to ensure it is not manipulated or corrupted. The receiver generates a new hash value and then compares it with the hash value generated by the sender. If the hash value differs, the receiver knows the

message is not the original one. Otherwise, the receiver can believe that the message is not modified.

Appendix 7.1 is the SHA1 hash function program which calculates the digests of the input string using the SHA1 algorithm to produce a byte array hash.

2.2 RSA + SHA1 method

The methodology discussed in this section combines SHA1 with the RSA algorithm, which uses the hashing algorithm previously discussed in Section 1.1 and uses both public and private keys to share information securely. In a public-key cryptosystem, the encryption key is public. RSA users create and publish a public key based on large secret prime numbers and an accessory value. Anyone with the public key can encrypt the message, but only those who know the prime numbers. (Rivest et al., 1978)

The RSA + SHA1 algorithm generates a hash value, which is then encrypted by the public key. This key is generated as part of the key pair. the key produces a cipher text and then sends the ciphertext message to the receiver. The verifier can use the same algorithm to generate their hash. The message cannot be trusted if the hash is not matched.

Appendix 7.2 is an example used to create a program to detect manipulated messages.

2.3 DSA method

The DSA method is similar to section 2.2, but DSA is not encrypting the message hash like the RSA method. DSA provides message authentication, integrity and non-repudiation.

The DSA algorithm involves the following operations

Sender

- DSA method gets the origin message, generates the key pair and creates a hash.
- Sender should publish the public key through a reliable, but not have to be a secret way to the receivers.
- Sign the message using the private key to produce a unique signature.

Verifier

- When the receiver gets the message, they can take the original message to generate a message hash.
- Verifier takes the message hash value received and the public key to the verifier. The public matches the private key that used to sign the message

-If the verify function of the DSA algorithm returns “True” it means the message can be trusted. If the verify function return “False”, it means the message cannot be trusted, just like the other authentication method already discussed.

Appendix 7.3 is an example of detecting the manipulated message.

2.4 HMAC-SHA256

HMAC involve the following step:

- Sender and receiver have a shared key, and then the sender generates a MAC using the shared private key. The MAC send the original message to the receiver.

-When the receiver gets the message, they use the message and shared key to generate their own MAC

- The receiver compares the received MAC and the one they generated MAC. If the MAC is different, the message cannot be trusted.

Appendix 7.4 is an example of the HMAC algorithm used to create MAC value.

2.5 Diffie-Hellman Key Exchange Protocol

Following the Diffie-Hellman Key Exchange Protocol means that if more parties need to add to this process, more key needs to be shared. Each party in this process use 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:

$Y_A = a^{X_A} \; mod \; q$

$Y_B = a^{X_B} \; mod \; q$

$Y_C = a^{X_C} \; mod \; q$

$Y_D = a^{X_D} \; mod \; q$

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

A calculates $K = (Y_{BCD})^{X_A} \; mod \; q$

B calculates $K = (Y_{ACD})^{X_B} \; mod \; q$

C calculates $K = (Y_{ABD})^{X_C} \; mod \; q$

D calculates $K = (Y_{ABC})^{X_D} \; mod \; q$

Figure 1 is an example to explain how to reach the shared secret key.

