**Assignment: Cryptography Analysis and Implementation**

**Name: Gauravi Mittal**

**Date: 06.06.2023**

**ADVANCED ENCRYPTION STANDARD (AES)**

A more popular and widely used symmetric encryption algorithm that you are likely to encounter these days is the Advanced Encryption Standard (AES). This is at least 6 times faster than Triple DES.

A DES replacement was needed because the key size is too small. As computational power increased, it was deemed vulnerable to exhaustive key lookup attacks. Triple DES was designed to overcome this shortcoming, but proved to be slow. The features of AES are –

* Symmetric Key, Symmetric Block Cipher
* 128-bit data, 128/192/256-bit key
* More powerful and faster than Triple-DES
* Software that can be implemented in C and Java
* Provide full specifications and design details

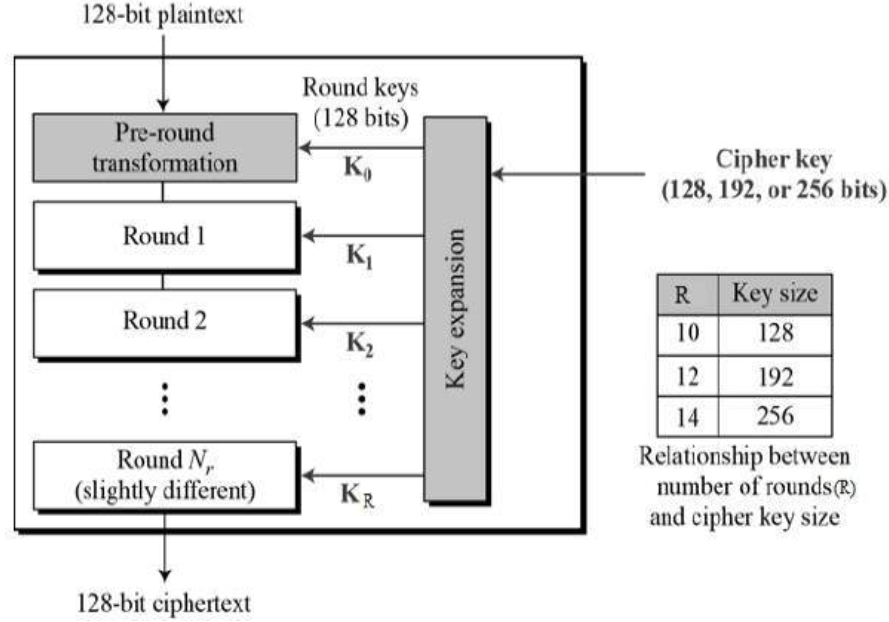
OPERATION OF AES

AES is an iterative cipher rather than a Feistel cipher. It is based on a "permutation network". It consists of a series of linked operations, some of which involve replacing an input with a particular output (permutation) or shuffling bits (permutation).

Interestingly, AES does all its calculations in bytes, not bits. Therefore, AES treats 128 bits of plaintext block as 16 bytes. These 16 bytes are arranged in 4 columns and 4 rows and are treated as a matrix.

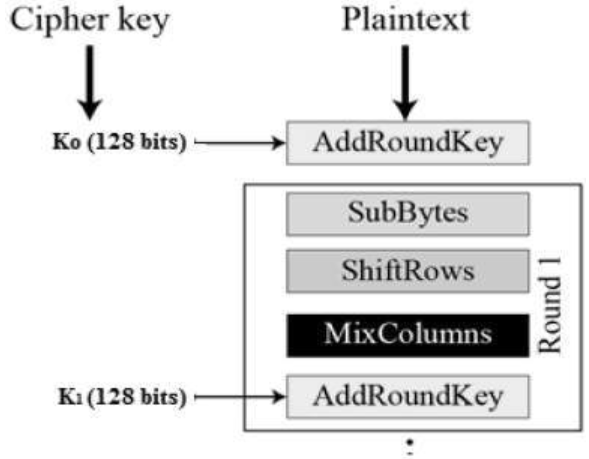
In contrast to DES, the number of rounds in AES is variable and dependent on key length. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key calculated from the original AES key.

The scheme of AES structure is shown in the diagram below –



ENCRYPTION PROCESS

Here we limit ourselves to a description of a typical AES encryption round. Each round consists of four subprocesses. The first-round process is shown below –



BYTE SUBSTITUTION (SUBBYTES)

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

SHIFTROWS

Each of the four rows of the matrix is shifted to the left. Any entries that ‘fall off’ are re-inserted on the right side of row. Shift is carried out as follows −

* First row is not shifted.
* Second row is shifted one (byte) position to the left.
* Third row is shifted two positions to the left.
* Fourth row is shifted three positions to the left.
* The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

MIX COLUMNS

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.

