**Experiment No: 10**

* **Aim**

Use RSA for generation and verification of digital signature on file.

* **Introduction**

RSA (Rivest-Shamir-Adleman) is a widely used cryptographic algorithm that plays a fundamental role in securing digital communications and data integrity. One of its key applications is in the generation and verification of digital signatures for files. Digital signatures are essential for ensuring the authenticity and integrity of electronic documents and messages in an age where information is frequently transmitted and stored digitally.

RSA is a public-key cryptosystem that utilizes a pair of keys: a public key and a private key. The public key is available to anyone and is used for encryption and verification, while the private key is kept secret and is used for decryption and signature generation. The core principle behind RSA is the mathematical difficulty of factoring large composite numbers into their prime components, which serves as the foundation for its security.

In the context of generating and verifying digital signatures on files, RSA works as follows:

1. Signature Generation:

* The sender uses their private key to generate a unique digital signature for a file they want to sign. This signature is a mathematical representation of the file's content and a unique private key, and it serves as proof of the file's origin and integrity.
* The private key is used to perform a mathematical operation on a hash (a fixed-size representation) of the file. This operation creates the digital signature, which is then attached to the file.

1. Signature Verification:

* The recipient, or anyone interested in verifying the file's authenticity, uses the sender's public key to check the digital signature. The public key is used to reverse the signature into a hash value.
* The recipient also computes a hash of the received file. If the computed hash matches the one obtained by reversing the signature, it indicates that the file has not been tampered with and is genuinely from the claimed sender.

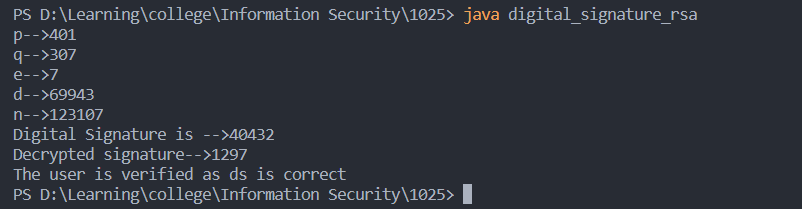
In this way, RSA digital signatures provide a reliable method for verifying the source and integrity of electronic files. Even if the public key is well-known and can be freely distributed, it is computationally infeasible for anyone to generate a valid signature without access to the corresponding private key. This fundamental property of RSA makes it a robust tool for ensuring the trustworthiness of digital documents in various applications, including secure email communication, software distribution, and digital contracts.

* **Program**

1. **For Authentication**

|  |
| --- |
| import java.util.Random;  class keySet{      long e;      long d;      long n;  }  public class digital\_signature\_rsa {      public static boolean **isPrime**(long a){          for (int i = 2; i < (int)a/2 ; i++) {              if(a%i == 0){                  return false;              }          }          return true;      }      public static boolean **isRelativelyPrime**(long a,long b){          long counter = 2;          long check = 0;          long temp = a>b ? b : a;          for (long i = 2; i <= temp; i++) {              if(a%i==0 && b%i ==0){                  return false;              }          }          return true;      }      public static keySet **generateDE**(long p,long q){          keySet a = new **keySet**();          long n = p\*q;          long totient = (p-1)\*(q-1);          long e = 2;          for (long i = 2; i < totient; i++) {              if(**isRelativelyPrime**(i,totient)){                  e = i;                  break;              }          }          long d = 0;          for (long i = 0; i < e; i++) {              long temp = i \* totient +1;              if(temp%e == 0){                  d = temp/e;                  break;              }          }          a.d = d;          a.e = e;          a.n = n;          return a;      }      public static long **getRandomPrime**(int a){          Random rand = new **Random**();          int min = 10\*4;          String temp  = "";          for (int i = 0; i <a ; i++) {              temp += "9";          }          int max = Integer.**parseInt**(temp);          while (true){              long randomNum = rand.**nextInt**(min,max);              if(**isPrime**(randomNum)){                  return randomNum;              }          }      }      public static long **powMod**(long x,long y,long n){          long power = 2;          long finalNum = 1;          while (power != y+1){              if(power == 2){                  long temp = finalNum \* x \* x ;                  finalNum = temp % n;              }              else {                  long temp = finalNum \* x;                  finalNum = temp % n;              }              power++;          }          return finalNum;      }      public static long **encryption**(long d,long n,long plain\_txt){          long cipher = **powMod**(plain\_txt,d,n);          return cipher;      }      public static long **decryption**(long e,long n,long cipher){          long plain = **powMod**(cipher,e,n);          return plain;      }      public static void **main**(String[] args) {          long p = **getRandomPrime**(3);          long q = **getRandomPrime**(3);          keySet keys = **generateDE**(p,q);          System.out.**println**("p-->"+p);          System.out.**println**("q-->"+q);          System.out.**println**("e-->"+keys.e);          System.out.**println**("d-->"+keys.d);          System.out.**println**("n-->"+keys.n);          long msg = 1297;          long cipher = **encryption**(keys.d,keys.n,msg);          System.out.**println**("Digital Signature is -->"+cipher);          long verifiedSign = **decryption**(keys.e,keys.n,cipher);          System.out.**println**("Decrypted signature-->"+verifiedSign);          if(msg == verifiedSign){              System.out.**println**("The user is verified as ds is correct");          }          else {              System.out.**println**("User is not verified");          }      }  } |

