**Ministerul Educației și Cercetării**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Raport**

Lucrarea de laborator nr.4

Disciplina: Securitatea Informațională

Tema: Algoritmi Combinaţi într-o aplicaţie complexă

**Efectuat**: st.gr. TI-202 Bunescu Gabriel

**Verificat**: asist. univ. Todos Alexandru

Chișinău 2023

**Scopul lucrării:**

Dezvoltarea unei aplicaţii complexe pe baza tuturor algoritmilor studiaţi anterior.

**Mersul lucrării:**

* Algoritmul AES
* Algoritmul RSA
* Algoritmul DSA

Introducere

Această aplicație Python oferă funcționalități pentru criptarea și decriptarea datelor folosind metode simetrice și asimetrice, precum și pentru generarea de semnături digitale. Scopul său principal este de a exemplifica utilizarea criptării și securității într-un mediu simplu și demonstrativ.

Codul sursă este prezentat în **ANEXA A**.

# Criptare simetrică (AES)

Criptarea simetrică folosește algoritmul AES (Standardul de Criptare Avansată) pentru a cripta și decripta datele. Funcțiile acestei secțiuni permit generarea unei chei simetrice și apoi criptarea/decriptarea datelor folosind această cheie.

Funcționalități:

* generate\_symmetric\_key(key\_size): Generează chei de 128, 192 sau 256 biți;
* symmetric\_encrypt(key, plaintext): Criptează textul clar folosind cheia generată;
* symmetric\_decrypt(key, encrypted\_message): Decriptează mesajul criptat folosind cheia.

