



T.C.
MARMARA UNIVERSITY
FACULTY of ENGINEERING

CSE4057 Information Systems and Security
Fall 2018 – Homework 1

Students

Emine Feyza MEMİŞ

150114077

Hilal EKİNCİ

150114057

-16.11.2018 -

1. Generate an RSA public-private key pair. K_A^+ and K_A^- .

The RSA algorithm was published in 70's by Ron Rivest, Adi Shamir and Leonard Adleman. It is an asymmetric system, which means that a key pair will be generated, a public key and a private key, obviously it keep the private key secure and pass around the public one.

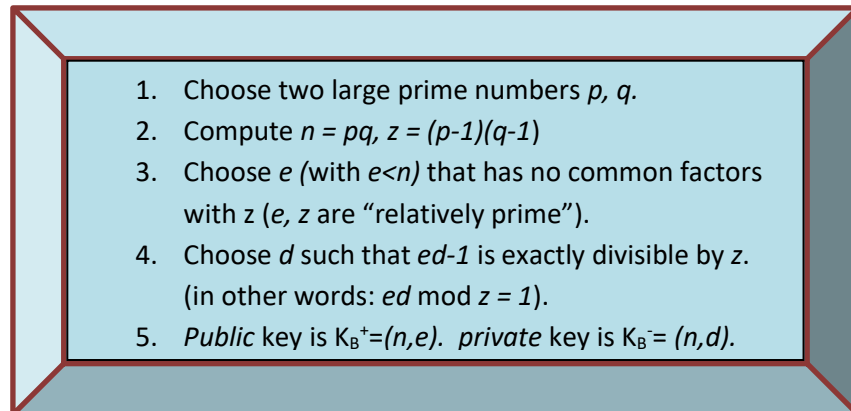


Figure1: Algorithm of RSA

Firstly, we created RSA Key Pair using *KeyPairGenerator* from a factory method by specifying the algorithm. After that, we initialized the *KeyPairGenerator* with the key size of 1024. We got the public key using *getPublic()* and the private key using *getPrivate()* from the *KeyPair* object.

```
/// Question 1
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);

KeyPair kp = kpg.generateKeyPair();
PrivateKey prk = kp.getPrivate();
PublicKey puk = kp.getPublic();

System.out.println("\n--RSA Public/Private Key Pair-- \n\n");
System.out.println("Public Key KA(+): \n" + enc.encodeToString(puk.getEncoded()) + "\n");
System.out.println("Private Key KA(-): \n" + enc.encodeToString(prk.getEncoded()) + "\n\n");
```

Figure2: Codes of question1

```
Problems @ Javadoc Console
<terminated> Security_hw [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.exe (Nov 14, 2018, 8:27:05 PM)
|-----RSA Public/Private Key Pair-----
|-----

Public Key KA(+):
MIGfMA0GCsQGSIB3DQEBAQUAA4GNADCBiQKBgQCYYuwgwQ0oG+jEAUBC5p9fS0rlaih0Roi0rA27CHG
sHv2RRueWedt2AHNEsCP8119BhtMdkqzc0s1CxnCwfSdAmcb3nZvZycQ2psRvQX2EAAfFfYpZPJ4Uag0
aPXc+wUjx8wNVFmWnKdfZVM6vcy42dFeSfg8x36yDrQkWM7GFwIDAQAB

Private Key KA(-):
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAJhi7CDBDSgb6MQBQELmn19I6uVqKGDR
GiLSsDbsIcawe/ZFG55Z523YAc0SwI/zWX0GG0x2SrNw6zULGcLB9J0CZxvedm9nJxDamxG9BfYQAC0V
9inM8nhRqDRo9dz7BSPHzA1UWZacp19lUzq9zLjZ0V5J+DzHfrIOtCRYzsYXAgMBAAECgYBFSw5GoqKB
75b7sfvnsGecigLOSWSNceiRly6r+totCEmGJcy2R/l1uHixM0rBS04RzPoS3e0AD0PVPfP73MiG+/JA
U0fLGM1Tv19Rf/qV3Y0VPDqM/b9PIQ1xHvQ2h92nE2EehCkz99Tpn0N8gXmC4HjC0YuaSjXe6xrnSdbG
uQJBAN+dXx1QiydMED0/Yhqq4wPu4zmlW75BmfsvRc7hG/0/unFFiaL2Aa9qAsRyxyFTHrxfMXq2VF8WT
zPfQ723GRmMCQCudLmoS5L2dvsUSiQ1+JX1JFfg/bm7PiXCj0IaZIM2LBdiaak3ju/fGfr8Mw5ht79+
qEoPfM/6D1iT681fZaw9AkEAurzh9koRuuJpUV2MhqqXCIYxGSGzs4n0by7djVE02sJvpGmHzc8yj7Y
KRdKLYqH7L7NffXY4TU5Y0YJXEnM8wJAPhprZQ/SN07Pvr9N+LCPm8y/3f80uQU8p+9DJiYV15kGkhAP
HyoRFhjHkjN+meaxksMR6kYwUqv8uFu+9nDw+QJAbkkJ66VoPQD0R+B0u3pUUezpRU8YyJv9QwvpVSzf
6hmMArA77Qp4JY9t1bZZM5gz18neOMvyXJm01yz7rMGV9w==
```

Figure3: Output of question1

2. **Generate two symmetric keys: 128-bit K1 and 256-bit K2. Print values of the keys on the screen. Encrypt them with KA+, print the results, and then decrypt them with KA-. Again, print the results. Provide a screenshot showing your results.**

The symmetric key algorithms was initiated a process to select a symmetric-key encryption algorithm to be used to protect sensitive (unclassified) Federal information in furtherance of NIST's statutory responsibilities by the National Institute of Standards and Technology (NIST) in 1997. It is an encryption system in which the same key is used for the encoding and decoding of the data. The safe distribution of the key is one of the drawbacks of this method, but what it lacks in security it gains in time complexity.

