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 #6

Hash Functions and Digital Signatures.

Author:
Alexandru Rudoi
std. gr. FAF-231

Verified:
Maia Zaica
asist. univ.

Chișinău 2025

Contents

Objective

The objective of this laboratory work is to study and implement **cryptographic hash functions** and **digital signature algorithms**. The work focuses on two widely used asymmetric signature schemes—**RSA** and **ElGamal**—combined with hash functions required to ensure message integrity and authenticity.

The specific objectives of this laboratory work are:

- To understand the role and properties of cryptographic hash functions.
- To generate RSA keys of at least **3072 bits** and perform signing and verification using the assigned hash function.
- To generate ElGamal signature keys using the provided prime p (2048 bits) and generator $g = 2$, and to perform signing and verification.
- To compute the hash of a message obtained in Laboratory Work #2 and use it in both signature schemes.
- To analyze the full signing and verification process using decimal representations, as required.

Task Description

The laboratory work consists of two major tasks:

- **Task 2 — RSA Digital Signature**
  Using the platform WolframAlpha / Mathematica or a programming language implementation, generate RSA keys with modulus size of at least **3072 bits**, compute the hash of the message m using the assigned hash function, sign the hash, and verify the signature.
- **Task 3 — ElGamal Digital Signature**
  Using the provided parameters:

  $$p = (\text{2048-bit prime given in the lab}), \qquad g = 2,$$

  generate ElGamal signing keys, compute the assigned hash of message m, sign the message, and verify the produced signature.

Hash function selection rule:

Each student uses a specific hash function determined by:

$$i = (k \bmod 24) + 1,$$

where:

- k is the student's position in the group list,
- i is the index of the hash function in the table.

For the present laboratory work:

$$k = 25 \Rightarrow i = (25 \bmod 24) + 1 = 2.$$

Thus, the assigned hash functions are:

- **For RSA: index 2 → MD5**
- **For ElGamal: index 2 → MD4**

These hash functions will be used to compute the decimal message digest required for the signature algorithms.

Theoretical Background

Cryptographic Hash Functions

A **cryptographic hash function** is an algorithm that takes an input message of arbitrary length and produces a fixed-length digest. Hash functions used in cryptography must satisfy:

- **Preimage resistance**: Given a hash value h, it should be infeasible to find any message m such that $H(m) = h$.
- **Second preimage resistance**: Given a message m_1, it should be infeasible to find a different message m_2 such that $H(m_1) = H(m_2)$.
- **Collision resistance**: It should be infeasible to find any two messages m_1, m_2 such that $H(m_1) = H(m_2)$.

Hash functions are essential in digital signatures because they compress large messages into fixed-size values, making signing efficient and secure.

MD5 (Used in RSA)

MD5 is a 128-bit cryptographic hash function developed by Ronald Rivest. Although MD5 is no longer secure against collisions, it remains useful in academic contexts.

Output size:

$$H_{\text{MD5}}(m) \in \{0,1\}^{128}$$

In this laboratory work, MD5 is used to compute the message digest that is signed using RSA.

MD4 (Used in ElGamal)

MD4 is another hash function designed by Rivest, producing a 128-bit output. It is cryptographically broken but fully acceptable for educational demonstration.

Output size:

$$H_{\text{MD4}}(m) \in \{0,1\}^{128}$$

Digital Signatures Overview

A **digital signature** provides:

- **Integrity** — proves the message was not modified.
- **Authenticity** — proves the identity of the signer.
- **Non-repudiation** — the signer cannot deny their action.

The standard signature process includes:

1. Compute the message hash: $h = H(m)$
2. Sign the hash using the private key: $s = f_{\text{priv}}(h)$
3. Verify the signature using the public key: $h' = f_{\text{pub}}(s)$
4. If $h = h'$, the signature is valid.

Note: Digital signatures do not encrypt the message; they authenticate it.

