

Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automatics

# Report

Laboratory Work Nr.5

# Cryptography and Security

Performed by:

Felicia Lupascu, st. gr. FAF-212

Verified by: Cǎtǎlin MÎŢU

Chișinău – 2023

**Topic:** Public key cryptography

**Tasks:**

1. Study the recommended educational materials related to the topic posted on ELSE.
   1. Using the [wolframalpha.com](https://www.wolframalpha.com/) platform or the Wolfram Mathematica application, generate the keys and perform the encryption and decryption of the message:

using the RSA algorithm. The value of must be at least bits.

* 1. Using the [wolframalpha.com](https://www.wolframalpha.com/) platform or the Wolfram Mathematica application, generate the keys and perform the encryption and decryption of the message:

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

1. Using the [wolframalpha.com](https://www.wolframalpha.com/) platform or the Wolfram Mathematica application, perform the Diffie-Hellman key exchange between Alice and Bob, utilizing the AES algorithm with a 256-bit key. The secret numbers and must be randomly chosen in accordance with the algorithm's requirements (the prime number and the generator are provided below).

**Note**:

For tasks **2.1** and **2.2**, use the decimal numeric representation of the message, reaching it through the hexadecimal representation of characters, following the ASCII encoding. For convenience in conversion, you can use the page

<https://www.rapidtables.com/convert/number/hex-to-decimal.html>.

For tasks **2.2** and **3**, please consider:

p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039

the 2048-bit prime number and the generator .

**Theory:**

Useful functions in Wolfram:

* **Prime[n]** – returns the -th prime number from the list of prime numbers ( is bounded).
* **RandomPrime[{imin, imax}]** – returns a pseudo-random prime number between imin and imax.
* **RandomInteger[imax]** – returns a pseudo-random integer between and imax.
* **Mod[a, n]** – returns the remainder of the division of by .
* **PowerMod[a, b, n]** – returns the remainder of the division of by .
* **FactorInteger[n]** – returns the list of prime factors of n along with their exponents.
* **IntegerDigits[n, b]** – returns the list of digits in base of the integer number .
* **Length[lst]** – returns the length of the list lst.

In RSA, *p* and *q* are the two prime numbers used in the key generation process to create the public and private keys. These primes are an essential part of the RSA algorithm, and they are typically kept secret.

Here's a brief overview of how RSA key generation works:

1. **Key Generation:**
   * Two large prime numbers, *p* and *q*, are chosen independently.
2. **Modulus Calculation:**
   * The modulus *n* is calculated as the product of *p* and ().
3. **Euler's Totient Function:**
   * The totient function (*ϕ*(*n*)) is calculated as *ϕ*(*n*)=(*p*−1)×(*q*−1). This function is important for choosing the public exponent (*e*) and calculating the private exponent (*d*).
4. **Public Key (*e*, *n*):**
   * The public key consists of the modulus (*n*) and the public exponent (*e*). The public exponent is typically a small, fixed value, often 65537.
5. **Private Key (*d, n*):**
   * The private key consists of the private exponent (*d*), which is calculated as the modular multiplicative inverse of *e* modulo *ϕ*(*n*).

In summary, while the public key (*e*, *n*) is shared openly and can be used for encryption, the private key (*d*) must be kept secret. The security of RSA relies on the difficulty of factoring the modulus *n* into its prime factors *p* and *q*, given that *n* is publicly known.

Security of the RSA system is not perfect!!!

It rests on the difficulty of factoring .

Recall that the encryption key is public. Therefore, integers e and n are known. But in order to find d, someone needs to factor to get and .