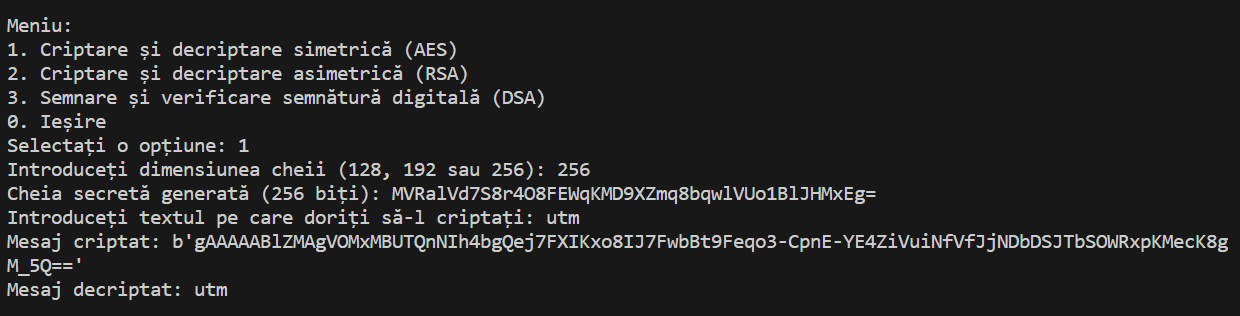


Figura 1.1 – Criptarea Algoritmul AES

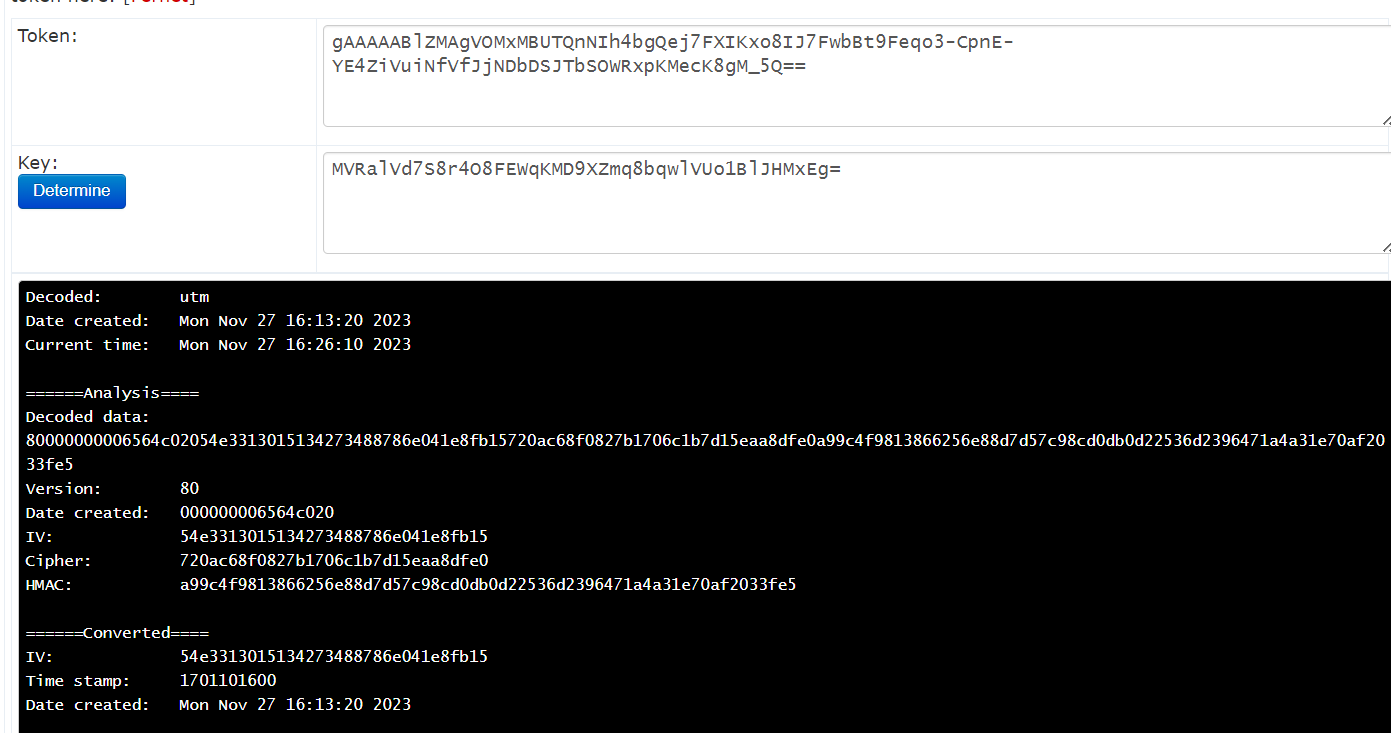


Figura 1.2 – Decriptarea Algoritmul AES

# Criptare asimetrică (RSA)

Criptarea asimetrică se bazează pe algoritmul RSA, care utilizează o pereche de chei (publică și privată) pentru criptare și decriptare. Aici se prezintă funcțiile pentru generarea cheilor RSA și procesul de criptare și decriptare a datelor folosind aceste chei.

Funcționalități:

* generate\_rsa\_keys(): Generează o pereche de chei RSA (publică și privată);
* asymmetric\_encrypt(public\_key, plaintext): Criptează textul clar folosind cheia publică;
* asymmetric\_decrypt(private\_key, ciphertext): Decriptează mesajul criptat folosind cheia privată.