ADD ROUND KEY

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

DECRYPTION PROCESS

The process of decryption of an AES ciphertext is similar to the encryption process in the reverse order. Each round consists of the four processes conducted in the reverse order −

ADD ROUND KEY

MIX COLUMNS

SHIFT ROWS

BYTE SUBSTITUTION

Since sub-processes in each round are in reverse manner, unlike for a Feistel Cipher, the encryption and decryption algorithms needs to be separately implemented, although they are very closely related.

AES ANALYSIS

In present day cryptography, AES is widely adopted and supported in both hardware and software. Till date, no practical cryptanalytic attacks against AES has been discovered. Additionally, AES has built-in flexibility of key length, which allows a degree of ‘future-proofing’ against progress in the ability to perform exhaustive key searches.

However, just as for DES, the AES security is assured only if it is correctly implemented and good key management is employed.

STRENGTHS

* Security: AES is considered highly secure and resistant to various types of attacks. It has undergone extensive analysis and evaluation by the cryptographic community, and no practical vulnerabilities have been found when used properly.
* Efficiency: AES is a computationally efficient algorithm, making it suitable for a wide range of applications. It can encrypt and decrypt data quickly, allowing for efficient processing and transmission of encrypted information.
* Standardization: AES is an internationally recognized and widely adopted standard. Its popularity and usage across different platforms, operating systems, and programming languages ensure interoperability and compatibility.
* Key sizes and modes: AES supports key sizes of 128, 192, and 256 bits, providing flexibility in choosing the desired level of security. It also supports different modes of operation, such as CBC (Cipher Block Chaining) and GCM (Galois/Counter Mode), allowing for customization based on specific requirements.

WEAKNESS

* Key management: The strength of AES heavily depends on the proper management and protection of encryption keys. If the keys are weak or compromised, the entire encryption system becomes vulnerable. Implementing and maintaining robust key management practices is crucial to ensure the security of AES.
* Side-channel attacks: AES itself is resistant to cryptographic attacks, but it can be susceptible to side-channel attacks. Side-channel attacks exploit information leaked during the encryption process, such as power consumption or electromagnetic radiation, to extract the encryption key. Implementing countermeasures, such as secure hardware and software implementations, can mitigate this weakness.
* Quantum computing: While AES is secure against classical computing attacks, it may be vulnerable to attacks from future quantum computers. Quantum computers have the potential to break certain types of cryptographic algorithms, including those used in AES. To address this, researchers are actively working on developing quantum-resistant encryption algorithms.

CODE

import javax.crypto.Cipher;

import javax.crypto.SecretKey;

import javax.crypto.SecretKeyFactory;

import javax.crypto.spec.IvParameterSpec;

import javax.crypto.spec.PBEKeySpec;

import javax.crypto.spec.SecretKeySpec;

import java.nio.charset.StandardCharsets;

import java.security.InvalidAlgorithmParameterException;

import java.security.InvalidKeyException;

import java.security.NoSuchAlgorithmException;

import java.security.spec.InvalidKeySpecException;

import java.security.spec.KeySpec;

import java.util.Base64;

import javax.crypto.BadPaddingException;

import javax.crypto.IllegalBlockSizeException;

import javax.crypto.NoSuchPaddingException;

public class Main

{

/\* Private variable declaration \*/

private static final String SECRET\_KEY = "123456789";

private static final String SALTVALUE = "abcdefg";

/\* Encryption Method \*/

public static String encrypt(String strToEncrypt)

{

try

{

/\* Declare a byte array. \*/

byte[] iv = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

IvParameterSpec ivspec = new IvParameterSpec(iv);

/\* Create factory for secret keys. \*/

SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");

/\* PBEKeySpec class implements KeySpec interface. \*/

KeySpec spec = new PBEKeySpec(SECRET\_KEY.toCharArray(), SALTVALUE.getBytes(), 65536, 256);

SecretKey tmp = factory.generateSecret(spec);

SecretKeySpec secretKey = new SecretKeySpec(tmp.getEncoded(), "AES");

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

cipher.init(Cipher.ENCRYPT\_MODE, secretKey, ivspec);

/\* Retruns encrypted value. \*/

return Base64.getEncoder()

.encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets.UTF\_8)));

}

catch (InvalidAlgorithmParameterException | InvalidKeyException | NoSuchAlgorithmException | InvalidKeySpecException | BadPaddingException | IllegalBlockSizeException | NoSuchPaddingException e)