Symmetric key cryptography

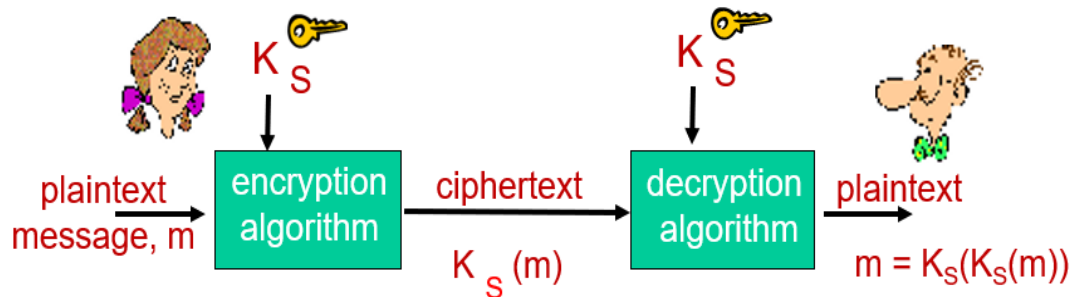


Figure4: Diagram of Symmetric Key Cryptography

Firstly, we create AES Symmetric Key using *KeyGenerator* from a factory method by specifying the algorithm. And then, we initialized the *KeyGenerator* with the key size of 128(K1) and 256(K2). We got the public key using *getPublic()* and the private key using *getPrivate()* from the *KeyPair* object. After that, we encrypted K1 and K2 with RSA asymmetric key K_A^+ and decrypted them with K_A^- .

```

/// Question 2
System.out.println("128 bit symmetric key K1:");
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(128);
Key k128 = generator.generateKey();
System.out.println("K1: " + new String(enc.encodeToString(k128.getEncoded())) + "\n");

System.out.println("256 bit symmetric key K2:");
KeyGenerator generator2 = KeyGenerator.getInstance("AES");
generator2.init(256);
Key k256 = generator2.generateKey();
System.out.println("K2: " + new String(enc.encodeToString(k256.getEncoded())) + "\n");

System.out.println("Encryption K1 with public key:\n");
byte [] encrypted128 = encrypt(puk, new String(enc.encodeToString(k128.getEncoded())));
System.out.println(new String(enc.encodeToString(encrypted128)));
System.out.println("\nDecryption K1 with private key:\n");
byte[] decrypted128 = decrypt(prk, encrypted128);
System.out.println(new String(decrypted128));    // This is a secret message

System.out.println("\n\nEncryption K2 with public key:\n");
byte [] encrypted256 = encrypt(puk, new String(enc.encodeToString(k256.getEncoded())));
System.out.println(new String(enc.encodeToString(encrypted256)));
System.out.println("\nDecryption K2 with private key:\n");
byte[] decrypted256 = decrypt(prk, encrypted256);
System.out.println(new String(decrypted256));    // This is a secret message

```

Figure5: Codes of question1

```
Problems @ Javadoc Console
<terminated> Security_hw [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.exe (Nov 14, 2018, 8:27:05 PM)
128 bit symmetric key K1:
K1: 6RY1ndWSs8v7BxhuLYqWaw==

256 bit symmetric key K2:
K2: CF7VrWs9mXCo/PAMrFDkDjg0bm5HmKYgNfdSVQ8x0g=

Encryption K1 with public key:

HH2lg8FDsMFYd2mtVjz4Q0lWtsLFWSUV4XlkhQPM+ocCFbHuE2lZqGLy0BHB1JIIfdC2iktefxxsXExBt
yUhURQEoU7qMgBwtJrPyJjLrqREwV5ieTzrtRyayRFQnBB1Zd25peDK4v6vCmhVrQ+WtqQEPIM5CBsolW
OmUhi1mEL+Y=

Decryption K1 with private key:

6RY1ndWSs8v7BxhuLYqWaw==
|

Encryption K2 with public key:

daUAtmxrm4hj5lMOB0rBokqK0o0h9cqahJwNe0y0mHEIIX5hs+OT3RttVtT4UgoavwiDV2n60HUESbWx
CslqVq6Fzf1ccY30RKQG8eGEZWQ9XZG08EMGmJtPF1ByUUCYpIb/EDsdxP0/rDZLb1Hfz+uErKefIRn1
U9uc7dGIXZ8=

Decryption K2 with private key:

6RY1ndWSs8v7BxhuLYqWaw==
```

Figure6: Outputs of question1

3. Consider a long text m . Apply SHA256 Hash algorithm (Obtain the message digest, $H(m)$). Then encrypt it with KA^- . (Thus, generate a digital signature.) Then verify the digital signature. (Decrypt it with KA^+ , apply Hash algorithm to the message, compare). Print m , $H(m)$ and digital signature on the screen. Provide a screenshot. (Or you may print in a file and provide the file).

The SHA256 designed by the NSA as a member of the SHA-2 cryptographic hash functions. It is a cryptographic hash function with digest length of 256 bits. It is a keyless hash function; that is, an MDC (Manipulation Detection Code).

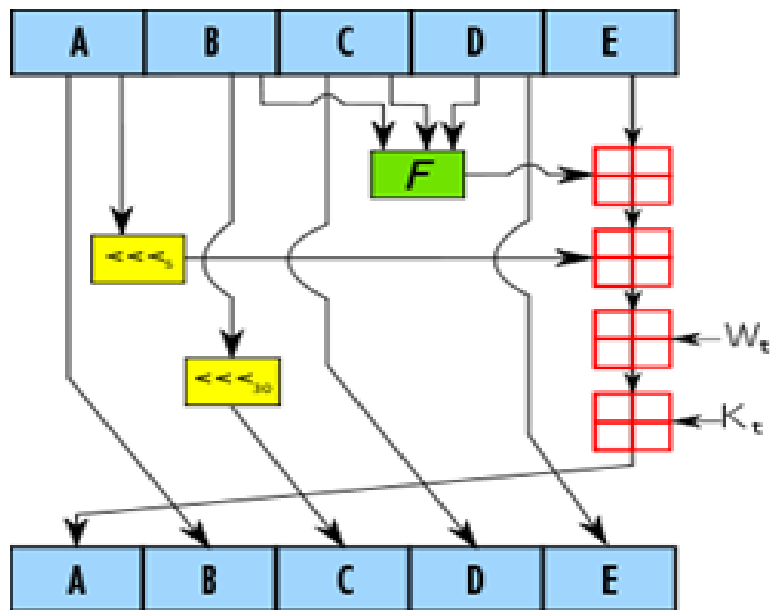


Figure7: Working principle of SHA256

We chose along text as a message. We digested message using MessageDigest from a factory method by specifying the SHA256 algorithm. And then, we encrypted hashed message with RSA asymmetric key K_A^- that is a digital signature. After that, we decrypted the digital signature with RSA asymmetric key K_A^+ . Finally, we compared hashed message ($H(m)$) and decrypted digital signature and we verified digital signature.

