

**COMSAT UNIVERSITY ISLAMABAD ATTOCK
CAMPUS**



INFORMATION SECURITY LAB TERMINAL

NAME: MENAHIL NOOR

REGISTRATION NO: SP24-BSE-035

SUBMITTED TO: MA'AM AMBREEN GUL

DEPARTMENT: SOFTWARE ENGINEERING

DATE: 16 DECEMBER 2025

QUESTION: 01

Secure Email System (RSA for Encryption + ElGamal for Digital Signatures)

Part 1: RSA (Encryption / Decryption)

RSA Key Generation

RSA generates keys using two prime numbers.

In the code:

- Two primes are chosen: $p = 23$, $q = 19$
- Compute modulus:
 - $n = p \times q$
- Compute Euler's totient:
 - $\phi = (p-1)(q-1)$
- Choose public exponent:
 - $e = 17$
- Compute private exponent:
 - $d = e^{-1} \pmod{\phi}$
This is done using the `modinv(e, phi)` function, which calculates the **modular inverse** using the extended Euclidean method.

Keys formed:

- Public key: (e, n)
- Private key: (d, n)

RSA Encryption

In RSA encryption, the sender converts the message into an integer `msg` and encrypts it using the receiver's public key:

`cipher=msge mod n`
`cipher = msg^e \bmod n`

In the code:

- `msg = 78`

- $\text{cipher} = \text{pow}(\text{msg}, e, n)$

This produces the encrypted message (ciphertext), which is safe to send over an insecure channel.

RSA Decryption

The receiver decrypts using their private key:

$\text{plain} = \text{cipher}^d \bmod n$

In the code:

- $\text{plain} = \text{pow}(\text{cipher}, d, n)$

This recovers the original message 65.

Result: RSA ensures **confidentiality** because only the receiver has d .

Part 2: ElGamal Digital Signature (Signing / Verification)

ElGamal Key Generation

ElGamal signatures work in a finite group modulo a prime p .

In the code:

- Select prime: $p = 199$
- Select generator: $g = 2$
- Choose private key:
 - $x = \text{random.randint}(1, p-2)$
- Compute public key:
 - $y = g^x \bmod p$

Keys formed:

- Public key: (p, g, y)
 - Private key: x
-

Signature Generation

The sender signs the message using a random value k.

In the code:

- Hash value is kept simple as:
 - $h = \text{msg}$
(So the message itself is treated like the hash.)
- Choose random k:
 - $k = \text{random.randint}(1, p-2)$
- Compute:
 - $r = g^k \bmod p$
- Compute s:

$$s = (h - x \cdot r) \cdot k^{-1} \bmod (p-1)$$

In the code:

- $s = ((h - x \cdot r) * \text{modinv}(k, p-1)) \% (p-1)$

Signature becomes the pair:

- **Signature (r, s)**

Purpose: This signature proves the sender created the message.

Signature Verification

The receiver verifies the signature using the sender's public key y.

Two values are computed:

1.

$$v1 = (y^r \cdot r^s) \bmod p$$

]

In the code:

- $v1 = (\text{pow}(y, r, p) * \text{pow}(r, s, p)) \% p$

2.

$$v2 = g^h \bmod p$$

]

In the code:

- $v2 = \text{pow}(g, h, p)$

If $v1 == v2$, the signature is valid:

- Signature Valid: True

This confirms:

- **Integrity:** message was not changed
 - **Authenticity:** signature was made by the sender's private key
-

Final Output Explanation

The program prints:

- **Encrypted Message:** RSA ciphertext (secure for sending)
- **Decrypted Message:** original message after RSA decryption
- **Digital Signature (r, s):** ElGamal signature values
- **Signature Valid:** shows whether verification passed or failed

```
import random

# ===== RSA =====
# Key Generation: Creates public and private keys for encryption
def modinv(e, phi):
    t, nt = 0, 1
    r, nr = phi, e
    while nr:
        q = r // nr
        t, nt = nt, t - q * nt
        r, nr = nr, r - q * nr
    return t % phi
```

```
p, q = 23, 19
n = p * q
phi = (p-1)*(q-1)
e = 17
d = modinv(e, phi)
```

```
# Encryption: Sender encrypts email using receiver's public key
```

```

msg = 78
cipher = pow(msg, e, n)

# Decryption: Receiver decrypts email using private key
plain = pow(cipher, d, n)

# ===== ElGamal =====
# Key Generation: Creates keys for digital signature
p = 199
g = 2
x = random.randint(1, p-2)      # private key
y = pow(g, x, p)                # public key

# Signature: Sender signs the message
h = msg
k = random.randint(1, p-2)
r = pow(g, k, p)
s = ((h - x*r) * modinv(k, p-1)) % (p-1)

# Verification: Receiver verifies the signature
v1 = (pow(y, r, p) * pow(r, s, p)) % p
v2 = pow(g, h, p)

# ===== OUTPUT =====
print("Encrypted Message:", cipher)
print("Decrypted Message:", plain)
print("Digital Signature (r, s):", (r, s))
print("Signature Valid:", v1 == v2)

```

OUTPUT:



```
"C:\Users\Discount Laptop\Desktop\python\.venv\Scripts\python.exe" "C:\Users\Discount Laptop\Desktop\p  
Encrypted Message: 371  
Decrypted Message: 78  
Digital Signature (r, s): (57, 3)  
Signature Valid: True  
Process finished with exit code 0
```

QUESTION: 02

You have developed a project during this course as your final submission. Using that same project:

ANSWERE

1. Justification of my security method

In my project I used salted PBKDF2-HMAC with SHA-256 to protect user passwords instead of storing plain text passwords or a single direct hash. This approach fits an online application very well because the main risk is that someone might steal the password database and then try to recover the original passwords. By adding a unique random salt to each password before hashing, users who choose the same password still end up with different stored values, so pre-computed tables and simple pattern matching do not work. PBKDF2 then applies SHA-256 many thousands of times, which makes every password guess deliberately slow. This does not bother a normal user who logs in a few times a day, but it makes large-scale brute-force and dictionary attacks much more expensive. PBKDF2-HMAC-SHA256 is also a standard and widely used construction, so it is a sensible and reliable choice for password storage in this kind of system.

2. One possible vulnerability or weakness

A remaining weakness in my current system is that it still depends on the strength of the passwords that users choose. Even with salt and PBKDF2, if many users pick very short or common passwords such as “123456” or simple dictionary words, an attacker who manages to

obtain a copy of the database can run an offline dictionary attack. They can take a large list of likely passwords, combine each one with the stored salt for a user, run PBKDF2 with the same parameters, and check whether the resulting hash matches the stored hash. Because this process can be automated, weak passwords will eventually be found, so the overall security is reduced by poor password choices even though the hashing scheme itself is strong.

3. One realistic improvement

A realistic improvement would be to add a clear strong-password policy and enforce it in the registration and password-change parts of the system. For example, the application could require a minimum length (such as 12 characters), a mix of letters, digits and symbols, and reject passwords that appear in lists of commonly used or previously breached passwords. This would raise the average entropy of user passwords and make offline dictionary attacks far less effective, because the attacker would have to search a much larger and less predictable password space. The existing salted PBKDF2-HMAC-SHA256 mechanism would then be combined with stronger passwords, giving a more robust overall level of security for the project.