{

System.out.println("Error occured during encryption: " + e.toString());

}

return null;

}

/\* Decryption Method \*/

public static String decrypt(String strToDecrypt)

{

try

{

/\* Declare a byte array. \*/

byte[] iv = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

IvParameterSpec ivspec = new IvParameterSpec(iv);

/\* Create factory for secret keys. \*/

SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");

/\* PBEKeySpec class implements KeySpec interface. \*/

KeySpec spec = new PBEKeySpec(SECRET\_KEY.toCharArray(), SALTVALUE.getBytes(), 65536, 256);

SecretKey tmp = factory.generateSecret(spec);

SecretKeySpec secretKey = new SecretKeySpec(tmp.getEncoded(), "AES");

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");

cipher.init(Cipher.DECRYPT\_MODE, secretKey, ivspec);

/\* Retruns decrypted value. \*/

return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));

}

catch (InvalidAlgorithmParameterException | InvalidKeyException | NoSuchAlgorithmException | InvalidKeySpecException | BadPaddingException | IllegalBlockSizeException | NoSuchPaddingException e)

{

System.out.println("Error occured during decryption: " + e.toString());

}

return null;

}

/\* Driver Code \*/

public static void main(String[] args)

{

/\* Message to be encrypted. \*/

String originalval = "This is Gauravi Mittal";

/\* Call the encrypt() method and store result of encryption. \*/

String encryptedval = encrypt(originalval);

/\* Call the decrypt() method and store result of decryption. \*/

String decryptedval = decrypt(encryptedval);

/\* Display the original message, encrypted message and decrypted message on the console. \*/

System.out.println("Original value: " + originalval);

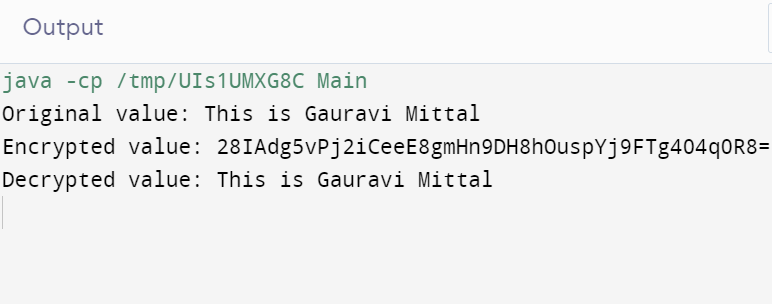
System.out.println("Encrypted value: " + encryptedval);

System.out.println("Decrypted value: " + decryptedval);

}

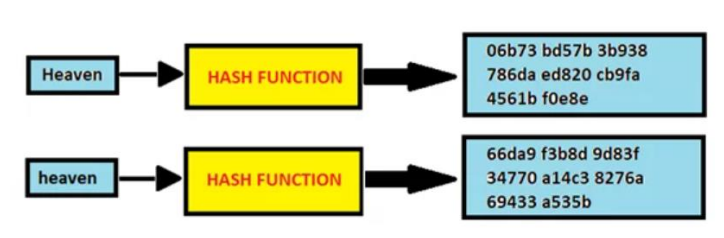
}

OUTPUT



**SHA-1 HASH**

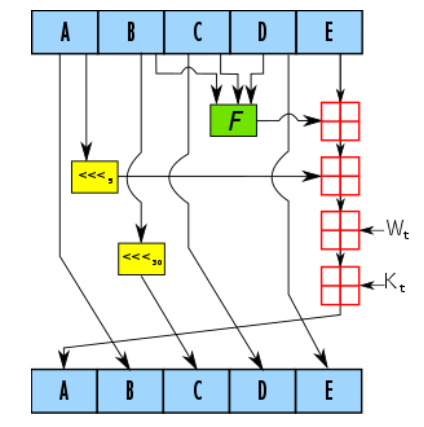
SHA stands for Secure Hash Algorithm. SHA is a modified version of MD5 and is used for hashing data and certificates. A hashing algorithm truncates input data into a smaller form that is incomprehensible using bitwise operations, modular addition, and compression functions. You may be wondering if the hash can be cracked or decrypted. Hashing is like encryption. The only difference between hashing and encryption is that hashing is one-way. Once the data has been hashed, the resulting hash digest cannot be cracked unless a brute force attack is used. See the image below for how the SHA algorithm works. SHA works this way, and even if his one character in the message changes, another hash will be generated. For example, hashes of two similar but different messages, namely Ten and Sora are different. However, there is only one case difference.