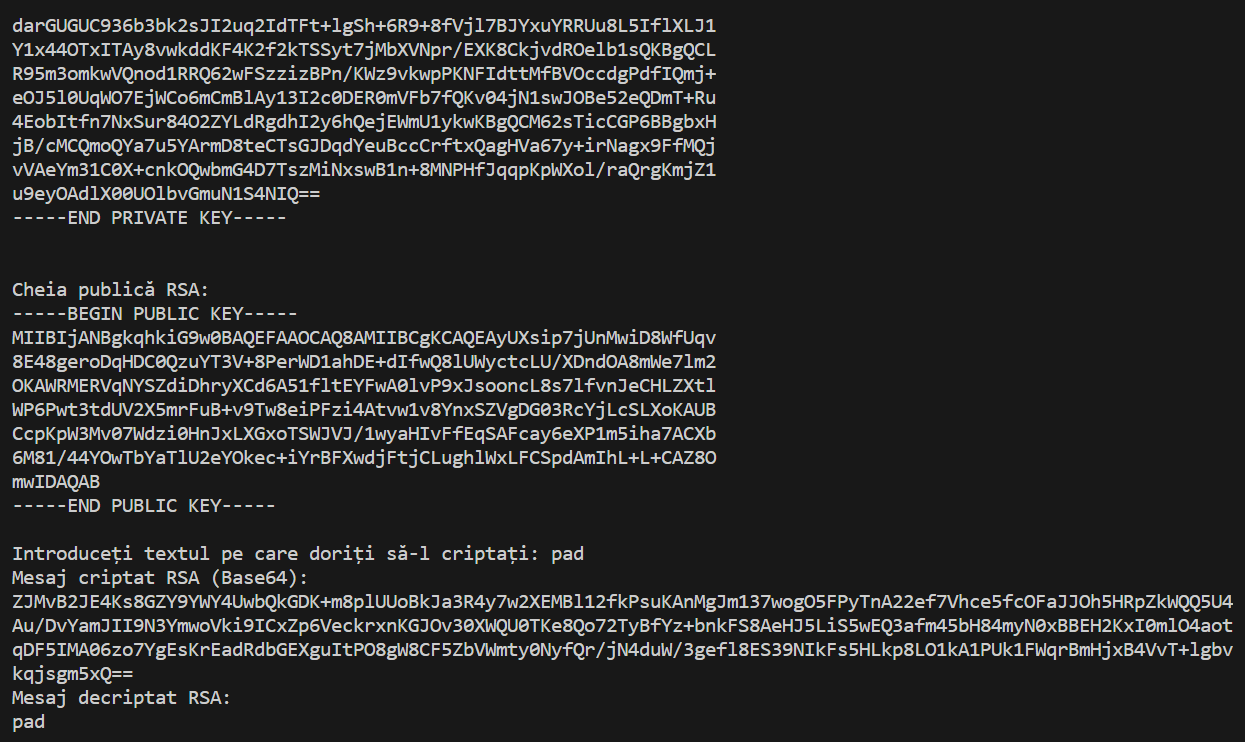


Figura 2.1 – Criptarea Algoritmul RSA

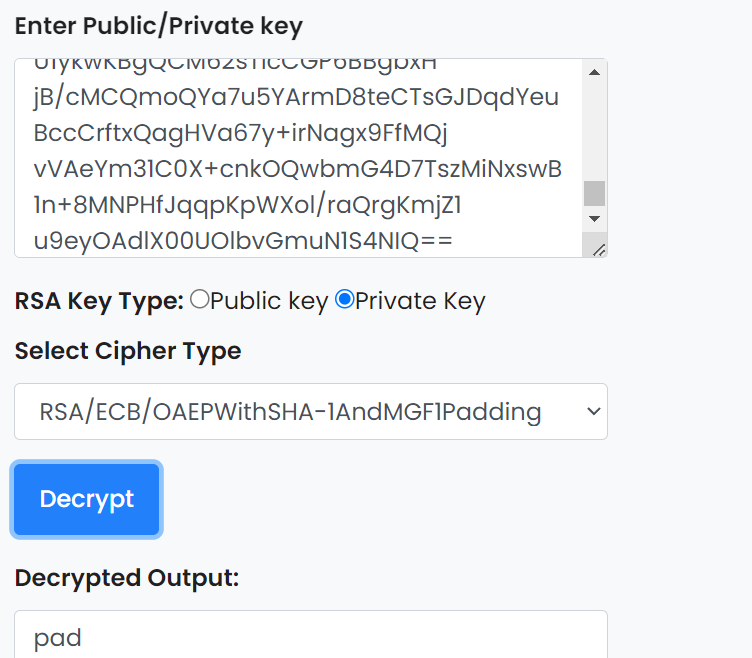


Figura 2.2 – Decriptarea Algoritmul AES

# Semnătură Digitală (DSA)

Semnătura digitală, bazată pe algoritmul DSA, este utilizată pentru a semna și verifica autenticitatea unui mesaj. Această secțiune prezintă funcțiile pentru generarea cheilor DSA, semnarea mesajului și verificarea semnăturii.