There are different factoring algorithms. Based on them here are a few recommendations for increased security:

1. and should be of slightly different sizes and d should be large in order to guard against any particular attack.
2. Choose p such that most of the digits are not predictable. If we choose our p by testing

numbers for primality of the form for a random 50-digit number N and

then the attacker can easily compute of the last

1. If is large enough, then is computed with greater difficulty.

**Implementation:**

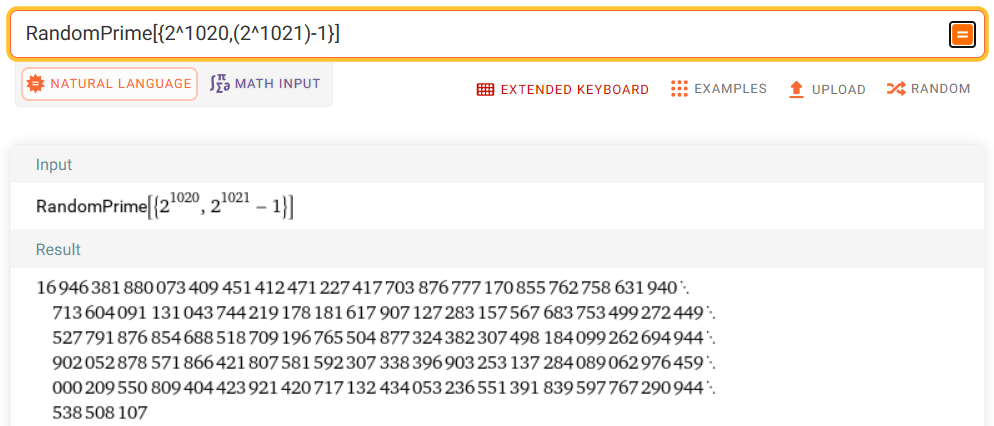
A 4096-bit RSA key provides a higher level of security compared to RSA 2048. They are recommended for applications that require an extra layer of security, especially for long-term data protection. RSA 4096 offers increased resistance to attacks, particularly those based on advances in computational power.

RSA 4096 keys require more computational overhead for encryption, decryption, and key exchange compared to RSA 2048. Taking this into consideration I will do the 2048-bit one.

**2.1 RSA**

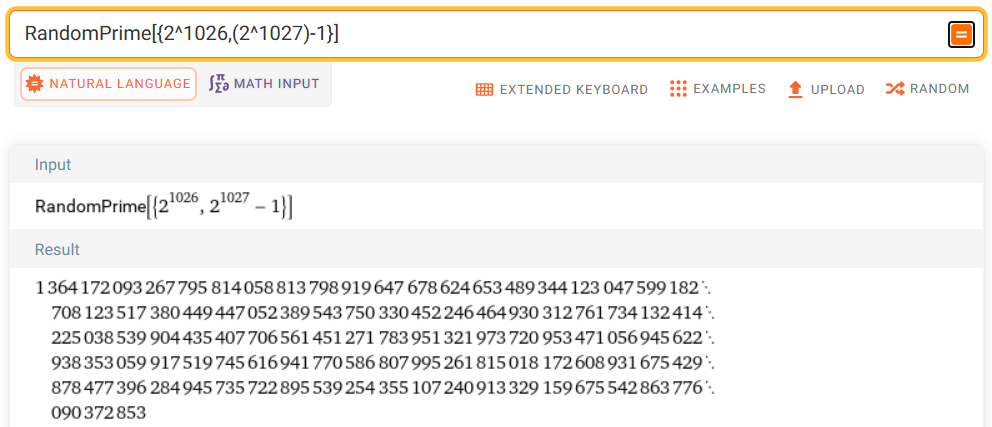
Using [wolframalpha.com](https://www.wolframalpha.com/) ’s function **RandomPrime[{imin, imax}]** (or **Prime[n]**)

Let



**Figure 1.** *Generating p using RandomPrime function*

p = 16946381880073409451412471227417703876777170855762758631940713604091131043744219178181617907127283157567683753499272449527791876854688518709196765504877324382307498184099262694944902052878571866421807581592307338396903253137284089062976459000209550809404423921420717132434053236551391839597767290944538508107



**Figure 2.** *Generating q using RandomPrime function*

1364172093267795814058813798919647678624653489344123047599182708123517380449447052389543750330452246464930312761734132414225038539904435407706561451271783951321973720953471056945622938353059917519745616941770586807995261815018172608931675429878477396284945735722895539254355107240913329159675542863776090372853