The original message is hashed with SHA-1 resulting in the hash digest "06b73bd57b3b938786daed820cb9fa4561bf0e8e". If a second similar message were hashed with his SHA-1, the hash digest would be: This is called the avalanche effect. This effect is important in cryptography because it means that even a small change in the input message will completely change the output. This prevents an attacker from understanding the initial contents of the hash digest and telling the recipient of the message whether the message has been modified in transit. SHA also helps reveal if the original message has been modified in any way. By looking at the original hash digest, the user can determine if even a single character has changed since the hash digest is completely different. One of the most important parts of SHA is that it is deterministic. This means that any computer or user can recreate the hash digest as long as the hash function used is known. SHA's determinism is one of the reasons why all SSL certificates on the Internet should be hashed with the SHA-2 function.

As mentioned earlier, all digital signatures and certificates associated with SSL/TLS connections require a secure hashing algorithm, but SHA has many more uses. Applications such as SSH, S-MIME (Secure/Multipurpose Internet Mail Extensions), and IPSec also use SHA. SHA is also used to hash passwords, so the server only needs to remember the hash, not the password. This way, if an attacker steals a database containing all hashes, they won't have direct access to all plaintext passwords and would also need to find a way to crack the hashes in order to be able to use the passwords. SHA also serves as an indicator of file integrity. If the file is modified in transit, the resulting hash digest created from the hash function will not match the hash digest originally created and sent by the owner of the file.

Now that you've learned what SHA is used for, why use a secure hashing algorithm? A common reason is its ability to thwart attackers. Some methods, such as brute force attacks, can reveal the plaintext of the hash digest, but SHA makes these tactics very difficult. Cracking passwords hashed with SHA-2 can take years or even decades, wasting resources and time on simple passwords and deterring many attackers. You can. Another reason to use SHA is the uniqueness of all hash digests. If SHA-2 is used, there may be few or no collisions. In other words, changing her one word in the message completely changes the hash digest. With few or no collisions, we find no pattern that would allow an attacker to easily defeat a secure hashing algorithm. These are just some of the reasons why SHA is so widely used.

****

STRENGTHS

* Simplicity: SHA-1 is relatively simple and straightforward to implement, making it efficient in terms of computational resources and processing time.
* Widely Supported: SHA-1 is widely supported and used in various applications and protocols, which makes it compatible with a wide range of systems.
* Performance: SHA-1 can generate hash values quickly, making it suitable for applications that require high-speed hashing.

WEAKNESS

* Vulnerability to Collision Attacks: SHA-1 is vulnerable to collision attacks, where two different inputs can produce the same hash value. This weakness makes it insecure for applications that require strong data integrity, such as digital signatures or certificate authorities.
* Weakening Cryptographic Security: Over time, cryptographic attacks have improved, and the computational power available to attackers has increased. As a result, SHA-1's security is considered weak by modern standards.
* Deprecation: Due to its vulnerabilities, SHA-1 has been deprecated by many organizations and industry standards bodies. It is no longer recommended for use in security-critical applications and has been replaced by more secure hashing algorithms like SHA-256 and SHA-3.

CODE

// Java program to calculate SHA-1 hash value

import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

public class Hash {

public static String encryptThisString(String input)

{

try {

// getInstance() method is called with algorithm SHA-1

MessageDigest md = MessageDigest.getInstance("SHA-1");

// 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);

}

}

public static void main(String args[]) throws

NoSuchAlgorithmException

{

System.out.println("HashCode Generated by SHA-1 for: ");

String s1 = "This is Gauravi Mittal";

System.out.println("\n" + s1 + " : " + encryptThisString(s1));

String s2 = "hello world";

System.out.println("\n" + s2 + " : " + encryptThisString(s2));

}

}

OUTPUT



**RSA**

**Generating Public Key**

1. Select two prime no's. Suppose P = 53 and Q = 59.

Now First part of the Public key : n = P\*Q = 3127.

2. We also need a small exponent say e :

But e Must be

-An integer.

-Not be a factor of n.

-1 < e < Φ(n) [Φ(n) is discussed below],

Let us now consider it to be equal to 3.

The public key has been made of n and e

**Generating Private Key**

1. We need to calculate Φ(n) :

Such that Φ(n) = (P-1)(Q-1)

so, Φ(n) = 3016

2. Now calculate Private Key, d :

d = (k\*Φ(n) + 1) / e for some integer k

3. For k = 2, value of d is 2011.

The private key has been made of d

**IMPLEMENTATION**

* Consider two prime numbers p and q.
* Compute n = p\*q
* Compute ϕ(n) = (p – 1) \* (q – 1)
* Choose e such gcd(e , ϕ(n) ) = 1
* Calculate d such e\*d mod ϕ(n) = 1
* Public Key {e,n} Private Key {d,n}
* Cipher text C = Pe mod n where P = plaintext
* For Decryption D = Dd mod n where D will refund the plaintext.