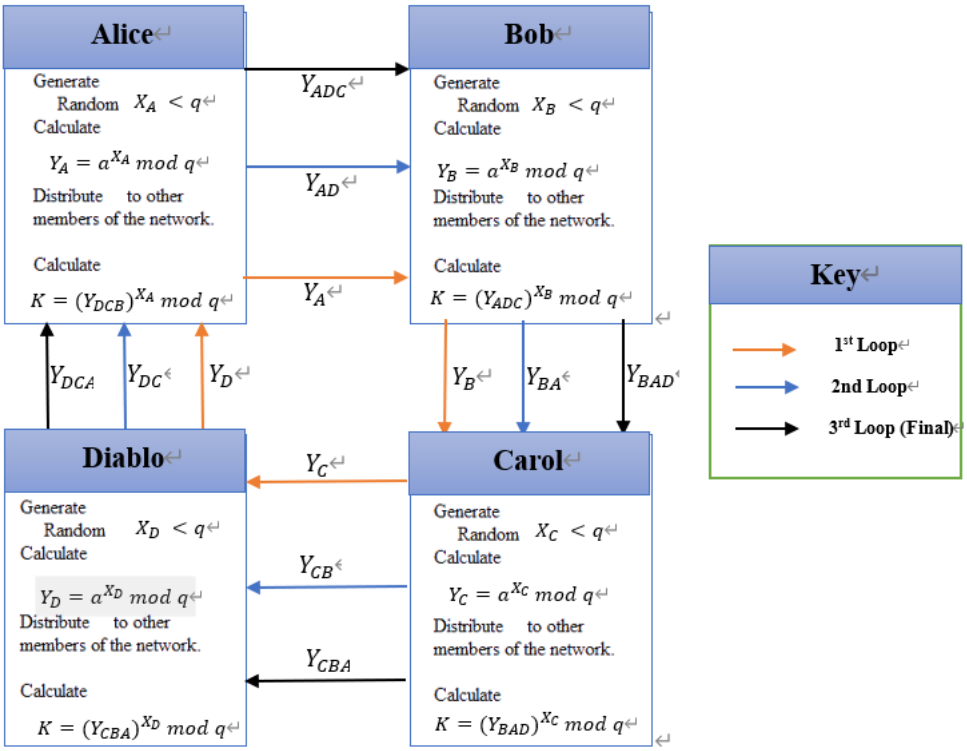


Figure 1: The process of exchange key

3. Results

3.1 Hash-function

Figure 2 is the outcome of *Appendix 7.1,* the hash function using the SHA1 algorithm produces a byte array hash.

Figure 2: The output of the hashing function

3.2 RSA + SHA1 method

Figure 3 is the outcome of *Appendix 7.2,* the user inputs the plaintext to the program and the verifier is received the message without interference. However, on the way to the receiver the message has been changed to “wrong”, As in Figure 4, because the message has been changed, the hash value has been changed too, so the message cannot be trusted.

Figure 3: the output of None intercepted

Figure 4: the output of intercepted message detected

3.3 DSA method

The figure below shows the result of *Appendix 7.3* that has been run the DSA algorithm.

Figure 5 shows that when the message is not intercepted and the original message and the signed hash are original, then the DSA algorithm can verify that the message has not been manipulated, it means the message can be trusted.

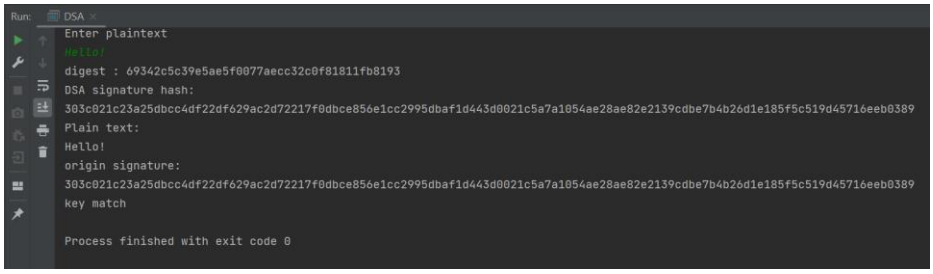


Figure 5: the output of None intercepted

As in Figure 6, because the message has been changed, the hash value is different, so the message cannot be trust.

Figure 6: the output of intercepted message detected

As in Figure 7, even the message is the same, but the signed hash has been manipulated, then the DSA will throw and exception, therefore, the message cannot be trusted.

Figure 7: the output of the wrong sign

3.4 HMAC-SHA256

Figure 8 is the output of HMAC. The function has been discussed in 1.4 and 2.4.

