

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.6

# Cryptography and Security

Performed by:

Felicia Lupascu, st. gr. FAF-212

Verified by: Cǎtǎlin MÎŢU

Chișinău – 2023

**Subject:** Hash functions and digital signatures

**Tasks:**

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

Task 2. Using the wolframalpha.com platform or the Wolfram app Mathematica, generate keys, perform signing and digital signature validation a to the message m that you obtained by completing laboratory work no. 2. The signing will be done by applying the RSA signature. The value of n must be of at least 3072 bits. The hash algorithm will be selected from the list below accordingly with the formula i = (k mod 24) +1, where k is the student's order number in the list group, i is the index of the hash function in the list (SHA3\_512)

Task 3. Using the wolframalpha.com platform or the Wolfram app Mathematica, perform signing and digital signature validation of message m on which you obtained by completing laboratory work no. 2. The signature will be achieved by applying the ElGamal signature (p and generator are given lower). The hash algorithm will be selected from the list below according to formula i = (k mod 24) +1, where k is the student's order number in the list group, i is the index of the hash function in the list (SHA3\_384)

**Note**:

For tasks **2.2** and **2.3**, 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**

P and Q = prime numbers

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

m = „Lupascu Felicia”  
# Function to choose a public exponent for RSA  
def choose\_public\_exponent(phi\_n):  
 e = 65537 # Commonly used public exponent  
  
 while not (1 < e < phi\_n and math.gcd(e, phi\_n) == 1):  
 e = random.randint(2, phi\_n - 1)  
  
 return e  
  
# Hash the message using SHA3-512  
hash\_object = hashlib.sha3\_512()  
# Update the hash object with the encoded message  
hash\_object.update(msg.encode())  
hashed\_message = int.from\_bytes(hash\_object.digest(), byteorder='big')  
  
# Ensure the hash has a specific bit length  
hash\_size = 512  
hashed\_message = hashed\_message << (hash\_size - hashed\_message.bit\_length())  
print("Hashed message:", hashed\_message)  
  
# Choose two large prime numbers  
bits = 1554  
prime1 = generate\_large\_prime(bits)  
prime2 = generate\_large\_prime(bits)  
  
# Calculate the modulus and Euler's totient function  
n = prime1 \* prime2  
phi\_n = (prime1 - 1) \* (prime2 - 1)  
  
# Choose a public exponent  
e = choose\_public\_exponent(phi\_n)  
  
# Calculate the private exponent  
d = pow(e, -1, phi\_n)  
  
# Calculate the digital signature using the private exponent  
signature = pow(hashed\_message, d, n)  
  
# Verify the signature using the public exponent  
verification = pow(signature, e, n)  
  
# Print the result of signature verification  
print("Signature Validation Result:", verification == hashed\_message)

It uses SHA3-512 for hashing, generating large prime numbers, and selecting public and private exponents. The signature is calculated using the private exponent and then verified using the public exponent.

**Steps:**

**Generate Large Prime Numbers:**

The generate\_large\_prime function generates a large prime number with a specified number of bits, ensuring it is an odd number.

**Choose Public Exponent for RSA:**

The choose\_public\_exponent function selects a public exponent (e) that is commonly used (65537) but ensures it is coprime with Euler's totient function (phi\_n).

**Hash the Message:**

The script defines a message (msg) and hashes it using SHA3-512. The hash is then adjusted to a specific bit length.

**Choose Two Large Prime Numbers:**

Two large prime numbers (prime1 and prime2) are generated with a specified number of bits.

**Calculate Modulus and Euler's Totient Function:**

The modulus (n) and Euler's integral function (phi\_n) are calculated based on the chosen prime numbers.

**Choose Public Exponent:**

The public exponent (e) is selected using the choose\_public\_exponent function.

**Calculate Private Exponent:**

The private exponent (d) is calculated using modular inverse.

**Calculate Digital Signature:**

The digital signature is calculated using the private exponent.

**Verify the Signature:**

The script verifies the digital signature using the public exponent.

**Print Signature Verification Result:**

The result of the signature verification is printed, indicating whether the digital signature is valid for the given message.

The code demonstrates the RSA digital signature process, where a message is hashed, a private key is used to sign the hash, and a public key is used to verify the signature. It also emphasizes the importance of choosing appropriate prime numbers, exponents, and ensuring the security of the RSA algorithm.

