جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الامن السيبراني

DEPARTMENT OF CYBER SECURITY

SUBJECT:

PUBLIC KEY ENCRYPTION

CLASS:

THIRD

LECTURER:

ASST. LECTURER QUSAI AL-DURRAH

LECTURE (7 & 8):

ELGAMAL PUBLIC-KEY ENCRYPTION

1. Introduction

The **ElGamal public-key cryptosystem**, introduced by *Taher ElGamal in 1985*, is an asymmetric encryption algorithm based on the **Discrete Logarithm Problem (DLP)**. It extends the ideas of the **Diffie–Hellman key exchange** and enables both **encryption** and **digital signatures**.
ElGamal is widely used in secure communication protocols such as **PGP**, **GPG**, and cryptographic libraries that implement public-key encryption.
ElGamal's security depends on the hardness of computing discrete logarithms in a cyclic group:
even if an attacker knows g^x and g^k, computing g^{x*k} is computationally infeasible.

2. Learning Outcomes

By the end of this lecture, students will be able to:

1. **Explain** how ElGamal encryption is derived from the Diffie–Hellman key exchange.
2. **Perform** key generation, encryption, and decryption using ElGamal.
3. **Describe** the flow of the ElGamal encryption process.
4. **Implement** a Python version of the ElGamal algorithm.
5. **Identify** the main advantages, disadvantages, and security considerations of ElGamal.

3. Components of the ElGamal Algorithm

ElGamal operates in the multiplicative group Z^*_p (The notation (read: "Z sub p star") refers to the **multiplicative group modulo p**, where:

- p is a **prime number**
- Z^*_p is the set of **all integers from 1 to p − 1**
- The group operation is **multiplication modulo p**

Formally:
$Z^*_p = \{1,2,3,\ldots,p-1\}$
then:

- (p) is a large prime number,
- (g) is a primitive generator of the group,
- All arithmetic is done **mod p**.

3.1 Key Generation

1. Select a large prime number (p).
2. Select a generator (g) of the group Z^*_p
3. Choose a private key (x) such that $1 \le x \le p-2$.

4. Compute the public key:
   $h = g^x \bmod p$

Public Key: (p, g, h)
Private Key: (x)

4. Encryption Process

To encrypt a message (M) for a receiver whose public key is (p, g, h):

1. Choose a random integer (k) such that ($1 \leq k \leq p-2$).
2. Compute:
   $C_1 = g^k \bmod p$
   $C_2 = M * h^k \bmod p$

