

MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA  
TECHNICAL UNIVERSITY OF MOLDOVA  
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS  
SOFTWARE ENGINEERING DEPARTMENT

CRYPTOGRAPHY AND SECURITY

LABORATORY WORK #5

---

## Block Ciphers. DES Algorithm.

---

*Author:*

Alexandru RUDOI

std. gr. FAF-231

*Verified:*

Maia ZAICA

asist. univ.

Chişinău 2025

# Contents

<b>Objective . . . . .</b>	<b>3</b>
<b>Task Description . . . . .</b>	<b>3</b>
<b>Theoretical Background . . . . .</b>	<b>3</b>
<b>Technical Implementation . . . . .</b>	<b>6</b>
<b>Results . . . . .</b>	<b>11</b>
<b>Conclusion . . . . .</b>	<b>14</b>
<b>Bibliography . . . . .</b>	<b>15</b>

## Objective

The goal of this laboratory work is to implement and demonstrate the functionality of three fundamental cryptographic algorithms: RSA, ElGamal, and Diffie-Hellman. Through the implementation of these algorithms, the principles of **public-key cryptography**, asymmetric encryption/decryption mechanisms, and **secure key exchange** processes will be explored.

The specific objectives include:

- Implementing the RSA algorithm with at least 2048-bit keys.
- Implementing the ElGamal algorithm with specified parameters.
- Demonstrating the Diffie-Hellman key exchange to generate an AES-256 key.

## Task Description

The laboratory work consists of the following tasks:

- **Task 2.1 - RSA Algorithm:** Generate RSA keys with at least 2048 bits and perform encryption and decryption for the message ‘m = "Nume Prenume"’.
- **Task 2.2 - ElGamal Algorithm:** Generate ElGamal keys using specified parameters ‘p’ and ‘g’, and perform encryption and decryption for the message ‘m = "Nume Prenume"’.
- **Task 3 - Diffie-Hellman Key Exchange:** Perform Diffie-Hellman key exchange between Alice and Bob, using the algorithm to derive a 256-bit AES key.

For tasks 2.1 and 2.2, the message will be represented numerically using its ASCII hexadecimal representation.

## Theoretical Background

### RSA Algorithm

**RSA** is an asymmetric encryption algorithm based on the difficulty of factoring large prime numbers. It provides both confidentiality (encryption) and authenticity (digital signatures).

#### Key Generation:

1. Choose two large prime numbers  $p$  and  $q$ .
2. Compute  $n = p \times q$  and  $\phi(n) = (p - 1)(q - 1)$ .

3. Choose an encryption exponent  $e$  such that  $\gcd(e, \phi(n)) = 1$ .
4. Compute the decryption exponent  $d$ , which is the modular inverse of  $e$  modulo  $\phi(n)$ .
5. Public key:  $(n, e)$ , Private key:  $(n, d)$ .

#### Encryption and Decryption:

1. Encryption:  $c = m^e \mod n$ .
2. Decryption:  $m = c^d \mod n$ , where  $m$  is the message and  $c$  is the ciphertext.

**Security:** The security of RSA is based on the assumption that factoring large numbers is computationally infeasible.

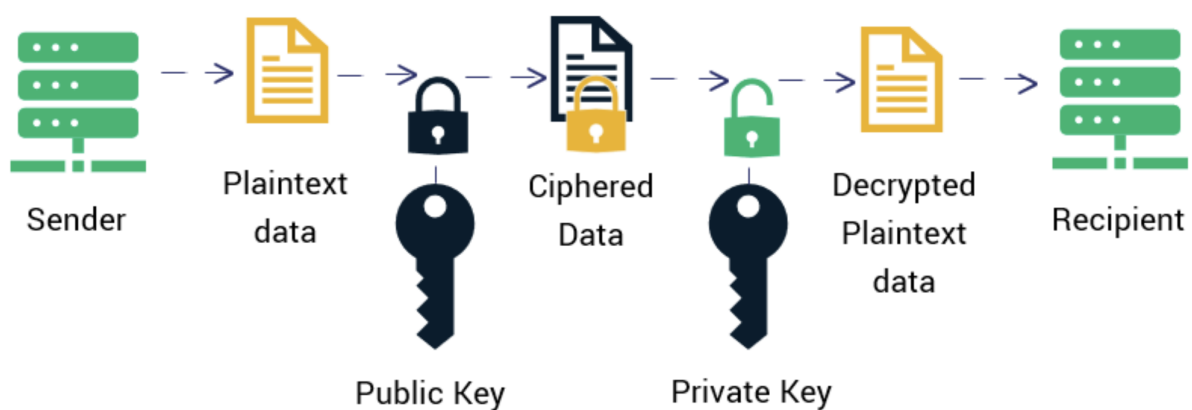


Figure 1: RSA Key Generation and Encryption/Decryption Process

## ElGamal Algorithm

**ElGamal** is an asymmetric encryption algorithm based on the difficulty of computing discrete logarithms in a finite field.