**Results:**

Hashed message: 12796356638614602874332370452957861330661627345288337574091967649295300133269317238029781378352350477028721997553528439791741999679272005438747803500445142

Signature: (3527693535449717909161144223309855757835746468287338243690679529233498119551303345455937509120095178328946632192618361744723763087255395518537657940322728919206388872753531784270182458437803555058719865288743993025615003874520685075745337286563129374649756515386786670096710984027334642389107645723848952702874196879316136947005667191467597282953470343851033922372282055314081341754155692975426235542591517340588447090218197972349614806112250315560047545937795281974424340028877880230247952413768096802762254681861517704583638294580343702854967743379577295278747606429529050841146784227406615102291082010664723738652, 9537375078311888036877763751846890991618581688019953422146621132434773013310759770929465218420801127877412849156028605688189398422382215013728327674896712490901974141427529565713374620549754998578024356272448339455510515333740801449570001833979789712470542307684081584476516510720059725613352514081057185880053214704365303105857895795834498721478143764154478378025999481893045493591614074450812699285226619079907966424945301138337600623079757403011498117249797434480496453501650800700764302726183350937481450608456086527657084590296856646151604673298260380354798126469858196925915643863264115514034294143208399549668)

Signature Verification: True

**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.

**Implimentation:**

**Define the Message:**

The message to be signed is defined as "Lupascu Felicia."

**Hash the Message Using SHA3-384:**

The message is hashed using the SHA3-384 algorithm.

The resulting hash is converted to an integer and adjusted to a specific bit length.

hash\_object = hashlib.sha3\_384()  
hash\_object.update(msg.encode())

**Choose Large Prime 'p' and Generator 'g':**

A large prime number 'p' and a generator 'g' are chosen for the ElGamal algorithm.

p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281003994966956119696956248629032338072839127039  
g = 2

**Choose Private Key 'a' Randomly and Compute Public Key 'b':**

A private key 'a' is randomly chosen within a specified range.

The public key 'b' is calculated using modular exponentiation.

# Choose private key 'a' randomly  
a = random.randint(1, p - 2)  
  
# Compute public key 'b'  
b = pow(g, a, p)

**Generate a Valid Random Value 'k':**

generates a random value 'k' until it is coprime with 'p-1' (using the greatest common divisor, gcd).

# Generate a random value 'k'  
k = random.randint(1, p - 2)  
gcd\_value = math.gcd(k, p - 1)

**Calculate Signature Components 'r' and 's':**

Signature components 'r' and 's' are calculated using modular exponentiation and modular inverse.

# Calculate signature components 'r' and 's'  
r = pow(g, k, p)  
s = (pow(k, -1, p - 1) \* (hashed\_message - a \* r)) % (p - 1)  
signature = (r, s)  
print("Signature:", signature)

**Verify the Signature:**

The script verifies the generated signature by checking if certain equations hold true.

If the verification is successful, the signature is considered authentic.

V1=(b^r \* r^2) mod p

V2=g^hash mod p

# Verify the signature  
v1 = (pow(b, r\_received, p) \* pow(r\_received, s\_received, p)) % p  
v2 = pow(g, hashed\_message, p)  
verification = (v1 == v2)

**Results:**

Hashed message: 8119348613806873601709361756078987288200299675043739187491350347134519799327970594425350859143022620856192080047599266667326918858065953777253184792690688

Signature: (10936870028635241633854529492338198670188211999130420516629210112053828141938191306370127698782053489557850094633180899565459292678117472481499666571983219300902742617779502173051775720561795849493566663244114266900074176324628152570130808120952452673869716304178770562717440745187725204543477248400425061158057112180409056475325327895258876291464646513851469099100566854474144821437192020540941381120202650003670679087864133060452804229458311003640000183203511182440351212199292986860617232930622125391616350136164153652196633421693928698290898073557611032586806352163158399052230898445500728164222236940727271616438, 7954005828261162443467417407297675573359156918107769654645375246118962352887153023290369624813459544561887769612676349742758586363762262402548387842574920808230129493873145068439866421203137587307419365396033034125304441034120794436142484834612392419382046964432890401596401700062108349344226077149243262184375900962657829559120302657671797441020734723689798041054502817517186041502281050171846575363310625933112273036742782303099179738053777153222116423483052456816063923614488415343307700039413167204271010088058127029273928801331223723708530983293535814573659457533738212375734903330401656816690775551869078745174)

Signature Verification: True

**Conclusion:**

In summary, this lab demanded the implementation of cryptographic algorithms—namely RSA and ElGamal—for digital signatures. The procedure included creating message hashes utilizing SHA3-512 and SHA3-384.

This experience accentuate the important role these algorithms play in information security and the continuous need for improvements in cryptographic techniques to solve cybersecurity issues in our changing digital environment.

<https://github.com/feliciaL3/CS/tree/main/Lab6>