RSA Digital Signature

RSA signatures rely on modular arithmetic:

$$s = h^d \mod n, \qquad h = s^e \mod n.$$

Steps:

1. Generate primes p and q and compute $n = pq$.
2. Compute Euler's totient: $\phi(n) = (p-1)(q-1)$.
3. Choose public exponent e and compute private exponent d.
4. Compute the MD5 hash of the message.
5. Sign: $s = h^d \mod n$.
6. Verify: $h = s^e \mod n$.

ElGamal Digital Signature

ElGamal signatures operate in a multiplicative group modulo a large prime p.

Signing:

$$r = g^k \mod p, \qquad s = k^{-1}(h - xr) \mod (p-1),$$

where x is the private key and k is a random number with $\gcd(k, p-1) = 1$.

Verification:

$$g^h \equiv y^r \cdot r^s \pmod{p},$$

where $y = g^x \mod p$ is the public key.

Technical Implementation

In this section, the implementation of the **RSA** and **ElGamal** digital signature algorithms is presented. Both implementations were carried out in Python, using the assigned hash functions **MD5** (for RSA) and **MD4** (for ElGamal), selected based on the formula:

$$i = (k \bmod 24) + 1, \qquad k = 25 \Rightarrow i = 2.$$

The message used for signing was the one obtained in Laboratory Work #2 and stored in a separate text file `message.txt`.

All hash values, signatures, and verification parameters were handled in their **decimal representation**, according to the laboratory requirements.

RSA Digital Signature Implementation

The RSA implementation consists of: **(1) key generation**, **(2) hashing**, **(3) signature creation**, and **(4) signature verification**.

A modulus of **3072 bits** was chosen, satisfying the requirement that n must be at least 3072 bits.

1. RSA Key Generation

The RSA key pair is generated using two large random primes p and q, each 1536 bits long. The modulus and private exponent are computed as:

```
1 from Crypto.Util import number
2
3 def generate_rsa_keys(bits=3072):
4     p = number.getPrime(bits // 2)
5     q = number.getPrime(bits // 2)
6     n = p * q
```

```
    phi = (p - 1) * (q - 1)
    e = 65537
    d = pow(e, -1, phi)

    return (n, e, d, p, q)
```

Listing 1: RSA Key Generation (3072-bit modulus)

Explanation:

- Two 1536-bit primes p and q are generated.
- The RSA modulus is computed as $n = pq$.
- The totient is $\phi(n) = (p-1)(q-1)$.
- The public exponent is fixed at $e = 65537$.
- The private exponent is computed using the modular inverse:

$$d = e^{-1} \bmod \phi(n).$$

2. RSA Signing (MD5 Hash)

Before signing, the message is hashed using the assigned function **MD5**. The hash is printed in both hexadecimal and decimal form, and then the signature is computed as:

$$s = h^d \mod n.$$

```
def md5_hash(message: str) -> int:
    h = hashlib.md5()
    h.update(message.encode())
    return int(h.hexdigest(), 16)

def sign(message: str, d: int, n: int):
    h = md5_hash(message)
    signature = pow(h, d, n)
    return signature, h
```

Listing 2: RSA Signing with MD5

3. RSA Verification

Verification recomputes the message hash and checks:

$$h \stackrel{?}{=} s^e \mod n.$$

```python
def verify(message: str, signature: int, e: int, n: int):
    h = md5_hash(message)
    h_prime = pow(signature, e, n)
    return h == h_prime
```

Listing 3: RSA Verification

A signature is valid if:

$$h = h'.$$

ElGamal Digital Signature Implementation

For ElGamal, the laboratory provided a fixed 2048-bit prime p and a generator $g = 2$. The signature scheme uses the assigned hash function **MD4**.

The implementation involves: **(1) key generation**, **(2) MD4 hashing**, **(3) signature creation**, and **(4) verification**.

