**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.4

*la Tehnologii de Securitatea Informațională*

A efectuat:

st. gr. TI-214 Buza Cătălin

A verificat: Bulai Rodica

Chişinău - 2023

**Scopul lucrării:**

**1. Algoritmul DES/AES**

* + Studierea algoritmului DES sau AES
  + Modelul matematic și funcționalul
  + Domeniile de aplicare

**2. Algoritmul RSA**

* + Studierea modelului matematic al algoritmului RSA
  + Generarea cheilor
  + Domenii de aplicare
  + Semnătura digitală

**3. Realizarea aplicației**

* Realizarea unei aplicații într-un limbaj de programare la alegere
* Interfața principală va reprezenta un meniu cu algoritmii DES/AES, RSA și semnătura digitală
* Opțiunile la alegere sunt: Implementarea DES sau AES, la semnătura digitală fie RSA sau DSA

Studierea algoritmului AES

**AES** (de la **Advanced Encryption Standard** – in limba engleza, Standard Avansat de Criptare), cunoscut si sub numele de Rijndael, este un algoritm standardizat pentru criptarea simetrica, pe blocuri, folosit astazi pe scara larga in aplicatii si adoptat ca standard de organizatia guvernamentala americana NIST. Standardul oficializeaza algoritmul dezvoltat de doi criptografi belgieni, Joan Daemen si Vincent Rijmen si trimis la NIST pentru selectie sub numele Rijndael.

Deoarece DES devenise vulnerabil din cauza lungimii prea mici a cheii, NIST a recomandat utilizarea 3DES, un algoritm care consta in esenta in aplicarea de trei ori a DES. Desi 3DES s-a dovedit a fi un algoritm puternic, el este relativ lent in implementarile software, motiv pentru care NIST a lansat in 1997 o cerere de propuneri pentru un algoritm care sa-l inlocuiasca. S-a pornit de la 21 de propuneri acceptate initial, apoi prin eliminari numarul lor a fost redus la 15, si apoi la 5, din care a fost ales in cele din urma algoritmul propus de doi criptografi belgieni, Joan Daemen si Vincent Rijmen. La votarea finala, algoritmul, denumit de autorii sai Rijndael, a invins la vot patru alte propuneri, printre care si algoritmul RC6, propus de o echipa de criptografi in care se afla si reputatul informatician Ron Rivest.

Modelul matematic și funcționalul

•AES operează cu matrici de 4x4 octeți.

•Pentru criptare, fiecare rundă AES execută următorii pași:

1.SubBytes() – substituție în care fiecare octet este înlocuit cu un altul cu ajutorul unei “look-up table”

2.ShiftRows() – transpoziție ciclică cu un anumit număr de pași

3.MixColumns – o operație de amestec ce folosește o funcție polinomială

4.AddKeyRound() – fiecare octet este combinat cu o cheie specifică rundei, ce a fost în prealabil calculată din cheia inițială

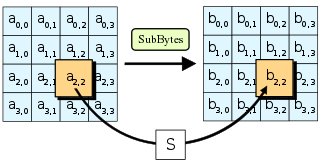
**Pasul SubBytes :**

Pasul SubBytes este un cifru cu substituție, fără punct fix, denumit Rijndael S-box, care rulează independent pe fiecare octet din state. Această transformare este neliniară și face astfel întreg cifrul să fie neliniar, ceea ce îi conferă un nivel sporit de securitate.

Fiecare octet este calculat astfel:

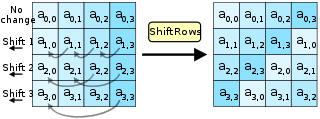


unde bi este bitul corespunzător poziției i din cadrul octetului, iar ci este bitul corespunzător poziției i din octetul ce reprezintă valoarea hexazecimală 63, sau, pe biți, 01100011. Maparea octeților se poate reține într-un tabel, explicitat în FIPS PUB 197, în care este specificat rezultatul operației de mai sus efectuată pe fiecare din cele 256 de valori posibile reprezentabile pe un octet.