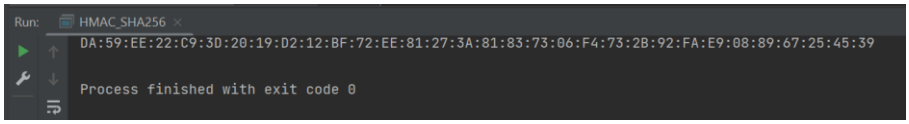


Figure 8: the output of HMAC

3.5 Diffie-Hellman Key Exchange Protocol

As discussed above, to follow Diffie-Hellman, every network member must exchange all the keys. Figure 10 shows how the key exchange in multiple iterations. At the first iteration, every member knows their own key, but through other iteration, every member's key will be collected, util everyone will get other member key. Figure 9 shows the output of the Diffie-Hellman Key Exchange Protocol.

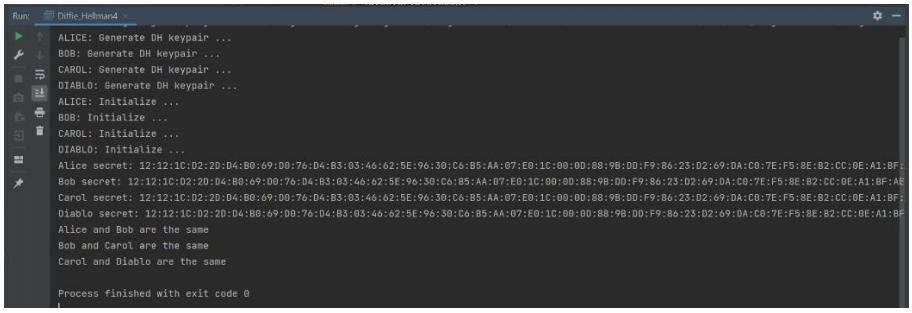


Figure 9: the output of Diffie-Hellman Key Exchange Protocol

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

Figure 10: key exchange order

4. Discussion

4.1 Hash-function

The advantage of using the Hash function is easy and very useful to judge whether the message can be trusted. If anyone changes the message, even a small change should change the hash value significantly. (Al-Kuwari et al., 1970)

The disadvantage of the Hash function is that the message is not encrypted, so basically everyone can read the message, and there has much research shows that the hash value is weak. In a recent report, Google has announced they already cracked this function. (Brewster, 2021)

4.2 RSA + SHA1 method

The advantage of using a large enough key is used there is no method to defeat it. (Castelvecchi, 2020) Another advantage is that it allows the user to protect the message before the user sends any message.

The disadvantage is that it is a slow and expensive algorithm, which must generate a new key after sending each message.

4.3 DSA method

The advantage of using DSA method is like the RSA method except DSA reduce the message been manipulate or copied, the digital signature is real and non-repudiation.

The disadvantage is that it is a slow and expensive algorithm as it must generate a new key after each message is sent.

4.4 HMAC-SHA256

The advantage of using HMAC is that it is the only-way function, so the HMAC function is non-reversible.

The disadvantage is that it is easy to get another MAC when they do not know the key to one of the methods of HMAC. Some research gives the same conclusion. (Preneel & van Oorschot, 1995)

4.5 Diffie-Hellman Key Exchange Protocol

The advantage of using this method is that the sender and verifier do not know any prior knowledge of each other. Once the key is exchanged, they can communicate in an insecure way.

The disadvantage of using it is the algorithm's lack of authentication procedure, and because there is no authentication procedure, it is vulnerable to a man-in-the-middle attack. (vazhakkat, 2020)

5. Conclusion

In this report, Diffie-Hellman's four-person exchange protocol is designed by itself. Four methods (hash function, RSA+SHA1 method, DSA method and HMAC-SHA256) are analysed, and their properties every method has been following:

Integrity:

Hash-Functions, RSA-SHA1/DSA, HMAC

Authentication:

RSA-SHA1/DSA, HMAC

Non-repudiation

RSA-SHA1/DSA

6. Reference

Al-Kuwari, S., Davenport, J.H. and Bradford, R.J. (1970) *Cryptographic hash functions: Recent design trends and security notions, Cryptology ePrint Archive.* Available at: https://eprint.iacr.org/2011/565 (Accessed: November 30, 2022).