1. ElGamal Key Generation

The private key x is randomly chosen, and the public key is:

$$y = g^x \mod p.$$

```python
def generate_elgamal_keys(p: str, g: str):
    p_int = int(p)
    g_int = int(g)

    x = random.randint(2, p_int - 2)
    y = pow(g_int, x, p_int)
    return x, y, p_int, g_int
```

Listing 4: ElGamal Key Generation (p and g provided)

2. ElGamal Signing (MD4 Hash)

Signing requires generating a random value k such that:

$$\gcd(k, p-1) = 1.$$

The signature is computed as:

$$r = g^k \mod p, \qquad s = k^{-1}(h - xr) \mod (p-1).$$

```python
from Crypto.Hash import MD4

def md4_hash(message: str) -> int:
    h = MD4.new()
    h.update(message.encode())
    return int(h.hexdigest(), 16)

def sign(message: str, x: int, p: int, g: int):
    h = md4_hash(message)

    while True:
        k = random.randint(2, p - 2)
        if number.GCD(k, p - 1) == 1:
            break

    r = pow(g, k, p)
    k_inv = pow(k, -1, p - 1)
    s = ((h - x * r) * k_inv) % (p - 1)

    return (r, s, h)
```

Listing 5: ElGamal Signing with MD4

3. ElGamal Verification

Verification computes:

$$g^h \pmod{p} \quad \text{and} \quad y^r \cdot r^s \pmod{p},$$

and checks if they are equal.

```python
def verify(message: str, signature, y: int, p: int, g: int):
    r, s, _ = signature
    h = md4_hash(message)

    left = pow(g, h, p)
    right = (pow(y, r, p) * pow(r, s, p)) % p

    return left == right
```

Listing 6: ElGamal Signature Verification

Integration Script

The following script integrates all steps: loading the message, generating keys, signing, and verifying.

```
def main():
    message = read_message()

    # RSA
    n, e, d, p, q = generate_rsa_keys(3072)
    signature_rsa, hash_rsa = rsa.sign(message, d, n)
    valid_rsa = rsa.verify(message, signature_rsa, e, n)

    # ElGamal
    x, y, p_int, g_int = generate_elgamal_keys(p_elgamal, g_elgamal)
    signature_elgamal = elgamal.sign(message, x, p_int, g_int)
    valid_elgamal = elgamal.verify(message, signature_elgamal, y, p_int,
        g_int)
```

Listing 7: Main Python Integration Script

This script completes the digital signature workflow for both RSA and ElGamal.

Results

In this section, the results produced by the implementation of the RSA and ElGamal digital signature schemes are presented. Each subsection includes the corresponding console output screenshots and a detailed explanation of the signing and verification process.

All computations were performed using:

- RSA modulus size: **3072 bits**
- ElGamal prime: the **2048-bit** value provided in the laboratory instructions
- Hash functions assigned according to $i = (k \bmod 24) + 1 = 2$:
  - **MD5** — for RSA signatures
  - **MD4** — for ElGamal signatures

The message used for hashing and signing was loaded from `message.txt`, generated in Laboratory Work #2.