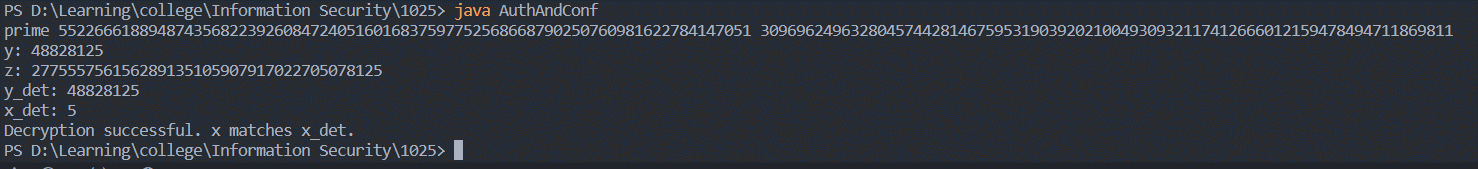
* **Output (Program)**

****

1. **For Authentication and Confidentiality**

|  |
| --- |
| import java.math.BigInteger;  import java.util.Random;  public class AuthAndConf {      public static BigInteger[] **select\_prime\_number**() {          Random rand = new **Random**();          BigInteger[] prime\_number = new BigInteger[2];          int max\_attempts = 1000;          int count = 0;          while (count < 2 && max\_attempts > 0) {              BigInteger randNumber = new **BigInteger**(256, rand);              if (randNumber.**isProbablePrime**(100)) {                  prime\_number[count] = randNumber;                  count++;              }              max\_attempts--;          }          return prime\_number;      }      public static BigInteger **calculate\_n**(BigInteger[] prime\_number) {          return prime\_number[0].**multiply**(prime\_number[1]);      }      public static BigInteger **calculate\_fi\_n**(BigInteger[] prime\_numbers) {          BigInteger p1 = prime\_numbers[0].**subtract**(BigInteger.ONE);          BigInteger p2 = prime\_numbers[1].**subtract**(BigInteger.ONE);          return p1.**multiply**(p2);      }      public static boolean **calculate\_gcd**(BigInteger e, BigInteger fi\_n) {          BigInteger temp;          while (!fi\_n.**mod**(e).**equals**(BigInteger.ZERO)) {              temp = fi\_n.**mod**(e);              fi\_n = e;              e = temp;          }          return e.**equals**(BigInteger.ONE);      }      public static BigInteger **find\_e**(BigInteger fi\_n) {          BigInteger e = BigInteger.TWO;          while (e.**compareTo**(fi\_n) < 0) {              boolean for\_e = **calculate\_gcd**(e, fi\_n);              if (for\_e) {                  return e;              } else {                  e = e.**add**(BigInteger.ONE);              }          }          return e;      }      public static BigInteger **find\_d**(BigInteger a, BigInteger m) {          BigInteger m0 = m;          BigInteger x0 = BigInteger.ZERO;          BigInteger x1 = BigInteger.ONE;          while (a.**compareTo**(BigInteger.ONE) > 0) {              BigInteger q = a.**divide**(m);              BigInteger t = m;              m = a.**mod**(m);              a = t;              t = x0;              x0 = x1.**subtract**(q.**multiply**(x0));              x1 = t;          }          if (x1.**compareTo**(BigInteger.ZERO) < 0) {              x1 = x1.**add**(m0);          }          return x1;      }      public static void **main**(String[] args) {          BigInteger[] prime\_a = **select\_prime\_number**();          BigInteger[] prime\_b = **select\_prime\_number**();          System.out.**println**("prime " + prime\_a[0] + " " + prime\_a[1]);          BigInteger n\_a = **calculate\_n**(prime\_a);          BigInteger n\_b = **calculate\_n**(prime\_b);          // System.out.println("n: " + n\_a + " " + n\_b);          BigInteger fi\_n\_a = **calculate\_fi\_n**(prime\_a);          BigInteger fi\_n\_b = **calculate\_fi\_n**(prime\_b);          // System.out.println("fi: " + fi\_n\_a + " " + fi\_n\_b);          BigInteger e\_a = **find\_e**(fi\_n\_a);          BigInteger e\_b = **find\_e**(fi\_n\_b);          // System.out.println("e: " + e\_a + " " + e\_b);          BigInteger d\_a = **find\_d**(e\_a, fi\_n\_a);          BigInteger d\_b = **find\_d**(e\_b, fi\_n\_b);          // System.out.println("d: " + d\_a + " " + d\_b);          BigInteger x = BigInteger.**valueOf**(5);          BigInteger y = x.**modPow**(e\_a, n\_a);          BigInteger z = y.**modPow**(e\_b, n\_b);          System.out.**println**("y: " + y);          System.out.**println**("z: " + z);          BigInteger y\_det = z.**modPow**(d\_b, n\_b);          BigInteger x\_det = y\_det.**modPow**(d\_a, n\_a);          System.out.**println**("y\_det: " + y\_det);          System.out.**println**("x\_det: " + x\_det);          if (x.**equals**(x\_det)) {              System.out.**println**("Decryption successful. x matches x\_det.");          } else {              System.out.**println**("Decryption failed. x does not match x\_det.");          }      }  } |