/// Question 3

```
String message = "A security is a fungible, negotiable financial "
    + "instrument that holds some type of monetary value. It "
    + "represents an ownership position in a publicly-traded "
    + "corporation (via stock), a creditor relationship with a "
    + "governmental body or a corporation (represented by owning "
    + "that entity's bond), or rights to ownership as represented "
    + "by an option";

MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] hashInBytes = md.digest(message.getBytes(StandardCharsets.UTF_8));

// bytes to hex
StringBuilder sb = new StringBuilder();
for (byte b : hashInBytes) {
    sb.append(String.format("%02x", b));
}
System.out.println("\nH(m): " + sb.toString());

byte[] encHm = signEncrypt(prk, sb.toString());
System.out.println("KA(-)(H(m)): " + new String(enc.encodeToString(encHm)));
byte[] decHm = signDecrypt(puk, encHm);
System.out.println("KA(+)(KA(-)(H(m))): " + new String(decHm));
```

Figure8: Codes of question3

```

Problems @ Javadoc Console
<terminated> Security_hw [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.exe (Nov 14, 2018, 8:27:05 PM)
H(m): b3727a5999d95697b561e3b53f57b792dae9a8b3fe00ed7206635c12895bc963
KA(-)(H(m)): 1tS1G41M01gzY/FNqjyE1Y+10gKVpnRBymwFndErupUcpdFndtXJ/EFYHMH/pio7Xsk0zP7T
RmAhx5AEodWrc8APKGZj9HhE31yvIUgz3uZpSt94Sd4hmbYkTSJJpSHyLdV4zi01/4LbAYPboHPE/rK2K2K+I
5fbL8vCeioWnaI=
KA(+)(KA(-)(H(m))): b3727a5999d95697b561e3b53f57b792dae9a8b3fe00ed7206635c12895bc963

```

Figure9: Outputs of question3

4. Consider a text m . Apply HMAC using $K1$ and SHA256 algorithms. Print m and result of HMAC on the screen and provide a screenshot.

HMAC is a Keyed-Hashing for message authentication that is almost like digital signature. It is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input.