RSA Digital Signature Results

- **Screenshot 1: RSA Key Generation (3072-bit modulus)**

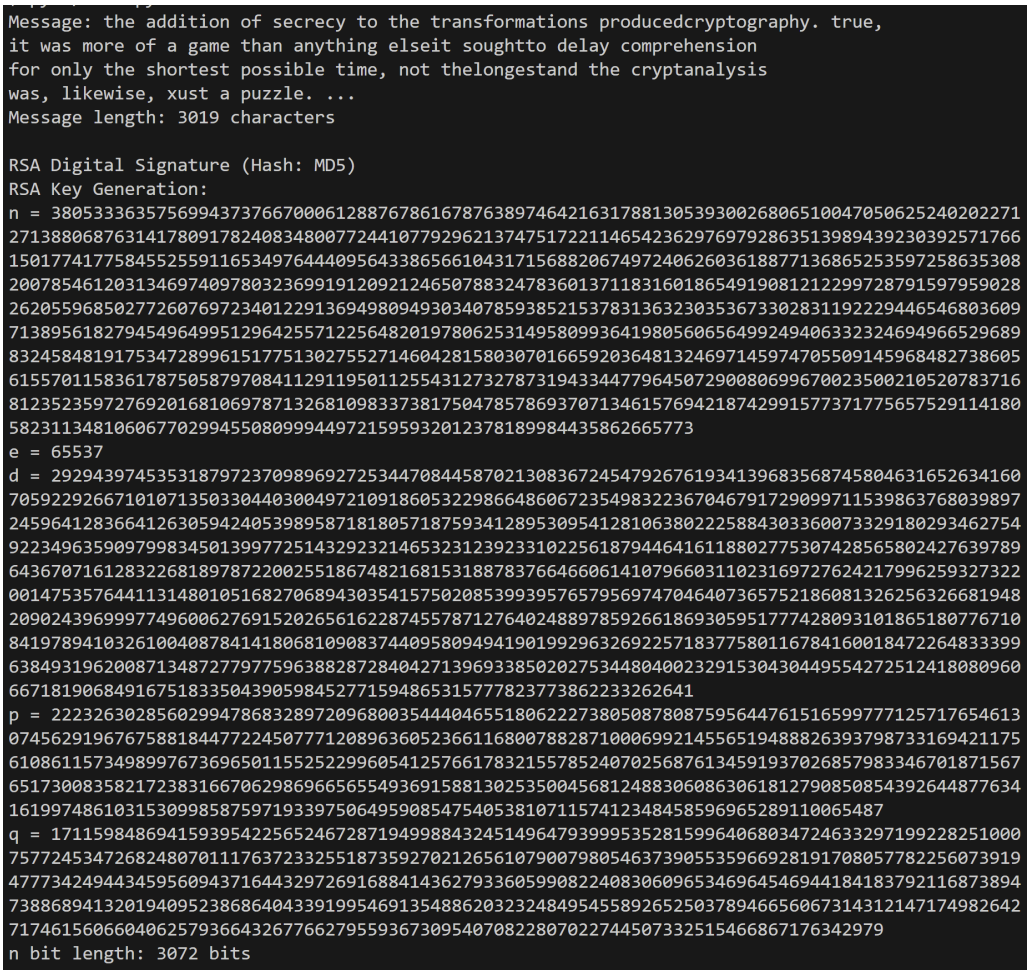

Figure 1: RSA Key Generation: 3072-bit Modulus, Public Exponent e, Private Exponent d

This screenshot shows the successful generation of the RSA key pair. The values displayed include:

- large primes p and q,
- modulus $n = pq$ (verified to be 3072 bits),
- Euler's totient $\phi(n)$,
- public exponent $e = 65537$,
- private exponent d, computed as $e^{-1} \bmod \phi(n)$.

These parameters form the core of the RSA digital signature scheme.

- **Screenshot 2: MD5 Hash Computation and RSA Signature**

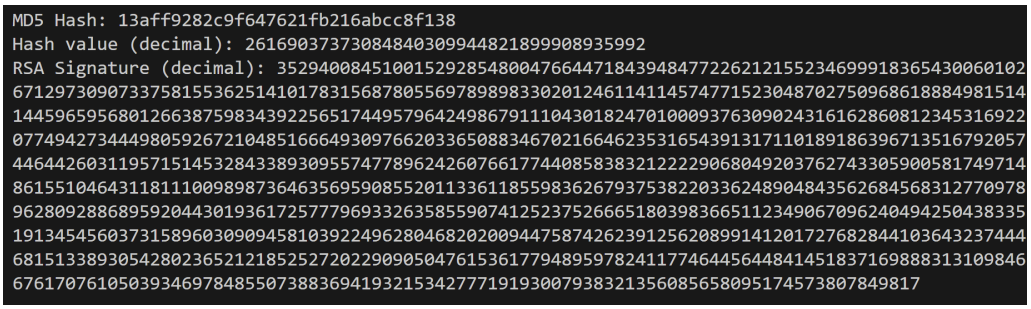

Figure 2: MD5 Hash (Hex and Decimal) and RSA Signature Computation

This output shows:

- the MD5 hash of the message in both hexadecimal and decimal representation,
- the signature computed as:

$$s = h^d \bmod n.$$

The signature value is extremely large (close to 3072 bits), as expected for RSA of this size.

