Ovidius University of Constanţa

Faculty of Mathematics and Informatics

Cybersecurity and Machine Learning

Digital signature techniques as an authentication method

Students: Memedula Edna

Nicolaev Andrei

2025-2026

Table of Contents

Chapter 1 – Introduction

1.1 Computer Science

Computer science is a comprehensive discipline concerned with the design, analysis, and application of computational systems. It encompasses a broad spectrum of theoretical and practical domains, from abstract algorithmic reasoning to concrete hardware-software interaction. Its goal is to solve complex problems, automate processes, and process large-scale data efficiently and securely.

Core areas of computer science include:

- **Algorithms and Data Structures** – methods for organizing and manipulating data efficiently.
- **Computer Architecture and Systems Design** – focusing on how hardware components interact with software layers.
- **Operating Systems and Software Engineering** – ensuring reliability, scalability, and maintainability in complex applications.
- **Networking and Distributed Systems** – enabling communication and resource sharing between remote systems.
- **Databases and Information Systems** – managing, querying, and safeguarding large datasets.
- **Artificial Intelligence and Machine Learning** – systems that simulate intelligent behavior and adapt based on data.
- **Human-Computer Interaction (HCI)** – optimizing user interfaces and usability.

Computer science draws heavily from **mathematics** (logic, combinatorics, graph theory) and **engineering** (digital electronics, control theory). Additionally, it employs **statistical methods** for performance evaluation, machine learning, and stochastic modeling.

A key aspect of the discipline is its focus on **empirical validation**. Researchers and practitioners rely on:

- **Experimental simulations**
- **Benchmarking** of algorithms and systems
- **Formal verification** to ensure correctness

Increasingly, **security and privacy** have become central concerns, influencing the development of secure protocols, access control mechanisms, cryptographic primitives, and data protection strategies.

Computer science is the engine behind today's digital transformation, enabling everything from cloud computing and robotics to mobile applications and quantum computing.

1.2 Cybersecurity

Cybersecurity is a specialized subfield of computer science dedicated to defending systems, networks, and data against unauthorized access, cyberattacks, and digital threats.

As digital devices and online services become integral to modern life, securing these systems is more critical than ever.

Key areas of cybersecurity include:

- **Network Security** – protecting communication protocols and preventing intrusion in networked environments.
- **Application Security** – ensuring software is free of vulnerabilities during design and deployment.
- **Information Security (InfoSec)** – guarding the confidentiality, integrity, and availability of data.
- **Identity and Access Management (IAM)** – controlling who has access to what information and systems.
- **Cryptographic Protocols** – encrypting data to prevent eavesdropping and forgery.
- **Incident Response and Forensics** – identifying, analyzing, and responding to breaches or malicious behavior.

Cybersecurity threats come in various forms:

- **Malware** (e.g., viruses, ransomware)
- **Phishing attacks** (fraudulent communication to extract sensitive data)
- **Denial-of-Service (DoS/DDoS)** attacks
- **Insider threats**
- **Advanced Persistent Threats (APTs)** – often involving **state-sponsored** actors

Historical cases such as **Marcus Hess**, a hacker who infiltrated US defense systems for the KGB, illustrate how vulnerabilities—if unpatched—can be exploited by attackers for political, financial, or ideological motives.

Hacktivists, cybercriminals, and nation-state actors each have different agendas, but all exploit weaknesses in systems, human behavior, or supply chains. Cybersecurity professionals must anticipate, detect, and mitigate these threats in real time.

As more systems shift online, from healthcare to critical infrastructure, cybersecurity becomes not just a technical concern but a national and global imperative.

1.3 Cryptography

Cryptography is the mathematical science of secure communication. It ensures that data remains **confidential**, **authentic**, and **tamper-proof**—whether stored or in transit. It plays a foundational role in cybersecurity, banking, blockchain technology, and secure messaging.

The term comes from the Greek *kryptós* (“hidden”) and *gráfein* (“to write”). While early cryptographic techniques such as the **Caesar cipher** were used by historical figures like Julius Caesar to secure military messages, modern cryptography involves highly complex mathematical transformations.

Cryptography is divided into:

- **Symmetric Cryptography** – the same key is used for encryption and decryption (e.g., AES).
- **Asymmetric Cryptography** – uses a **public-private key pair**, enabling secure key exchange, digital signatures, and zero-knowledge proofs (e.g., RSA, ECC).
- **Cryptanalysis** – the study of methods for breaking cryptographic systems.
- **Post-Quantum Cryptography** – developing algorithms resistant to quantum computer attacks.
- **Quantum Cryptography** – using quantum physics to achieve unbreakable encryption, like in Quantum Key Distribution (QKD).

