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

Public Key Cryptography.

Author:
Alexandru Rudoi
std. gr. FAF-231

Verified:
Maia Zaica
asist. univ.

Chișinău 2025

Contents

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.

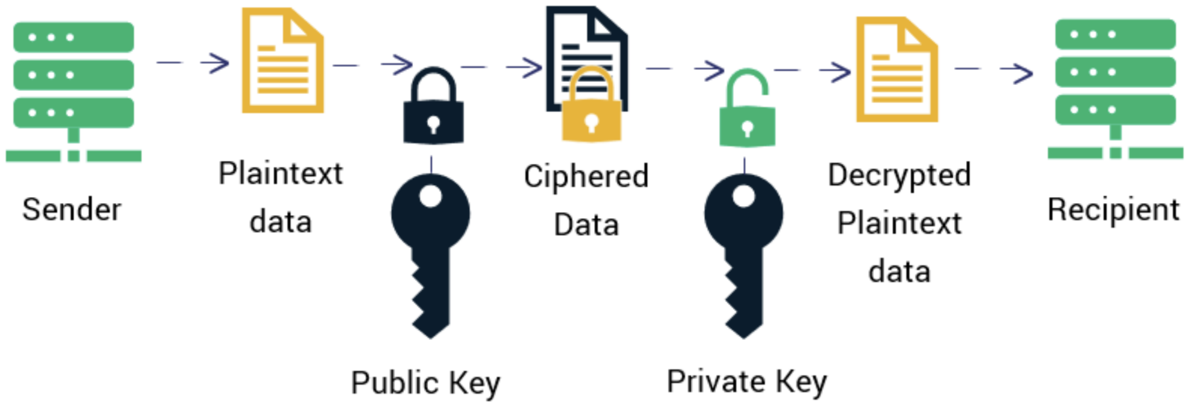


Figure 1: RSA Key Generation and Encryption/Decryption Process

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.

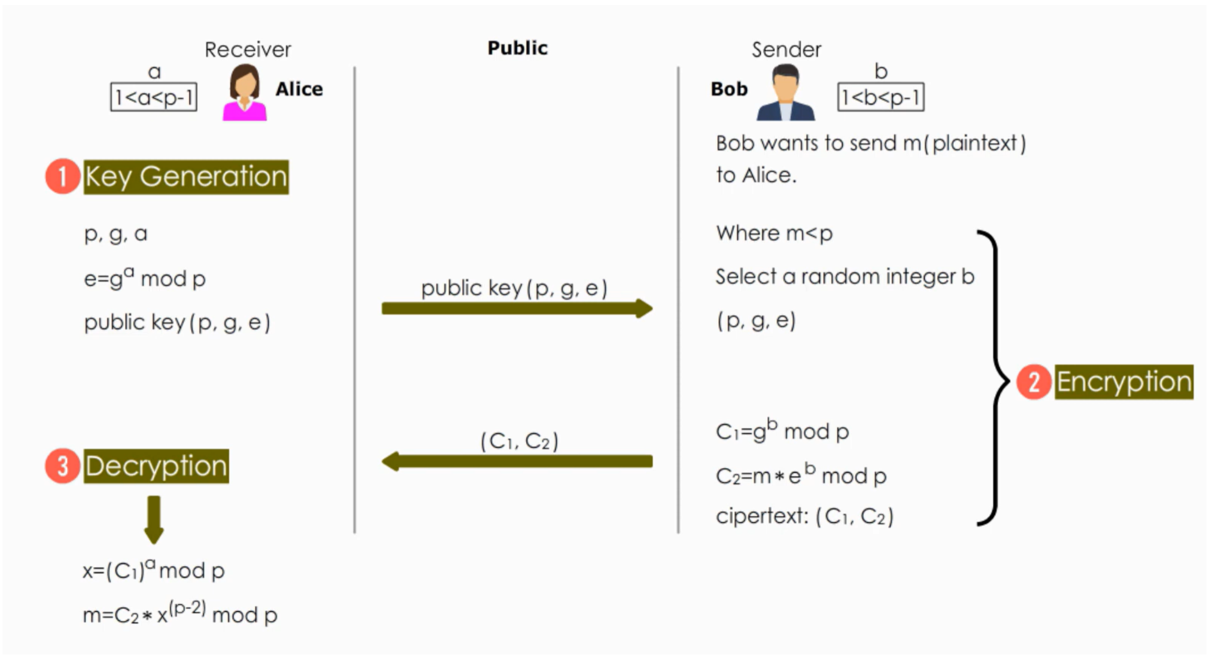


Figure 2: ElGamal Encryption and Decryption Process

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.