Then

**n = p q =**

23117781242655188095905122840715182750678296330747746970594340554239631372432753686826605557190247206738728930282361613235742156016716691686835356107510403169346768285892424748684120322800628901533382599983580111006824136756640861766966757014692323101601279493187166922840957351596569495338673256377073781041317256670690501971080988484140356556579505276203573455470883304832804431862903886080918476516351089257823024829122033222008894029658886194873087029042079984935328081488540133968599472920537580403156852458564976313930401667047357551405470718127957961126973306071032338727611650754441296368826111641777993219271

The number has 2048 bits.

m = Lupascu Felicia

23117781242655188095905122840715182750678296330747746970594340554239631372432753686826605557190247206738728930282361613235742156016716691686835356107510403169346768285892424748684120322800628901533382599983580111006824136756640861766966757014692323101601279493187166922840957351596569495338673256377073781039936138195542632747570762213993291174078074616003687649239759883105195920369712614513193108278771559635325028313888628358256063612899762268457328812265418709231046862350969814328031632514599091016989427935202082167538236598891900853410818829249271014032623146426716082340822490276976575369552801487057364338312

How to choose and ?

Integers and are closely related to numbers and .

Choose to be any large random integer that is relatively prime to .

Then will be the multiplicative inverse of modulo

.

**RandomInteger[imax]** function we can select the e

Select public exponent such that

a=e

b=-1

n=phi

Getting we can use the **PowerMod[a, b, n]** function

d =

15353533826202141035786904752505745523994438654380371921847037047222701605452004775513295290938900162053647793146925730163097381369052759851094735949700135928543989447361851772452584359525461549478035476858649433322901984779125534419778065303897005589518856389832684246815928554642297997377223433435593838496486875369626144513340021608042357624592554980181828716920362370447789151624458111590418286325919875866871495222398285516905769638754826319435947539293294089414995280996815724528768555729289409579405249558380850945637873051915236836375777047498417232353997819115449962616028193414341248753149445924068218566449

Using <https://www.rapidtables.com/convert/number/hex-to-decimal.html>

Getting a hexadecimal representation of the message:

Then decimal:

(4C7570617363752046656C69636961)₁₆ = (4 × 16²⁹) + (12 × 16²⁸) + (7 × 16²⁷) + (5 × 16²⁶) + (7 × 16²⁵) + (0 × 16²⁴) + (6 × 16²³) + (1 × 16²²) + (7 × 16²¹) + (3 × 16²⁰) + (6 × 16¹⁹) + (3 × 16¹⁸) + (7 × 16¹⁷) + (5 × 16¹⁶) + (2 × 16¹⁵) + (0 × 16¹⁴) + (4 × 16¹³) + (6 × 16¹²) + (6 × 16¹¹) + (5 × 16¹⁰) + (6 × 16⁹) + (12 × 16⁸) + (6 × 16⁷) + (9 × 16⁶) + (6 × 16⁵) + (3 × 16⁴) + (6 × 16³) + (9 × 16²) + (6 × 16¹) + (1 × 16⁰) = (396996506886017839616402540514732385)₁₀

The encryption key is a pair of integers and

The decryption key is a pair .

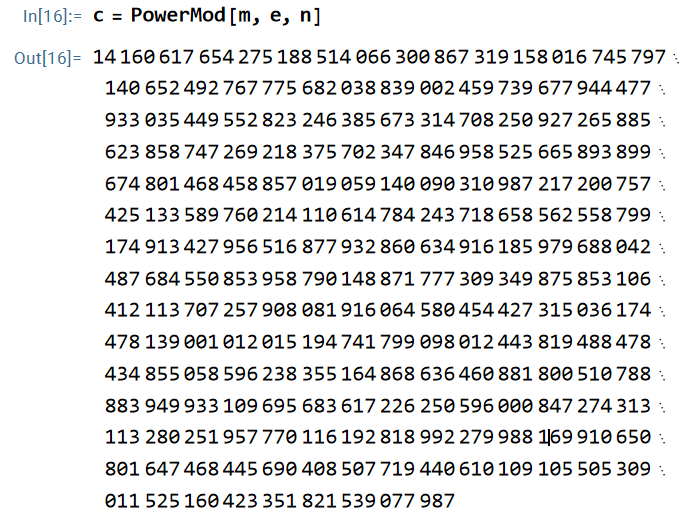
Given a message , in order to encrypt it, we would represent it as a number between and . If the message is too large, break it in blocks, as long as every block is between and .

