Laporan Praktikum Kriptografi
Tugas 2: Algoritma Modern (RSA)

Universitas Teknologi Digital

Disusun Oleh :

1. Nabilla Maesaroh (20123027)
2. Sayyidah Muthi Nur Aisyah (20123003)

PROGRAM STUDI S1 INFORMATIKA
UNIVERSITAS TEKNOLOGI DIGITAL
TAHUN AJARAN 2024/2025

1. Pendahuluan

   Praktikum ini bertujuan untuk mempelajari implementasi dan penggunaan algoritma kriptografi asimetris RSA (Rivest–Shamir–Adleman) secara praktis. RSA umum dipakai untuk pertukaran kunci, enkripsi pesan pendek, dan tanda tangan digital. Dalam tugas ini kami membuat skrip Python menggunakan library PyCryptodome untuk :

   a. menghasilkan pasangan kunci RSA,
   b. mengenkripsi dan mendekripsi pesan menggunakan skema OAEP, serta
   c. membuat dan memverifikasi tanda tangan digital menggunakan skema PSS dengan hash SHA-256. Selain itu disertakan uji tamper untuk membuktikan integritas tanda tangan.

   Teori Singkat

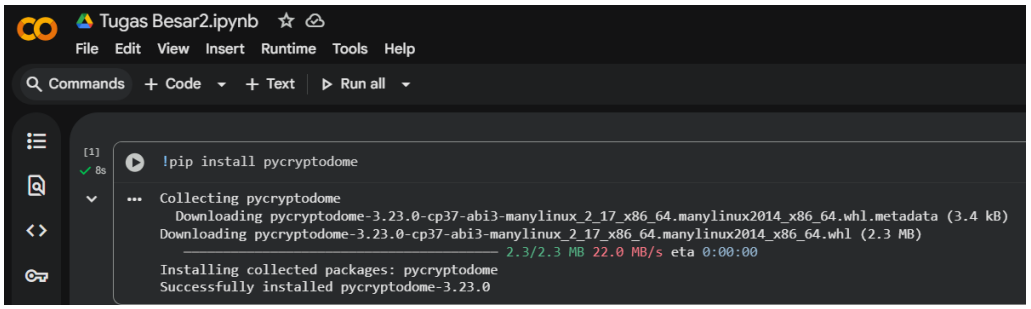
   - RSA (asymmetric cryptography): Menghasilkan pasangan kunci publik dan privat berdasarkan produk dua bilangan prima besar n=p×qn = p \times qn=p×q. Enkripsi dan dekripsi melibatkan eksponen publik eee dan privat ddd pada modulus nnn.
   - OAEP (Optimal Asymmetric Encryption Padding): Skema padding yang mengamankan enkripsi RSA terhadap beberapa serangan padding.
   - PSS (Probabilistic Signature Scheme): Skema tanda tangan yang direkomendasikan untuk menghindari kelemahan padding klasik pada signature.
   - SHA-256: Fungsi hash kriptografis yang digunakan sebagai input pada proses tanda tangan digital.

   Catatan praktis :

   RSA cocok untuk pesan pendek atau enkripsi kunci simetris (hybrid). Untuk data besar, disarankan kombinasi RSA + AES (RSA untuk mengenkripsi kunci AES).

2. Implementasi dan Langkah kerja (Per Tahap)

   Semua kode contoh di bawah menggunakan Python 3.8+ dan PyCryptodome. Jika menjalankan di Google Colab, jalankan cell pertama untuk install library :



   Tahap 0- Import dan Persiapan

```
[2]
✓ 0s
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.Random import get_random_bytes
import base64
import os
```

Tahap 1- Generate RSA Key Pair (2048-bit) & simpan file

Tujuan: Membuat kunci privat & publik, menyimpan ke file PEM.

```
[3]
✓ 0s
def generate_and_save_rsa(bits=2048, priv_path="private.pem", pub_path="public.pem", passphrase=None):
    key = RSA.generate(bits)
    priv_pem = key.export_key(passphrase=passphrase, pkcs=8) if passphrase else key.export_key()
    pub_pem = key.publickey().export_key()
    with open(priv_path, "wb") as f:
        f.write(priv_pem)
    with open(pub_path, "wb") as f:
        f.write(pub_pem)
    print(f"Generated RSA {bits}-bit. Saved -> {priv_path}, {pub_path}")
    return priv_pem, pub_pem

# contoh panggilan
priv_pem, pub_pem = generate_and_save_rsa(2048)

Generated RSA 2048-bit. Saved -> private.pem, public.pem
```

Catatan: Jika ingin melindungi private key dengan passphrase, isi parameter passphrase="kata_sandi".