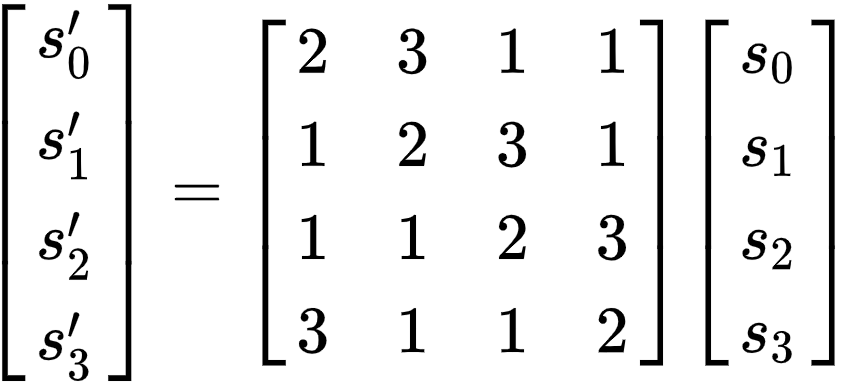
**Pasul ShiftRows :**

Pasul ShiftRows operează la nivel de rând al matricii de stare state. Pasul constă în simpla deplasare ciclică a octeților de pe rânduri, astfel: primul rând nu se deplasează; al doilea rând se deplasează la stânga cu o poziție; al treilea rând se deplasează la stânga cu două poziții; al patrulea se deplasează la stânga cu trei poziții. Rezultatul acestui pas este că fiecare coloană din tabloul state rezultat este compusă din octeți de pe fiecare coloană a stării inițiale. Acesta este un aspect important, din cauză că tabloul state este populat inițial pe coloane, iar pașii ulteriori, inclusiv AddRoundKey în care este folosită cheia de criptare, operațiile se efectuează pe coloane.



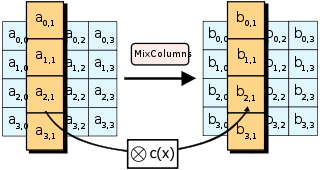
**Pasul MixColumns :**

În acest pas, fiecare coloană a tabloului de stare este considerată un polinom de gradul 4 peste corpul Galois F28 . Fiecare coloană, tratată ca polinom, este înmulțită, modulo x4 +1 cu polinomul a ( x ) = 3x3 + x2 + x + 2 . Operatia se poate scrie ca inmultire de matrice astfel :



Unde s’i sunt elementele de pe un vector coloană rezultate în urma înmulțirii, iar si sunt elementele de pe același vector înaintea aplicării pasului.

Rezultatul are proprietatea că fiecare element al său depinde de toate elementele de pe coloana stării dinaintea efectuării pasului. Combinat cu pasul ShiftRows, acest pas asigură că după câteva iterații, fiecare octet din stare depinde de fiecare octet din starea inițială (tabloul populat cu octeții mesajului în clar). Acești doi pași, împreună, sunt principala sursă de difuzie în algoritmul Rijndael. Coeficienții polinomului a(x) sunt toți 1, 2 și 3, din motive de performanță, criptarea fiind mai eficientă atunci când coeficienții sunt mici. La decriptare, coeficienții pasului corespunzător acestuia sunt mai mari și deci decriptarea este mai lentă decât criptarea. S-a luat această decizie pentru că unele din aplicațiile în care urma să fie folosit algoritmul implică numai criptări, și nu și decriptări, deci criptarea este folosită mai des.



**Pasul AddRoundKey și planificarea cheilor :**

Pasul AddRoundKey este pasul în care este implicată cheia. El constă într-o simplă operație de „sau” exclusiv pe biți între stare și cheia de rundă (o cheie care este unică pentru fiecare iterație, cheie calculată pe baza cheii secrete). Operația de combinare cu cheia secretă este una extrem de simplă și rapidă, dar algoritmul rămâne complex, din cauza complexității calculului cheilor de rundă (Key Schedule), precum și a celorlalți pași ai algoritmului.

Cheia de rundă este calculată după algoritmul următor:

KeyExpansion(byte key[4\*Nk], word w[Nb\*(Nr+1)], Nk)

begin

word temp

i = 0

while (i < Nk)

w[i] = word(key[4\*i], key[4\*i+1], key[4\*i+2], key[4\*i+3])

i = i+1

end while

i = Nk

while (i < Nb \* (Nr+1)]

temp = w[i-1]

if (i mod Nk = 0)

temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]

else if (Nk > 6 and i mod Nk = 4)

temp = SubWord(temp)