Then

and

The **encryption key** is a pair of integers

(65537, 23117781242655188095905122840715182750678296330747746970594340554239631372432753686826605557190247206738728930282361613235742156016716691686835356107510403169346768285892424748684120322800628901533382599983580111006824136756640861766966757014692323101601279493187166922840957351596569495338673256377073781041317256670690501971080988484140356556579505276203573455470883304832804431862903886080918476516351089257823024829122033222008894029658886194873087029042079984935328081488540133968599472920537580403156852458564976313930401667047357551405470718127957961126973306071032338727611650754441296368826111641777993219271)



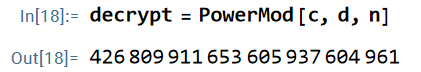
**Figure 3.** *Encryption. Ciphertext*

**Encryption**

c =

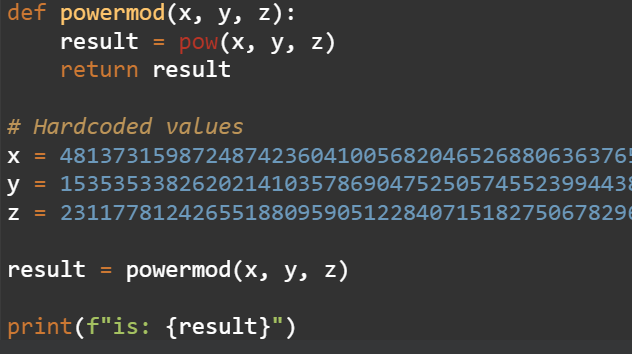
4813731598724874236041005682046526880636376527702292233863347999805196906596426671663679527309581190810826590790201928937734018739138965336068623856700669815309080584501584178235772064558541294862900246419804464379625607276557990626078300472122089064173435818189003301536865461167097001173246307612242321356995858289322788301165832421968156107273324670241781070269234109906088973213450260504726024562763652664906080234980076576283230244968475195854892066615104874769398938287171413468083340507800904052383188993664360152332917159033966906631085001673679747675072269060536086812051590558639046048971822645729619433905