STRENGTHS

* Security: RSA is considered a strong encryption algorithm when used properly. The security of RSA relies on the difficulty of factoring large composite numbers into their prime factors, which is computationally intensive and time-consuming.
* Public-Key Encryption: RSA is one of the most widely used algorithms for public-key encryption. It allows secure communication between two parties without the need for a shared secret key.
* Digital Signatures: RSA can be used to create digital signatures, which provide authenticity, integrity, and non-repudiation of digital documents. Digital signatures generated using RSA can verify that a document was signed by a particular entity and detect any tampering with the document.
* Key Exchange: RSA can also be used for secure key exchange protocols, such as the Diffie-Hellman key exchange. It enables two parties to establish a shared secret key over an insecure communication channel.

WEAKNESS

* Key Length: The strength of RSA depends on the size of the keys used. As computing power increases, the recommended key lengths for RSA need to be periodically updated to maintain security. With advancements in technology, it is important to use longer key lengths to resist attacks from increasingly powerful computers.
* Computational Cost: RSA encryption and decryption operations are computationally expensive, especially for large key sizes. This can pose challenges in scenarios where real-time encryption and decryption are required, such as in high-throughput systems or resource-constrained devices.
* Vulnerability to Quantum Computers: RSA is vulnerable to attacks by quantum computers. Shor's algorithm, if successfully implemented on a large-scale quantum computer, could factor large numbers efficiently, rendering RSA insecure. As quantum computing technology advances, it becomes crucial to develop and adopt post-quantum cryptographic algorithms that are resistant to quantum attacks.
* Lack of Perfect Forward Secrecy: RSA does not provide perfect forward secrecy. If an attacker gains access to a private key, they can decrypt past communications encrypted with the corresponding public key. This is in contrast to some other encryption algorithms like Diffie-Hellman, which provide forward secrecy by generating ephemeral keys for each session.

CODE

import java.math.\*;

import java.util.\*;

class RSA {

public static void main(String args[])

{

int p, q, n, z, d = 0, e, i;

// The number to be encrypted and decrypted

int msg = 12;

double c;

BigInteger msgback;

// 1st prime number p

p = 3;

// 2nd prime number q

q = 11;

n = p \* q;

z = (p - 1) \* (q - 1);

System.out.println("the value of z = " + z);

for (e = 2; e < z; e++) {

// e is for public key exponent

if (gcd(e, z) == 1) {

break;

}

}

System.out.println("the value of e = " + e);

for (i = 0; i <= 9; i++) {

int x = 1 + (i \* z);

// d is for private key exponent

if (x % e == 0) {

d = x / e;

break;

}

}

System.out.println("the value of d = " + d);

c = (Math.pow(msg, e)) % n;

System.out.println("Encrypted message is : " + c);

// converting int value of n to BigInteger

BigInteger N = BigInteger.valueOf(n);

// converting float value of c to BigInteger

BigInteger C = BigDecimal.valueOf(c).toBigInteger();

msgback = (C.pow(d)).mod(N);

System.out.println("Decrypted message is : "

+ msgback);

}

static int fn(int e, int z)

{

if (e == 0)

return z;

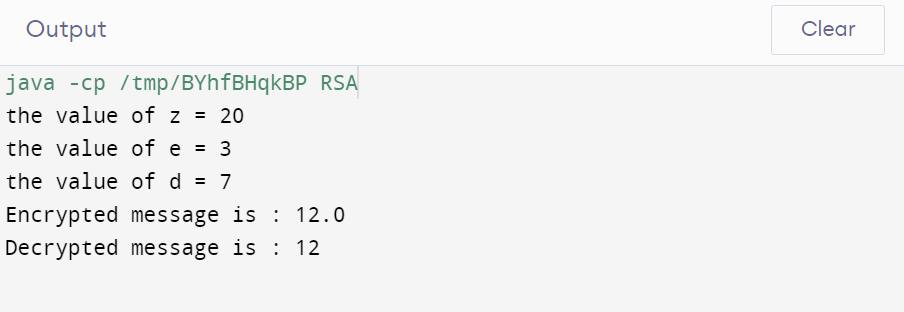
else

return fn(z % e, e);

}

}

OUTPUT



-----THANK YOU-----