end if

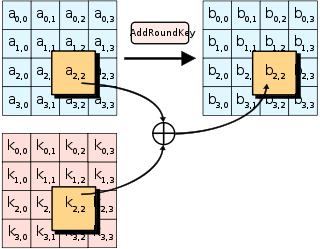
w[i] = w[i-Nk] xor temp

i = i + 1

end while

end

Acest algoritm lucrează pe cheia algoritmului, de lungime Nk cuvinte de 4 octeți (4, 6 sau 8, conform standardului), populând un tabel de Nb x ( Nr +1 ) cuvinte , Nb fiind numărul de cuvinte al blocului (în versiunea standardizată, 4), iar Nr numărul de runde (iterații), dependent de lungimea cheii. Algoritmul de planificare a cheilor folosește transformarea SubWord, care este o substituție a octeților identică cu cea din pasul SubBytes. RotWord este o rotație ciclică la stânga cu un octet a octeților dintr-un cuvânt. Cu Rcon[i] se notează în algoritm un cuvânt format din octeții [2i-1 , {00} , {00} , {00} ] . Operația de ridicare la putere este aici cea valabilă în corpul Galois F28 . Tabloul w conține la finalul prelucrării cuvintele de pe coloanele cheilor de rundă, în ordinea în care urmează să fie aplicate.



Aplicarea

Rijndael, ca și toți ceilalți algoritmi ajunși în etapa finală de selecție pentru standardul AES, a fost revizuit de NSA și, ca și ceilalți finaliști, este considerat suficient de sigur pentru a fi folosit la criptarea informațiilor guvernamentale americane neclasificate. În iunie 2003, guvernul SUA a decis ca AES să poată fi folosit pentru informații clasificate. Până la nivelul SECRET, se pot folosi toate cele trei lungimi de cheie standardizate, 128, 192 și 256 biți. Informațiile TOP SECRET (cel mai înalt nivel de clasificare) pot fi criptate doar cu chei pe 256 biți.

Atacul cel mai realizabil împotriva AES este îndreptat împotriva variantelor Rijndael cu număr redus de iterații. AES are 10 iterații la o cheie de 128 de biți, 12 la cheie de 192 de biți și 14 la cheie de 256 de biți. La nivelul anului 2008, cele mai cunoscute atacuri erau accesibile la 7, 8, respectiv 9 iterații pentru cele trei lungimi ale cheii.

*Algoritmul RSA*

În criptografie, RSA este un algoritm criptografic cu chei publice, primul algoritm utilizat atât pentru criptare, cât și pentru semnătura electronică. Algoritmul a fost dezvoltat în 1977 și publicat în 1978 de Ron Rivest, Adi Shamir și Leonard Adleman la MIT și își trage numele de la inițialele numelor celor trei autori.

Puterea sa criptografică se bazează pe dificultatea problemei factorizării numerelor întregi, problemă la care se reduce criptanaliza RSA și pentru care toți algoritmii de rezolvare cunoscuți au complexitate exponențială. Există însă câteva metode de criptanaliză care ocolesc factorizarea efectivă, exploatând maniere eronate de implementare efectivă a schemei de criptare.

RSA este un algoritm de criptare pe blocuri. Aceasta înseamnă că atât textul clar cât și cel cifrat sunt numere între 0 și n-1, cu un n ales. Un mesaj de dimensiune mai mare decât logn este împărțit în segmente de lungime corespunzătoare, numite blocuri, care sunt cifrate rând pe rând. De asemenea, ca algoritm criptografic cu chei publice, funcționează pe baza unei perechi de chei legate matematic între ele: o cheie publică, cunoscută de toată lumea, și una secretă, cunoscută doar de deținătorul acesteia.

**Generarea cheilor :**

Perechea de chei se generează după următorii pași :

1. Se generează două numere prime, de preferat mari, p și q ;
2. Se calculeaza n = pq is ф = ( p – 1 )( q – 1 ) ;
3. Se alege un intreg aleator e , 1 < e < ф astfel incat cmmdc( e , φ) = 1 . Perechea ( n , e ) este cheie publica .
4. Folosind algoritmul lui Euclid extins, se calculează întregul d, unicul cu proprietatea că de ≡ 1 mod ф . ( n , d ) constituie cheia secreta .

Decizia cu privire la care dintre e și d este cheia publică și care este cea secretă este, din punct de vedere matematic, arbitrară, oricare dintre cele două numere poate juca oricare dintre roluri[4]. În practică însă, pentru a mări viteza de criptare, și întrucât dintre cele două numere e este cel ales arbitrar, e este cheia publică iar valoarea sa este aleasă un număr mic, de regulă 3, 17 sau 65537 (216+1). Aceasta conduce la un număr minim de înmulțiri, deci la o performanță sporită, deoarece toate aceste numere au doar două cifre 1 în reprezentarea lor binară.