**Decryption**

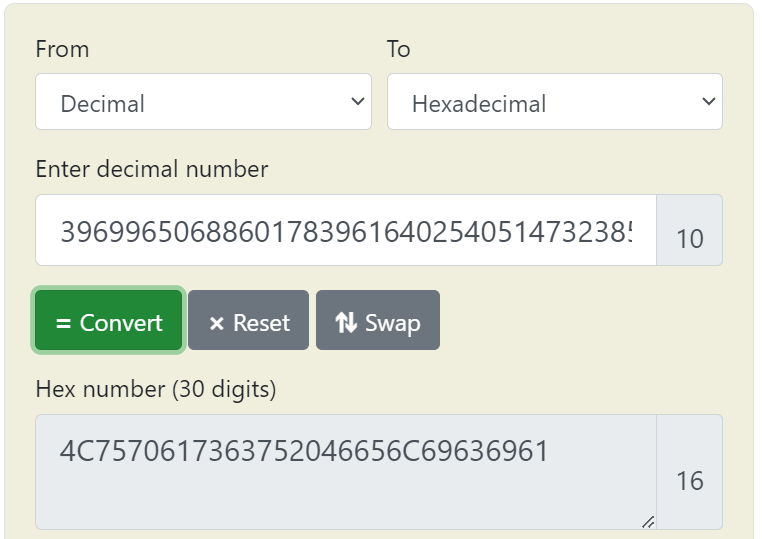
****

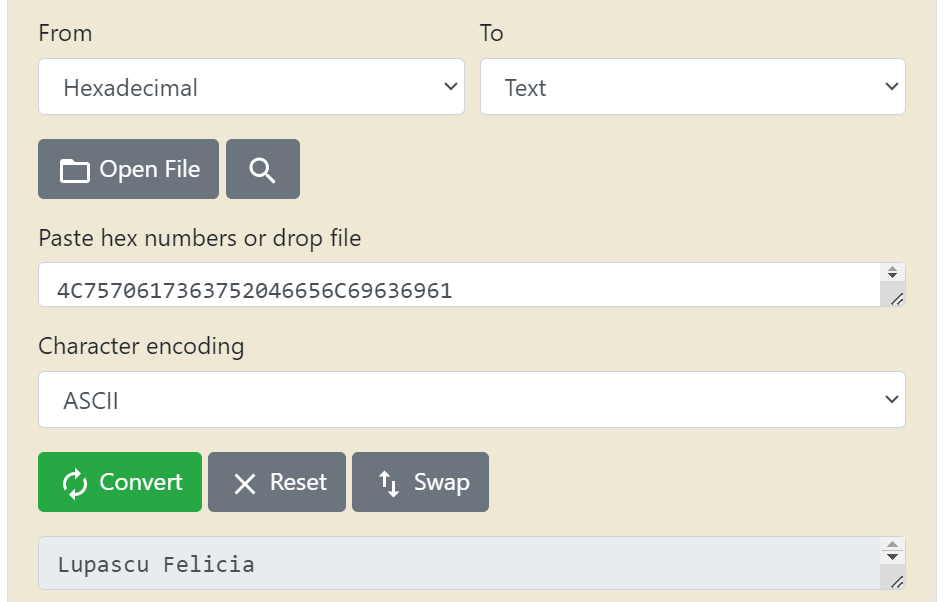
**Figure 4.** *Decryption. Message*

We found c,n, and d previously. I used this python script for POWERMOD()



m =396996506886017839616402540514732385

**  
Figure 5.** *Convert from decimal to hexadecimal*

****

**Figure 6.** *Convert from hexadecimal to text*

We got the initial message

**2.2. ElGamal algorithm**

We already have a prime number *p* and a generator *g*, you can proceed with the ElGamal key generation, encryption, and decryption algorithm.

Key Generation:

1. Choose a large prime number *p*.
2. Choose a primitive root modulo *p*, denoted as *g*.
3. Select a private key *x* randomly from the range [1, *p*−2].
4. Compute the public key

Encryption:

1. Choose a random integer *k* from the range [1, *p*−2].
2. Compute .
3. Compute , where is the plaintext message.

Decryption:

1. Compute .
2. Compute the modular multiplicative inverse mod .
3. Compute mod.

p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039

the 2048-bit prime number and the generator .

**Key generation**

Select private key an integer randomly chosen such that

x=RandomInteger[p-2]

x=

15080660497767353311199391654257060851133604176491794617723597373154574152972282841715079495968419359450113658704681173560751987131616366418305225407453855773844662183486380869985586345226612239125890868989999561492822953278117734677007914976120441264221498530340423521874664941192611647333053801433507149253367653843560284956058598974652833942590011749900485428759735791152649447129054568470028677677647601150802020973899613756566112861992439534625232209748216361849570088089244842547626251566370292742203298913208342142588849836794385272887664832460507417142508157005351172629794181388074399089480150590024438537471

Public key

y=

19581015610963267939039072167706628253351854539667096574011844206462358016847362949502873715672714352026223086529386956388021614342078079178912557476715549171976849314961601519569445088231871406390992563564135197608357305608873941126931768765013273348455197234393236174796949768981458992044848418592639513137265286600410137823966097146892352016779054547214707869376779044298385644792566900333877223914403588899632680877973319201330363285132135867355930881797672524431241701690242680048488683233856404009605249389961191673393446199910739795009926518277529417931771966920585525284377560586754386595206170676388400134267