#### Key Generation:

1. Choose a large prime  $p$  and a generator  $g$ .
2. Choose a private key  $x$  and compute the public key  $y = g^x \mod p$ .
3. Public key:  $(p, g, y)$ , Private key:  $x$ .

#### Encryption and Decryption:

1. Choose a random integer  $k$ .
2. Compute ciphertext as  $c_1 = g^k \mod p$  and  $c_2 = m \times y^k \mod p$ .
3. Decrypt the ciphertext by calculating  $s = c_1^x \mod p$  and then  $m = c_2 \times s^{-1} \mod p$ .

**Security:** The security of ElGamal is based on the difficulty of the discrete logarithm problem.

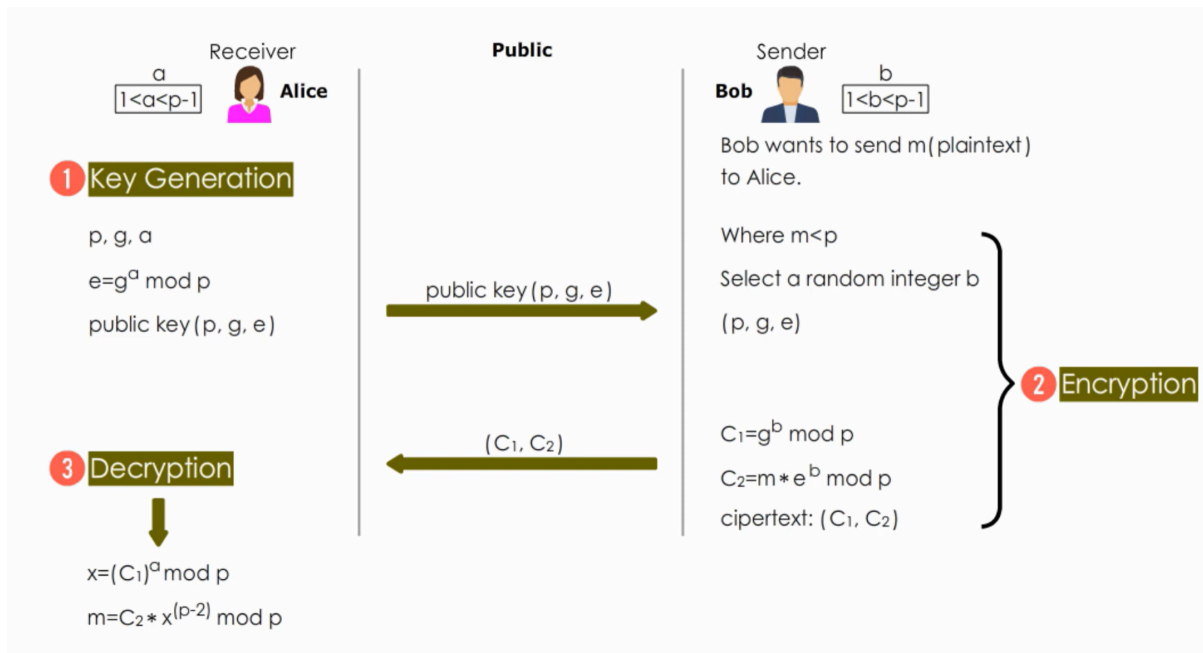


Figure 2: ElGamal Encryption and Decryption Process

## Diffie-Hellman Key Exchange

**Diffie-Hellman** is a key exchange protocol that allows two parties to securely establish a shared secret over an insecure channel.

### Key Exchange Process:

1. Choose a large prime  $p$  and a generator  $g$ .
2. Alice and Bob each select a private key  $a$  and  $b$ , respectively.
3. Alice computes  $A = g^a \mod p$ , and Bob computes  $B = g^b \mod p$ .
4. Alice and Bob exchange  $A$  and  $B$ , respectively.
5. Alice computes the shared key  $K_A = B^a \mod p$ , and Bob computes  $K_B = A^b \mod p$ .
6. The shared keys  $K_A$  and  $K_B$  are the same.