Bellare, M., Canetti, R. and Krawczyk, H. (1996) *Keying hash functions for message authentication, SpringerLink. Springer Berlin Heidelberg.* Available at: https://link.springer.com/chapter/10.1007/3-540-68697-5_1 (Accessed: November 28, 2022).

Brewster, T. (2021) *Google just 'shattered' an old crypto algorithm -- here's why that's big for web security, Forbes. Forbes Magazine.* Available at: https://www.forbes.com/sites/thomasbrewster/2017/02/23/google-sha-1-hack-why-it-matters/#3f73df04c8cd (Accessed: November 30, 2022).

Castelvecchi, D. (2020) *Quantum-computing pioneer warns of complacency over internet security, Nature News. Nature Publishing Group.* Available at: https://www.nature.com/articles/d41586-020-03068-9 (Accessed: November 30, 2022).

Diffie, W. et al. (1976) *New Directions in cryptography, IEEE Transactions on Information Theory.* Available at: https://dl.acm.org/doi/10.1109/TIT.1976.1055638 (Accessed: November 28, 2022).

Jena, B.K. (2022) *Digital Signature Algorithm (DSA) in Cryptography: A complete guide: Simplilearn, Simplilearn.com. Simplilearn.* Available at: https://www.simplilearn.com/tutorials/cryptography-tutorial/digital-signature-algorithm (Accessed: November 28, 2022).

Preneel, B. and van Oorschot, P.C. (1995) *MDX-MAC and Building Fast Macs from hash functions, SpringerLink. Springer Berlin*

Heidelberg. Available at: https://link.springer.com/chapter/10.1007/3-540-44750-4_1 (Accessed: November 30, 2022).

Rivest, R.L., Shamir, A. and Adleman, L., 1978. *A method for obtaining digital signatures and public-key cryptosystems.* Communications of the ACM, 21(2), pp.120-126.

vazhakkat, swetha (2020) Applications and limitations of Diffie-Hellman algorithm, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applications-and-limitations-of-diffie-hellman-algorithm/ (Accessed: November 30, 2022).

7. Appendices

7.1 Hash-function

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

        Scanner input = new Scanner(System.in);
        System.out.println("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()));


    }
}
```

7.2 RSA + SHA1 method

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


public class RSA_SHA1 {
    public static void main( String[]    args) throws Exception
    {

        //Sender
        Scanner input = new Scanner(System.in);
        System.out.println("Enter plaintext");
        String Gettext = input.nextLine();
        //String Wrong = "wrong";
        Cipher cipher = Cipher.getInstance( transformation: "RSA/ECB/PKCS1Padding");

        //SHA1
        byte[] hashed = MessageDigestExample.MessageDigest(Gettext);
        SecureRandom random = new SecureRandom();


        // create the keys
        KeyPairGenerator generator = KeyPairGenerator.getInstance( algorithm: "RSA");
```

```java
        generator.initialize( keysize: 512,random);

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

        // encryption step

        cipher.init(Cipher.ENCRYPT_MODE, pubKey);

        byte[] cipherText = cipher.doFinal(hashed);

        System.out.println("cipher : " + Utils.toHex(cipherText));
        System.out.println("Message sent was : " + Gettext);

        //Verifier
        cipher.init(Cipher.DECRYPT_MODE, privKey);

        byte[] plainText = cipher.doFinal(cipherText);
        String Sender_digest = Utils.toHex(plainText) ;
        System.out.println("Message received: "+ Gettext);
        //System.out.println("Message received: "+ Wrong);
        byte[] Verifier_Input = MessageDigestExample.MessageDigest(Gettext);
        String Verifier_digest = Utils.toHex(Verifier_Input);
```

```java

        if (Verifier_digest.equals(Sender_digest))
        {
            System.out.println("Value match\n"+ "Sender hash :" + Sender_digest +"\nVerifier has : "+Verifier
        }
        else
        {
            System.out.println("Value not match\n"+ "Sender hash :" + Sender_digest +"\nVerifier has : "+Veri
        }
    }
}
```