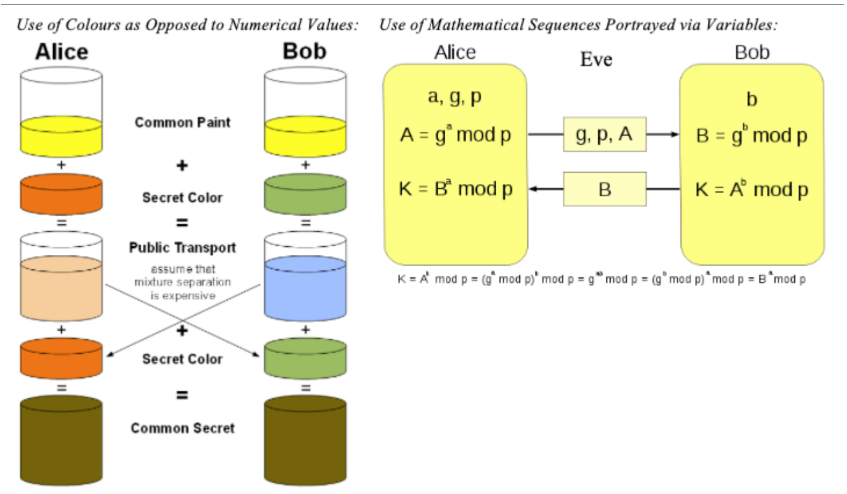

Use of Colours as Opposed to Numerical Values:
Alice
Bob
Common Paint
+
Secret Color
=
Public Transport
assume that mixture separation is expensive
+
Secret Color
=
Common Secret
Use of Mathematical Sequences Portrayed via Variables:
Alice
Eve
Bob
a, g, p
A = g^a mod p
g, p, A
b
B = g^b mod p
K = B^a mod p
B
K = A^b mod p
K = A^b mod p = (g^a mod p)^b mod p = g^ab mod p = (g^b mod p)^a mod p = B^a mod p

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 $\phi(n)$.

```python
from sympy import randprime, gcd, mod_inverse

def generate_rsa_keys(bits=2048):
    p = randprime(2**(bits//2 - 1), 2**(bits//2))
    q = randprime(2**(bits//2 - 1), 2**(bits//2))
    n = p * q
    phi_n = (p - 1) * (q - 1)
    e = 65537
    while gcd(e, phi_n) != 1:
        e += 2
    d = mod_inverse(e, phi_n)
    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 $c = m^e \mod n$.

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

Listing 2: RSA Encryption

Explanation:

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

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

Listing 3: RSA Decryption

Explanation:

1. Decrypt the ciphertext by computing $m = c^d \mod n$ 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 $y = g^x \mod p$.

```python
import random

def generate_elgamal_keys():
    p = 32317006071311007300153513477825163362488057133489075174588434139 26 # large prime p
    g = 2 # generator g
    x = random.randint(2, p - 2) # private key
    y = pow(g, x, p) # public key y = g^x mod p
    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 $y = g^x \mod p$.

2. Encryption

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

```python
def elgamal_encrypt(message, public_key):
    p, g, y = public_key
    m = message_to_number(message) # Convert message to number
    k = random.randint(2, p - 2) # choose random k
    c1 = pow(g, k, p) # g^k mod p
    c2 = (m * pow(y, k, p)) % p # m * y^k mod p
    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 $c_1 = g^k \mod p$ and $c_2 = m \times y^k \mod p$.

3. Decryption

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

```python
def elgamal_decrypt(ciphertext, private_key, p):
    c1, c2 = ciphertext
    x = private_key
    s = pow(c1, x, p) # s = c1^x mod p
    s_inv = mod_inverse(s, p) # modular inverse of s
    m = (c2 * s_inv) % p # m = c2 * s_inv mod p
    message = number_to_message(m)
    return message
```

Listing 6: ElGamal Decryption

Explanation:

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

—

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 $A = g^a \mod p$ and $B = g^b \mod p$, exchange them, and compute the shared secret key.