**Security:** The security of Diffie-Hellman relies on the difficulty of computing discrete logarithms.

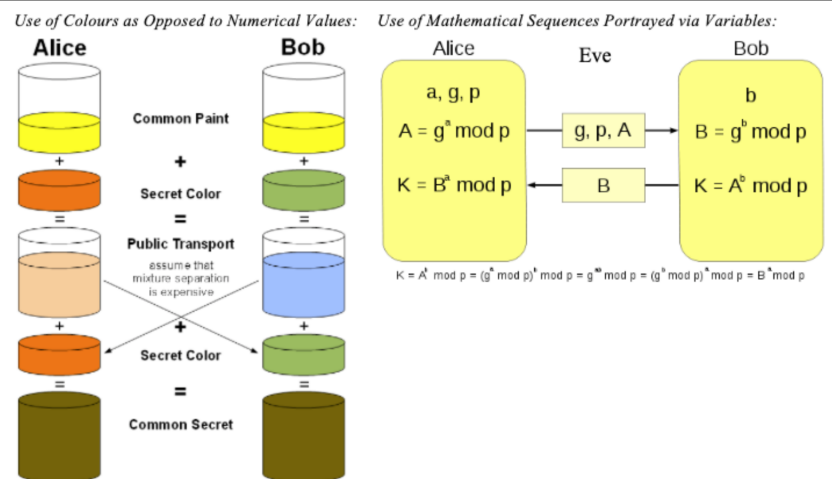


Figure 3: Diffie-Hellman Key Exchange Process

## Technical Implementation

In this section, we describe the implementation of the **RSA**, **ElGamal**, and **Diffie-Hellman** algorithms. For each algorithm, we will provide the **key generation**, **encryption**, and **decryption** steps along with the corresponding Python code snippets.

### RSA Algorithm Implementation

The RSA algorithm is based on the difficulty of factoring large prime numbers and relies on the use of a public and private key pair for encryption and decryption.

#### 1. Key Generation

In RSA, we first generate two large prime numbers,  $p$  and  $q$ , and compute the modulus  $n = p \times q$ . We also calculate the totient function  $\phi(n) = (p-1)(q-1)$  and choose a public exponent  $e$ . The private exponent  $d$  is calculated as the modular inverse of  $e$  modulo  $\phi(n)$ .

```
1 from sympy import randprime, gcd, mod_inverse
2
3 def generate_rsa_keys(bits=2048):
4     p = randprime(2**(bits//2 - 1), 2**(bits//2))
5     q = randprime(2**(bits//2 - 1), 2**(bits//2))
6     n = p * q
7     phi_n = (p - 1) * (q - 1)
8     e = 65537
9     while gcd(e, phi_n) != 1:
```

```

10     e += 2
11     d = mod_inverse(e, phi_n)
12     return {'public_key': (n, e), 'private_key': (n, d), 'p': p, 'q': q}

```

Listing 1: RSA Key Generation

**Explanation:**

1. Generate two large prime numbers,  $p$  and  $q$ .
2. Compute  $n = p \times q$  and  $\phi(n) = (p - 1)(q - 1)$ .
3. Choose the public exponent  $e$  (commonly  $e = 65537$ ).
4. Compute the private exponent  $d$  using the modular inverse of  $e$  modulo  $\phi(n)$ .

**2. Encryption**

To encrypt a message  $m$  using the RSA algorithm, we use the public key  $(n, e)$  and compute the ciphertext  $c = m^e \bmod n$ .

```

1 def rsa_encrypt(message, public_key):
2     n, e = public_key
3     m = message_to_number(message) # Convert message to number
4     if m >= n:
5         raise ValueError("Message is too large for the RSA key")
6     c = pow(m, e, n) # c = m^e mod n
7     return c

```

Listing 2: RSA Encryption

**Explanation:**

1. Convert the message  $m$  to its numeric representation using ASCII encoding.
2. Compute the ciphertext  $c = m^e \bmod n$  using the public key.

**3. Decryption**

Decryption is done using the private key  $(n, d)$  and the ciphertext  $c$ . The message  $m$  is recovered by calculating  $m = c^d \bmod n$ .

```

1 def rsa_decrypt(ciphertext, private_key):
2     n, d = private_key
3     m = pow(ciphertext, d, n) # m = c^d mod n
4     message = number_to_message(m) # Convert number back to message
5     return message

```

## Listing 3: RSA Decryption

**Explanation:**

1. Decrypt the ciphertext by computing  $m = c^d \bmod n$  using the private key.
2. Convert the numeric message back to its string representation.

