Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5: Cryptography and Security

Public key cryptography

Elaborated:
st. gr. FAF-213 Afteni Maria

Verified:
asist. univ. Cătălin Mîțu

Chişinău - 2023

ALGORITHM ANALYSIS

Tasks:

Task 1. Study teaching materials recommended for the assignment placed on ELSE.

Task 2.1. Using the wolframalpha.com platform or the Wolfram app Mathematica, generate the keys and perform the encryption and decryption of the message

m = Name First name

applying the RSA algorithm. The value of n must be at least 2048 bits.

Task 2.2. Using the wolframalpha.com platform or the Wolfram app Mathematica, generate the keys and perform the encryption and decryption of the message

m = Name First name

applying the ElGamal algorithm (p and generator are given below).

Task 3. Using the wolframalpha.com platform or the Wolfram app Mathematica, perform the Diffie-Helman key exchange between Alice and Bob, which uses AES algorithm with 256-bit key. The secret numbers a and b must be chosen randomly according to algorithm requirements (p and generator are given below).

Note:

For tasks 2.1 and 2.2 use the decimal numerical representation of a the message, reaching it through the hexadecimal representation of the characters, in according to ASCII encoding. For convenience in conversion you can use the page https://www.rapidtables.com/convert/number/hex-todecimal.html.

For tasks 2.2 and 3 considered

p=32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039,

which has 2048 bits and the generator g=2.

IMPLEMENTATION

RSA Algorithm:

First, I pick two big prime numbers, p1 and p2, that are crucial for the security of the RSA algorithm. To create the modulus n, I multiply p1 and p2 together. The length of n in decimal is printed out for reference.

I calculate Euler's totient function, PhiN, by multiplying (p1 - 1) and (p2 - 1).

```
PhiN = (p1 - 1) * (p2 - 1)
```

Now, I need a random number e that's coprime with PhiN. I keep trying different values until I find one that works (i.e., gcd(e, PhiN) equals 1).

```
while True:
    e = random.randint(1, PhiN - 1)
    gcdEPhiN = math.gcd(e, PhiN)
    # Check if e is coprime with PhiN
```

```
        if gcdEPhiN == 1:
            break
```

The private exponent d is calculated using modular arithmetic. It's the inverse of e modulo PhiN.

```
    d = pow(e, -1, PhiN)
```

I define two functions for encryption and decryption. The rsa_encrypt function uses modular exponentiation to encrypt a decimal message, and rsa_decrypt does the same for decryption.

```
    def rsa_encrypt(decimal, n, e):
        encrypted = pow(decimal, e, n)
        return encrypted

    def rsa_decrypt(message, n, d):
        decrypted = pow(message, d, n)
        return decrypted
```

Results:

Enter an ASCII string: Afteni Maria
n length in decimal: 617
Decimal message: 20240385737577211699886451041
Encrypted message:
2580894116495674270674563326427525695848565159503853532338178099731351482938867189252377595187086410799052503564584260438035840312452392979580645579059540679594885795453252566599082384784597129509973492823474850624900130575609723901482651839867621200483174974007092645451637864238174308315210130160578439806582536607019283433868884641791691703772611553562253894696574024635761044971973261211772036278765520065208818996642461274563230604648644097887320929971243680390150996685969011505222602246830075518196113362252930259546454341834961501568984748341135434652036136721876754480822619234123199581930400311168944181329
Decrypted decimal message: 20240385737577211699886451041
Decrypted ASCII message: Afteni Maria

ElGaman Algorithm:

For the ElGaman implementation, I firstly created the mod_exp function that uses a binary exponentiation algorithm to iteratively square the base and reduce the exponent until the exponent becomes zero.I need a fast way to calculate modular exponentiation so this function is crucial for the efficiency of the ElGamal encryption algorithm.

```
    def mod_exp(base, exponent, modulus):
        result = 1
        base = base % modulus
        while exponent > 0:
            if exponent % 2 == 1:
                result = (result * base) % modulus
            exponent = exponent // 2
            base = (base * base) % modulus
        return result
```