```python
def diffie_hellman_exchange():
    p = 32317006071311007300153513477825163362488057133489075174588434139269 # prime p
    g = 2 # generator g
    a = random.randint(2, p - 2) # Alice's private key
    b = random.randint(2, p - 2) # Bob's private key
    A = pow(g, a, p) # Alice computes A = g^a mod p
    B = pow(g, b, p) # Bob computes B = g^b mod p
    shared_key_alice = pow(B, a, p) # Alice computes shared key
    shared_key_bob = pow(A, b, p) # Bob computes shared key
    assert shared_key_alice == shared_key_bob, "Keys do not match!"
    return {'shared_key': shared_key_alice}
```

Listing 7: Diffie-Hellman Key Exchange

Explanation:

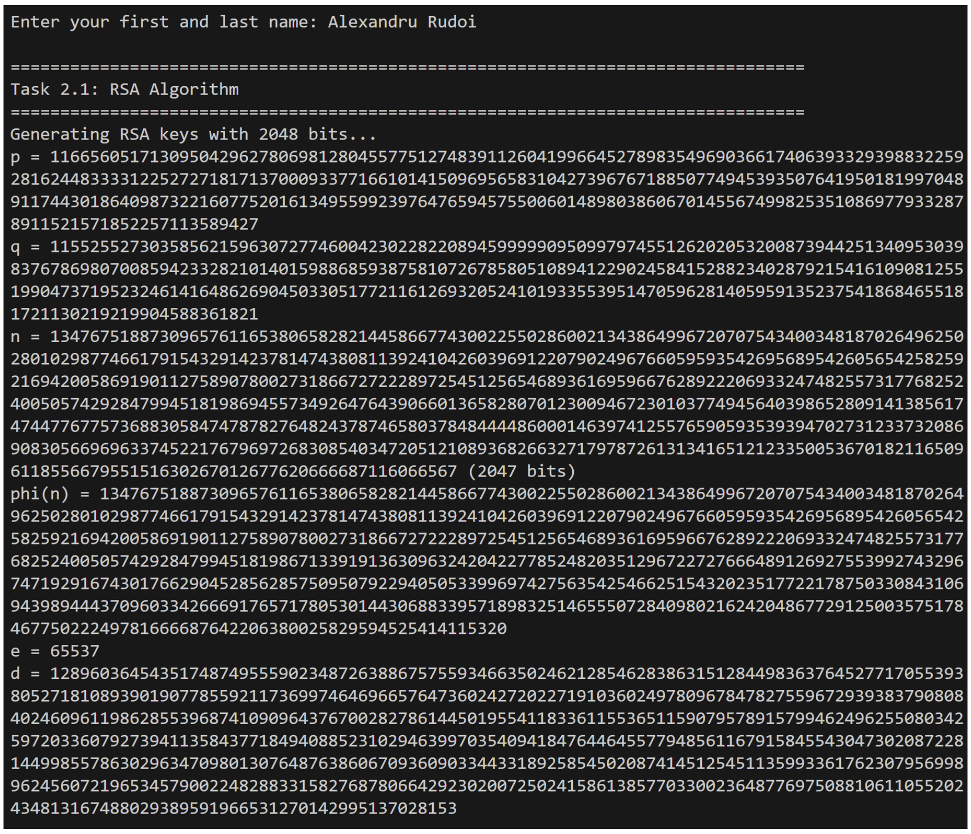
1. Alice and Bob select their private keys, a and b.
2. They compute their corresponding public values $A = g^a \mod p$ and $B = g^b \mod p$.
3. Alice and Bob compute the shared secret key independently using the public values.

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



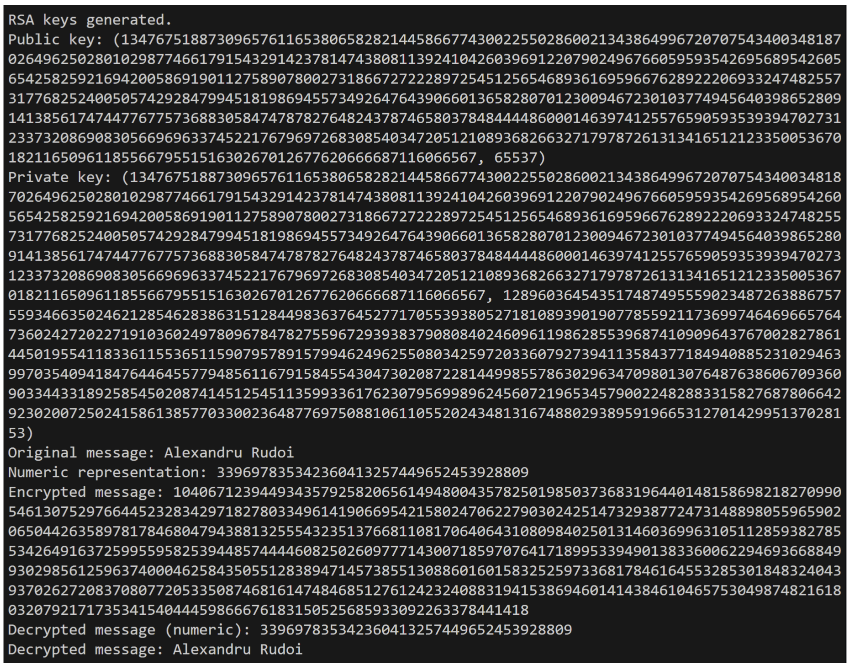
```
Enter your first and last name: Alexandru Rudoi