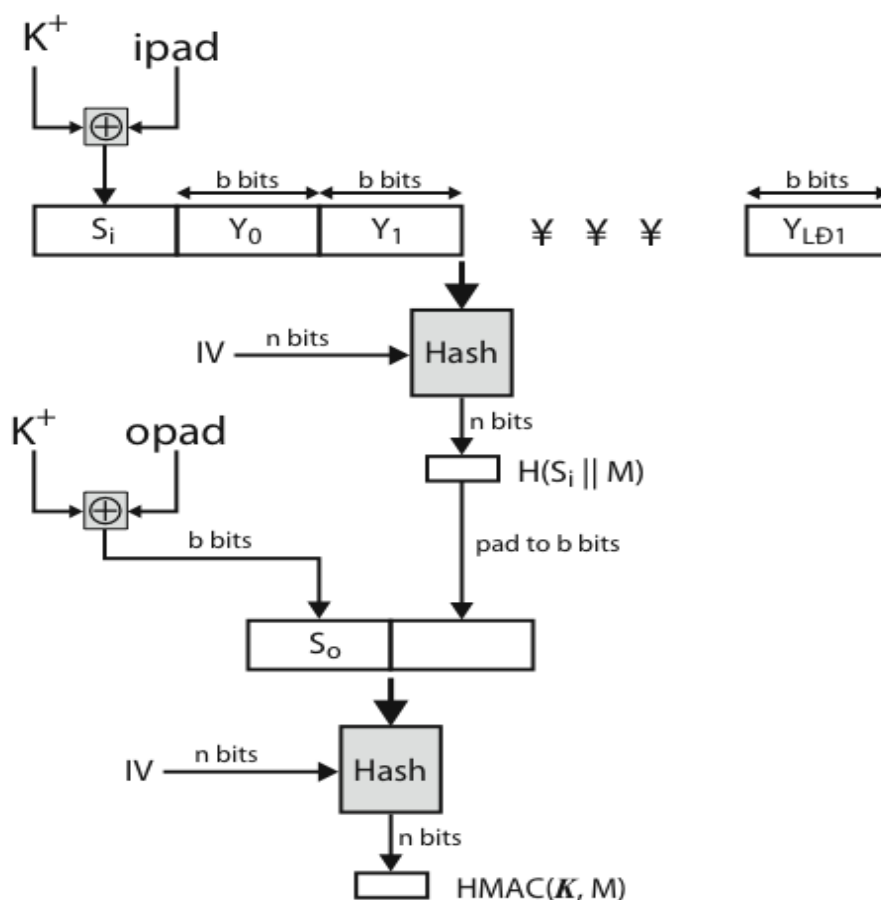


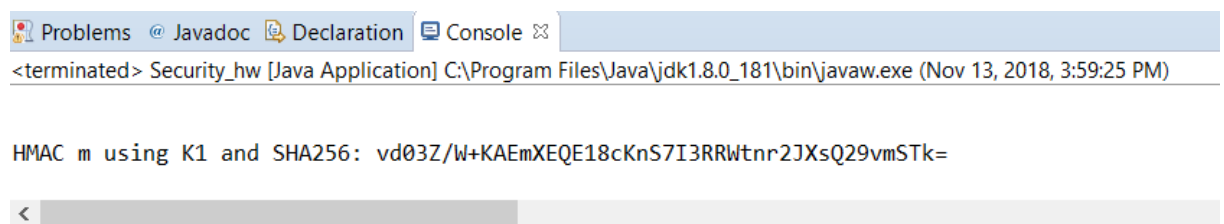
Figure10: Structure of HMAC Algorithm

We chose along text as a message. We digested message using MessageDigest from a factory method by specifying the SHA256 algorithm and we initialized the *Mac* with the key size of 128. After that, we applied HMAC algorithm to using hashed message and AES symmetric key K1.

```
String message = "A security is a fungible, negotiable financial "
    + "instrument that holds some type of monetary value. It "
    + "represents an ownership position in a publicly-traded "
    + "corporation (via stock), a creditor relationship with a "
    + "governmental body or a corporation (represented by owning "
    + "that entity's bond), or rights to ownership as represented "
    + "by an option";

/// Question 4
Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
sha256_HMAC.init(k128);
System.out.println("\n\nHMAC m using K1 and SHA256: " + enc.encodeToString(sha256_HMAC.doFinal(message.getBytes("UTF-8")))+"\n");
```

Figure11: Codes of question4



The screenshot shows a Java IDE with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the program: "HMAC m using K1 and SHA256: vd03Z/W+KAEmXEQE18cKnS7I3RRWtnr2JXsQ29vmSTk=". The output is displayed in a monospaced font with a light blue background.

Figure12: Codes of question4

5. Generate or find any file of size 1MB. Now consider following three algorithms:

- i) AES (128 bit key) in CBC mode.**
- ii) AES (256 bit key) in CBC mode.**
- iii) DES in CBC mode (you need to generate a 56 bit key for this).**

For each of the above algorithms, do the following:

- a) Encrypt the file of size 1MB. Store the result (and submit it with the homework) (Note: IV should be randomly generated, Key = K1 or K2).**
- b) Decrypt the file and store the result. Show that it is the same as the original file.**
- c) Measure the time elapsed for encryption. Write it in your report. Comment on the result.**
- d) For the first algorithm, change Initialization Vector (IV) and show that the corresponding ciphertext changes for the same plaintext (Give the result for both).**