**Criptarea si decriptarea :**

Presupunând că mesajul clar este sub forma unui număr m, mai mic decât n, atunci mesajul cifrat, notat cu c este c = me (mod n)

unde e este cheia publică a destinatarului mesajului. Pentru a decripta mesajul, destinatarul își folosește cheia sa secretă d, care are proprietatea foarte importantă că: de ≡ 1 mod ф

Astfel , mesajul clar reste recuperat calculand m = cd (mod n)

Oricine poate cripta mesaje cu cheia publică a destinatarului, dar numai acesta din urmă poate decripta, deoarece trebuie să folosească cheia sa secretă.

Algoritmul poate fi folosit și pentru semnătura electronică, folosind cheile invers. Dacă o entitate criptează un mesaj (sau mai degrabă un hash al acestuia) cu cheia sa secretă și atașează rezultatul mesajului său, atunci oricine poate verifica, decriptând cu cheia publică a semnatarului și comparând rezultatul cu mesajul clar (sau cu hash-ul acestuia), că într-adevăr acea entitate este autorul mesajului.

Demonstrarea formulei de decriptare :

Formula de decriptare este valabila , deoarece :

Cd mod n = med mod n

ed ≡ 1 ( mod ф ) si fiindca ф = (p - 1)(q -1) , atunci

ed ≡ 1 ( mod p - 1) si ed ≡ 1 ( mod q - 1)

si deci se poate scrie :

ed = k( p - 1) + 1

ed = h( q - 1) + 1

Dar, cum p este prim, și deci prim cu m, conform micii teoreme a lui Fermat, rezultă că

mp-1 ≡ 1 ( mod p )

Astfel med = mk(p-1)+1 = (mp-1)km ≡ 1km = m ( mod p ) .

Dacă p nu este totuși prim cu m, atunci înseamnă că m este multiplu al lui p, caz trivial în care m este congruent cu 0 modulo p, și deci ridicat la orice putere este congruent cu 0 și deci cu el însuși.

Analog si pentru q , med ≡ m ( mod q) , conform teoremei chinezesti a resturilor,deoarece p si q sunt numere prime rezulta med ≡ m ( mod pq) .

Implementarea :

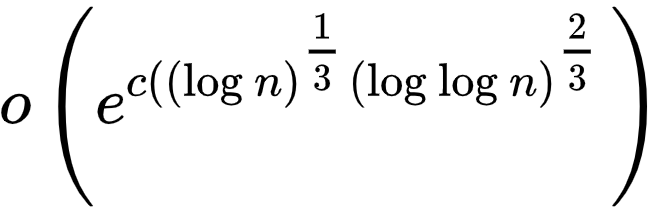
În general, deoarece se bazează pe o operație destul de costisitoare din punct de vedere al timpului de calcul și al resurselor folosite, și anume exponențierea modulo n, viteza RSA este mult mai mică decât a algoritmilor de criptare cu cheie secretă. Bruce Schneier estima, pe baza unor calcule efectuate în anii 1990, că o implementare hardware de RSA este de 1000 de ori mai lentă decât o implementare DES, iar în software, RSA este de 100 de ori mai lent.

Există anumite modificări care pot aduce performanțe sporite, precum alegerea unui exponent de criptare mic, care astfel reduce calculele necesare criptării, rezolvând în același timp și unele probleme de securitate.[8] De asemenea, operațiile cu cheia secretă pot fi accelerate pe baza teoremei chinezești a resturilor, dacă se stochează p, q și unele rezultate intermediare, folosite des. Cu toate acestea, îmbunătățirile nu sunt mari, iar ordinul de mărime al diferențelor de performanță față de implementările algoritmilor cu cheie secretă rămâne același. De aceea, în sistemele de comunicație în timp real, în care viteza de criptare și decriptare este esențială (cum ar fi, de exemplu, aplicațiile de streaming video sau audio securizate), RSA se folosește doar la începutul comunicației, pentru a transmite cheia secretă de comunicație, care ulterior este folosită într-un algoritm cu cheie secretă, cum ar fi 3DES sau AES.