To encrypt I randomly generate a number between 2 and p - 2. Then I calculate c1 and c2. c1 is calculated as g^k mod p, where g is a generator, and p is a prime number. c2 is calculated as (plaintext * public_key^k) mod p.

```
def el_encrypt(p, g, public_key, plaintext):
    k = random.randint(2, p - 2)
    c1 = mod_exp(g, k, p)
    c2 = (plaintext * mod_exp(public_key, k, p)) % p
    return c1, c2
```

When I receive a ciphertext, I use my private key to calculate s as c1^private_key mod p. I find the modular multiplicative inverse of s and then compute the decrypted message using (c2 * s_inverse) mod p.

```
def el_decrypt(p, private_key, c1, c2):
    s = mod_exp(c1, private_key, p)
    s_inverse = pow(s, -1, p)
    decrypted_message = (c2 * s_inverse) % p
    return decrypted_message
```

To encrypt and decrypt a message I generate a private and public keys using random numbers. When a message is sent, it is encrypted using the ElGamal encryption function, producing r and t. The recipient (in this case, myself) decrypts the message using their private key, resulting in the original message.

Results:

Decimal message: 2024038573757721169988645104‍1

Encrypted message:

1511502565238688563478860993404178533264617637305599076274023374045056687171496063608127329187986833111640951131436044038135244073459836094217734000697443188682952841361519461149705355690616987727965203228326022415080611377419118873848650119791564684294002775447156804318265661217792456356117998844378656679782770586316125167701753394401293799063468701627754841318889712428263416273481594147847711501635873045146705571240423171335635174795526008697998371208244889917255678265260589595208647150924947229311022391181855273637936807006754794501032414614448646666408075818542956039840642647933515884887231159892709907054‍1

3015656260795468067224573033857845495119859505585135950597840788066738617957419473653852813364777357842893252170514196446726945534182285836485733582626398838482938067013903470068614113585832110845435538016203347806966414754742030966359641264347106784752298414789231153134991923446373976881724410799537095657364986490663663371960533021184773757088428165189834638919671157929241356480678713006643374644358412679331999120383884763290375495611916515882873210729442549426338558225064371310296866236450171494490165098369594862097536018068351636712349031227319012101596839659226032972986226955769585430462925052890833328976

Decrypted decimal message:

164194747534075861827566295558737566437646208818745904897640046748064154420871201535762342222524932449039708577644013961533507405986029156008564492615258597740060638093441419391622125174697527551807300316164701235607328115477129453983607043519508831018603654761389736000486895255567255524526850545439098598019260041872354048431119667872719286460061379590825435639438163799326387865529657252979937468049199641358252401157208260014871140796573287

48354730809424944543106148155016532994513306396257731031162635492220716342
21972233540739826664048444196547514515727853104942338398282211678964378052
534883258069485751834269

Diffie-Hellman Algorithm:
For the Diffie-Hellman algorithm I'm randomly generating private keys for two parties engaged in secure communication. Using a base g and a prime number p, I compute their corresponding public keys.

```
private1 = random.randint(2, p - 2)
private2 = random.randint(2, p - 2)

public1 = pow(g, private1, p)
public2 = pow(g, private2, p)
```

Each party raises the other party's public key to the power of their own private key, modulo p. This results in two shared secrets which are now identical for both parties.

```
shared1 = pow(public2, private1, p)
shared2 = pow(public1, private2, p)
```

o ensure a consistent size for the shared secret, I'm converting it to 256 bits

Results:
Shared 1 hex: 4fec4a360e80ea1541a91ad4416988c8e0cb10b207ee8816e662314fc59f4dc2
Shared 1 decimal:
36150203131959930297254650587734923740244155282224872222652933735074873953
730
Shared 2 hex: 4fec4a360e80ea1541a91ad4416988c8e0cb10b207ee8816e662314fc59f4dc2
Shared 2 decimal:
36150203131959930297254650587734923740244155282224872222652933735074873953
730

CONCLUSIONS

In this laboratory, I explored the fascinating world of RSA, ElGamal, and Diffie-Hellman algorithms. I gained insights into RSA's secure prime number selection, ElGamal's key generation, and the art of securely sharing secrets through Diffie-Hellman. These experiences have provided a solid foundation in understanding how these cryptographic techniques contribute to secure communication and data protection.