- **Screenshot 3: RSA Signature Verification**

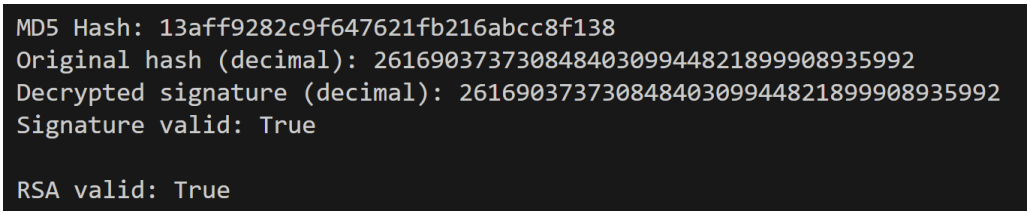

Figure 3: Verification: Checking if $h = s^e \bmod n$

The verification process successfully confirms the integrity and authenticity of the signature:

$$s^e \bmod n = h.$$

The result **Signature valid: True** shows that the RSA digital signature process was implemented correctly.

ElGamal Digital Signature Results

- **Screenshot 4: ElGamal Key Generation (Using Provided 2048-bit p)**

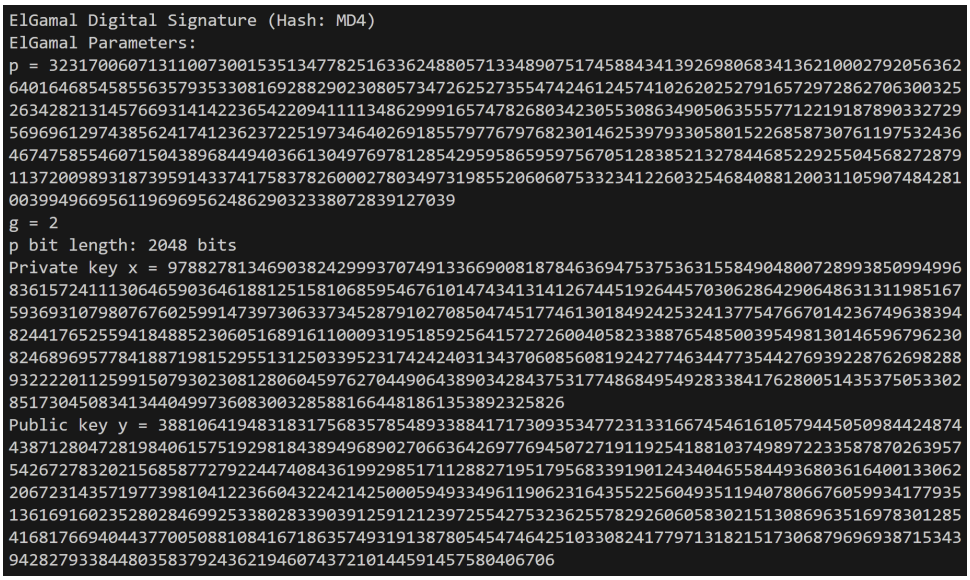

Figure 4: ElGamal Key Generation: Private Key x, Public Key $y = g^x \bmod p$

This screenshot displays:

- the fixed 2048-bit prime p,
- generator $g = 2$,
- randomly selected private key x,
- public key computed as $y = g^x \bmod p$.

These parameters are used to sign and verify the MD4 hash of the message.