7.3 DSA method

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

public class DSA {
    public static void main( String[]    args) throws Exception
    {

        //Sender
        //input_text from user
        Scanner input = new Scanner(System.in);
        System.out.println("Enter plaintext");
        String cleartext  = input.nextLine();
        String Wrong = "wrong";


        Cipher cipher = Cipher.getInstance( transformation: "RSA/ECB/PKCS1Padding");
        SecureRandom random = new SecureRandom();

        //SHA1
        byte[] hashed = MessageDigestExample.MessageDigest(cleartext);
        System.out.println("digest : " + Utils.toHex(hashed));

        // create the keys
        KeyPairGenerator generator = KeyPairGenerator.getInstance( algorithm: "DSA");
        Signature DSA = Signature.getInstance( algorithm: "SHA256withDSA");
        Signature Wrong_DSA = Signature.getInstance( algorithm: "SHA256withDSA");

        generator.initialize( keysize: 2048,random);
        KeyPair pair = generator.generateKeyPair();
        PublicKey pubKey = pair.getPublic();
        PrivateKey privKey = pair.getPrivate();


        //Sign the hash using the private key
        DSA.initSign(privKey);
        DSA.update(hashed);
        byte[] signed = DSA.sign();

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


        //Verifier
        String Message = "This is a message";

        //received plain text to verifier
        System.out.println("Plain text: \n" + cleartext);
        //System.out.println("Plain text: "+ Wrong);
        System.out.println("origin signature: \n" + Utils.toHex(signed));
```

```
50
51          byte[] verier = MessageDigestExample.MessageDigest(cleartext);
52          //byte[] wrong_sign = Wrong_DSA.sign();
53          //start verify progress
54          DSA.initVerify(pubKey);
55
56          DSA.update(verier);
57
58          if (DSA.verify(signed))
59          {
60              System.out.println("key match");
61          }
62          else
63          {
64              System.out.println("do not match");
65          }
66      }
67  }
```

7.4 HMAC-SHA256

```
33  import javax.crypto.*;
34
35  /**
36   * This program demonstrates how to generate a secret-key object for
37   * HMACSHA256, and initialize an HMACSHA256 object with it.
38   */
39
40  public class HMAC_SHA256 {
41
42      public static void main(String[] args) throws Exception {
43
44          // Generate secret key for HmacSHA256
45          KeyGenerator kg = KeyGenerator.getInstance( algorithm: "HmacSHA256");
46          SecretKey sk = kg.generateKey();
47
48          // Get instance of Mac object implementing HmacSHA256, and
49          // initialize it with the above secret key
50          Mac mac = Mac.getInstance( algorithm: "HmacSHA256");
51          mac.init(sk);
52          byte[] result = mac.doFinal("Hi".getBytes());
53          System.out.println(toHexString(result));
54
55      }
```

```java
    1 usage
    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
     */
    1 usage
    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();
    }
```

7.5 Diffie-Hellman Key Exchange Protocol

```java
/.../
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
 * 3 parties: Alice, Bob, and Carol using a shared 2048-bit DH parameter.
 */
public class Diffie_Hellman4 {
    private Diffie_Hellman4() {}
    public static void main(String argv[]) throws Exception {

        // All member have their own public keys
        // Alice creates her own DH key pair with 2048-bit key size
        System.out.println("ALICE: Generate DH keypair ...");
        KeyPairGenerator aliceKpairGen = KeyPairGenerator.getInstance( algorithm: "DH");
        aliceKpairGen.initialize( keysize: 2048);
        KeyPair aliceKpair = aliceKpairGen.generateKeyPair();