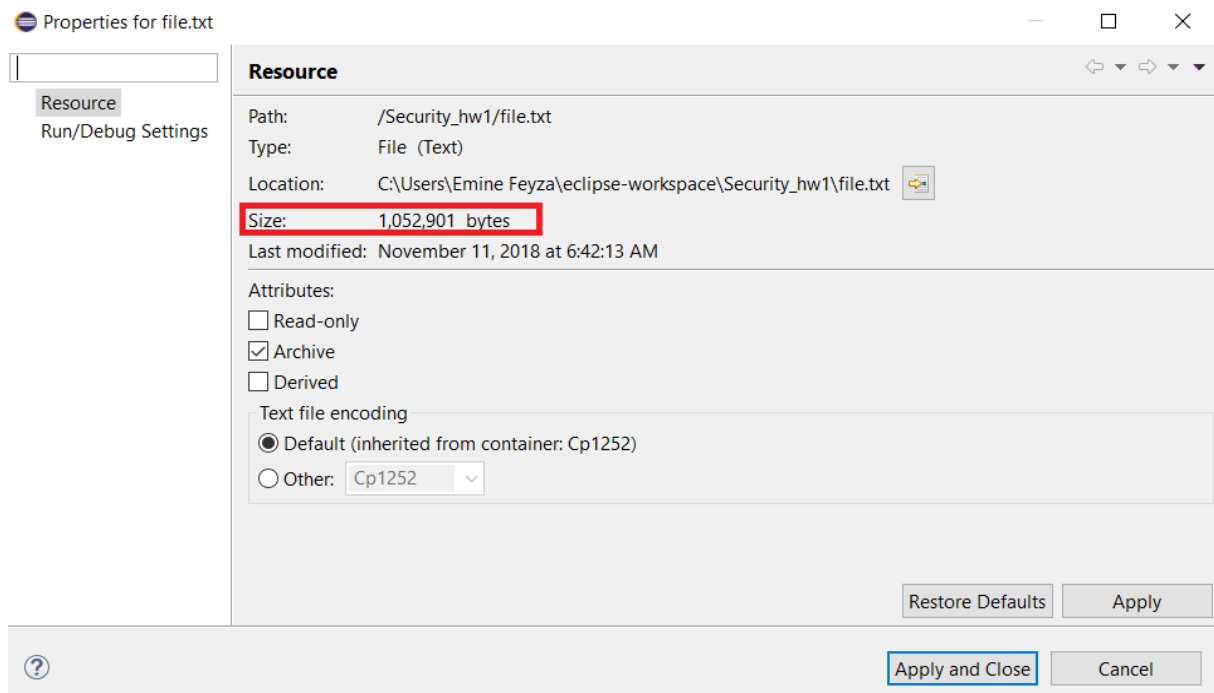


Figure13: Image of file size

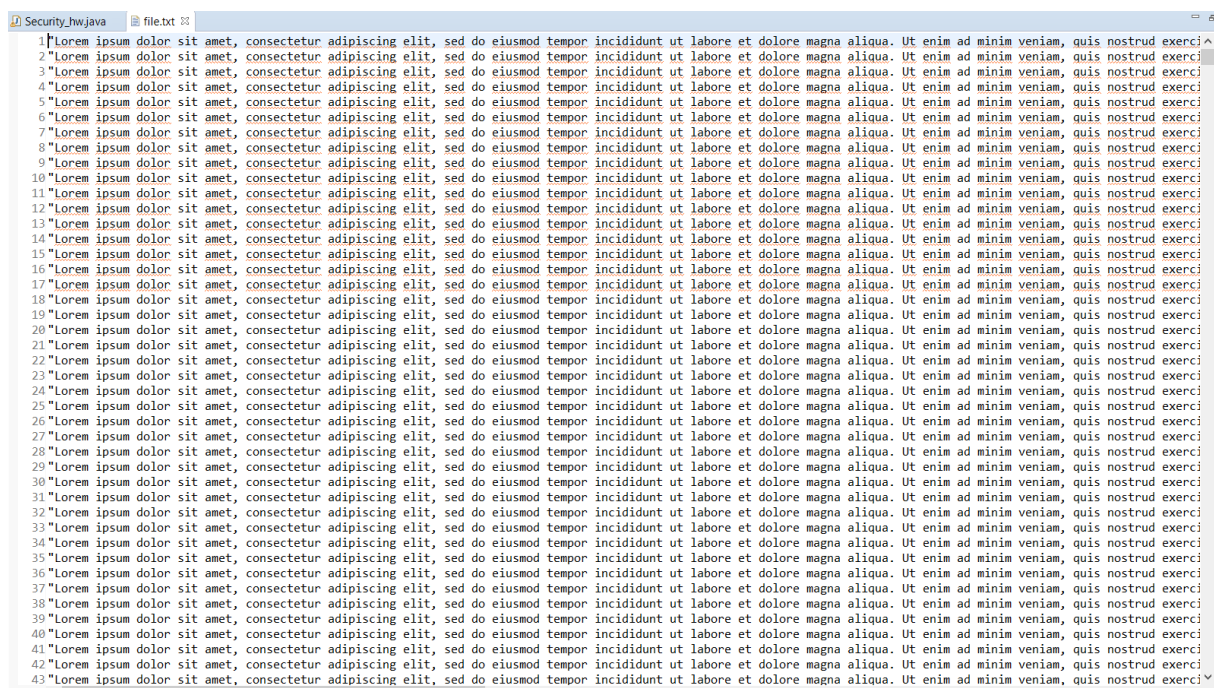


Figure14: Image of file.txt

The CBC encryption mode was invented in IBM in 1976. This mode is about adding XOR each plaintext block to the ciphertext block that was previously produced. The result is then encrypted using the cipher algorithm in the usual way. As a result, every subsequent ciphertext

block depends on the previous one. The first plaintext block is added XOR to a random initialization vector (commonly referred to as IV). The vector has the same size as a plaintext block.

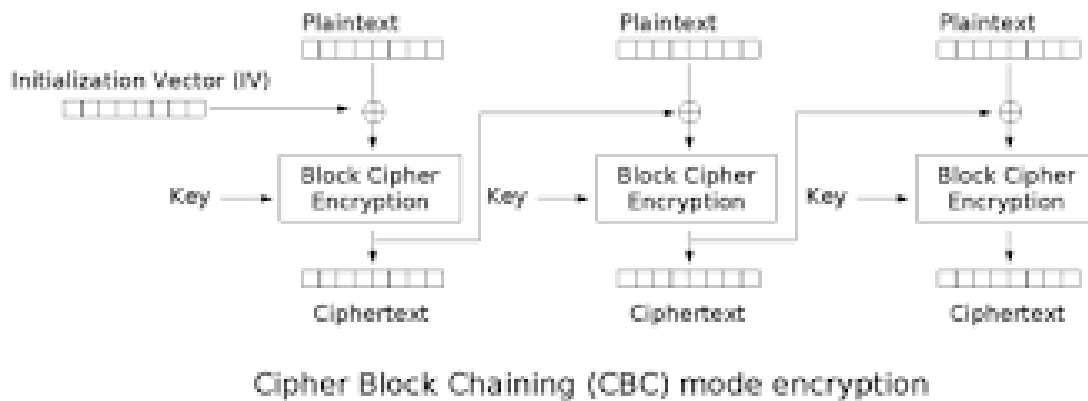


Figure14: Diagram of CBC encryption

During decrypting of a ciphertext block, one should add XOR the output data received from the decryption algorithm to the previous ciphertext block. Because the receiver knows all the ciphertext blocks just after obtaining the encrypted message, he can decrypt the message using many threads simultaneously.

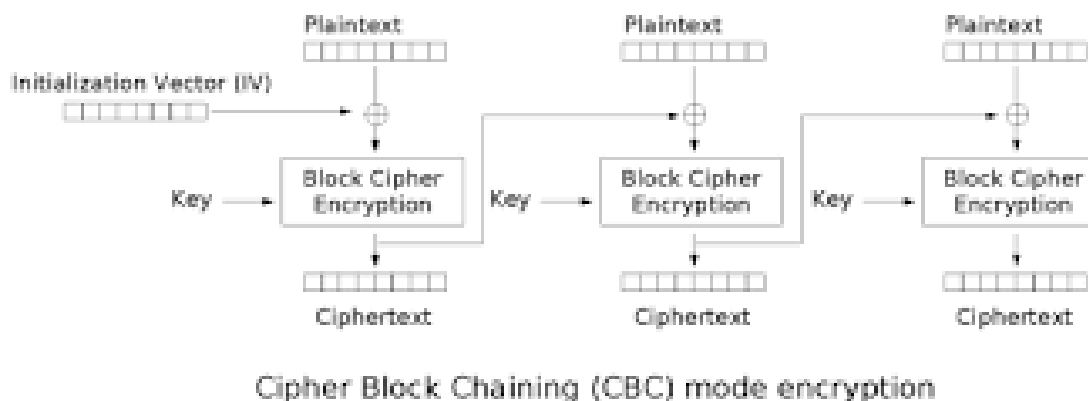


Figure14: Diagram of CBC decryption

We created an IV to initialize CBC encryption or decryption that the same secret key will create different cipher texts. We encrypted the file of size 1MB for AES (128 bit key) in CBC mode, AES (256 bit key) in CBC mode and DES in CBC mode (you need to generate a 56 bit key for

this). And then, we wrote file that encrypted with AES (128 bit key) in CBC mode into encK1.txt file, encrypted with AES (256 bit key) in CBC mode into encK2.txt file and encrypted with DES in CBC mode into encDES.txt file. After that, we decrypt all encrypted files that are encK1.txt, encK2.txt and encDES.txt and wrote decrypted encK1.txt into decK1.txt, encK2.txt into decK2.txt and encDES.txt into decDES.txt.

```

    /// Question 5

    byte[] iv = new byte[16];
    AlgorithmParameterSpec paramSpec = new IvParameterSpec(iv);