- **Screenshot 5: MD4 Hash and ElGamal Signature Parameters r and s**

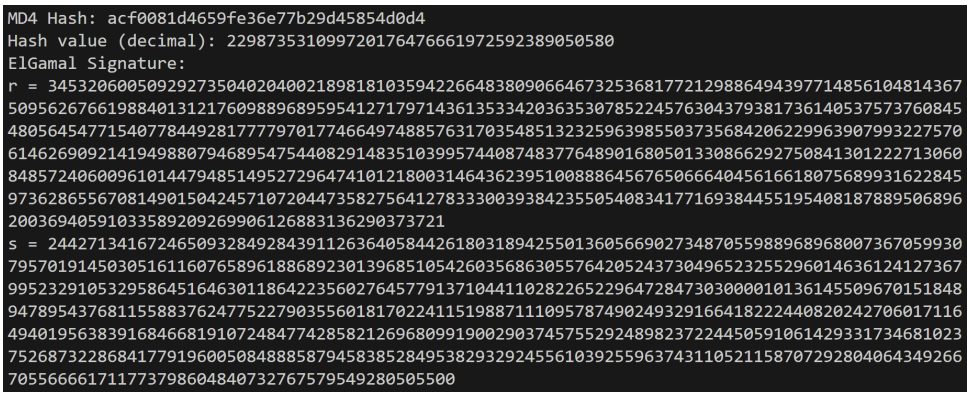

Figure 5: ElGamal Signature Generation: Hash h, Values r and s

This screenshot demonstrates:

– the MD4 hash of the message (hex + decimal),

– the randomly chosen k such that $\gcd(k, p-1) = 1$,

– the signature components:

$$r = g^k \bmod p, \qquad s = k^{-1}(h - xr) \bmod (p-1).$$

Both r and s appear as large random values, as expected.

- **Screenshot 6: Verification of the ElGamal Signature**

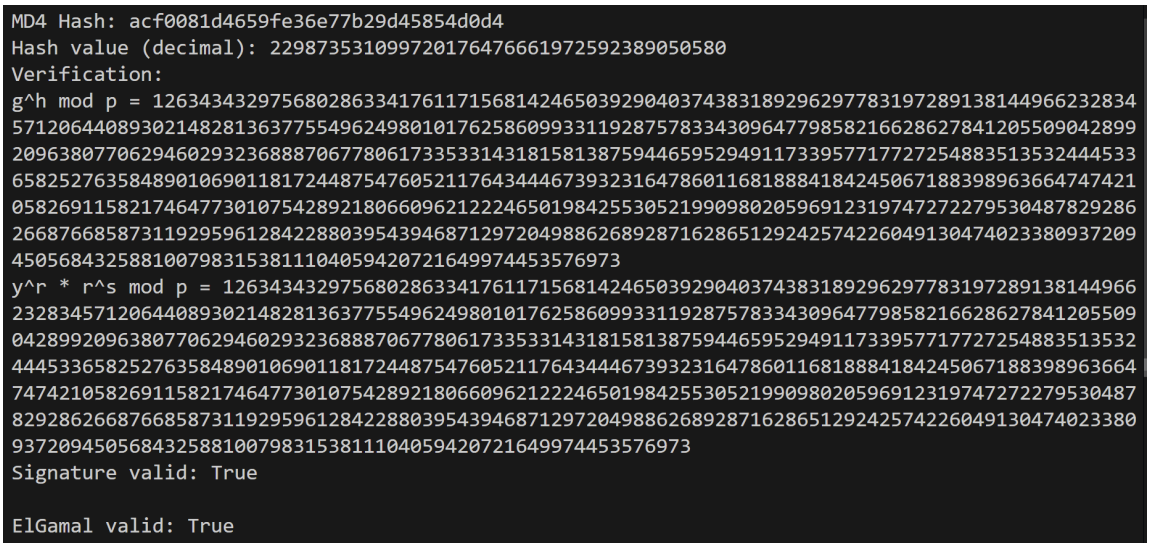

Figure 6: ElGamal Verification: Checking if $g^h \equiv y^r r^s \pmod{p}$

The verification compares:

$$\text{left} = g^h \bmod p, \qquad \text{right} = y^r \cdot r^s \bmod p.$$

Since the values match, the output prints:

$$\text{Signature valid: True.}$$

This confirms that the ElGamal signature implementation behaves correctly for the assigned parameters.