Funcționalități:

* generate\_dsa\_keys(): Generează o pereche de chei DSA (publică și privată);
* sign\_message(private\_key, message): Semnează un mesaj folosind cheia privată;
* verify\_signature(public\_key, message, signature): Verifică semnătura unui mesaj folosind cheia publică.

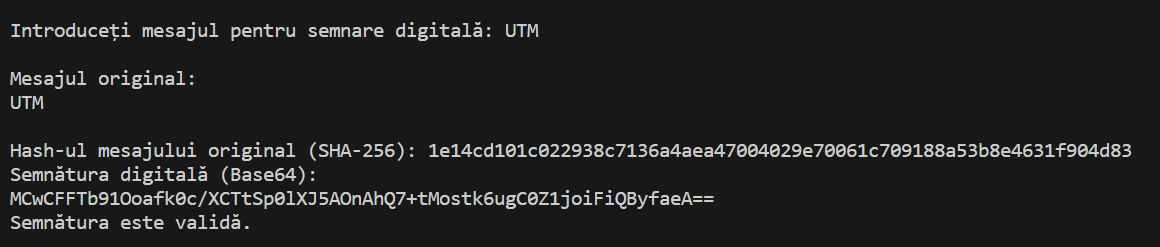


Figura 3.1 – Semnătură Digitală (DSA)

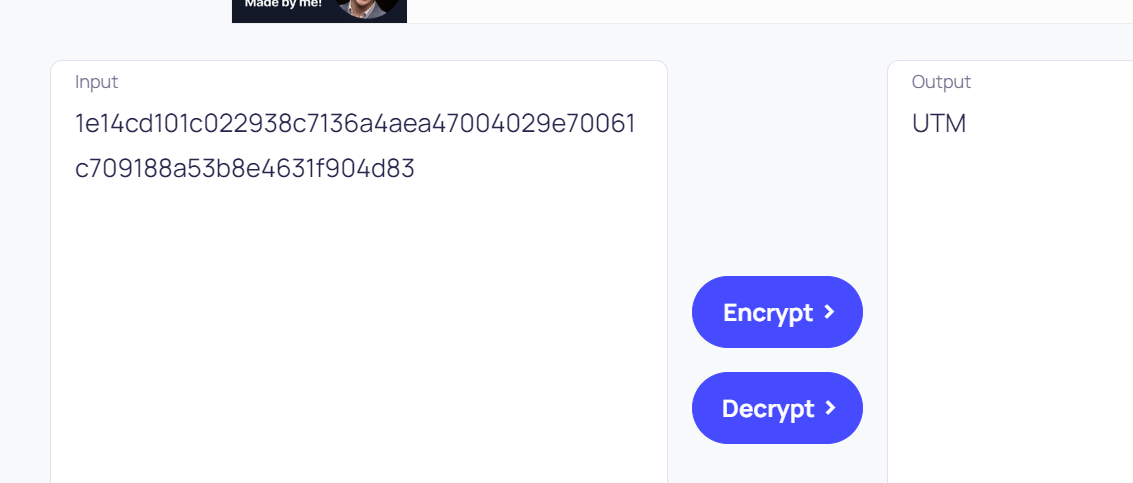


Figura 3.2 – verificarea cheii publice și a semnăturii (DSA)

# Meniu și interacțiunea cu utilizatorul

Modul în care utilizatorul interacționează cu codul prin intermediul sistemului de meniu.