================================================================================
Task 2.1: RSA Algorithm
================================================================================
Generating RSA keys with 2048 bits...
p = 116656051713095042962780698128045577512748391126041996645278983549690366174063933293988322592816244833331225272718171370009337716610141509695658310427396767188507749453935076419501819970489117443018640987322160775201613495599239764765945755006014898038606701455674998253510869779332878911521571852257113589427
q = 115525527303585621596307277460042302282208945999990950997974551262020532008739442513409530398376786980700859423328210140159886859387581072678580510894122902458415288234028792154161090812551990473719523246141648626904503305177211612693205241019335539514705962814059591352375418684655181721130219219904588361821
n = 134767518873096576116538065828214458667743002255028600213438649967207075434003481870264962502801029877466179154329142378147438081139241042603969122079024967660595935426956895426056542582592169420058691901127589078002731866727222897254512565468936169596676289222069332474825573177682524005057429284799451819869455734926476439066013658280701230094672301037749456403986528091413856174744776775736883058474787827648243787465803784844448600014639741255765905935393947027312337320869083056696963374522176796972683085403472051210893682663271797872613134165121233500536701821165096118556679551516302670126776206666871160665 67 (2047 bits)
phi(n) = 134767518873096576116538065828214458667743002255028600213438649967207075434003481870264962502801029877466179154329142378147438081139241042603969122079024967660595935426956895426056542582592169420058691901127589078002731866727222897254512565468936169596676289222069332474825573177682524005057429284799451819867133919136309632420422778524820351296722727666489126927553992743296747192916743017662904528562857509507922940505339969742756354254662515432023517722178750330843106943989444370960334266691765717805301443068833957189832514655507284098021624204867729125003575178467750222497816666876422063800258295945254141153 20
e = 65537
d = 128960364543517487495559023487263886757559346635024621285462838631512844983637645277170553938052718108939019077855921173699746469665764736024272022719103602497809678478275596729393837908084024609611986285539687410909643767002827861445019554118336115536511590795789157994624962550803425972033607927394113584377184940885231029463997035409418476446455779485611679158455430473020872281449985578630296347098013076487638606709360903344331892585450208741451254511359933617623079569989624560721965345790022482883315827687806642923020072502415861385770330023648776975088106110552024348131674880293895919665312701429951370281 53
```

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 private exponent ‘d‘.

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



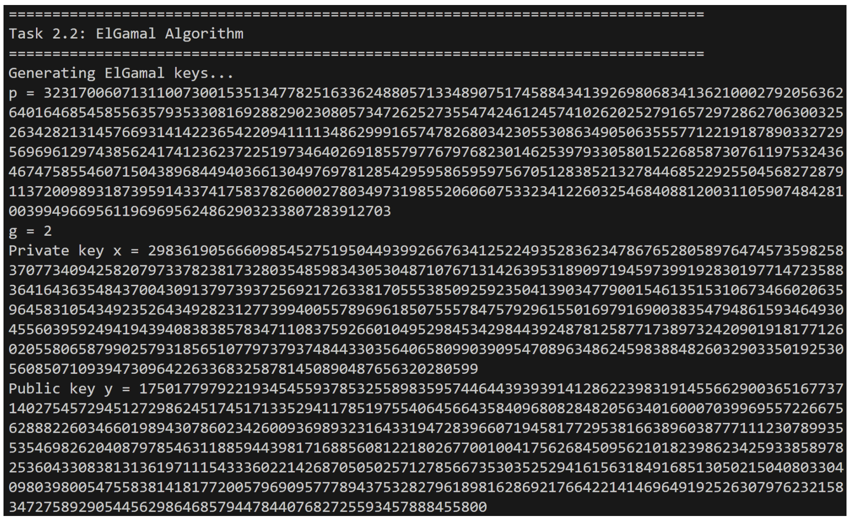
```
RSA keys generated.
Public key: (134767518873096576116538065828214458667743002255028600213438649967207075434003481870
026496250280102987746617915432914237814743808113924104260396912207902496766059593542695689542605
654258259216942005869190112758907800273186672722289725451256546893616959667628922206933247482557
317768252400505742928479945181986945573492647643906601365828070123009467230103774945640398652809
141385617474477677573688305847478782764824378746580378484444860001463974125576590593539394702731
233732086908305669696337452217679697268308540347205121089368266327179787261313416512123350053670
18211650961185566795515163026701267762066668711606656 7, 65537)
Private key: (1347675188730965761165380658282144586677430022550286002134386499672070754340034818
702649625028010298774661791543291423781474380811392410426039691220790249676605959354269568954260
565425825921694200586919011275890780027318667272228972545125654689361695966762892220693324748255
731776825240050574292847994518198694557349264764390660136582807012300946723010377494564039865280
914138561747447767757368830584747878276482437874658037848444486000146397412557659059353939470273
123373208690830566969633745221767969726830854034720512108936826632717978726131341651212335005367
01821165096118556679551516302670126776206666871160665 67, 12896036454351748749555902348726388675 7
559346635024621285462838631512844983637645277170553938052718108939019077855921173699746469665764
736024272022719103602497809678478275596729393837908084024609611986285539687410909643767002827861
445019554118336115536511590795789157994624962550803425972033607927394113584377184940885231029463
997035409418476446455779485611679158455430473020872281449985578630296347098013076487638606709360
903344331892585450208741451254511359933617623079569989624560721965345790022482883315827687806642
923020072502415861385770330023648776975088106110552024348131674880293895919665312701429951370281
53)
Original message: Alexandru Rudoi
Numeric representation: 339697835342360413257449652453928809
Encrypted message: 10406712394493435792582065614948004357825019850373683196440148158698218270990
546130752976644523283429718278033496141906695421580247062279030242514732938772473148898055965902
065044263589781784680479438813255543235137668110817064064310809840250131460369963105112859382785
534264916372599559582539448574444608250260977714300718597076417189953394901383360062294693668849
930298561259637400046258435055128389471457385513088601601583252597336817846164553285301848324043
937026272083708077205335087468161474846851276124232408883194153869460141438461046575304987482161 8
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 Algorithm Results