        // This DH parameters can also be constructed by creating a
        // DHParameterSpec object using agreed-upon values
        DHParameterSpec dhParamShared = ((DHPublicKey)aliceKpair.getPublic()).getParams();
```

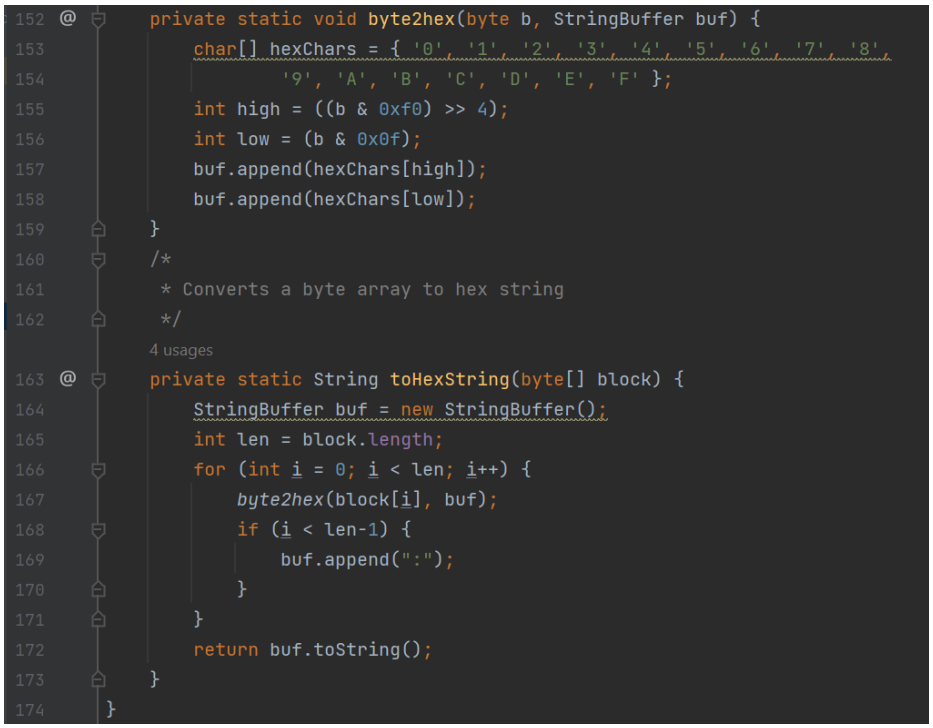
```
54
55          // Bob creates his own DH key pair using the same params
56          System.out.println("BOB: Generate DH keypair ...");
57          KeyPairGenerator bobKpairGen = KeyPairGenerator.getInstance( algorithm: "DH");
58          bobKpairGen.initialize(dhParamShared);
59          KeyPair bobKpair = bobKpairGen.generateKeyPair();
60
61          // Carol creates her own DH key pair using the same params
62          System.out.println("CAROL: Generate DH keypair ...");
63          KeyPairGenerator carolKpairGen = KeyPairGenerator.getInstance( algorithm: "DH");
64          carolKpairGen.initialize(dhParamShared);
65          KeyPair carolKpair = carolKpairGen.generateKeyPair();
66
67          // Diablo creates his own DH key pair using the same params
68          System.out.println("DIABLO: Generate DH keypair ...");
69          KeyPairGenerator DiabloKpairGen = KeyPairGenerator.getInstance( algorithm: "DH");
70          DiabloKpairGen.initialize(dhParamShared);
71          KeyPair DiabloKpair = DiabloKpairGen.generateKeyPair();
72
73          // Alice initialize
74          System.out.println("ALICE: Initialize ...");
75          KeyAgreement aliceKeyAgree = KeyAgreement.getInstance( algorithm: "DH");
76          aliceKeyAgree.init(aliceKpair.getPrivate());
77
78          // Bob initialize
```