```

Figure15: Codes of creating IV for using in CBC mode

```

//AES 128 CBC
try{
    ecipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    dcipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

    // CBC requires an initialization vector
    ecipher.init(Cipher.ENCRYPT_MODE, k128, paramSpec);
    dcipher.init(Cipher.DECRYPT_MODE, k128, paramSpec);

    // Encrypt
    long startTime3=System.nanoTime();
    encryptAES(new FileInputStream("file.txt"),new FileOutputStream("encK1.txt"));
    long endTime3=System.nanoTime();
    long time3=endTime3-startTime3;
    System.out.println("AES128 encryption took "+time3 +" nanosecond");
    // Decrypt
    long startTime4=System.nanoTime();
    decryptAES(new FileInputStream("encK1.txt"),new FileOutputStream("decK1.txt"));
    long endTime4=System.nanoTime();
    long time4=endTime4-startTime4;
    //System.out.println("AES128 decryption took "+time4 +" nanosecond");
}
catch (Exception e){
    e.printStackTrace();
}

```

Figure15: Encryption and decryption of AES(128) in CBC mode

```

//AES 256 CBC
try{

    ecipher.init(Cipher.ENCRYPT_MODE, k256, paramSpec);
    dcipher.init(Cipher.DECRYPT_MODE, k256, paramSpec);

    // Encrypt
    long startTime5=System.nanoTime();
    encryptAES(new FileInputStream("file.txt"),new FileOutputStream("encK2.txt"));
    long endTime5=System.nanoTime();
    long time5=endTime5-startTime5;
    System.out.println("AES256 encryption took "+time5+" nanosecond");
    // Decrypt
    decryptAES(new FileInputStream("encK2.txt"),new FileOutputStream("decK2.txt"));
}
catch (Exception e){
    e.printStackTrace();
}

```

Figure16: Encryption and decryption of AES(256) in CBC mode

```

//DES CBC
try {

    SecretKey secret_key = KeyGenerator.getInstance("DES")
        .generateKey();

    AlgorithmParameterSpec algorithm_specs = new IvParameterSpec(
        initialization_vector);

    ecipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
    ecipher.init(Cipher.ENCRYPT_MODE, secret_key, algorithm_specs);

    dcipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
    dcipher.init(Cipher.DECRYPT_MODE, secret_key, algorithm_specs);

    long startTime6=System.nanoTime();
    encryptDES(new FileInputStream("file.txt"), new FileOutputStream("encDES.txt"));
    long endTime6=System.nanoTime();
    long time6=endTime6-startTime6;
    System.out.println("DES encryption took "+time6+" nanosecond");

    decryptDES(new FileInputStream("encDES.txt"), new FileOutputStream("decDES.txt"));

} catch (Exception e) {
    e.printStackTrace();
}