Securitatea :

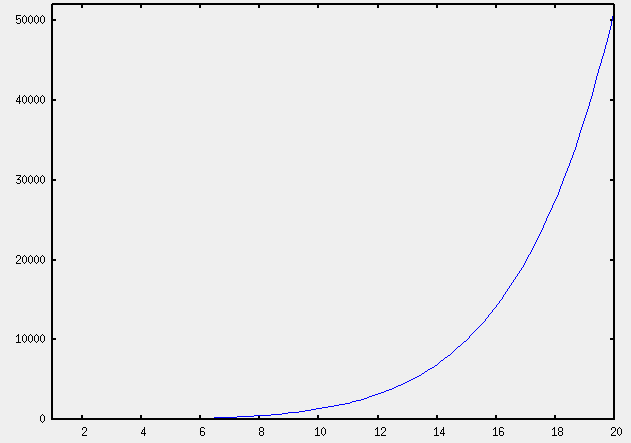
Problema decriptării unui mesaj criptat cu RSA este denumită problema RSA. Aceasta constă în obținerea radicalului de ordin e modulo n, unde e și n au proprietatea că n este produsul a două numere prime mari p și q, iar e este prim cu produsul dintre p-1 și q-1. În acest moment, cea mai eficientă metodă de a realiza aceasta este descompunerea în factori primi a lui n, și obținerea astfel a cheii secrete d pe baza lui e. Astfel, este demonstrat că dificultatea spargerii unui mesaj criptat cu RSA nu este mai dificilă decât problema factorizării. Nu a fost descoperită încă o altă soluție generală a problemei RSA, dar nici nu s-a demonstrat matematic că nu există o altă soluție.

Factorizarea întregilor prin metodele comune ajută la găsirea soluțiilor în timp util doar pentru numere mici. Pentru numere mari, algoritmii de factorizare, cu complexitate exponențială, dau soluția după foarte mult timp. Cea mai rapidă metodă de factorizare a întregilor, algoritmul general al ciurului câmpurilor de numere, are o complexitate de



Aici, c este un număr ce ia valori în jur de 1,9 pentru numere de tipul lui n, adică numere cu doi factori primi. Cel mai mare număr factorizat vreodată prin acest algoritm, rulat în anul 2005, de către specialiști de la Agenția Federală Germană pentru Securitatea Tehnologiei Informației, are 200 de cifre zecimale, iar reprezentarea binară a factorilor primi obținuți ocupă 663 de biți.[14][15] Cheile de criptare RSA cele mai sigure au lungimi de peste 1024 de biți.

Atacul RSA prin metoda forței brute, adică încercarea fiecărei chei secrete posibile, consumă chiar mai mult timp decât factorizarea.



Graficul complexităţii celei mai bune metode de factorizare a întregilor în funcţie de lungimea reprezentării binare a numărului factorizat (pe abscisă, log n, adică numărul de cifre al numărului de factorizat; pe ordonată, ordinul de mărime al duratei de factorizare). Se observă că această complexitate este exponenţială, crescând foarte mult pentru numere mari

**Atacuri impotriva RSA :**

Deși securitatea algoritmului RSA constă în legătura dintre acesta și factorizarea întregilor, el trebuie folosit cu grijă în implementări, deoarece, în caz de folosire eronată, sistemele bazate pe RSA pot fi atacate în anumite maniere care ocolesc factorizarea efectivă a modulului, atacatorul ajungând să obțină mesajul clar sau cheia secretă.

**Atac cu text cifrat :**

În cazul atacului cu text cifrat ales, atacatorul dispune de cheia publică a entității atacate (exponentul de criptare e și modulul n), și interceptează mesaje cifrate trimise acestuia. Pentru a obține mesajul clar m dintr-un mesaj cifrat c, atacatorul poate proceda, de exemplu, astfel:

1. Calculeaza x = ( c x 2e) ( mod n)
2. Trimite entitatii atacate spre semnare pe x , obtinand y = xd ( mod n )
3. Se observa ca x = c x 2e ( mod n ) = me x 2e ( mod n ) = (2m)e (mod n)
4. Se rezolva ecuatia y = (2m) (mod n)

Atacatorul obține astfel mesajul cifrat. Există mai multe feluri de atacuri cifrate, dar sunt câteva moduri de apărare împotriva lor. Unele pot fi evitate dacă pur și simplu entitatea protejată cu chei secrete refuză să semneze texte arbitrare trimise de terți. Dacă acest lucru nu este posibil (ca de exemplu în cazul unui notar public care trebuie să semneze documente electronice prezentate de persoane străine), atunci atacul poate fi prevenit prin folosirea unei perechi diferite de chei pentru criptare și pentru semnătura electronică. De asemenea, este necesar să se folosească și un padding aleator pentru mesaj înainte de criptare sau, în cazul semnăturii, să nu se semneze mesajul clar, ci un hash al acestuia. De asemenea, atacul poate fi evitat și dacă se impune o anumită structură predefinită mesajelor primite spre semnare.

