

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.

Chişinău 2025

Contents

Objective	3
Task Description	3
Theoretical Background	4
Technical Implementation	6
Results	10
Conclusion	14
Bibliography	16

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:

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

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 \rightarrow MD5**
- **For ElGamal: index 2 \rightarrow 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:

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

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:

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

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:

$$i = (k \mod 24) + 1, \quad k = 25 \Rightarrow i = 2.$$

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
```

```

7     phi = (p - 1) * (q - 1)
8     e = 65537
9     d = pow(e, -1, phi)
10
11     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:

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

```

1 def md5_hash(message: str) -> int:
2     h = hashlib.md5()
3     h.update(message.encode())
4     return int(h.hexdigest(), 16)
5
6 def sign(message: str, d: int, n: int):
7     h = md5_hash(message)
8     signature = pow(h, d, n)
9     return signature, h

```

Listing 2: RSA Signing with MD5

3. RSA Verification

Verification recomputes the message hash and checks:

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

```

1 def verify(message: str, signature: int, e: int, n: int):
2     h = md5_hash(message)
3     h_prime = pow(signature, e, n)
4     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:

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

```

1 def generate_elgamal_keys(p: str, g: str):
2     p_int = int(p)
3     g_int = int(g)
4
5     x = random.randint(2, p_int - 2)
6     y = pow(g_int, x, p_int)
7     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:

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


```

1 from Crypto.Hash import MD4
2
3 def md4_hash(message: str) -> int:
4     h = MD4.new()
5     h.update(message.encode())
6     return int(h.hexdigest(), 16)
7
8 def sign(message: str, x: int, p: int, g: int):
9     h = md4_hash(message)
10
11     while True:
12         k = random.randint(2, p - 2)
13         if number.GCD(k, p - 1) == 1:
14             break
15
16     r = pow(g, k, p)
17     k_inv = pow(k, -1, p - 1)
18     s = ((h - x * r) * k_inv) % (p - 1)
19
20     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.

```

1 def verify(message: str, signature, y: int, p: int, g: int):
2     r, s, _ = signature
3     h = md4_hash(message)
4
5     left = pow(g, h, p)
6     right = (pow(y, r, p) * pow(r, s, p)) % p
7
8     return left == right

```

Listing 6: ElGamal Signature Verification

Integration Script

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

```
1 def main():
2     message = read_message()
3
4     # RSA
5     n, e, d, p, q = generate_rsa_keys(3072)
6     signature_rsa, hash_rsa = rsa.sign(message, d, n)
7     valid_rsa = rsa.verify(message, signature_rsa, e, n)
8
9     # ElGamal
10    x, y, p_int, g_int = generate_elgamal_keys(p_elgamal, g_elgamal)
11    signature_elgamal = elgamal.sign(message, x, p_int, g_int)
12    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)

```

Message: the addition of secrecy to the transformations produced cryptography. true,
it was more of a game than anything else it sought to delay comprehension
for only the shortest possible time, not the longest and the cryptanalysis
was, likewise, just a puzzle. ...
Message length: 3019 characters

RSA Digital Signature (Hash: MD5)
RSA Key Generation:
n = 38053336357569943737667000612887678616787638974642163178813053930026806510047050625240202271
271388068763141780917824083480077244107792962137475172211465423629769792863513989439230392571766
150177417758455255911653497644409564338656610431715688206749724062603618877136865253597258635308
200785461203134697409780323699191209212465078832478360137118316018654919081212299728791597959028
262055968502772607697234012291369498094930340785938521537831363230353673302831192229446546803609
713895618279454964995129642557122564820197806253149580993641980560656499249406332324694966529689
832458481917534728996151775130275527146042815803070166592036481324697145974705509145968482738605
615570115836178750587970841129119501125543127327873194334477964507290080699670023500210520783716
812352359727692016810697871326810983373817504785786937071346157694218742991577371775657529114180
58231134810606770299455080999449721595932012378189984435862665773
e = 65537
d = 29294397453531879723709896927253447084458702130836724547926761934139683568745804631652634160
705922926671010713503304403004972109186053229866486067235498322367046791729099711539863768039897
245964128366412630594240539895871818057187593412895309541281063802225884303360073329180293462754
9223496359097998345013997725143292321246532312392331022561879446416118802775307428565802427639789
643670716128322681897872200255186748216815318878376646606141079660311023169727624217996259327322
001475357644113148010516827068943035415750208539939576579569747046407365752186081326256326681948
209024396999774960062769152026561622874557871276402488978592661869305951777428093101865180776710
841978941032610040878414180681090837440958094941901992963269225718377580116784160018472264833399
638493196200871348727797759638828728404271396933850202753448040023291530430449554272512418080960
66718190684916751833504390598452771594865315777823773862233262641
p = 22232630285602994786832897209680035444046551806222738050878087595644761516599777125717654613
074562919676758818447722450777120896360523661168007882871000699214556519488826393798733169421175
610861157349899767369650115525229960541257661783215578524070256876134591937026857983346701871567
651730083582172383166706298696656554936915881302535004568124883060863061812790850854392644877634
16199748610315309985875971933975064959085475405381071157412348458596965289110065487
q = 17115984869415939542256524672871949988432451496479399953528159964068034724633297199228251000
757724534726824807011176372332551873592702126561079007980546373905535966928191708057782256073919
477734249443459560943716443297269168841436279336059908224083060965346964546944184183792116873894
738868941320194095238686404339199546913548862032324849545589265250378946656067314312147174982642
71746156066040625793664326776627955936730954070822807022744507332515466867176342979
n bit length: 3072 bits