A message in its readable form is called **plaintext**, and once encrypted, becomes **ciphertext**. This transformation is done via a combination of:

- A **cryptographic algorithm** (e.g., AES, RSA)
- A **key** (a number or bit string that drives the algorithm's operation)

Key concepts include:

- **Encryption** – the process of transforming plaintext into ciphertext to prevent unauthorized access.
- **Decryption** – converting ciphertext back into readable plaintext using the correct key.
- **Hash Functions** – irreversible one-way functions that map data to a fixed-length output, essential in digital signatures and blockchain.

A secure cryptographic system allows the encryption method to be public, as long as the **key** remains secret. Without the correct key, decrypting the message should be computationally infeasible, even with full knowledge of the algorithm.

Cryptography has evolved from simple codes into an advanced scientific domain embedded in protocols like HTTPS, VPNs, digital currencies, and secure multi-party computation.

Chapter 2 – Digital Signature

2.1 – Terminology

A **signer** is a person who initiates a digital signature using a signature creation device. This individual may act in a personal capacity or represent another party—such as a company, government body, or other legal entity. In the context of secure digital communications, the signer is responsible for ensuring that the signature is generated using trusted means and in a secure environment.

Signature creation data refers to any unique digital credentials—typically cryptographic in nature—used to generate a digital signature. These may include private keys, secure tokens, or specialized credentials embedded in hardware modules like smart cards or USB tokens. This data must remain confidential and securely stored, as unauthorized access could compromise the integrity of the signature process.

A **digital signature creation device** can be software-based (e.g., cryptographic libraries, mobile applications) or hardware-based (e.g., hardware security modules or smart cards). These devices are responsible for securely applying the private key to the message digest or content, producing a signature that is mathematically tied to the signer and the data being signed.

2.2 – Definition

A **digital signature scheme** is a cryptographic framework designed to provide authentication, data integrity, and non-repudiation in digital communications. It typically involves the following three core algorithms:

- **Key Generation Algorithm**: This algorithm generates a key pair—comprising a private key (kept secret) and a corresponding public key (shared openly). These keys are mathematically linked through a one-way function, ensuring the private key cannot be derived from the public key.
- **Signing Algorithm**: This algorithm takes two inputs—a message (or its hash) and a private key—and outputs a digital signature. The result is a unique fingerprint that is cryptographically bound to the content and the identity of the signer.
- **Verification Algorithm**: This algorithm takes the message, the digital signature, and the signer's public key. It checks whether the signature is valid by comparing the decrypted signature against a freshly computed hash of the message. If they match, authenticity and data integrity are confirmed.

Two essential security properties of digital signature schemes are:

1. **Authenticity** – A digital signature created using a specific private key and message must be verifiable by the corresponding public key. This ensures the signature was indeed created by the legitimate signer.
2. **Unforgeability** – It must be **computationally infeasible** for an attacker to generate a valid signature without knowing the private key, even if they have access to multiple message-signature pairs.

A **digital signature** functions as an electronic seal of authenticity. The signer "locks" their approval onto a digital document using a cryptographic method, and this seal can be verified by others using the corresponding public key. A well-known implementation is the **Digital Signature Algorithm (DSA)**, standardized by NIST for government and commercial applications.

An **advanced electronic signature (AES)** must meet additional legal and technical criteria:

- **Uniquely linked** to the signer through a secure private key
- **Capable of identifying** the signer without ambiguity
- **Created using means** under the sole control of the signer
- **Bound to the signed data** so that any tampering can be detected immediately

2.3 – Who Can Sign Digitally

Any individual—whether a private citizen, employee, or government official—can apply for a **digital certificate** from a recognized Certificate Authority (CA) and use it to sign documents electronically. The scope of use can be categorized as:

- **Optional Use** – For non-binding purposes, such as confirming information in casual digital correspondence.
- **Mandatory Use** – For submitting legally binding documents, especially in regulated sectors (e.g., taxation, public procurement, or financial reporting).

Unlike handwritten signatures, **electronic signatures can be delegated**. Authorized individuals can sign:

- On **their own behalf**, or
- On **behalf of an organization**, with appropriate legal authorization and role-based permissions.

However, it's important to note that **a basic electronic signature** (e.g., a scanned image or typed name) **does not encrypt or secure** the document. It offers **no guarantee of confidentiality**, and the content remains readable to any viewer. By contrast, **advanced or qualified electronic signatures**, backed by valid digital certificates, **are legally equivalent to handwritten signatures** under EU regulations like eIDAS and are enforceable in courts.

2.4 – Digital vs. Handwritten Signature

Handwritten signatures are visually recognizable marks that can be forged or copied through manual or digital reproduction. Although they carry legal weight, their verification often requires forensic expertise, especially in dispute resolution scenarios.

Digital signatures, on the other hand, are **mathematically generated codes** that bind the signer's identity to the content of the document. They are tamper-evident and **non-transferable**—a signature created for one document cannot be reused or applied to another without invalidating the cryptographic binding.