Conclusion

In conclusion, in this laboratory work, I studied and implemented **hash functions** and **digital signature schemes** using RSA and ElGamal. The main focus was on how cryptographic hash functions are combined with asymmetric algorithms to ensure **integrity**, **authenticity**, and **non-repudiation** of messages.

For the RSA digital signature, I generated keys with a modulus of at least **3072 bits**, satisfying the security requirement specified in the task. The hash of the message (obtained from Laboratory Work #2) was computed using the assigned hash function **MD5** (index 2 from the first list, based on $i = (k \bmod 24) + 1$ with $k = 25$). The MD5 digest was converted to its decimal representation and then signed using the RSA private key. During verification, the signature was correctly validated by recomputing the hash and checking that

$$h = s^e \bmod n,$$

which confirmed the correctness of the implementation.

For the ElGamal digital signature, I used the **given 2048-bit prime** p and generator $g = 2$, as required by the lab specification. The signing and verification were performed using the assigned hash function **MD4** (index 2 from the second list). The MD4 hash of the same message was computed, converted to its decimal form, and used in the ElGamal signature equations. The verification step confirmed that

$$g^h \equiv y^r \cdot r^s \pmod{p},$$

showing that the ElGamal signature scheme was also implemented correctly.

Through this work, I reinforced the following key ideas:

- Digital signatures are always applied to the **hash** of a message, not directly to the full message.
- Different hash functions (MD5 and MD4 in this case) can be integrated into the same digital signature framework.
- RSA and ElGamal signatures use different mathematical structures (factoring vs. discrete logarithms), but both provide authenticity and integrity.
- The choice of **key size** (3072-bit RSA, 2048-bit ElGamal prime) is crucial for practical security.

Even though MD5 and MD4 are no longer recommended for real-world cryptographic applications due to known vulnerabilities, they serve as useful tools for understanding the principles of hash-based digital signatures. Overall, this laboratory work provided practical experience with the complete signing and verification pipeline, from hashing the message to validating the resulting signature in two different asymmetric schemes.

Git repository: `https://github.com/AlexandruRudoi/CS_Labs/tree/main/Lab_6`

Bibliography

1. Course materials: *Cryptography and Security*, UTM–FCIM, 2025.
   Lecture notes and lab instructions for public-key cryptography, hash functions, and digital signatures.

2. Rivest, R. L. (1992). *The MD5 Message-Digest Algorithm.* RFC 1321, Internet Engineering Task Force (IETF).
   Defines the MD5 hash function used in this laboratory work for the RSA digital signature.

3. Rivest, R. L. (1991). *The MD4 Message-Digest Algorithm.* RFC 1186, Internet Engineering Task Force (IETF).
   Describes the MD4 hash function used in this laboratory work for the ElGamal digital signature.

4. ElGamal, T. (1985). *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms.* IEEE Transactions on Information Theory, 31(4), 469–472.
   Original paper that introduces the ElGamal encryption and signature schemes based on the discrete logarithm problem.

5. Rivest, R. L., Shamir, A., & Adleman, L. (1978). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21(2), 120–126.
   Foundational paper describing the RSA cryptosystem and its use for digital signatures.

6. Stinson, D. R., & Paterson, M. (2018). *Cryptography: Theory and Practice* (4th ed.). CRC Press.
   A comprehensive textbook covering hash functions, RSA, ElGamal, and digital signature schemes from both theoretical and practical perspectives.

7. Schneier, B., Ferguson, N., & Kohno, T. (2010). *Cryptography Engineering: Design Principles and Practical Applications.* Wiley.
   Practical guidance on designing and implementing cryptographic systems, including hash-based signatures and key size recommendations.