**ElGamal Algorithm Implementation**

The ElGamal algorithm is based on the difficulty of computing discrete logarithms in a finite field.

**1. Key Generation**

In ElGamal, a large prime  $p$  and a generator  $g$  are chosen. A private key  $x$  is selected, and the corresponding public key  $y$  is computed as  $y = g^x \bmod p$ .

```

1 import random
2
3 def generate_elgamal_keys():
4     p = 3231700607131100730015351347782516336248805713348907517458843413926 #
        large prime p
5     g = 2 # generator g
6     x = random.randint(2, p - 2) # private key
7     y = pow(g, x, p) # public key y = g^x mod p
8     return {'public_key': (p, g, y), 'private_key': x}

```

## Listing 4: ElGamal Key Generation

**Explanation:**

1. Select a large prime  $p$  and a generator  $g$ .
2. Choose a private key  $x$  and compute the corresponding public key  $y = g^x \bmod p$ .

**2. Encryption**

To encrypt a message  $m$ , a random integer  $k$  is chosen, and the ciphertext is computed as  $c_1 = g^k \bmod p$  and  $c_2 = m \times y^k \bmod p$ .

```

1 def elgamal_encrypt(message, public_key):
2     p, g, y = public_key

```



```

3     m = message_to_number(message) # Convert message to number
4     k = random.randint(2, p - 2) # choose random k
5     c1 = pow(g, k, p) # g^k mod p
6     c2 = (m * pow(y, k, p)) % p # m * y^k mod p
7     return (c1, c2)

```

Listing 5: ElGamal Encryption

**Explanation:**

1. Select a random integer  $k$  such that  $2 \leq k \leq p - 2$ .
2. Compute the ciphertext components  $c_1 = g^k \bmod p$  and  $c_2 = m \times y^k \bmod p$ .

**3. Decryption**

The ciphertext  $(c_1, c_2)$  is decrypted using the private key  $x$  by calculating  $s = c_1^x \bmod p$  and recovering the message  $m = c_2 \times s^{-1} \bmod p$ .

```

1 def elgamal_decrypt(ciphertext, private_key, p):
2     c1, c2 = ciphertext
3     x = private_key
4     s = pow(c1, x, p) # s = c1^x mod p
5     s_inv = mod_inverse(s, p) # modular inverse of s
6     m = (c2 * s_inv) % p # m = c2 * s_inv mod p
7     message = number_to_message(m)
8     return message

```

Listing 6: ElGamal Decryption

**Explanation:**

1. Calculate  $s = c_1^x \bmod p$ , where  $x$  is the private key.
2. Compute the inverse  $s^{-1}$  modulo  $p$ , and recover the message by calculating  $m = c_2 \times s^{-1} \bmod p$ .

**Diffie-Hellman Key Exchange Implementation**

The Diffie-Hellman algorithm allows two parties to exchange a shared secret key over an insecure channel.

## 1. Key Exchange

Alice and Bob each select private keys  $a$  and  $b$ , respectively. They compute their corresponding public values  $A = g^a \bmod p$  and  $B = g^b \bmod p$ , exchange them, and compute the shared secret key.

```
1 def diffie_hellman_exchange():
2     p = 3231700607131100730015351347782516336248805713348907517458843413926 #
        prime p
3     g = 2 # generator g
4     a = random.randint(2, p - 2) # Alice's private key
5     b = random.randint(2, p - 2) # Bob's private key
6     A = pow(g, a, p) # Alice computes A = g^a mod p
7     B = pow(g, b, p) # Bob computes B = g^b mod p
8     shared_key_alice = pow(B, a, p) # Alice computes shared key
9     shared_key_bob = pow(A, b, p) # Bob computes shared key
10    assert shared_key_alice == shared_key_bob, "Keys do not match!"
11    return {'shared_key': shared_key_alice}
```

Listing 7: Diffie-Hellman Key Exchange

### Explanation:

1. Alice and Bob select their private keys,  $a$  and  $b$ .
2. They compute their corresponding public values  $A = g^a \bmod p$  and  $B = g^b \bmod p$ .
3. Alice and Bob compute the shared secret key independently using the public values.

## Results

In this section, the results of the RSA, ElGamal, and Diffie-Hellman implementations are shown through the corresponding screenshots. Each screenshot is accompanied by a brief explanation of the operations and outcomes.