In traditional contracts, signatures may appear on the final page, leaving earlier pages vulnerable to tampering. **Digital signatures cover the entire content**, such that **even a single character change** will result in verification failure. This ensures **document integrity** and prevents post-signature modifications.

Furthermore, **digital signatures are verifiable by anyone**, without relying on expert knowledge. Verification tools are widely available in PDF readers, document management systems, and email clients, making this technology both **accessible and transparent**.

2.5 – Purpose of Document Signing

Document signing serves multiple essential purposes in legal, organizational, and technical contexts:

- **Proof of Origin**: Confirms the **identity of the signer**. A unique signature can be attributed to an individual based on exclusive access to the private key or personal signing method.
- **Legal Authority**: Indicates that the signer understands the **legal implications** of the content and is willingly assuming responsibility for it. This is crucial in contracts, government filings, and declarations.
- **Explicit Consent or Agreement**: In regulated sectors, signatures represent formal **authorization or approval**, signifying that the signer has agreed to the document's terms or decisions.
- **Operational Efficiency**: Replaces manual processes, shortens approval cycles, reduces printing/scanning costs, and speeds up decision-making—especially important in remote or large-scale operations.

Digital signature technology surpasses traditional handwritten signing in every dimension—from automation and traceability to security and legal defensibility. In a world increasingly reliant on digital workflows and cross-border transactions, the digital signature has become **a cornerstone of trust** in the digital economy.

Chapter 3 – Mathematical Algorithms Used in Electronic Signatures

3.1 – Hash Functions and Message Digests

In practical applications, **public-key cryptographic algorithms**—while highly secure—can be computationally expensive, especially when applied to large datasets or high-frequency operations like digital signatures. To address performance limitations, cryptographic systems typically **combine public-key encryption with cryptographic hash functions**, allowing only a small, fixed-length digest of the message to be signed rather than the full message.

This hybrid approach provides strong security guarantees while optimizing performance.

Core Steps in Secure Digital Signing and Verification:

1. **Hashing**: A **cryptographic hash function** (e.g., SHA-256) is applied to the input message or document, producing a fixed-size string called a **digest** or hash value. This digest uniquely represents the content, such that even a one-bit change in the document alters the hash completely.

2. **Signing**: The hash is then **encrypted with the sender's private key**, producing the digital signature. This step links the signature to both the content and the identity of the signer.

3. **Transmission**: The original document and its digital signature are **sent together** to the recipient.

4. **Verification**: The recipient performs three tasks:

   - Recomputes the hash of the received document.
   - Decrypts the received signature using the sender's public key to recover the original hash.
   - Compares both hashes. If they match, the signature is deemed valid, and the document is verified as **authentic, unaltered, and signed by the rightful party**.

This mechanism provides **integrity, authentication, and non-repudiation**, which are fundamental pillars of digital security.

Widely Used Hash Algorithms:

Hash functions condense data into a compact representation while ensuring:

- **Determinism**: The same input always produces the same hash.
- **Preimage resistance**: It's computationally infeasible to reverse a hash back into its original input.
- **Collision resistance**: Two different inputs shouldn't produce the same hash.

Commonly used hash algorithms include:

- **MD2, MD4, MD5** – Designed by Ronald Rivest; now largely deprecated due to vulnerabilities.
- **SHA-1** – Once standard, now discouraged for cryptographic use due to collision attacks.
- **SHA-2 family (SHA-224, SHA-256, SHA-384, SHA-512)** – Currently recommended by NIST for digital signatures.
- **SHA-3** – A newer, Keccak-based algorithm, designed as a secure alternative to SHA-2.

3.2 – RSA Algorithm

RSA (Rivest–Shamir–Adleman) is one of the most important public-key cryptosystems and a foundational technology for secure digital communications. Developed at MIT in

1977, it is considered a benchmark for cryptographic security and is used in countless systems including TLS/SSL, email encryption, software licensing, and digital signatures.

Security Principle:

RSA relies on the computational difficulty of the **integer factorization problem**: given a large composite number *n*, it is practically impossible to determine its prime factors *p* and *q* in a reasonable amount of time.

Mathematical Foundations:

Let's define the key parameters:

1. **p and q** – Two large prime numbers (e.g., 1024-bit or 2048-bit), randomly chosen and kept secret.
2. **n = p × q** – The modulus used for both encryption and decryption; made public.
3. **φ(n) = (p−1)(q−1)** – Euler's totient function, used in key generation; not publicly revealed.
4. **PRIV (Private Key)** – A number chosen such that it is relatively prime to φ(n).
5. **PUB (Public Key)** – The modular inverse of the private key with respect to φ(n), computed using the Extended Euclidean Algorithm.
6. **M** – The message or document to be signed.
7. **H(M)** – The message digest, obtained by applying a secure hash function to *M*.