```

Figure17: Encryption and decryption of DES in CBC mode



Figure18: File that encrypted with AES (128 bit key) in CBC mode

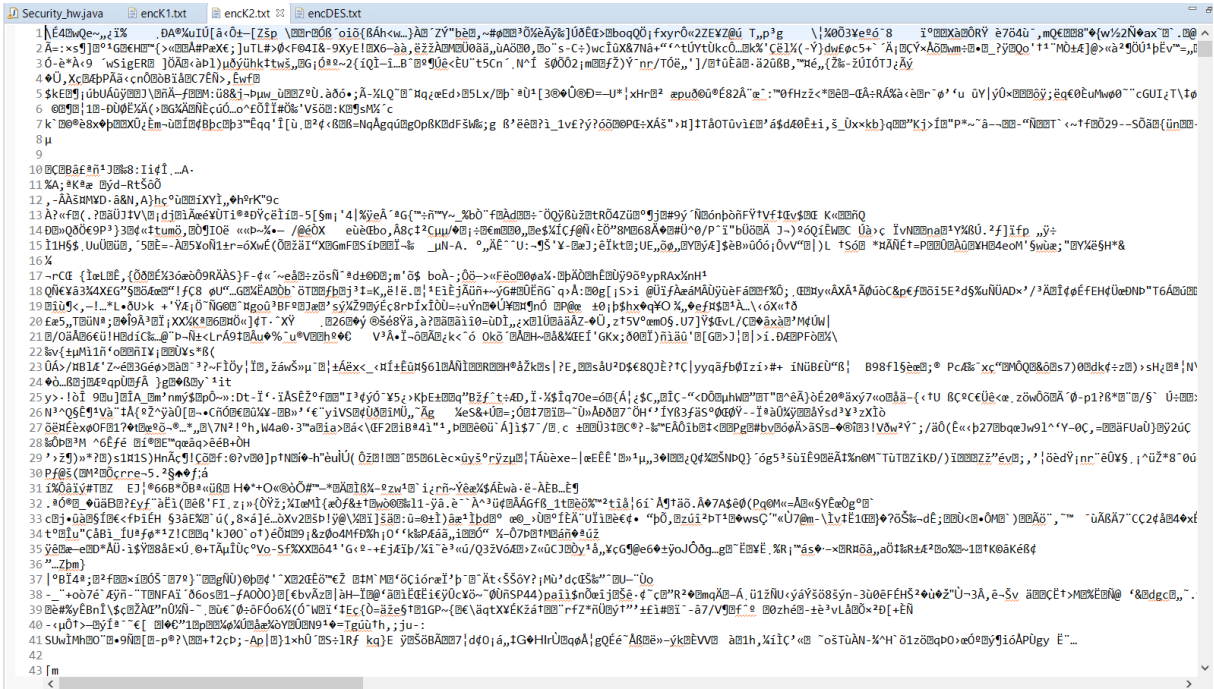


Figure19: File that encrypted with AES (256 bit key) in CBC mode

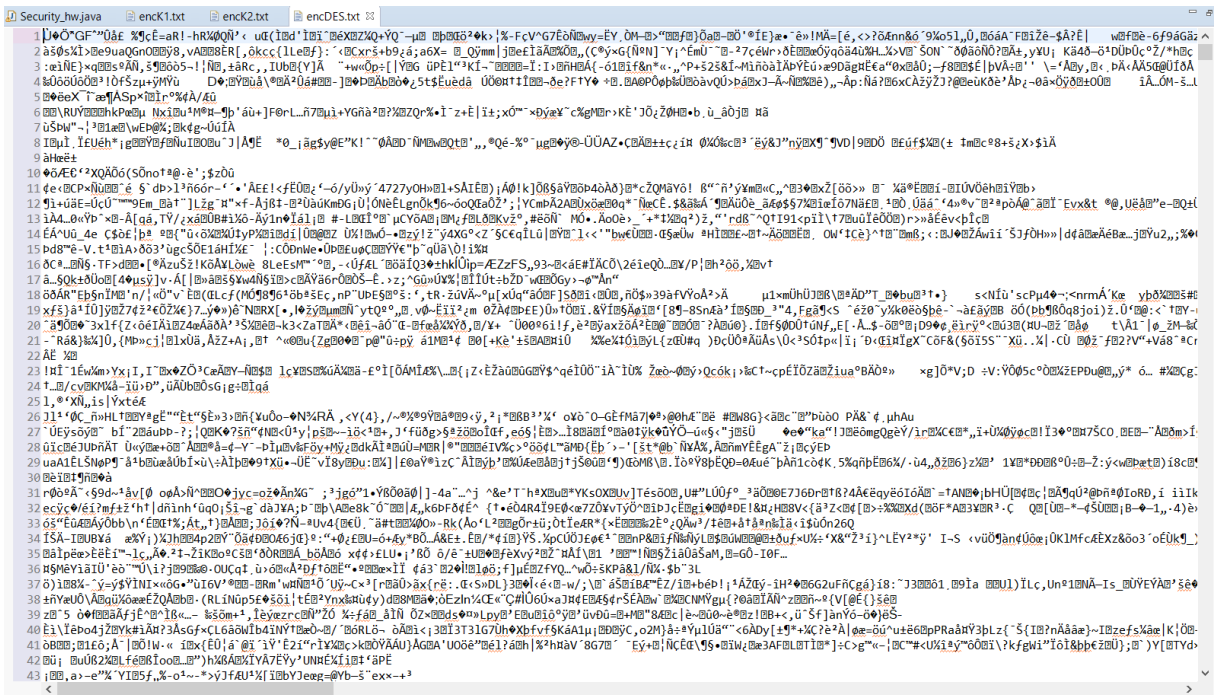


Figure20: File that encrypted with DES in CBC mode

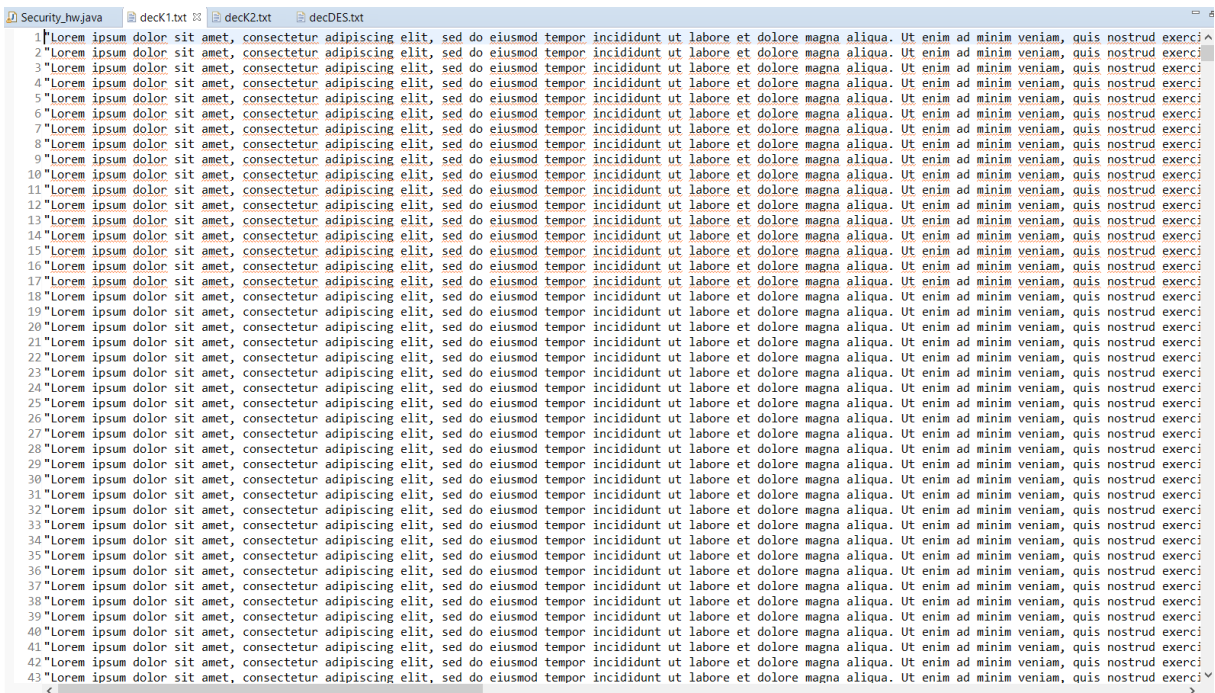
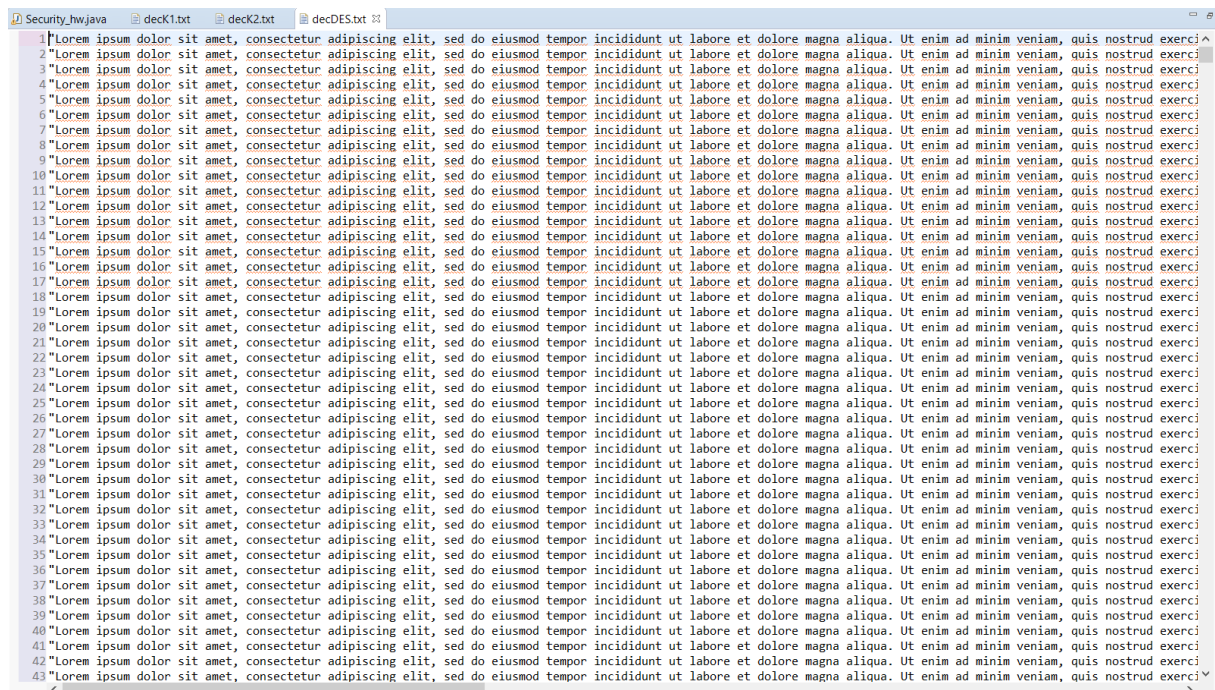
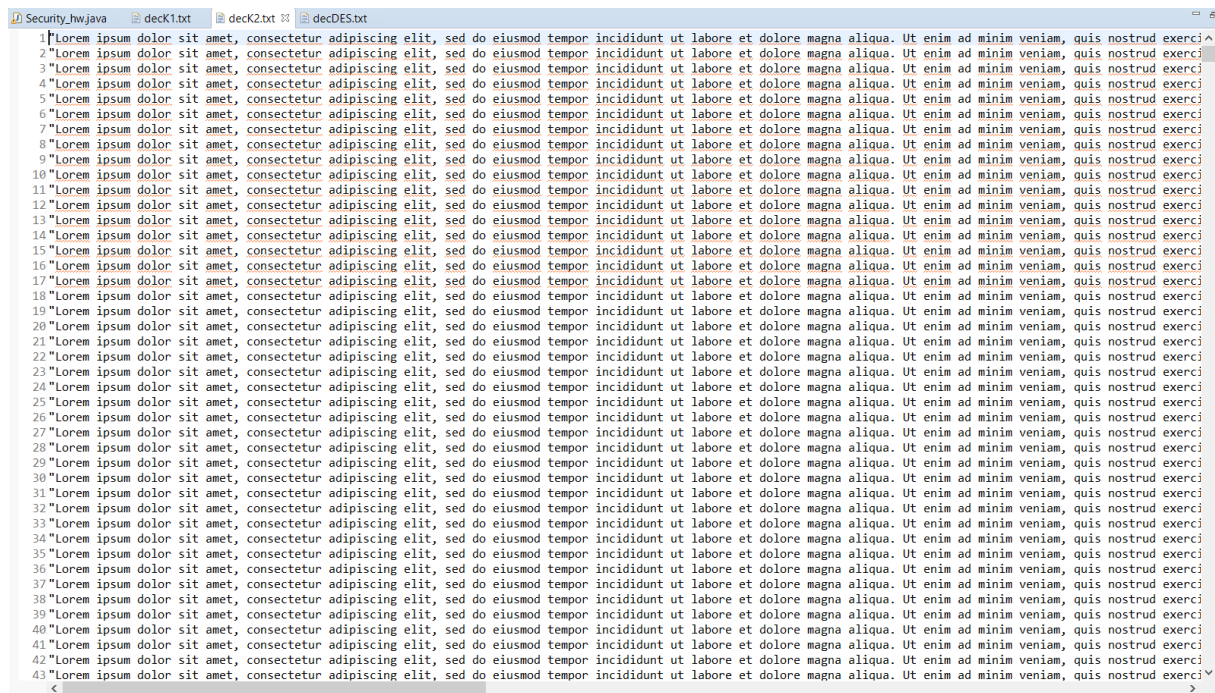


Figure21: File that decrypted with AES (128 bit key) in CBC mode