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

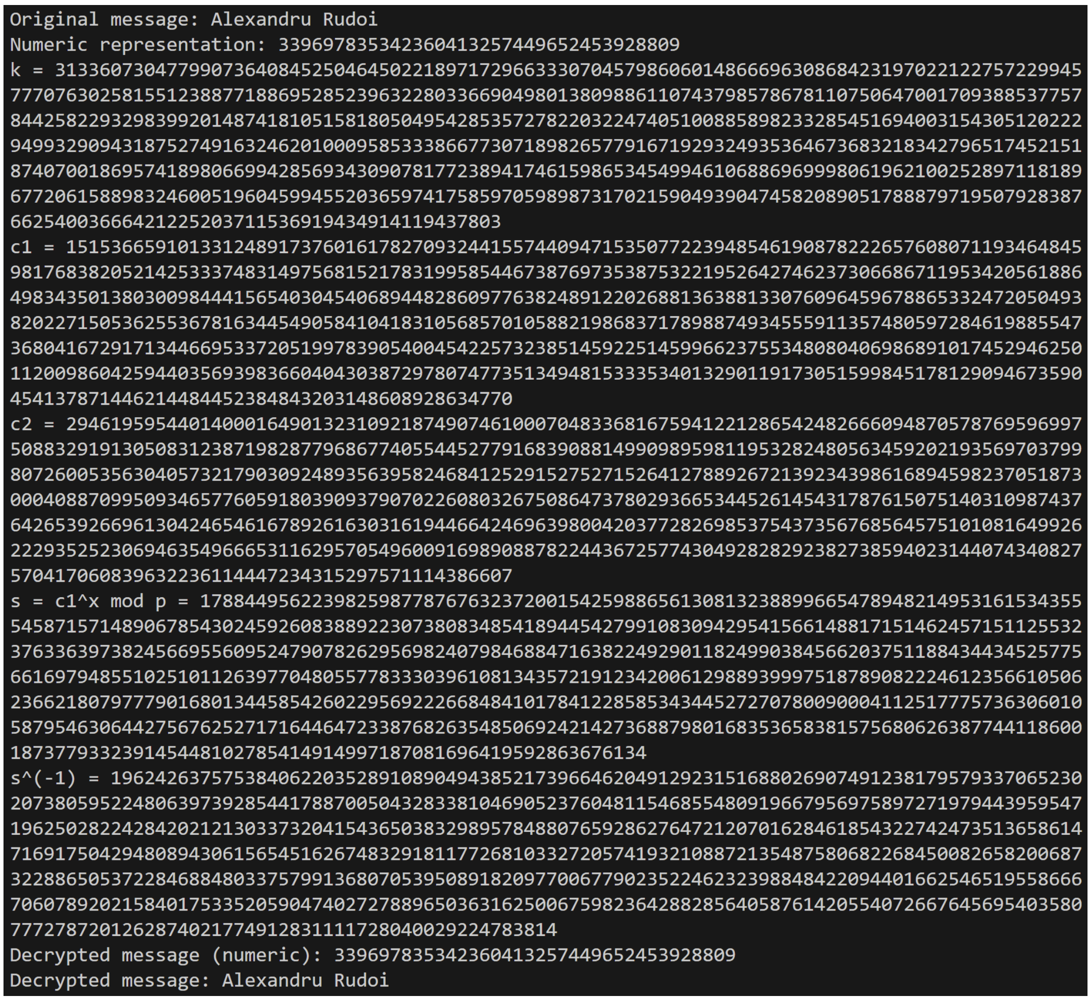


```
================================================================================
Task 2.2: ElGamal Algorithm
================================================================================
Generating ElGamal keys...
p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362
640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325
263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729
569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436
467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879
113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281
003994966956119696956248629032338072839127 03
g = 2
Private key x = 29836190566609854527519504493992667634125224935283623478676528058976474573598258
370773409425820797337823817328035485983430530487107671314263953189097194597399192830197714723588
364164363548437004309137973937256921726338170555385092592350413903477900154613515310673466020635
964583105434923526434928231277399400557896961850755578475792961550169791690038354794861593464930
455603959249419439408383857834711083759266010495298453429844392487812587717389732420901918177126
020558065879902579318565107797379374844330356406580990390954708963486245983884826032903350192530
56085071093947309642263368325878145089048765632028 0599
Public key y = 17501779792219345455937853255898359574464439393914128622398319145566290036516773 7
140275457294512729862451745171335294117851975540645664358409680828482056340160007039969557226675
628882260346601989430786023426009369893231643319472839660719458177295381663896038777111230789935
535469826204087978546311885944398171688560812218026770010041756268450956210182398623425933858978
253604330838131361971115433360221426870505025712785667353035252941615631849168513050215040803304
098039800547558381418177200579690957778943753282796189816286921766422141469649192526307976232158
34727589290544562986468579447844076827255934578884558 00
```