Opțiuni de Meniu:

* Opțiunea 1: Criptare Simetrică(AES);
* Opțiunea 2: Criptare Asimetrică (RSA);
* Opțiunea 3: Semnătură Digitală (DSA);
* Opțiunea 0: Ieșire.

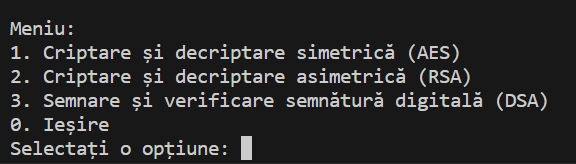


Figura 4.1 – Meniul aplicație

**Concluzii**

Aplicația dezvoltată evidențiază utilizarea diverselor tehnici criptografice, precum criptarea simetrică și asimetrică, precum și generarea și verificarea semnăturilor digitale. Implementând algoritmi precum AES, RSA și DSA prin intermediul bibliotecii cryptography, această aplicație oferă o platformă interactivă pentru criptarea, decriptarea și securizarea datelor.

Prin intermediul funcționalităților prezentate, utilizatorul are ocazia să experimenteze diferite aspecte ale criptografiei, inclusiv generarea și utilizarea cheilor de diferite dimensiuni, criptarea și decriptarea datelor folosind cheile simetrice și asimetrice, precum și semnarea și verificarea mesajelor pentru autenticitate.

**Bibliografii**

Resurse Electronice [27.11.2023] Disponibil:

<https://8gwifi.org/fernet.jsp>

https://asecuritysite.com/tokens/ferdecode

Resurse Electronice [27.11.2023] Disponibil:

<https://www.devglan.com/online-tools/rsa-encryption-decryption>

Resurse Electronice [27.11.2023] Disponibil:

https://10015.io/tools/sha256-encrypt-decrypt

**Anexa A:**

from cryptography.fernet import Fernet

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import serialization, hashes

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives.asymmetric import dsa

from cryptography.exceptions import InvalidSignature

import base64

import hashlib

def generate\_symmetric\_key(key\_size):

    if key\_size == 128:

        return Fernet.generate\_key(), 128

    elif key\_size == 192:

        return Fernet.generate\_key(), 192

    elif key\_size == 256:

        return Fernet.generate\_key(), 256

    else:

        raise ValueError("Dimensiunea cheii nu este validă. Alegeți una dintre: 128, 192 sau 256.")

def symmetric\_encrypt(key, plaintext):

    f = Fernet(key)

    encrypted\_message = f.encrypt(plaintext.encode())

    return encrypted\_message

def symmetric\_decrypt(key, encrypted\_message):

    f = Fernet(key)

    decrypted\_message = f.decrypt(encrypted\_message).decode()

    return decrypted\_message

def generate\_rsa\_keys():

    private\_key = rsa.generate\_private\_key(

        public\_exponent=65537,

        key\_size=2048,

        backend=default\_backend()

    )

    private\_pem = private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.PKCS8,

        encryption\_algorithm=serialization.NoEncryption()

    )

    public\_key = private\_key.public\_key()

    public\_pem = public\_key.public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_pem.decode(), public\_pem.decode()

def asymmetric\_encrypt(public\_key, plaintext):

    public\_key = serialization.load\_pem\_public\_key(public\_key.encode(), backend=default\_backend())

    ciphertext = public\_key.encrypt(

        plaintext.encode(),

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA1()),

            algorithm=hashes.SHA1(),

            label=None

        )

    )

    return base64.b64encode(ciphertext).decode()

def asymmetric\_decrypt(private\_key, ciphertext):

    private\_key = serialization.load\_pem\_private\_key(private\_key.encode(), password=None, backend=default\_backend())

    decoded\_ciphertext = base64.b64decode(ciphertext.encode())

    decrypted\_message = private\_key.decrypt(

        decoded\_ciphertext,

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA1()),

            algorithm=hashes.SHA1(),

            label=None

        )