### RSA Algorithm Results

- **Screenshot 1: RSA Key Generation** In this screenshot, the RSA algorithm

```

Enter your first and last name: Alexandru Rudoi

=====
Task 2.1: RSA Algorithm
=====
Generating RSA keys with 2048 bits...
p = 11665605171309504296278069812804557751274839112604199664527898354969036617406393329398832259
28162448333322527271817137000933771661014150969565831042739676718850774945393507641950181997048
911744301864098732216077520161349559923976476594575500601489803860670145567499825351086977933287
8911521571852257113589427
q = 11552552730358562159630727746004230228220894599999095099797455126202053200873944251340953039
837678698070085942332821014015988685938758107267858051089412290245841528823402879215416109081255
199047371952324614164862690450330517721161269320524101933553951470596281405959135237541868465518
1721130219219904588361821
n = 13476751887309657611653806582821445866774300225502860021343864996720707543400348187026496250
280102987746617915432914237814743808113924104260396912207902496766059593542695689542605654258259
216942005869190112758907800273186672722289725451256546893616959667628922206933247482557317768252
400505742928479945181986945573492647643906601365828070123009467230103774945640398652809141385617
47447767757368830584747872764824378746580378484444860001463974125576590593539394702731233732086
90830566969337452217679697268308540347205121089368266327179787261313416512123350053670182116509
611855667955151630267012677620666687116066567 (2047 bits)
phi(n) = 134767518873096576116538065828214458667743002255028600213438649967207075434003481870264
962502801029877466179154329142378147438081139241042603969122079024967660595935426956895426056542
582592169420058691901127589078002731866727222897254512565468936169596676289222069332474825573177
682524005057429284799451819867133919136309632420422778524820351296722727666489126927553992743296
747192916743017662904528562857509507922940505339969742756354254662515432023517722178750330843106
943989444370960334266691765717805301443068833957189832514655507284098021624204867729125003575178
46775022249781666687642206380025829594525414115320
e = 65537
d = 12896036454351748749555902348726388675755934663502462128546283863151284498363764527717055393
805271810893901907785592117369974646966576473602427202271910360249780967847827559672939383790808
402460961198628553968741090964376700282786144501955411833611553651159079578915799462496255080342
597203360792739411358437718494088523102946399703540941847644645577948561167915845543047302087228
144998557863029634709801307648763860670936090334433189258545020874145125451135993361762307956998
962456072196534579002248288331582768780664292302007250241586138577033002364877697508810611055202
434813167488029389591966531270142995137028153

```

Figure 4: RSA Key Generation and Display of Public/Private Keys

has successfully generated the `**public and private keys**`. The public key consists of ‘n’ and ‘e’, and the private key contains ‘n’ and ‘d’. The process involves selecting two large primes, computing ‘n’, and determining the corresponding private exponent ‘d’.

- **Screenshot 2: RSA Encryption and Decryption** This screenshot shows the

```

RSA keys generated.
Public key: (13476751887309657611653806582821445866774300225502860021343864996720707543400348187
026496250280102987746617915432914237814743808113924104260396912207902496766059593542695689542605
654258259216942005869190112758907800273186672722289725451256546893616959667628922206933247482557
317768252400505742928479945181986945573492647643906601365828070123009467230103774945640398652809
1413856174744776775736883058474787264824378746580378484444860001463974125576590593539394702731
23373208690830566966337452217679697268308540347205121089368266327179787261313416512123350053670
182116509611855667955151630267012677620666687116066567, 65537)
Private key: (1347675188730965761165380658282144586677430022550286002134386499672070754340034818
702649625028010298774661791543291423781474380811392410426039691220790249676605959354269568954260
565425825921694200586919011275890780027318667272228972545125654689361695966762892220693324748255
731776825240050574292847994518198694557349264764390660136582807012300946723010377494564039865280
9141385617474477677573688305847478726482437874658037848444486000146397412557659059353939470273
12337320869083056696633745221767969726830854034720512108936826632717978726131341651212335005367
0182116509611855667955151630267012677620666687116066567, 128960364543517487495559023487263886757
559346635024621285462838631512844983637645277170553938052718108939019077855921173699746469665764
736024272022719103602497809678478275596729393837908080424609611986285539687410909643767002827861
445019554118336115536511590795789157994624962550803425972033607927394113584377184940885231029463
99703540941847644645577948561167915845543047302087228144998578630296347098013076487638606709360
903344331892585450208741451254511359933617623079569989624560721965345790022482883315827687806642
92302072502415861385770330023648776975088106110552024348131674880293895919665312701429951370281
53)
Original message: Alexandru Rudoi
Numeric representation: 339697835342360413257449652453928809
Encrypted message: 10406712394493435792582065614948004357825019850373683196440148158698218270990
546130752976644523283429718278033496141906695421580247062279030242514732938772473148898055965902
065044263589781784680479438813255543235137668110817064064310809840250131460369963105112859382785
5342649163725995595825394485744446082502609777143007185970764171899533949013833600622946933668849
930298561259637400046258435055128389471457385513088601601583252597336817846164553285301848324043
937026272083708077205335087468161474846851276124232408831941538694601414384610465753049874821618
032079217173534154044459866676183150525685933092263378441418
Decrypted message (numeric): 339697835342360413257449652453928809
Decrypted message: Alexandru Rudoi