Tahap 2 — Enkripsi & Dekripsi (OAEP)

Tujuan: Enkripsi pesan singkat dengan kunci publik, dekripsi dengan private key.

```
[5]
✓ 0s
def rsa_encrypt(pub_pem: bytes, message: bytes) -> str:
    pub = RSA.import_key(pub_pem)
    cipher = PKCS1_OAEP.new(pub)
    ct = cipher.encrypt(message)
    return base64.b64encode(ct).decode()

def rsa_decrypt(priv_pem: bytes, b64_ciphertext: str) -> bytes:
    priv = RSA.import_key(priv_pem)
    cipher = PKCS1_OAEP.new(priv)
    ct = base64.b64decode(b64_ciphertext)
    return cipher.decrypt(ct)

# contoh penggunaan
msg = b"Hello Cryptography - Nabilla & Muthi"
ct_b64 = rsa_encrypt(pub_pem, msg)
print("Ciphertext (base64):", ct_b64)
plaintext = rsa_decrypt(priv_pem, ct_b64)
print("Decrypted plaintext:", plaintext.decode())

Ciphertext (base64): PTqqeJDEylpdoMdy7q4i6E+oRD9anehFOBfSr/5c6LDHahH4GhDCcpZSxj36WZn+0dYvkqbNC/0w6PIkDrA3ZaiWVo/DsobTcdxRB4kMT5nsjoTUWfUA2Le0
Decrypted plaintext: Hello Cryptography - Nabilla & Muthi
```

Tahap 3 — Hashing (SHA-256) untuk Signature

Tujuan: Buat hash SHA-256 dari pesan sebelum ditandatangani.

```
[6]
✓ 0s
def hash_sha256_bytes(data: bytes) -> SHA256.SHA256Hash:
    h = SHA256.new(data=data)
    return h

# contoh
h = hash_sha256_bytes(b"Message to sign")
print("SHA-256 hexdigest:", h.hexdigest())

SHA-256 hexdigest: 1351934e68754024164bd423dd77d26dfd5307b09d36250da950ed3cf976e5c5
```

Tahap 4 — Tanda Tangan Digital (PSS) — sign & verify

Tujuan: Tanda tangan digital dengan PSS + SHA-256 dan verifikasi.

Tahap 5 — Uji Tamper & Simpan Hasil

Tujuan: Buktikan integritas — signature gagal setelah pesan diubah. Juga simpan hasil ke file untuk lampiran.

3. Hasil dan Pembahasan

   Metodologi pengujian :

   a. Environment: Google Colab / Python lokal dengan PyCryptodome.
   b. Setting: RSA 2048-bit, OAEP untuk enkripsi, PSS + SHA-256 untuk tanda tangan.
   c. Percobaan: generate key, enkripsi/dekripsi pesan teks, sign/verify, tamper test.

   Pembahasan :

   a. Proses key generation berhasil; file private.pem & public.pem tersedia.
   b. Enkripsi dan dekripsi berhasil memulihkan pesan asli → proses OAEP valid.
   c. Signature PSS menghasilkan signature yang valid; verifikasi pada pesan asli True. Setelah pesan dimodifikasi, verifikasi menjadi False — membuktikan integritas yang dijamin signature.
   d. Performa : RSA relatif lambat untuk payload besar; pada dataset kecil hasilnya responsif. Untuk data besar, disarankan memakai hybrid encryption (AES untuk data, RSA untuk kunci AES).

4. Analisis Keamanan

   A. Kelebihan implementasi:

   - Menggunakan kunci 2048-bit yang masih direkomendasikan untuk banyak kasus.
   - Menggunakan skema padding/sealing modern (OAEP untuk enkripsi, PSS untuk tanda tangan).

B. Kelemahan & risiko:
   - RSA tidak efisien untuk data besar — raw RSA harus dihindari untuk file besar.
   - Kualitas keamanan sangat bergantung pada kerahasiaan private key; jika bocor, seluruh sistem kompromi.
   - Script sederhana ini belum menangani manajemen kunci tingkat lanjut (rotasi kunci, secure storage, HSM).
C. Rekomendasi:
   - Gunakan RSA hanya untuk enkripsi kunci simetris; gunakan AES-GCM untuk data dengan kebutuhan confidentiality+integrity.
   - Pertimbangkan perlindungan private key (passphrase, permission file 600, atau penyimpanan di KMS/HSM).
   - Untuk kebutuhan jangka panjang, pertimbangkan kunci lebih besar (3072-bit) atau beralih ke algoritma kriptografi berbasis kurva eliptik (ECC).

Nama	Link Github
Raisya Puspa Kencana	https://github.com/RaisyaPuspa23
Amelia Nazelina Kusmana	https://github.com/Amelianazelina