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



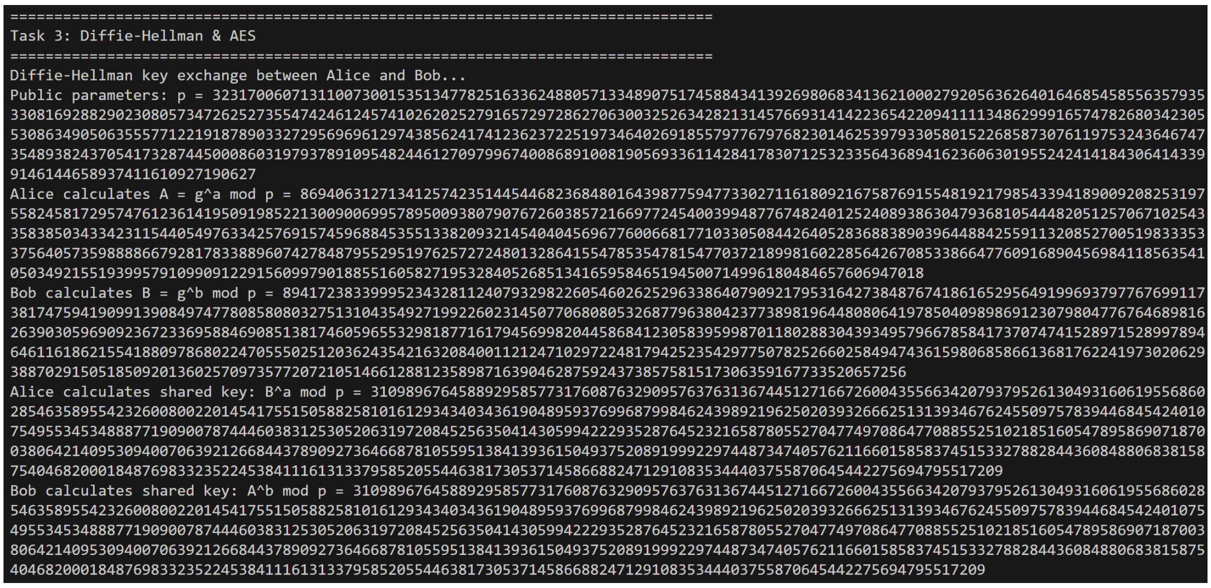
```
Original message: Alexandru Rudoi
Numeric representation: 339697835342360413257449652453928809
k = 31336073047799073640845250464502218971729663330704579860601486669630868423197022122757229945
7770763025815551238877188695285239632280336690498013809886110743798578678110750647001709388537757
8442582293298399201487418105158180504954285357278220322474051008858982332854516940031543051202222
9499329094318752749163246201000958533386677307189826577916719293249353646736832183427965174521511
8740700186957418980669942856934309078177238941746159865345499461068869699980619621002528971181899
6772061588983246005196045994552036597417585970598987317021590493904745820890517888797195079283877
66254003666421225203711536919434914119437803
c1 = 15153665910133124891737601617827093244155744094715350772239485461908782226576080711934648455
9817683820521425333748314975681521783199585446738769735387532219526427462373066867119534205618866
4983435013803009844415654030454068944828609776382489122026881363881330760964596788653324720504933
8202271505362553678163445490584104183105685701058821986837178988749345559113574805972846198855477
3680416729171344669533720519978390540045422573238514592251459966237553480804069868910174529462500
1120098604259440356939836604043038729780747735134948153335340132901191730515998451781290946735900
45413787144621448445238484320314860892863477O
c2 = 29461959544014000164901323109218749074610007048336816759412212865424826660948705787695969977
5088329191305083123871982877968677405544527791683908814990989598119532824805634592021935697037999
8072600535630405732179030924893563958246841252915275271526412788926721392343986168945982370518733