**Mesaje necriptate :**

Întrucât RSA se bazează pe ridicarea la putere (modulo un număr n), există anumite părți de mesaje care nu sunt criptate, părți rezultate în urma împărțirii mesajului pe blocuri. Astfel de mesaje sunt mesajele m cu proprietatea că m=mx (mod n) oricare ar fi x, ca de exemplu m=0, m=1, m=n-1. Numărul exact al acestor mesaje decriptate este ( 1 + cmmdc(e – 1 , p – 1 )) \* ( 1 + cmmdc(e – 1 , q – 1)) și deci este de minim 9 (deoarece e, p și q sunt impare). Pentru a micșora numărul de astfel de părți de mesaj, este util să se folosească un exponent public e cât mai mic.

**Exponentul de criptare mic :**

În unele aplicații, se folosește un exponent de criptare (public) mic, de exemplu 3, pentru a mări performanța, dar și pentru a rezolva unele probleme de securitate. Dacă mai multe entități care comunică folosesc același exponent public (dar fiecare are propriul modul și deci propria cheie secretă), atunci același mesaj trimis mai multor destinatari are următoarele valori :

с1 = me (mod n1)

с2 = me (mod n2)

с3 = me (mod n3)

unde ni sunt modulele celor trei destinatari, e este exponentul comun acestora iar m este mesajul trimis tuturor celor trei. Un atacator poate folosi algoritmul lui Gauss pentru a descoperi o soluție mai mică decât n1n2n3 a unui sistem compus din următoarele ecuații:

x = me (mod n1)

x = me (mod n2)

x = me (mod n3)

Această soluție este, conform teoremei chinezești a resturilor, cubul mesajului m. Soluția pentru această problemă este cea denumită sărarea mesajului (din engleză salting), adică adăugarea unui padding format din numere pseudoaleatoare, padding diferit pentru fiecare expediere a mesajului.

Exponentul de decriptare mic :

Dacă exponentul de decriptare (cel secret) este mic, pe lângă faptul că multe părți din mesaj nu se criptează, așa cum s-a arătat mai sus, există un algoritm rapid de găsire a lui d, cunoscând informațiile e și n. Acest algoritm nu este eficient dacă d este de același ordin de mărime cu n, deci dacă e este mic, acesta fiind unul din motivele pentru care se alege în general e un număr mic, pentru ca d să fie cât mai mare.

**Realizarea aplicatiei**

În cadrul acestui punct trebuia sa elaboram o aplicatie utilizand unul dintre limbajele de programare propuse , in cazul nostru am utilizat limbajul python ,deoarece mi-a parut cel mai convenabil pentru realizarea task-urilor propuse . Aplicatia trebuie sa contina un meniu cu algoritmii DES/AES ,Rsa si semntatura digitala .

Codul:

import tkinter as tk

from tkinter import ttk

from Crypto.Cipher import DES, AES

from Crypto.PublicKey import RSA

from Crypto.Signature.pkcs1\_15 import PKCS115\_SigScheme

from Crypto.Hash import SHA256

from Crypto.Util.Padding import pad

import rsa

import binascii

keyPair = RSA.generate(bits=1024)

pubKey = keyPair.publickey()

def generateKeys():

    (publicKey, privateKey) = rsa.newkeys(2048)

    with open('publicKey.pem', 'wb') as p:

        p.write(publicKey.save\_pkcs1('PEM'))

    with open('privateKey.pem', 'wb') as p:

        p.write(privateKey.save\_pkcs1('PEM'))

def loadKeys():

    with open('publicKey.pem', 'rb') as p:

        publicKey = rsa.PublicKey.load\_pkcs1(p.read())

    with open('privateKey.pem', 'rb') as p:

        privateKey = rsa.PrivateKey.load\_pkcs1(p.read())

    return privateKey, publicKey

def encryptRSA(message, key):

    return rsa.encrypt(message.encode('ascii'), key)

def decryptRSA(ciphertext, key):

    try:

        return rsa.decrypt(ciphertext, key).decode('ascii')

    except:

        return False

class Application(tk.Frame):

    def \_\_init\_\_(self, master=None):

        super().\_\_init\_\_(master)