```
78          // Bob initialize
79          System.out.println("BOB: Initialize ...");
80          KeyAgreement bobKeyAgree = KeyAgreement.getInstance( algorithm: "DH");
81          bobKeyAgree.init(bobKpair.getPrivate());
82
83          // Carol initialize
84          System.out.println("CAROL: Initialize ...");
85          KeyAgreement carolKeyAgree = KeyAgreement.getInstance( algorithm: "DH");
86          carolKeyAgree.init(carolKpair.getPrivate());
87
88          // Diablo initialize
89          System.out.println("DIABLO: Initialize ...");
90          KeyAgreement DiabloKeyAgree = KeyAgreement.getInstance( algorithm: "DH");
91          DiabloKeyAgree.init(DiabloKpair.getPrivate());
92
93          // Alice uses Diablo's public key
94          Key ad = aliceKeyAgree.doPhase(DiabloKpair.getPublic(),  lastPhase: false);
95          // Bob uses Alice's public key
96          Key ba = bobKeyAgree.doPhase(aliceKpair.getPublic(),  lastPhase: false);
97          // Carol uses Bob's public key
98          Key cb = carolKeyAgree.doPhase(bobKpair.getPublic(),  lastPhase: false);
99          // Diablo uses Carol's public key
100         Key dc = DiabloKeyAgree.doPhase(carolKpair.getPublic(),  lastPhase: false);
```

```
        // Alice uses Diablo's result from above
        Key adc = aliceKeyAgree.doPhase(dc, lastPhase: false);
        // Bob uses Alice's result from above
        Key bad = bobKeyAgree.doPhase(ad, lastPhase: false);
        // Carol uses Bob's result from above
        Key cba = carolKeyAgree.doPhase(ba, lastPhase: false);
        //Diablo uses Carols result from above
        Key dcb = DiabloKeyAgree.doPhase(cb, lastPhase: false);

        //get key
        // Alice uses Diablo's result from above
        aliceKeyAgree.doPhase(dcb, lastPhase: true);
        // Bob uses Alice's result from above
        bobKeyAgree.doPhase(adc, lastPhase: true);
        // Carol uses Bob's result from above
        carolKeyAgree.doPhase(bad, lastPhase: true);
        // Diablo uses Carol's result from above
        DiabloKeyAgree.doPhase(cba, lastPhase: true);

        // all member have public key from others

        // Alice, Bob and Carol compute their secrets
        byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();
        System.out.println("Alice secret: " + toHexString(aliceSharedSecret));
        byte[] bobSharedSecret = bobKeyAgree.generateSecret();
```

Security\Lab\Lab_6

```
        byte[] bobSharedSecret = bobKeyAgree.generateSecret();
        System.out.println("Bob secret: " + toHexString(bobSharedSecret));
        byte[] carolSharedSecret = carolKeyAgree.generateSecret();
        System.out.println("Carol secret: " + toHexString(carolSharedSecret));
        byte[] DiabloSharedSecret = DiabloKeyAgree.generateSecret();
        System.out.println("Diablo secret: " + toHexString(DiabloSharedSecret));

        // Compare Alice and Bob
        if (!java.util.Arrays.equals(aliceSharedSecret, bobSharedSecret))
            throw new Exception("Alice and Bob differ");
        System.out.println("Alice and Bob are the same");
        // Compare Bob and Carol
        if (!java.util.Arrays.equals(bobSharedSecret, carolSharedSecret))
            throw new Exception("Bob and Carol differ");
        System.out.println("Bob and Carol are the same");
        // Compare Carol and Diablo
        if (!java.util.Arrays.equals(carolSharedSecret, DiabloSharedSecret))
            throw new Exception("Carol and Diablo differ");
        System.out.println("Carol and Diablo 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
     */
    4 usages
    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();
    }
}
```