**Encryption**

k =

404765664390454018777994344431476693571285200699221492029204504258763076934921073412738470822445800525077465761008353473603095374674726978971619469587102942356211872795937369863865436695079170552280637628111402666772877861788413631830766676791215451521076886169611378657397391774220840362234387554280150301127585379453855406113591220948581713479685153274725099244765637820095722812596587727839985172670646249139617191400610998177593022271032005987467209781417544811570376410709058923725787333642349214013586929270782893731881573691489473076104007343817019519824988464018085319391700853583127493283809543314747641071

a =

24206671740990526354756577550040682635148620021640447612794398068963323305302599136660934936297824889985791802549517946789873218582946858009510187995341781679431088555749478492252335852061534725194539630095725675650346515034624656721907634884526057017807709134626756991420401249873026122303192723253062025890833003275059283702701585384622744414024854360016158530523837865151850530068190946857200978579190395410737672582583597401182767107834242392325512958780929032418127557589299158757254282111981935928743157450194155136544127385093768707082749278416579152878587628955209263338169320451120524189951031474555058996396

to power , i get the overflowing error so used this rule

b =

28098186326555927608674987550111432893551352048247867335288523032128035267446389048926684212809862866653953098913347669812934031633837418097616472780224536564076329668948986270783564489575473562449784980661033556599918578633249808120823329313432801740719894841224263407887700958804753574221804404110693951484752979457585061740544006217270467340127692710799201526473300089285506024410218163321581199059591676102006909812723987949312376972253861359158993323279575278017243418053666943392758044538584021759764633891973743255866105558086858414051343336377262692049030581630107633747451244517302092860927340843060207569962

Ciphertext is the pair of

**Decryption**

s =

9884105298994022956540139906345924592600193716370235401337999206095865500431326709249297480654707370713330289239453304594373209087997247017014247654424160458828490727743462848227980448731028056110398484297392962015662093902808774060581623846424667815943956095180318929422191472574100876069899408476498256044920031202310345098286913515067294892810166702520810387925988035297575045503740974527968089841727151868302131548992170078583618237185645990404858326858932477072289318838586607396652812762500019198507872901713995446085852559063710621998262001404305592534003499814678483257832644298063262572836294707977852538580

Modular multiplicative inverse =

8991929139061903423660745483687582366381412891882979118704107162676175270162628053299201739834009837566752467895477007321569498011690161579956979508468299918545254102819817611085645760905128346899317648876999805131565297890345561247211261216479892497466243356871071426277088508752061628663711461596410636881791190599555535611130249140466542668702560051209404440708511395042275563099388002336340030566484437030548046157053796821218216636765365273902616936337664789522453113621246941896862181086314667140877441579161922292442296062633915861267364003365974249090081121635805680459726836796373478442385168412674548984122

y1=Mod[b,p]

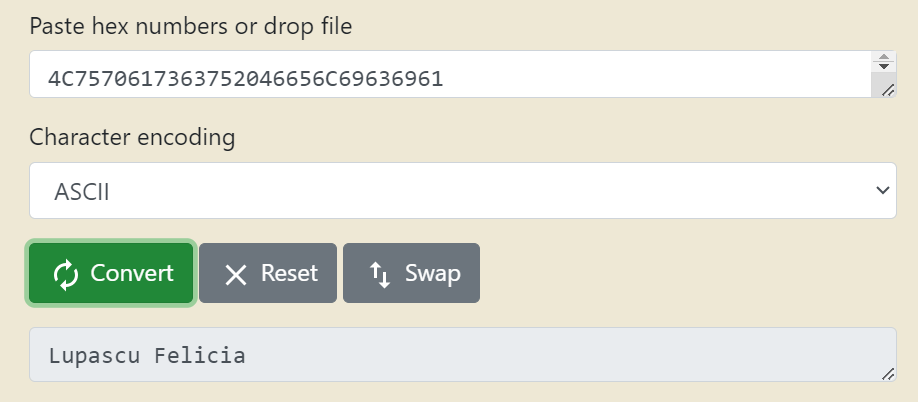
y2=PowerMod[s,-1,p]

decryption=Mod[y1\*y2,p]

**Figure 7.** *Decryption. Message*

m =396996506886017839616402540514732385 (hexadecimal)

(hexadecimal-decimal-text)



**3. Diffie-Hellman**

The Diffie-Hellman key exchange algorithm is a cryptographic protocol that allows two parties to establish a shared secret key over an insecure communication channel. This shared key can then be used for secure communication, such as encrypting and decrypting messages.