    )

    return decrypted\_message.decode()

def generate\_dsa\_keys():

    private\_key = dsa.generate\_private\_key(key\_size=1024, backend=default\_backend())

    private\_key\_pem = private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.PKCS8,

        encryption\_algorithm=serialization.NoEncryption()

    )

    public\_key = private\_key.public\_key()

    public\_key\_pem = public\_key.public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_key\_pem.decode(), public\_key\_pem.decode()

def sign\_message(private\_key, message):

    private\_key = serialization.load\_pem\_private\_key(private\_key.encode(), password=None, backend=default\_backend())

    message\_hash = hashlib.sha256(message.encode()).digest()

    signature = private\_key.sign(

        message\_hash,

        algorithm=hashes.SHA256()

    )

    return base64.b64encode(signature).decode()

def verify\_signature(public\_key, message, signature):

    public\_key = serialization.load\_pem\_public\_key(public\_key.encode(), backend=default\_backend())

    message\_hash = hashlib.sha256(message.encode()).digest()

    signature = base64.b64decode(signature.encode())

    try:

        public\_key.verify(

            signature,

            message\_hash,

            algorithm=hashes.SHA256()

        )

        return True

    except InvalidSignature:

        return False

# Meniu

while True:

    print("\nMeniu:")

    print("1. Criptare și decriptare simetrică (AES)")

    print("2. Criptare și decriptare asimetrică (RSA)")

    print("3. Semnare și verificare semnătură digitală (DSA)")

    print("0. Ieșire")

    choice = input("Selectați o opțiune: ")

    if choice == "1":

        try:

            key\_size = int(input("Introduceți dimensiunea cheii (128, 192 sau 256): "))

            if key\_size not in [128, 192, 256]:

                raise ValueError("Dimensiunea cheii nu este validă. Alegeți una dintre: 128, 192 sau 256.")

            key, key\_size\_in\_bits = generate\_symmetric\_key(key\_size)

            print(f"Cheia secretă generată ({key\_size\_in\_bits} biți):", key.decode())

            plaintext = input("Introduceți textul pe care doriți să-l criptați: ")

            encrypted\_message = symmetric\_encrypt(key, plaintext)

            print("Mesaj criptat:", encrypted\_message)

            decrypted\_message = symmetric\_decrypt(key, encrypted\_message)

            print("Mesaj decriptat:", decrypted\_message)

        except ValueError as e:

            print(e)

    elif choice == "2":

        try:

            private\_key, public\_key = generate\_rsa\_keys()

            print("Cheia privată RSA:")

            print(private\_key)

            print("\nCheia publică RSA:")

            print(public\_key)

            plaintext = input("Introduceți textul pe care doriți să-l criptați: ")

            ciphertext = asymmetric\_encrypt(public\_key, plaintext)

            print("Mesaj criptat RSA (Base64):")

            print(ciphertext)

            decrypted\_message = asymmetric\_decrypt(private\_key, ciphertext)

            print("Mesaj decriptat RSA:")

            print(decrypted\_message)

        except Exception as e:

            print("A apărut o eroare:", e)

    elif choice == "3":

        try:

            private\_key, public\_key = generate\_dsa\_keys()

            print("Cheia privată DSA:")

            print(private\_key)

            print("\nCheia publică DSA:")

            print(public\_key)

            message = input("Introduceți mesajul pentru semnare digitală: ")

            message\_hash = hashlib.sha256(message.encode()).hexdigest()

            print("\nMesajul original:")

            print(message)

            print(f"\nHash-ul mesajului original (SHA-256): {message\_hash}")

            signature = sign\_message(private\_key, message)

            print("Semnătura digitală (Base64):")

            print(signature)

            verified = verify\_signature(public\_key, message, signature)

            if verified:

                print("Semnătura este validă.")

            else:

                print("Semnătura nu este validă.")

        except Exception as e:

            print("A apărut o eroare:", e)

    elif choice == "0":

        print("La revedere!")

        break

    else:

        print("Opțiune invalidă. Vă rugăm să introduceți o opțiune validă.")