*self*.master = master

*self*.pack()

*self*.create\_widgets()

    def create\_widgets(self):

*self*.algo\_label = tk.Label(*self*, text="Select Algorithm")

*self*.algo\_label.pack()

*self*.algo\_var = tk.StringVar(value="DES")

*self*.algo\_combobox = ttk.Combobox(*self*, textvariable=*self*.algo\_var, state="readonly", values=["DES", "AES", "RSA", "Digital Signature"])

*self*.algo\_combobox.pack()

*self*.input\_label = tk.Label(*self*, text="Enter text to be encrypted/decrypted:")

*self*.input\_label.pack()

*self*.input\_text = tk.Text(*self*, height=4)

*self*.input\_text.pack()

*self*.output\_label = tk.Label(*self*, text="Output:")

*self*.output\_label.pack()

*self*.output\_text = tk.Text(*self*, height=4)

*self*.output\_text.pack()

*self*.encrypt\_button = tk.Button(*self*, text="Encrypt", command=*self*.encrypt)

*self*.encrypt\_button.pack(side="left",padx=10)

*self*.decrypt\_button = tk.Button(*self*, text="Decrypt", command=*self*.decrypt)

*self*.decrypt\_button.pack(side="left",padx=10)

*self*.sign\_button = tk.Button(*self*, text="Sign", command=*self*.sign)

*self*.sign\_button.pack(side="left",padx=10)

*self*.verify\_button = tk.Button(*self*, text="Verify", command=*self*.verify)

*self*.verify\_button.pack(side="left",pady=10)

*self*.clear\_input\_button = tk.Button(*self*, text="Clear Input", command=*self*.clear\_input)

*self*.clear\_input\_button.pack(side="left",padx=10)

*self*.clear\_output\_button = tk.Button(*self*, text="Clear Output", command=*self*.clear\_output)

*self*.clear\_output\_button.pack(side="left",padx=10)

*self*.copy\_button = tk.Button(*self*, text="Copy Input", command=*self*.copy\_text\_input)

*self*.copy\_button.pack(side="left", padx=10)

*self*.copy\_button = tk.Button(*self*, text="Copy Output", command=*self*.copy\_text\_output)

*self*.copy\_button.pack(side="left", padx=10)

*self*.paste\_input\_button = tk.Button(*self*, text="Paste Input", command=*self*.paste\_input)

*self*.paste\_input\_button.pack(side="left",padx=10)

*self*.paste\_output\_button = tk.Button(*self*, text="Paste Output", command=*self*.paste\_output)

*self*.paste\_output\_button.pack(side="left",padx=10)

*self*.quit\_button = tk.Button(*self*, text="Quit", command=*self*.master.destroy)

*self*.quit\_button.pack(side="right",padx=20)

    def copy\_text\_input(self):

*self*.master.clipboard\_clear()

*self*.master.clipboard\_append(*self*.input\_text.get("1.0", "end-1c"))

    def copy\_text\_output(self):

*self*.master.clipboard\_clear()

*self*.master.clipboard\_append(*self*.output\_text.get("1.0", "end-1c"))

    def paste\_input(self):

*self*.input\_text.insert("end", *self*.master.clipboard\_get())

    def paste\_output(self):

*self*.output\_text.insert("end", *self*.master.clipboard\_get())

    def clear\_input(self):

*self*.input\_text.delete("1.0", "end")

    def clear\_output(self):

*self*.output\_text.delete("1.0", "end")

    def encrypt(self):

        algo = *self*.algo\_var.get()

        plaintext = *self*.input\_text.get("1.0", "end-1c")

        if algo == "DES":

            key = b"mydeskey"

            cipher = DES.new(key, DES.MODE\_ECB)

            ciphertext = cipher.encrypt(pad(plaintext.encode(), DES.block\_size))

        elif algo == "AES":

            key = b"1myaeskey1234567"

            cipher = AES.new(key, AES.MODE\_ECB)

            ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block\_size))

        elif algo == "RSA":

            generateKeys()

            publicKey, privateKey = loadKeys()

            ciphertext = encryptRSA(plaintext, publicKey)

        else:

*self*.output\_text.insert("end", "Encryption not supported for selected algorithm.")

            return

*self*.output\_text.delete("1.0", "end")

*self*.output\_text.insert("end", ciphertext.hex())

    def decrypt(self):

        algo = *self*.algo\_var.get()

        ciphertext = bytes.fromhex(*self*.input\_text.get("1.0", "end-1c"))

        if algo == "DES":

            key = b"mydeskey"

            cipher = DES.new(key, DES.MODE\_ECB)

            plaintext = cipher.decrypt(ciphertext).decode()

        elif algo == "AES":

            key = b"1myaeskey1234567"

            cipher = AES.new(key, AES.MODE\_ECB)

            plaintext = cipher.decrypt(ciphertext).decode()

        elif algo == "RSA":

            publicKey, privateKey = loadKeys()

            plaintext = decryptRSA(ciphertext, publicKey)

        else:

*self*.output\_text.insert("end", "Decryption not supported for selected algorithm")

*self*.output\_text.delete("1.0", "end")

*self*.output\_text.insert("end", plaintext)

    def sign(self):

        plaintext = *self*.input\_text.get("1.0", "end-1c")

        msg = bytes(plaintext, encoding = 'utf-8')

        hash = SHA256.new(msg)

        signer = PKCS115\_SigScheme(keyPair)

        signature = signer.sign(hash)

        print(signature,'\n')

        with open('signature.txt', 'wb') as p:

            p.write(signature)

*self*.output\_text.delete("1.0", "end")

*self*.output\_text.insert("end", binascii.hexlify(signature))

    def verify(self):

        plaintext = *self*.input\_text.get("1.0", "end-1c")

        msg = bytes(plaintext, encoding = 'utf-8')

        hash = SHA256.new(msg)

        with open('signature.txt', 'rb') as p:

            signature = p.read()

            print(signature)

        verifier = PKCS115\_SigScheme(pubKey)

        try:

            verifier.verify(hash, signature)

*self*.output\_text.delete("1.0", "end")

*self*.output\_text.insert("end", "Signature is valid.")

            print("Signature is valid.")

        except:

*self*.output\_text.delete("1.0", "end")

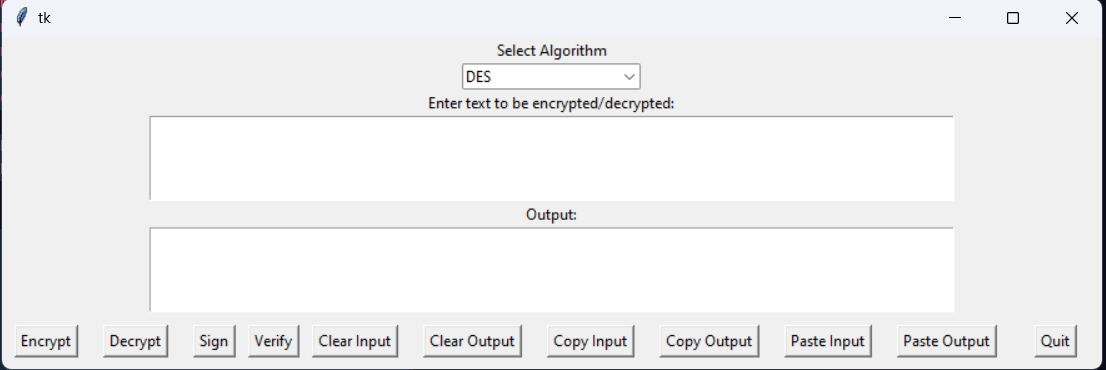
*self*.output\_text.insert("end", "Signature is not valid.")

            print("Signature is invalid.")

root = tk.Tk()

app = Application(master=root)

app.mainloop()

Interfata aplicatiei 

Concluzie

În concluzie, laboratorul 4 la Tehnologii de Securitate Informaționale a presupus studiul algoritmilor DES/AES și RSA, precum și realizarea unei aplicații care să ofere opțiuni pentru implementarea acestor algoritmi și semnătura digitală folosind RSA sau DSA. Algoritmii DES și AES sunt utilizați în principal pentru criptarea datelor, în timp ce algoritmul RSA este utilizat pentru criptarea cheilor de criptare și semnături digitale. Implementarea acestor algoritmi în aplicații de securitate informațională este esențială pentru protejarea informațiilor și a datelor personale.