* **Output (Program)**

****

* **Cryptanalysis**

1. Computational Intensity: RSA encryption and decryption operations involve complex mathematical operations, especially when working with large prime numbers. This computational intensity can be a disadvantage for devices with limited processing power, as it can result in slower performance when generating and verifying digital signatures.
2. Key Management: Managing the public and private keys in an RSA system can be challenging, particularly in large-scale applications. Key distribution and key storage require careful planning and security measures to prevent key exposure or loss.
3. Key Length and Security: The security of RSA is highly dependent on the length of the keys used. As computational power advances, longer key lengths are required to maintain the same level of security. Longer keys can slow down cryptographic operations and increase the amount of data required for storage and transmission.
4. Size of Signatures: RSA digital signatures tend to be larger in size compared to some other signature algorithms like elliptic curve cryptography (ECC). This can result in larger files and more bandwidth usage, which may be a concern in certain applications, particularly when dealing with limited network resources.
5. Vulnerability to Quantum Computing: While not a current concern, RSA is theoretically vulnerable to attacks by future quantum computers. Quantum computers could potentially break the RSA algorithm by factoring large numbers quickly. This potential threat necessitates a transition to quantum-resistant cryptographic algorithms in the long term.

* **Application**

1. Secure Communication:

* Email Encryption: RSA is often used to digitally sign and encrypt emails. Senders can use their private key to sign emails, proving their authenticity to recipients. Recipients use the sender's public key to verify the digital signature, ensuring that the email has not been tampered with during transit.
* Secure Messaging Apps: Many secure messaging apps use RSA for end-to-end encryption, ensuring that only the intended recipient can read the messages. Digital signatures can also verify the authenticity of messages.

1. Software Distribution:

* Software Updates: Software vendors and developers use RSA signatures to ensure the integrity and authenticity of software updates and patches. Users can verify that the updates have not been altered or compromised during download or installation.
* Code Signing: Developers digitally sign their code with RSA keys to prove its authenticity and origin. This is crucial for applications, plugins, and drivers, as users can verify that the code is from a trusted source.

3. Document Verification and Authentication:

* Legal Contracts and Documents: In legal and business contexts, RSA digital signatures are used to verify the authenticity and integrity of contracts, agreements, and legal documents. This ensures the non-repudiation of signed documents, meaning the signatory cannot deny their signature.
* Certificate Authorities (CAs): Digital certificates issued by CAs use RSA for signature generation. These certificates are used to verify the identity of websites (SSL/TLS certificates) and individuals (digital ID certificates) in secure online transactions and communications.
* **References**

1. <https://chat.openai.com>
2. <https://www.geeksforgeeks.org/rsa-and-digital-signatures/>