```

Figure 5: RSA Encryption and Decryption Process

encryption of the message "Nume Prenume" using the public key, resulting in an encrypted ciphertext. Following this, the ciphertext is decrypted back to the original message using the private key. The correct decryption confirms the functionality of the RSA algorithm.

## ElGamal Encryption Results

- **Screenshot 3: ElGamal Key Generation** This screenshot demonstrates the

```

=====
Task 2.2: ElGamal Algorithm
=====
Generating ElGamal keys...
p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362
640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325
26342821314576693141422365422094111134862999165747826803423055308634905063557712219187890332729
569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436
46747585460715043896844940366130497697812854295958659597567051283852132784468522925504568272879
113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281
00399496695611969695624862903233807283912703
g = 2
Private key x = 29836190566609854527519504493992667634125224935283623478676528058976474573598258
370773409425820797337823817328035485983430530487107671314263953189097194597399192830197714723588
364164363548437004309137973937256921726338170555385092592350413903477900154613515310673466020635
964583105434923526434928231277399400557896961850755578475792961550169791690038354794861593464930
455603959249419439408383857834711083759266010495298453429844392487812587717389732420901918177126
020558065879902579318565107797379374844330356406580990390954708963486245983884826032903350192530
560850710939473096422633683258781450890487656320280599
Public key y = 175017797922193454559378532558983595744644393939141286223983191455662900365167737
140275457294512729862451745171335294117851975540645664358409680828482056340160007039969557226675
62888226034660198943078602342600936989321643319472839660719458177295381663896038777111230789935
535469826204087978546311885944398171688560812218026770010041756268450956210182398623425933858978
253604330838131361971115433360221426870505025712785667353035252941615631849168513050215040803304
09803980054758381418177200579690957778943753282796189816286921766422141469649192526307976232158
3472758929054456298646857944784407682725593457888455800

```

Figure 6: ElGamal Key Generation and Public Key Display

**\*\*ElGamal key generation\*\***, where a **\*\*private key\*\*** ‘x’ is randomly chosen and the corresponding **\*\*public key\*\*** ‘y’ is computed. The parameters ‘p’ and ‘g’ are predefined, and ‘y’ is computed as  $g^x \mod p$ .