```

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**

```
MD5 Hash: 13aff9282c9f647621fb216abcc8f138
Hash value (decimal): 26169037373084840309944821899908935992
RSA Signature (decimal): 35294008451001529285480047664471843948477226212155234699918365430060102
671297309073375815536251410178315687805569789898330201246114114574771523048702750968618884981514
144596595680126638759834392256517449579642498679111043018247010009376309024316162860812345316922
077494273444980592672104851666493097662033650883467021664623531654391317110189186396713516792057
446442603119571514532843389309557477896242607661774408583832122229068049203762743305900581749714
861551046431181110098987364635695908552011336118559836267937538220336248904843562684568312770978
962809288689592044301936172577796933263585590741252375266651803983665112349067096240494250438335
191345456037315896030909458103922496280468202009447587426239125620899141201727682844103643237444
681513389305428023652121852527202290905047615361779489597824117746445644841451837169888313109846
67617076105039346978485507388369419321534277719193007938321356085658095174573807849817
```

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**

```
MD5 Hash: 13aff9282c9f647621fb216abcc8f138
Original hash (decimal): 26169037373084840309944821899908935992
Decrypted signature (decimal): 26169037373084840309944821899908935992
Signature valid: True

RSA valid: True
```

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)

```

ElGamal Digital Signature (Hash: MD4)
ElGamal Parameters:
p = 32317006071311007300153513477825163362488057133489075174588434139269806834136210002792056362
640164685458556357935330816928829023080573472625273554742461245741026202527916572972862706300325
263428213145766931414223654220941111348629991657478268034230553086349050635557712219187890332729
569696129743856241741236237225197346402691855797767976823014625397933058015226858730761197532436
467475855460715043896844940366130497697812854295958659597567051283852132784468522925504568272879
113720098931873959143374175837826000278034973198552060607533234122603254684088120031105907484281
003994966956119696956248629032338072839127039
g = 2
p bit length: 2048 bits
Private key x = 97882781346903824299937074913366900818784636947537536315584904800728993850994996
836157241113064659036461881251581068595467610147434131412674451926445703062864290648631311985167
593693107980767602599147397306337345287910270850474517746130184924253241377547667014236749638394
824417652559418488523060516891611000931951859256415727260040582338876548500395498130146596796230
824689695778418871981529551312503395231742424031343706085608192427746344773544276939228762698288
932222011259915079302308128060459762704490643890342843753177486849549283384176280051435375053302
85173045083413440499736083003285881664481861353892325826
Public key y = 388106419483183175683578548933884171730935347723133166745461610579445050984424874
438712804728198406157519298184389496890270663642697769450727191192541881037498972233587870263957
542672783202156858772792244740843619929851711288271951795683391901243404655844936803616400133062
206723143571977398104122366043224214250005949334961190623164355225604935119407806676059934177935
136169160235280284699253380283390391259121239725542753236255782926060583021513086963516978301285
416817669404437700508810841671863574931913878054547464251033082417797131821517306879696938715343
9428279338448035837924362194607437210144591457580406706

```

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

```

MD4 Hash: acf0081d4659fe36e77b29d45854d0d4
Hash value (decimal): 229873531099720176476661972592389050580
ElGamal Signature:
r = 34532060050929273504020400218981810359422664838090664673253681772129886494397714856104814367
509562676619884013121760988968959541271797143613533420363530785224576304379381736140537573760845
48056454771540778449281777970177466497488576317035485132325963985503735684206229963907993227570
614626909214194988079468954754408291483510399574408748377648901680501330866292750841301222713060
848572406009610144794851495272964741012180031464362395100888645676506664045616618075689931622845
973628655670814901504245710720447358275641278333003938423550540834177169384455195408187889506896
20036940591033589209269906126883136290373721
s = 24427134167246509328492843911263640584426180318942550136056690273487055988968968007367059930
795701914503051611607658961886892301396851054260356863055764205243730496523255296014636124127367
995232910532958645164630118642235602764577913710441102822652296472847303000010136145509670151848
947895437681155883762477522790355601817022411519887111095787490249329166418222440820242706017116
494019563839168466819107248477428582126968099190029037457552924898237224450591061429331734681023
752687322868417791960050848885879458385284953829329245561039255963743110521158707292804064349266
705566661711773798604840732767579549280505500

```

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:

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

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

• Screenshot 6: Verification of the ElGamal Signature

```
MD4 Hash: acf0081d4659fe36e77b29d45854d0d4
Hash value (decimal): 229873531099720176476661972592389050580
Verification:
g^h mod p = 126343432975680286334176117156814246503929040374383189296297783197289138144966232834
571206440893021482813637755496249801017625860993311928757833430964779858216628627841205509042899
209638077062946029323688870677806173353314318158138759446595294911733957717727254883513532444533
658252763584890106901181724487547605211764344467393231647860116818884184245067188398963664747421
058269115821746477301075428921806609621222465019842553052199098020596912319747272279530487829286
266876685873119295961284228803954394687129720498862689287162865129242574226049130474023380937209
4505684325881007983153811104059420721649974453576973
y^r * r^s mod p = 126343432975680286334176117156814246503929040374383189296297783197289138144966
232834571206440893021482813637755496249801017625860993311928757833430964779858216628627841205509
042899209638077062946029323688870677806173353314318158138759446595294911733957717727254883513532
444533658252763584890106901181724487547605211764344467393231647860116818884184245067188398963664
747421058269115821746477301075428921806609621222465019842553052199098020596912319747272279530487
829286266876685873119295961284228803954394687129720498862689287162865129242574226049130474023380
9372094505684325881007983153811104059420721649974453576973
Signature valid: True

ElGamal valid: True
```

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

The verification compares:

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

Since the values match, the output prints:

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.