Ciphertext:
$C = (C_1, C_2)$

5. Decryption Process

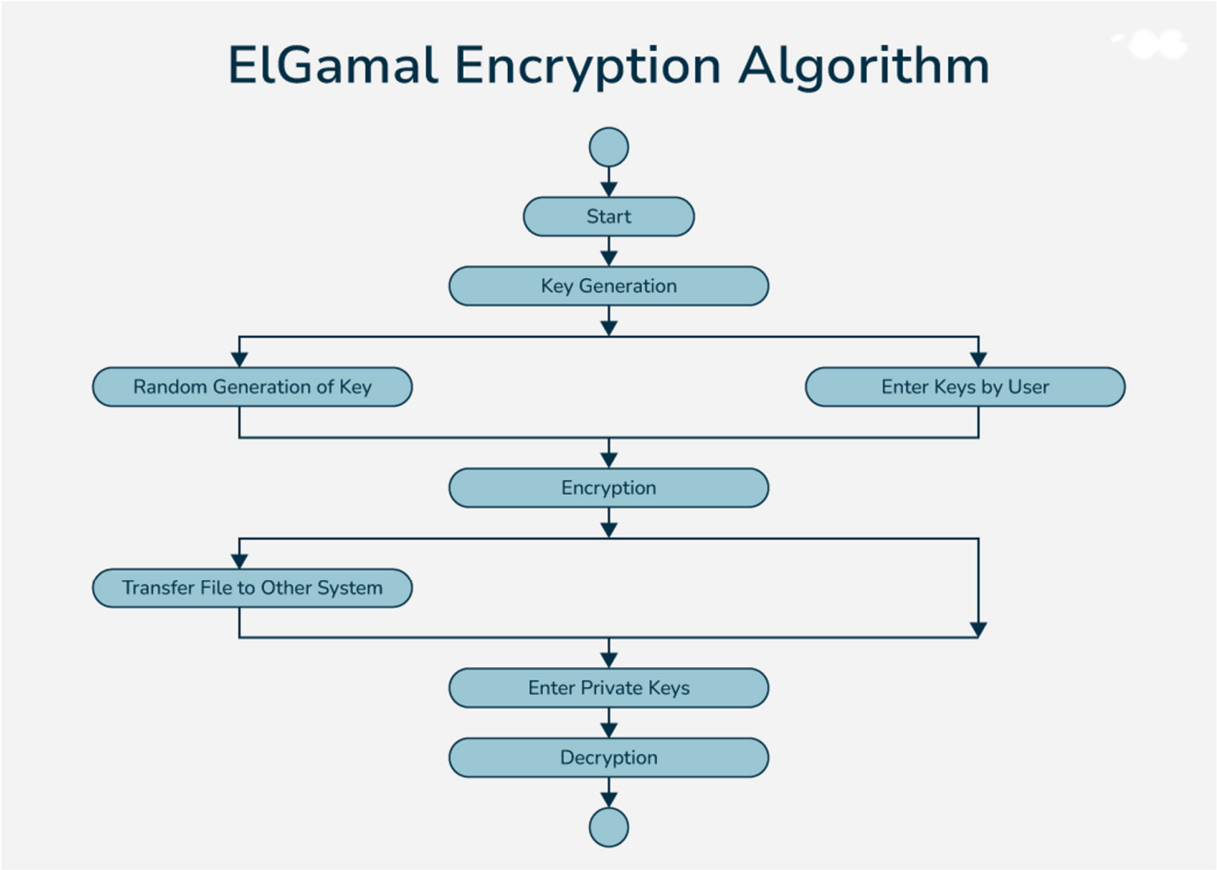
To decrypt (C_1, C_2):

1. Compute the shared secret key:
   $s = C_1^x \bmod p$
2. Compute the modular inverse ($s^{-1} \bmod p \equiv 1 \bmod p$).
3. Recover the original message:
   $M = C_2 * s^{-1} \bmod p$

Why does this work?

Since:
$s = C_1^x = (g^k)^x = g^{k*x}$
And:
$C_2 = M * h^k = M * (g\text{^}x)\text{^}k = M * g^{k*x}$
Then:
$M = C_2 / s \bmod p$

6. ElGamal Encryption Flowchart



This figure illustrates:

- Key generation
- Encryption path
- Decryption path

7. Conceptual Example

To avoid information loss when using small primes, we convert letters to numbers using the mapping:

A = 0, B = 1, C = 2, … Z = 25

And after decryption, we convert back to letters by:

ASCII = M + 65

7.1 System Parameters

- Prime: (p = 23)
- Generator: (g = 5)
- Private key: (x = 6)

Compute public component:
$h = g^x \bmod p = 5^6 \bmod 23 = 15625 \bmod 23 = 8$

Public Key: (p = 23, g = 5, h = 8)
Private Key: (x = 6)

7.2 Convert "HELLO" to Numbers

So uppercase letters start at 0:

Letter	Number
H	7
E	4
L	11
L	11
O	14

7.3 Encryption (Sender)

Choose random:
k = 7

Compute:
$C_1 = g^k \bmod p = 5^7 \bmod 23 = 17$

Now encrypt each symbol using:
$C_2 = M * h^k \bmod p$

Encrypting all letters

Letter	M	$C_2 = M \times 8^7 \bmod 23$
H	7	($7\times8^7 \bmod 23 = 15$)
E	4	($4\times8^7 \bmod 23 = 2$)
L	11	($11\times8^7 \bmod 23 = 17$)
L	11	17
O	14	($14\times8^7 \bmod 23 = 7$)

Final Ciphertext

$C = (C_1 = 17, C_2 = [15, 2, 17, 17, 7])$

7.4 Decryption (Receiver)

Compute:

$s = C_1^x \bmod p = 17^6 \bmod 23 = 12$

Find modular inverse:

$s^{-1} = 12^{-1} \bmod 23$

$12*s^{-1} \equiv 1 \bmod 23$

x	12 × x	mod 23
1	12	12
2	24	1
3	36	13
4	48	2
5	60	14

$s^{-1} = 12^{-1} \bmod 23 = 2$

Since:

$12*2 = 24 \equiv 1 \bmod 23$

Recover plaintext numbers:

$M = C_2 * s^{-1} \bmod p$

Decrypting each element

C_2	$M = C_2 \times 2 \bmod 23$
15	30 mod 23 = 7
2	4
17	34 mod 23 = 11
17	11
7	14

Numeric Plaintext
[7, 4, 11, 11, 14]

7.5 Convert Numbers Back to Letters

We use:
ASCII = M + 65

M	ASCII	Letter
7	72	H
4	69	E
11	76	L
11	76	L
14	79	O

8. Python Implementation

```
def char_to_num(ch):
    # Map 'A'..'Z' → 0..25
    return ord(ch) - 65

def num_to_char(n):
    # Map 0..25 → 'A'..'Z' (via +65)
    return chr(n + 65)

def mod_inverse(a, p):
    # Compute modular inverse of a mod p
```

```
    return pow(a, -1, p)

# System parameters
p = 23          # prime
g = 5           # generator
x = 6           # private key of receiver (A)

# compute public key h = g^x mod p
h = pow(g, x, p)

print("System parameters:")
print("p =", p, ", g =", g, ", x (private) =", x)
print("h (public) = g^x mod p =", h)
print("-" * 40)

# 2. Plaintext message
message = "HELLO"
print("Original message:", message)

# Convert to numbers using A=0, B=1, ..., Z=25
M_list = [char_to_num(ch) for ch in message]
print("Numeric plaintext (A=0 mapping):", M_list)

# 3. Encryption side (sender)
k = 7   # random key

# Shared key s = h^k mod p
C1 = pow(g, k, p)

print("\nEncryption:")
print("k =", k)
print("C1 = g^k mod p =", C1)

# Encrypt each symbol: C2 = M * s mod p
C2_list = [(M * (h**k)) % p for M in M_list]
print("C2 components:", C2_list)
print("Ciphertext = (C1, C2_list) = (", C1, ",", C2_list, ")")
```

```
# 4. Decryption side (receiver)
print("\nDecryption:")
# Recompute s = C1^x mod p
s = pow(C1, x, p)
print("s (from C1^x mod p) =", s)

# Compute modular inverse of s
s_inv = mod_inverse(s, p)
print("s^-1 mod p =", s_inv)

# Recover each plaintext number: M = C2 * s_inv mod p
M_dec_list = [(C2 * s_inv) % p for C2 in C2_list]
print("Recovered numeric plaintext:", M_dec_list)

# Convert numbers back to letters using +65
decrypted_message = "".join(num_to_char(M) for M in M_dec_list)
print("Decrypted message:", decrypted_message)

# 5. Check correctness
if decrypted_message == message:
    print("\n ElGamal successful: plaintext recovered correctly.")
else:
    print("\n Error: plaintext not recovered correctly.")
```

9. Applications

ElGamal is used in:

- Secure message encryption
- Digital signatures
- GPG (GNU Privacy Guard) / PGP (Pretty Good Privacy)
- Cryptographic libraries
- Distributed secure key exchange

10. Advantages and Disadvantages

Advantages

- High security based on DLP(Discrete Logarithm Problem)
- Supports both encryption and signatures
- Easy key distribution due to asymmetric nature

Disadvantages

- Slower than RSA
- Requires larger key sizes
- Vulnerable to DLP-related attacks if parameters are weak

11. Conclusion

The ElGamal cryptosystem is a powerful asymmetric encryption technique that offers strong security through the discrete logarithm problem.
It is widely used in modern cryptosystems and remains foundational in public-key cryptography.
Although its ciphertext expansion and computational cost are higher than RSA, its randomness and security properties make it valuable for secure communications.