Signing and Verification:

- To sign: Encrypt **H(M)** with **PRIV**, producing the signature *S*.
- To verify: Decrypt *S* using **PUB** and compare the result with a newly computed **H(M)**.

Dual Use:

Unlike DSA or ElGamal (which are dedicated to signatures), RSA can also be used for **encryption and decryption**, enabling it to support secure message exchange and key distribution.

Practical Security:

RSA's strength depends on the key size:

- A 1024-bit key is considered **minimum secure**.
- 2048-bit and 3072-bit keys are standard for high-security environments.
- Keys smaller than 1024 bits are no longer considered safe due to advances in factoring techniques and computational power.

Secure File Transmission with RSA (SSFTP-style workflow)

Though RSA itself is not a file transfer protocol, it is often used **alongside symmetric encryption** in secure systems such as **SSFTP** (Secure Signed File Transfer Protocol):

1. Generate a message digest **H(M)** using a secure hash function.
2. Encrypt the digest with the **sender's private key** (digital signature).
3. Encrypt the signed digest with the **recipient's public key** (confidentiality).
4. Encrypt the original document with the recipient's public key.
5. Send both the **encrypted document and encrypted signature**.
6. The **recipient decrypts** the document with their **private key**.
7. Recalculates the hash of the decrypted document.
8. Decrypts the received signature with their **private key**.
9. Further decrypts the result using the **sender's public key**.
10. Compares the hash from step 7 with the one obtained in step 9. If they match, the signature is **authentic and the document is untampered**.

The RSA algorithm remains one of the most trusted methods for digital signatures and public-key encryption. Its rigorous mathematical foundations, proven resistance to cryptanalysis, and widespread support across platforms and protocols make it a cornerstone of modern cryptographic infrastructure.

3.3 – Elliptic Curve Digital Signature Algorithm (ECDSA)

While RSA has been the dominant public-key algorithm for decades, **Elliptic Curve Cryptography (ECC)** has emerged as a modern alternative offering **equivalent security with significantly smaller key sizes**. This makes ECDSA particularly attractive for resource-constrained environments such as mobile devices, IoT systems, and blockchain applications.

Mathematical Foundation:

ECDSA is based on the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**, which is computationally harder to solve than integer factorization (RSA's basis) for equivalent key sizes. This means:

- A **256-bit ECC key** provides security comparable to a **3072-bit RSA key**.
- Smaller keys result in **faster computations** and **reduced storage requirements**.

SECP256R1 Curve:

The most widely adopted curve for ECDSA is **SECP256R1** (also known as **P-256** or **prime256v1**), standardized by NIST. This curve is used in:

- **TLS/SSL certificates** for secure web communications
- **Bitcoin and Ethereum** blockchain transactions (though Bitcoin uses SECP256K1)
- **Smart cards and hardware security modules**
- **Mobile device authentication**

Key Properties of ECDSA:

- **Smaller signatures**: ECDSA signatures are typically 64 bytes (for 256-bit curves), compared to 256 bytes for RSA-2048.

- **Faster signing**: Signature generation is computationally efficient.
- **Non-deterministic by default**: Each signature includes a random component, though deterministic variants (RFC 6979) eliminate this randomness for reproducibility.

ECDSA vs. RSA Comparison:

Feature	RSA-2048	ECDSA (SECP256R1)
Security Level	~112 bits	~128 bits
Key Size	2048 bits	256 bits
Signature Size	256 bytes	64 bytes
Signing Speed	Moderate	Fast
Verification Speed	Very Fast	Moderate
Quantum Resistance	Vulnerable	Vulnerable

Both algorithms are currently considered secure for most applications, though **post-quantum cryptography** will eventually be needed as quantum computers advance.

Implementation Considerations:

ECDSA requires careful implementation to avoid vulnerabilities:

- **Random number generation** must be cryptographically secure (weak randomness led to the PlayStation 3 private key compromise).
- **Side-channel attacks** can leak private keys if not properly protected.
- **Deterministic ECDSA** (RFC 6979) eliminates randomness risks by deriving the nonce from the message and private key.

3.4 – Signature Algorithm Comparison

RSA – Based on the hardness of factoring large integers. - **Advantages**: Widely supported, simple to implement, dual-use (encryption + signatures) - **Disadvantages**: Large key sizes, slower key generation

ECDSA – Based on elliptic curve discrete logarithm problem. - **Advantages**: Smaller keys, faster operations, efficient for mobile/embedded systems - **Disadvantages**: More complex mathematics, requires careful implementation

DSA (Digital Signature Algorithm) – Standardized by NIST; uses discrete logarithms over finite fields. - **Advantages**: NIST standardized, signature-specific design - **Disadvantages**: Only for signatures (cannot encrypt), being superseded by ECDSA

ElGamal – A probabilistic encryption algorithm used in digital signature schemes. - **Advantages**: Semantic security, used in variants like Schnorr signatures - **Disadvantages**: Large signature sizes, less common in practice

Each of these schemes offers unique performance and security trade-offs. Modern systems increasingly favor **ECDSA** for its efficiency, while **RSA** remains dominant in legacy systems and scenarios requiring encryption capabilities.