- **Screenshot 4: ElGamal Encryption and Decryption** The screenshot shows

```
Original message: Alexandru Rudoi
Numeric representation: 339697835342360413257449652453928809
k = 31336073047799073640845250464502218971729663330704579860601486669630868423197022122757229945
777076302581551238877188695285239632280336690498013809886110743798578678110750647001709388537757
844258229329839920148741810515818050495428535727822032247405100885898233285451694003154305120222
949932909431875274916324620100095853338667730718982657791671929324935364673683218342796517452151
874070018695741898066994285693430907817723894174615986534549946106886969998061962100252897118189
677206158898324600519604599455203659741758597059898731702159049390474582089051788879719507928387
66254003666421225203711536919434914119437803
c1 = 1515366591013312489173760161782709324415574409471535077223948546190878222657608071193464845
981768382052142533374831497568152178319958544673876973538753221952642746237306686711953420561886
498343501380300984441565403045406894482860977638248912202688136388133076096459678865332472050493
820227150536255367816344549058410418310568570105882198683717898874934555911357480597284619885547
368041672917134466953372051997839054004542257323851459225145996623755348080406986891017452946250
112009860425944035693983660404303872978074773513494815333534013290119173051599845178129094673590
454137871446214484452384843203148608928634770
c2 = 2946195954401400016490132310921874907461000704833681675941221286542482666094870578769596997
508832919130508312387198287796867740554452779168390881499098959811953282480563459202193569703799
807260053563040573217903092489356395824684125291527527152641278892672139234398616894598237051873
000408870995093465776059180390937907022608032675086473780293665344526145431787615075140310987437
642653926696130424654616789261630316194466424696398004203772826985375437356768564575101081649926
222935252306946354966653116295705496009169890887822443672577430492828292382738594023144074340827
570417060839632236114447234315297571114386607
s = c1^x mod p = 1788449562239825987787676323720015425988656130813238899665478948214953161534355
545871571489067854302459260838892230738083485418944542799108309429541566148817151462457151125532
376336397382456695560952479078262956982407984688471638224929011824990384566203751188434434525775
661697948551025101126397704805577833303961081343572191234200612988939997518789082224612356610506
236621807977790168013445854260229569222668484101784122858534344527270780090004112517775736306010
587954630644275676252717164464723387682635485069242142736887980168353658381575680626387744118600
187377933239145448102785414914997187081696419592863676134
s^(-1) = 196242637575384062203528910890494385217396646204912923151688026907491238179579337065230
207380595224806397392854417887005043283381046905237604811546855480919667956975897271979443959547
196250282242842021213033732041543650383298957848807659286276472120701628461854322742473513658614
716917504294808943061565451626748329181177268103327205741932108872135487580682268450082658200687
322886505372284688480337579913680705395089182097700677902352246232398848422094401662546519558666
706078920215840175335205904740272788965036316250067598236428828564058761420554072667645695403580
7772787201262874021774912831111728040029224783814
Decrypted message (numeric): 339697835342360413257449652453928809
Decrypted message: Alexandru Rudoi
```

Figure 7: ElGamal Encryption and Decryption Process

the encryption of the message “Nume Prenume” using the ElGamal public key. The message is split into two ciphertext components,  $c_1$  and  $c_2$ . These components are then decrypted back into the original message using the private key, successfully restoring the plaintext.

## Diffie-Hellman Key Exchange Results

- **Screenshot 5: Diffie-Hellman Key Exchange Process** This screenshot illustrates the **\*\*Diffie-Hellman key exchange\*\*** between Alice and Bob. Both parties select private keys, compute their corresponding public values, exchange them, and



```

=====
Task 3: Diffie-Hellman & AES
=====
Diffie-Hellman key exchange between Alice and Bob...
Public parameters: p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935
3308169288290230805734726252735547424612457410262025279165729728627063003252634282131457669314142236542209411113486299916574782680342305
5308634905063555771221918789033272956969612974385624174123623722519734640269185579776797682301462539793305801522685873076119753243646747
3548938243705417328744500086031979378910954824461270979967400868910081905693361142841783071253233564368941623606301955242414184306414339
9146144658937411610927190627
Alice calculates A = g^a mod p = 8694063127134125742351445446823684801643987759477330271161809216758769155481921798543394189009208253197
5582458172957476123614195091985221300900699578950093807907672603857216697724540039948776748240125240893863047936810544482051257067102543
358385034342311544054976334257691574596884535513382093214540404569677600668177103305084426405283688389039644884255911320852700519833353
375640573598886679281783388960742784879552951976257272480132864155478535478154770372189981602285642670853386647760916890456984118563541
05034921551939957910990912291560997901885516058271953284052685134165958465194500714996180484657606947018
Bob calculates B = g^b mod p = 894172383399952343281124079329822605460262529633864079092179531642738487674186165295649199693797767699117
3817475941909913908497477808580803275131043549271992260231450770680805326877963804237738981964480806419785040989869123079804776764689816
2639030596909236723369588469085138174605965532981877161794569982044586841230583959987011802883043934957966785841737074741528971528997894
64611618621554188097860224705502512036243542163208400112124710297224817942523542977507825260025849474361598068586613681762241973020629
38870291505185092136025709735772072105146612881235898716390462875924373857581517306359167733520657256
Alice calculates shared key: B^a mod p = 31098967645889295857731760876329095763763136744512716672600435566342079379526130493160619556860
28546358955423260080022014541755150588258101612934340343619048959376996879984624398921962502039326662513139346762455097578394468454240107
7549553453488877190900787444603831253052063197208452563504143059942229352876452321658780552704774970864770885525102185160547895869071870
0380642140953094007063921266844378909273646687810559513841393615049375208919992297448734740576211660158583745153327882844360848806838158
75404682000184876983323522453841116131337958520554463817305371458668824712910835344403755870645442275694795517209
Bob calculates shared key: A^b mod p = 3109896764588929585773176087632909576376313674451271667260043556634207937952613049316061955686028
546358955423260080022014541755150588258101612934340343619048959376996879984624398921962502039326662513139346762455097578394468454240107
4955345348887719090078744460383125305206319720845256350414305994222935287645232165878055270477497086477088552510218516054789586907187003
8064214095309400706392126684437890927364668781055951384139361504937520891999229744873474057621166015858374515332788284436084880683815875
404682000184876983323522453841116131337958520554463817305371458668824712910835344403755870645442275694795517209