Here's a simplified explanation of how the Diffie-Hellman algorithm works:

1. **Key Generation:**
   * Both parties, let's call them Alice and Bob, publicly agree on two things: a large prime number and a primitive root modulo , often denoted as .
   * Each party generates a private key. Let's say Alice's private key is and Bob's private key is .
2. **Public Key Calculation:**
   * Both parties independently calculate their public keys using the agreed-upon and .
   * Alice calculates and Bob calculates .
3. **Exchange Public Keys:**
   * Alice sends her public key to Bob, and Bob sends his public key to Alice. This exchange can occur over the insecure communication channel.
4. **Shared Secret Calculation:**
   * Both parties use the received public key and their own private key to compute the same shared secret.
   * Alice computes the shared secret , and Bob computes the shared secret using.
5. **Result:**
   * Both Alice and Bob now have the same shared secret , which can be used as a symmetric key for secure communication.

The security of the Diffie-Hellman key exchange relies on the difficulty of the discrete logarithm problem, making it computationally infeasible for an eavesdropper to determine the shared secret even if they know the public keys exchanged.

p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039

the 2048-bit prime number and the generator .

**Key Generation**

a=

32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039

b=

8036483926844834707462088463479860202871243865298716411387025881508822613518568484339890941484998741272156677108601719596136685667703157193104189371957786520323230013289734417776735918898510769066951734474225219155574145361432136856360765921656396823140661497671938939014613338412535771052775673677554381473043341662428460059222321998492383592317668222066079328337326979950495419197723533852762501858014105907842118839520101799662452870540526536045047386543151794543770450050741035966604256703292042709890919215371101610221826765689392776929257634872532271146995769033168908148049184082725101295002474231431475807445

**Public Key Calculation**

A=

31508297016437848017711058649769152741823482141000701690666987440758333733647078194169085314379454461804240883290357496769749792713730717588255454925661082778967143861901399260787111767858907062636765614372643690660725902424671496262426417921472923725850771337357557001304444959620482832647308921756141816444668490139940054344057399101197442091362801917576259794811657329090537957511388250351289818782096583965440556552924316155462477430613672985897449683357718068308003529377268886723580862775284738178211216744443531177687077118426128506452264784711523218396158288245720396910190057781932254098065946898871240286308

B=

9966234991605288302193552158637958180764366087107212328746167821901316226266156139034941051002947923187769037114958272362434918465939974305681606721957175627148336857211837159647454144871258577933754871697820981036065807294296574499234056673036779884727724487280350688592702595902612631847564243640729892374667725948078686609849673814264396796083740035187425922045717424976558665077666481536094195731006979879840632832859436641614557393656789268520955747706914062961022055854939731514068628963716828025054403857382284475377136606943927174990801440865778225660880332320689644652603011588812046324505102247197270225835

**Exchange Public Keys**

**Generated Secret Key**

Shared Secret

ka=

18126617209588703859779312305374451481726937067885313969869843102515789949808389444476182770153006900232459319560394351366094737936246701653205601802621590052082531718529338868822267767110527686819150180077021913328355408940621274481895358953321307160205520240630254680186941233505735252183821667669740794222118179753729454747698894358285806955450537823267517348250985066910862490616325167148859998848821530259404204408330971955914620338956274964209726190371467175895152771950283911893532806779244618055245536484359306669323923469108355550763599087930126897774668265448697385151166383400499221215953299227281971932306

kb=

18126617209588703859779312305374451481726937067885313969869843102515789949808389444476182770153006900232459319560394351366094737936246701653205601802621590052082531718529338868822267767110527686819150180077021913328355408940621274481895358953321307160205520240630254680186941233505735252183821667669740794222118179753729454747698894358285806955450537823267517348250985066910862490616325167148859998848821530259404204408330971955914620338956274964209726190371467175895152771950283911893532806779244618055245536484359306669323923469108355550763599087930126897774668265448697385151166383400499221215953299227281971932306

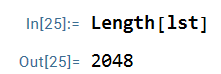
AES algorithm with a 256-bit key.