Chapter 4 – How Electronic Signatures Work

Electronic signatures, while not inherently encrypting the content of the document, are built upon **cryptographic algorithms** that ensure **integrity, authenticity, and non-repudiation**. These signatures rely on mathematical principles from public-key cryptography and are governed by standardized protocols that define how the signing and verification processes occur.

Understanding Cryptographic Algorithms

To understand how digital signatures work, let's first break down what a cryptographic algorithm does. Imagine the message: **"Ana has apples"**. If we wanted to obscure it, a basic method might be to shift each letter by one in the alphabet, resulting in: **"Bob bsf bqqmft"**. This is known as **Caesar cipher**—a very simple form of encryption. However, such basic ciphers are trivially easy to crack.

In modern systems, we use **encryption keys** in combination with algorithms. A **key** is a string of bits (characters, numbers, symbols) that, when applied through a cryptographic algorithm, transforms readable data (plaintext) into an unreadable format (ciphertext). Only someone with the appropriate key can decrypt the message.

Symmetric Encryption

In **symmetric cryptography**, **the same key is used** to both encrypt and decrypt the data. This method is fast and suitable for large data volumes, but it has a critical flaw: **key distribution**. If an unauthorized party gains access to the key, they can both decrypt and forge messages.

Common symmetric encryption algorithms include:

- **DES (Data Encryption Standard)** – now obsolete due to its limited key size.
- **AES (Advanced Encryption Standard)** – the modern standard, with 128/192/256-bit key options.

Asymmetric Encryption (Public-Key Cryptography)

Used in digital signatures, **asymmetric encryption** involves two keys:

- A **private key**, known only to the signer, used to create the digital signature.
- A **public key**, distributed freely, used by others to verify the signature.

This model ensures that:

- Only the holder of the private key can sign a message.
- Anyone can verify the authenticity of the signature using the public key.

This forms the foundation of public-key algorithms like **RSA** and **DSA**, which are secure due to the mathematical difficulty of problems such as **factoring large numbers** or **solving discrete logarithms**.

Asymmetric cryptography is computationally intensive, so it's generally used **only to sign a hash** of a message, not the full message itself.

Creating and Verifying a Digital Signature

The digital signature process combines hashing and asymmetric encryption.

Steps to Sign a Document:

1. **Hashing the document**: A **cryptographic hash function** (like SHA-256) converts the document into a fixed-length digest (e.g., a 64-character alphanumeric string). This **digest is unique** to the content—any change in the document results in a completely different digest.

2. **Encrypting the hash with the private key**: The signer encrypts the digest using their **private key**, creating the digital signature. This binds the signature to both the document content and the identity of the signer.

3. **Transmitting the document**: The **original document and the digital signature** are sent together to the recipient.

Steps to Verify the Signature:

1. The recipient **decrypts the signature** using the sender's **public key**, extracting the original hash.

2. They **recalculate the hash** of the received document.

3. The two hashes are **compared**:

    - If they **match**, the document is authentic, has not been tampered with, and was signed by the private key holder.
    - If they **don't match**, the document has been altered or the signature is forged.

This process guarantees:

- **Integrity** – The data hasn't changed.
- **Authentication** – The signer's identity is confirmed.
- **Non-repudiation** – The signer cannot later deny their involvement.

Key Management in Practice

Every user in a digital signature system maintains a **key pair**:

- The **public key** can be freely shared (via digital certificates).
- The **private key** must be kept **secret and secure**.

Key properties:

- It should be **computationally infeasible** to derive the private key from the public one.
- The system must implement **key generation**, **digital signing**, and **signature verification** algorithms securely.

Smart Cards and Hardware Security

For higher assurance, private keys should **not be stored on standard hard drives**. Instead:

- **Smart cards or USB tokens** can securely store private keys.
- These devices often require a **PIN code** for activation, enabling **two-factor authentication**.
- During signing, the document's hash is sent to the card, which performs the encryption internally and returns the signature—preventing direct access to the key.

Chapter 5 – File Signing and Document Integrity

5.1 – The Need for File Signing

While message signing is suitable for short communications, real-world business operations require the ability to sign **complete documents, contracts, reports, and multimedia files**. File signing extends digital signature technology to any type of digital content—from PDF contracts to software executables—ensuring that the file remains unaltered from the moment of signing.

Key Differences from Message Signing:

- **Arbitrary file types**: Unlike text messages, files can be binary data (PDFs, images, videos, executables).
- **Large file sizes**: Files may range from kilobytes to gigabytes, making direct signing impractical.
- **Persistent storage**: File signatures must be stored separately and remain verifiable long after signing.