0004088709950934657760591803909379070226080326750864737802936653445261454317876150751403109874377
6426539266961304246546167892616303161944664246963980042037728269853754373567685645751010816499266
2229352523069463549666531162957054960091698908878224436725774304928282923827385940231440743408277
570417060839632236114447234315297571114386607
s = c1^x mod p = 17884495622398259877876763237200154259886561308132388996654789482149531615343555
5458715714890678543024592608388922307380834854189445427991083094295415661488171514624571511255322
3763363973824566955609524790782629569824079846884716382249290118249903845662037511884344345257755
6616979485510251011263977048055778333039610813435721912342006129889399975187890822246123566105066
2366218079777901680134458542602295692226684841017841228585343445272707800900041125177757363060100
5879546306442756762527171644647233876826354850692421427368879801683536583815756806263877441186000
187377933239145448102785414914997187081696419592863676134
s^(-1) = 196242637575384062203528910890494385217396646204912923151688026907491238179579337065230
2073805952248063973928544178870050432833810469052376048115468554809196679569758972719794439595477
19625028224284202121303337320415436503832989578488076592862764721207016284618543227424735136586144
7169175042948089430615654516267483291811772681033272057419321088721354875806822684500826582006877
3228865053722846884803375799136807053950891820977006779023522462323988484220944016625465195586666
7060789202158401753352059047402727889650363162500675982364288285640587614205540726676456954035800
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

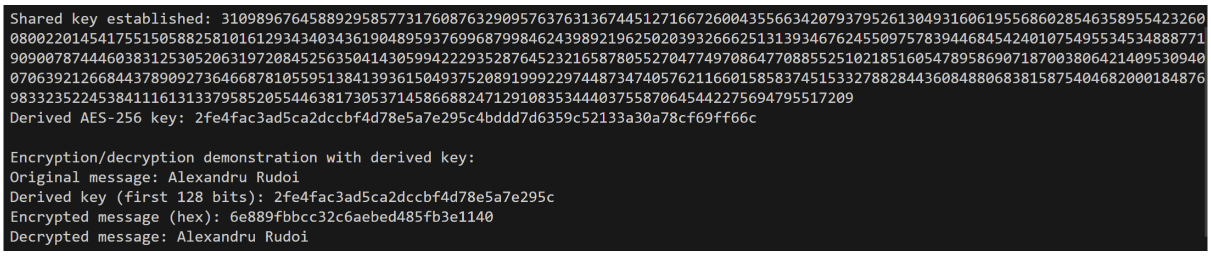