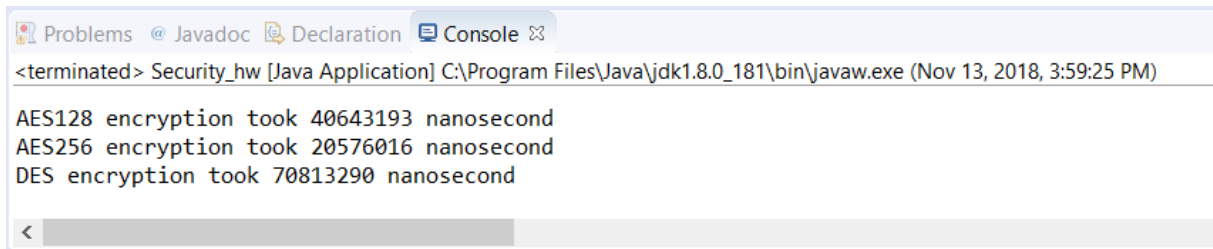


Figure5: Time elapsed for encryption



Figure5: AES (128 bit key) in CBC mode with first IV



Figure5: AES (128 bit key) in CBC mode with second IV

REFERENCES

<https://hackernoon.com/how-does-rsa-work-f44918df914b>

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4863838/>

<https://en.bitcoinwiki.org/wiki/SHA-256>

<https://www.jscape.com/blog/what-is-hmac-and-how-does-it-secure-file-transfers>

<https://searchsecurity.techtarget.com/definition/cipher-block-chaining>