The Hash-Then-Sign Approach:

Rather than signing the entire file (which would be computationally prohibitive for large files), cryptographic systems employ a **hash-then-sign** methodology:

1. **Compute file hash**: The entire file is processed through a cryptographic hash function (SHA-256, SHA-512, or SHA3-256), producing a fixed-size digest.
2. **Sign the hash**: Only the hash is encrypted with the private key, creating the signature.
3. **Store metadata**: The signature, hash, algorithm details, and signer information are stored in a separate signature file (typically JSON format).

This approach offers several advantages:

- **Efficiency**: Signing a 32-byte hash is much faster than signing a 10MB file.
- **Fixed signature size**: Regardless of file size, the signature remains constant.
- **Verifiability**: Anyone with the public key can verify the signature by recomputing the hash and comparing.

5.2 – Metadata and Audit Trail Requirements

In legal and regulatory contexts, a digital signature must include **contextual information** beyond the cryptographic signature itself. Industry standards (eIDAS, ESIGN, UETA) require signatures to capture:

Required Metadata Fields:

- **Signer Name**: The legal identity of the individual applying the signature.
- **Organization**: The entity the signer represents (company, government agency, etc.).
- **Email**: Contact information for verification purposes.
- **Reason**: The purpose of signing (e.g., "Approval", "Acknowledgment", "Authorization").
- **Location**: Geographic location of the signer (city, country).
- **Timestamp**: Exact date and time of signing (ISO 8601 format with timezone).

Optional but Recommended:

- **Signature ID**: Unique identifier (UUID) for tracking and auditing.
- **Hash Algorithm**: Documents which hash algorithm was used (SHA256, SHA512, SHA3_256).
- **Key Algorithm**: Specifies RSA or ECDSA for cryptographic verification.

Why Metadata Matters:

Without metadata, a signature cannot be: - **Linked to a specific person** – verification becomes impossible. - **Audited** – no record of when/why the signature was applied. - **Verified in the future** – the algorithm information is essential for long-term signature validation.

This metadata forms the foundation of **non-repudiation**: the signer cannot later claim they didn't sign the document, as their name, timestamp, and explicit reason are permanently attached to the signature.

5.3 – PDF Digital Signatures

PDF (Portable Document Format) has become the de facto standard for document exchange due to its compatibility across platforms. Modern PDF readers support embedded digital signature fields that provide:

- **Visual indication** of signature status (valid, invalid, or expired).
- **Signature appearance** showing signer name, timestamp, and reason.
- **Signature placement** in designated areas of the document.
- **Incremental signing**: Multiple signatures can be added without invalidating previous ones.

PDF Signature Workflow:

1. **Create signature field**: The PDF author defines areas where signatures can be applied.
2. **Sign the document**: The signer opens the PDF, selects the signature field, and applies their digital signature.
3. **Visual appearance**: The signature field displays signer name, timestamp, and graphical indication of validity.
4. **Store signature**: The signature is embedded in the PDF as a digital object.
5. **Verify signature**: Recipients can click the signature to view details and verify authenticity.

Advantages of PDF Signatures:

- **Integrated workflow**: No separate signature files needed.
- **Visual feedback**: Users see signature status immediately.
- **Standardization**: PDF/A-3 and PAdES (PDF Advanced Electronic Signatures) provide compliance frameworks.
- **Long-term validation**: Archived signatures can be validated years after creation.

5.4 – File Verification Process

The verification of a signed file follows a strict, reproducible process:

Verification Algorithm:

```
1. Load signed file and signature metadata
2. Extract the original hash from signature
3. Extract the algorithm information (RSA/ECDSA, hash type)
4. Recompute hash of the current file using same algorithm
5. Load signer's public key
6. Decrypt the signature using public key to extract embedded hash
7. Compare:
   - Recomputed hash (step 4)
   - Decrypted hash (step 6)
8. If they match:
```

```
   - Document is AUTHENTIC (not tampered with)
   - Signature is VALID (created by private key holder)
   - Display signer name, timestamp, and reason
9. If they don't match:
   - Document has been MODIFIED or CORRUPTED
   - Signature is INVALID
   - Display warning and reject
```

Failure Scenarios:

- **Hash mismatch**: File was modified after signing.
- **Decryption failure**: Signature was corrupted or tampered with.
- **Missing public key**: Cannot verify without signer's public key.
- **Algorithm mismatch**: Signature used different algorithm than expected.
- **Expired timestamp**: Signature predates document creation (time fraud).

Chapter 6 – Multi-Signature Documents and Workflows

6.1 – The Multi-Signature Challenge

In real-world business processes, documents often require approval from **multiple parties** before taking effect. Examples include:

- **Contracts**: Both buyer and seller must sign.
- **Authorizations**: Multiple levels of management approval.
- **Legal proceedings**: Multiple parties validate the same document.
- **Financial transactions**: Multiple signatories confirm validity.