```
================================================================================
Task 3: Diffie-Hellman & AES
================================================================================
Diffie-Hellman key exchange between Alice and Bob...
Public parameters: p = 323170060713110073001535134778251633624880571334890751745884341392698068341362100027920563626401646854585563579353308169288290230805734726252735547424612457410262025279165729728627063003252634282131457669314142236542209411113486299916574782680342305530863490506355577122191878903327295696961297438562417412362372251973464026918557977679768230146253979330580152268587307611975324364674735489382437054173287445000860319793789109548244612709799674008689100819056933611428417830712532335643689416236063019552424141843064143399146144658937411610927190627
Alice calculates A = g^a mod p = 869406312713412574235144544682368480164398775947733027116180921675876915548192179854339418900920825319755824581729574761236141950919852213009006995789500938079076726038572166977245400399487767482401252408938630479368105444820512570671025433583850343342311544054976334257691574596884535513382093214540404569677600668177103305084426405283688389039644884255911320852700519833353375640573598888667928178338896074278487955295197625727248013286415547853547815477037218998160228564267085338664776091689045698411856354105034921551939957910990912291560997901885516058271953284052685134165958465194500714996180484657606947018
Bob calculates B = g^b mod p = 894172383399952343281124079329822605460262529633864079092179531642738487674186165295649199693797767699117381747594190991390849747780858080327513104354927199226023145077068080532687796380423773898196448080641978504098986912307980477676468981626390305969092367233695884690851381746059655329818771617945699820445868412305839599870118028830439349579667858417370747415289715289978946461161862155418809786802247055502512036243542163208400112124710297224817942523542977507825266025849474361598068586613681762241973020629388702915051850920136025709735772072105146612881235898716390462875924373857581517306359167733520657256
Alice calculates shared key: B^a mod p = 310989676458892958577317608763290957637631367445127166726004355663420793795261304931606195568602854635895542326008002201454175515058825810161293434034361904895937699687998462439892196250203932666251313934676245509757839446845424010754955345348887719090078744460383125305206319720845256350414305994222935287645232165878055270477497086477088552510218516054789586907187003806421409530940070639212668443789092736466878105595138413936150493752089199922974487347405762116601585837451533278828443608488068381587540468200018487698332352245384111613133795852055446381730537145866882471291083534440375587064544227569479551720
Bob calculates shared key: A^b mod p = 310989676458892958577317608763290957637631367445127166726004355663420793795261304931606195568602854635895542326008002201454175515058825810161293434034361904895937699687998462439892196250203932666251313934676245509757839446845424010754955345348887719090078744460383125305206319720845256350414305994222935287645232165878055270477497086477088552510218516054789586907187003806421409530940070639212668443789092736466878105595138413936150493752089199922974487347405762116601585837451533278828443608488068381587540468200018487698332352245384111613133795852055446381730537145866882471291083534440375587064544227569479551720
```

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: 310989676458892958577317608763290957637631367445127166726004355663420793795261304931606195568602854635895542326008002201454175515058825810161293434034361904895937699687998462439892196250203932666251313934676245509757839446845424010754955345348887719090078744460383125305206319720845256350414305994222935287645232165878055270477497086477088552510218516054789586907187003806421409530940070639212668443789092736466878105595138413936150493752089199922974487347405762116601585837451533278828443608488068381587540468200018487698332352245384111613133795852055446381730537145866882471291083534440375587064544227569479551720
Derived AES-256 key: 2fe4fac3ad5ca2dccbf4d78e5a7e295c4bddd7d6359c52133a30a78cf69ff66c

Encryption/decryption demonstration with derived key:
Original message: Alexandru Rudoi
Derived key (first 128 bits): 2fe4fac3ad5ca2dccbf4d78e5a7e295c
Encrypted message (hex): 6e889fbbcc32c6aebed485fb3e1140
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`

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.