In order to get the AES key we need to convert to hexadecimal, we can use this function:

**IntegerDigits[n, b]**

We get

Getting the lenth of list we got



To be human-readable

{8,15,9,7,2,8,13,14,8,11,0,7,10,0,2,15,13,12,15,0,4,8,14,8,10,12,10,8,13,9,11,1,15,4,6,0,0,13,8,11,15,14,13,3,9,2,12,6,0,10,8,9,3,6,10,13,11,13,15,3,11,9,8,4,8,12,8,15,12,8,1,9,2,4,14,4,2,8,13,12,15,15,11,14,9,12,0,8,8,4,0,0,3,10,3,8,12,11,9,5,5,5,0,12,4,6,10,11,2,3,0,1,8,10,11,6,3,0,2,1,5,15,4,1,9,15,12,9,7,3,15,1,15,10,3,2,2,8,3,10,4,3,14,15,8,8,9,3,6,8,9,8,8,0,0,6,10,10,2,9,9,12,11,5,13,15,0,13,8,6,11,7,5,8,11,4,5,10,0,10,1,3,2,13,3,11,1,6,15,0,3,7,13,7,2,1,8,6,12,15,14,5,14,14,6,2,13,14,11,8,1,3,1,13,11,1,4,0,7,10,1,10,11,6,0,12,3,2,0,3,5,9,7,1,0,1,7,8,4,4,4,11,7,14,10,4,13,0,7,15,5,6,12,13,0,13,1,14,6,7,2,10,7,11,11,3,3,11,9,10,14,6,10,5,4,12,4,2,6,2,1,8,14,14,4,1,5,13,12,9,10,15,6,2,10,12,8,8,4,14,4,14,8,2,9,11,6,8,1,8,2,8,15,7,10,15,10,7,5,11,13,15,14,2,14,11,4,3,13,3,12,14,14,6,1,4,1,0,7,10,2,11,9,0,7,7,15,1,6,7,11,5,13,12,8,2,14,3,12,15,6,7,9,11,13,0,7,13,15,11,3,6,7,13,13,8,6,15,11,12,12,4,11,0,3,0,9,14,1,6,8,10,4,6,13,9,3,14,7,2,15,0,0,10,5,0,3,12,2,13,0,6,2,10,9,6,0,7,15,0,8,15,12,15,10,7,2,10,7,5,15,14,11,10,4,15,3,13,12,15,1,12,7,0,1,11,12,12,14,14,5,7,4,14,8,8,10,8,14,13,14,4,5,0,11,14,5,12,1,7,13,13,4,8,9,1,1,10,5,10,13,1,2,6,15,15,14,14,15,13,7,6,0,8,8,5,6,12,10,8,10,5,4,14,15,9,10,14,14,12,9,2}

The length would be 512

Derived AES Key (256 bits): abf8827dc02df24da27dae8782323696a622ddf780d0e1bff47434bbf37105ee

**Conclusion:**

In summary, the utilization of Wolfram Mathematica to implement Public Key Cryptography algorithms, such as RSA, ElGamal, and Diffie-Hellman, has yielded valuable insights into the practical aspects of securing communication and encrypting data. This laboratory work has enhanced our comprehension of the mathematical foundations and computational processes inherent in these cryptographic methods.

Wolfram Mathematica demonstrated its versatility as an effective tool for algorithm implementation, leveraging its symbolic computation capabilities, extensive mathematical functions, and visualization tools to express intricate mathematical operations clearly and succinctly. The interactive features of Mathematica further facilitated a thorough exploration of the algorithms, enabling experimentation with diverse parameters and providing a deeper understanding of their behavior.