```

Figure 8: Diffie-Hellman Key Exchange and Shared Secret Generation

then compute the shared secret key. The shared secret is derived independently by both Alice and Bob, ensuring they end up with the same key.

### • Screenshot 6: AES Key Derivation and Encryption The final screenshot

```

Shared key established: 3109896764588929585773176087632909576376313674451271667260043556634207937952613049316061955686028546358955423260
0800220145417551505882581016129343403436190489593769968799846243989219625020393266625131393467624550975783944684542401075495534534888771
9090078744460383125305206319720845256350414305994222935287645232165878055270477497086477088552510218516054789586907187003806421409530940
0706392126684437890927364668781055951384139361504937520891999229744873474057621166015858374515332788284436084880683815875404682000184876
98332352453841116131337958520554463817305371458668824712910835344403755870645442275694795517209
Derived AES-256 key: 2fe4fac3ad5ca2dccb4d78e5a7e295c4bdd7d6359c52133a30a78cf69ff66c

Encryption/decryption demonstration with derived key:
Original message: Alexandru Rudoi
Derived key (first 128 bits): 2fe4fac3ad5ca2dccb4d78e5a7e295c
Encrypted message (hex): 6e889fbcc32c6aebd485fb3e1140
Decrypted message: Alexandru Rudoi

```

Figure 9: AES Key Derivation from Diffie-Hellman Shared Secret

demonstrates the **\*\*AES-256 key derivation\*\*** from the shared secret established by Diffie-Hellman. The shared key is hashed using SHA-256 to generate the final AES key, which can then be used for symmetric encryption in secure communication.

## Conclusion

In this laboratory work, I implemented and tested three core cryptographic algorithms—RSA, ElGamal, and Diffie-Hellman—demonstrating key principles of public-key cryptography.

The RSA algorithm, based on large prime factorization, securely handles both encryption and digital signatures through key pairs. The implementation reaffirmed the importance of prime numbers and modular arithmetic in ensuring the security of cryptographic systems.

The ElGamal encryption system, which relies on the discrete logarithm problem, provided another layer of asymmetric cryptography. By using random values for key generation and encryption, it effectively ensures message security and confidentiality.

Finally, the Diffie-Hellman key exchange allowed us to securely establish a shared key over an insecure channel, which was then used to derive an AES-256 key for secure communication. This task highlighted the significance of modular exponentiation and secure key exchange protocols.

Through the successful implementation of these algorithms, I gained a deeper understanding of how public-key cryptography works to secure communications in real-world applications. The results underscore the importance of selecting secure parameters and random values in cryptographic systems to defend against potential attacks.

**Git repository:** [https://github.com/AlexandruRudoi/CS\\_Labs/tree/main/Lab\\_5](https://github.com/AlexandruRudoi/CS_Labs/tree/main/Lab_5)

## Bibliography

1. Course materials: *Cryptography and Security*, UTM-FCIM, 2025.
2. ElGamal, T. (1985). *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. IEEE Transactions on Information Theory, 31(4), 469-472.  
This paper presents the ElGamal encryption system, which is based on the difficulty of computing discrete logarithms in a finite field.
3. Schneier, B. (2007). *Cryptography Engineering: Design Principles and Practical Applications*. Wiley.  
This book covers the implementation of various cryptographic algorithms, including RSA, ElGamal, Diffie-Hellman, and AES. It provides in-depth discussions on the design principles behind modern cryptography.
4. Stinson, D. R. (2006). *Cryptography: Theory and Practice* (3rd ed.). CRC Press.  
A comprehensive textbook covering the theoretical background of cryptographic algorithms and their practical applications, including RSA, ElGamal, Diffie-Hellman, and AES.
5. NIST (National Institute of Standards and Technology). (2015). *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (SP 800-56A Rev. 3)*. NIST Special Publication 800-56A.  
This NIST document outlines recommended practices for the implementation of key exchange schemes like Diffie-Hellman.