Challenges of Multi-Signature Systems:

1. **Signature Independence**: Each signer must apply their own signature independently without affecting others.
2. **Sequence Management**: Track the order in which signatures were applied and by whom.
3. **Document Integrity**: The document must remain unchanged through all signing rounds.
4. **Algorithm Flexibility**: Different signers may prefer different algorithms (RSA vs ECDSA).
5. **Timestamp Accuracy**: Precise timestamps prevent time-based attacks.
6. **Revocation Handling**: If one signer revokes their signature, the status of all signatures must be re-evaluated.

6.2 – Multi-Signature Architecture

JSON-Based Multi-Signature Structure:

```
{
  "file_path": "/path/to/document.pdf",
  "file_hash": "a1b2c3d4e5f6... (hash of original file)",
  "creation_timestamp": "2025-12-15T10:00:00Z",
  "signatures": [
    {
      "signature_id": "550e8400-e29b-41d4-a716-446655440000",
      "signature": "base64_encoded_signature_data...",
      "signer_name": "Alice Johnson",
      "organization": "Legal Department",
      "email": "alice@company.com",
      "reason": "Contract Approval",
      "location": "New York",
      "timestamp": "2025-12-15T10:15:30.123456Z",
      "algorithm": "ECDSA",
      "hash_algorithm": "SHA512",
      "file_hash": "a1b2c3d4e5f6..."
    },
    {
      "signature_id": "660f9501-f39c-52e5-b827-557766551111",
      "signature": "base64_encoded_signature_data...",
      "signer_name": "Bob Smith",
      "organization": "Finance Department",
      "email": "bob@company.com",
      "reason": "Financial Approval",
      "location": "Boston",
      "timestamp": "2025-12-15T11:45:00.654321Z",
      "algorithm": "RSA",
      "hash_algorithm": "SHA256",
      "file_hash": "a1b2c3d4e5f6..."
    }
  ]
}
```


Key Properties:

- **File Hash Consistency**: All signatures reference the **same file hash**, ensuring they're all signing the same document.
- **Independent Signing**: Each signer uses their own key pair and algorithm.
- **Chronological Order**: Timestamps track who signed when.
- **Metadata Preservation**: Each signature includes complete context (name, reason, location, etc.).
- **Audit Trail**: Complete history available for compliance and forensics.

6.3 – Sequential Signing Workflow

Process Steps:

1. **Initial Document**: File is created and hashed (SHA256).

2. **First Signature**: Alice signs using her ECDSA key + SHA512.
   - Signature is added to signatures array.
   - File hash is recorded.
   - Timestamp recorded.
3. **Second Signature**: Bob signs using his RSA key + SHA256.
   - New signature added to array.
   - **Same file hash** is verified (document must be unchanged).
   - New timestamp recorded.
4. **Verification Later**:
   - Recompute original file hash.
   - For each signature in array:
     - Extract the algorithm (ECDSA, RSA).
     - Extract the hash algorithm (SHA512, SHA256).
     - Verify the signature using detected algorithms.
     - Confirm the file hash hasn't changed.

Critical Insight:

The **file hash remains the same** throughout the process. This ensures: - No one tampered with the document between signatures. - Each signer approved the exact same content. - Verification can be done in any order.

6.4 – Automatic Algorithm Detection in Multi-Signature

Challenge: When verifying multi-signature documents, you don't know in advance: - Which algorithm each signer used (RSA or ECDSA). - Which hash algorithm they chose (SHA256, SHA512, SHA3_256).

Solution: **Automatic Algorithm Detection**

The system extracts this information from the signature metadata:

```
def detect_signature_algorithms(signature_data):
    """Extract and validate algorithms from signature JSON"""
    key_algorithm = signature_data.get('algorithm', 'RSA')      # RSA or
ECDSA
    hash_algorithm = signature_data.get('hash_algorithm', 'SHA256')  #
SHA256/512/3_256

    # Validate against supported algorithms
    if key_algorithm not in ['RSA', 'ECDSA']:
        key_algorithm = 'RSA'  # Fallback

    if hash_algorithm not in ['SHA256', 'SHA512', 'SHA3_256']:
        hash_algorithm = 'SHA256'  # Fallback

    # Validate security
```

```
    security_info = validate_algorithm_security(key_algorithm,
hash_algorithm)

    return {
        'key_algorithm': key_algorithm,
        'hash_algorithm': hash_algorithm,
        'security_info': security_info
    }
```

Verification Process with Auto-Detection:

```
For each signature in multi-sig array:
  1. Extract algorithm from signature JSON
  2. Auto-detect: "ECDSA" + "SHA512"
  3. Validate security: "Very Strong"
  4. Verify signature using detected algorithms
  5. If valid: Mark as ✓ AUTHENTIC
  6. If invalid: Mark as ✗ FORGED or MODIFIED
```

6.5 – Visual Signature Rendering for PDFs

When embedding multi-signatures in PDFs, each signature is displayed with:

```
┌──────────────────────────────────┐
│  DIGITALLY SIGNED                │
│  By: Alice Johnson               │
│  Organization: Legal Dept        │
│  Date: 2025-12-15 10:15:30 UTC   │
│  Reason: Contract Approval       │
│  Location: New York              │
│  Status: ✓ VALID                 │
└──────────────────────────────────┘
```

PDF Libraries Used:

- **PyPDF2**: For PDF manipulation and reading.
- **ReportLab**: For rendering signature appearance.

6.6 – Legal and Regulatory Considerations

eIDAS Regulation (EU): - Requires signer identification and timestamp. - Multi-signatures must preserve chronological order. - Each signature must independently verify.

ESIGN Act (USA): - Electronic signatures are legally binding. - Metadata ensures enforceability. - Audit trail is essential for disputes.

UETA (USA - Uniform Electronic Transactions Act): - Does not mandate specific technology. - Any cryptographic method acceptable if parties agree. - Digital signatures with metadata are prima facie evidence.

Industry Standards: - **PAdES** (PDF/A-3): Compliance framework for PDFs. - **XAdES**: XML Advanced Electronic Signatures. - **CMS (RFC 5652)**: Cryptographic Message Syntax.

6.7 – Attack Scenarios and Mitigations

Attack 1: Signature Forging - **How**: Attacker modifies document, then uses stolen private key to sign. - **Mitigation**: Secure private key storage, multi-factor authentication, key revocation.

Attack 2: Hash Collision Attack - **How**: Attacker finds two documents with same hash, swaps them. - **Mitigation**: Use SHA256+ (not MD5/SHA1), collision resistance proven mathematically.

Attack 3: Timestamp Manipulation - **How**: Attacker backdates signature to before document creation. - **Mitigation**: Use trusted timestamp authority (RFC 3161), verify timestamp against external source.

Attack 4: Signature Reuse - **How**: Attacker copies signature from Document A to Document B. - **Mitigation**: Signature is mathematically tied to specific document hash, cannot be reused.

Attack 5: Private Key Compromise - **How**: Attacker steals private key and forges signatures. - **Mitigation**: Hardware security modules, key revocation certificates, short key validity.

Chapter 7 – Automatic Algorithm Detection System

7.1 – What is Automatic Algorithm Detection?

Automatic Algorithm Detection is a feature that analyzes loaded cryptographic keys and signature data to automatically determine:

1. **Key Type**: Is this an RSA or ECDSA key?
2. **Hash Algorithm**: Which hash function was used (SHA256, SHA512, SHA3_256)?
3. **Security Strength**: Is this combination cryptographically sound?

This eliminates the need for users to manually select algorithms, reducing errors and improving usability.

7.2 – How Key Type Detection Works

Technical Process:

```
def detect_key_type(key_object):
    """Identify RSA or ECDSA from key object"""
    if isinstance(key_object, rsa.RSAPrivateKey):
        return 'RSA'
```

```
    elif isinstance(key_object, ec.EllipticCurvePrivateKey):
        return 'ECDSA'
    else:
        return None
```

What Happens:

1. User loads a PEM key file: `private_key.pem`
2. Python cryptography library parses the file.
3. Key object is examined using Python's `isinstance()` function.
4. Key's class inheritance reveals its type (RSA vs ECDSA).
5. Type is automatically returned.
6. GUI combobox updates without user action.

Security Guarantee:

This process is **cryptographically sound** because: - Key objects inherit from fixed classes in cryptography library. - No way to forge a key's type without breaking the library itself. - Type detection is instantaneous and 100% accurate.

7.3 – How Hash Algorithm Detection Works

Technical Process:

```
def detect_signature_algorithms(signature_data):
    """Extract and validate algorithms from signature JSON"""
    key_algorithm = signature_data.get('algorithm', 'RSA')
    hash_algorithm = signature_data.get('hash_algorithm', 'SHA256')

    # Validate against supported algorithms
    if hash_algorithm not in ['SHA256', 'SHA512', 'SHA3_256']:
        hash_algorithm = 'SHA256'

    # Return with security info
    security_info = validate_algorithm_security(key_algorithm,
hash_algorithm)
    return {
        'key_algorithm': key_algorithm,
        'hash_algorithm': hash_algorithm,
        'security_info': security_info
    }
```

What Happens:

1. User loads a signature file: `document.json`
2. JSON is parsed into dictionary.
3. System looks for `algorithm` field: "ECDSA"
4. System looks for `hash_algorithm` field: "SHA512"
5. Both are validated against supported values.

6. Security strength is determined: “Very Strong”
7. GUI comboboxes update automatically.
8. User sees: “Detected: ECDSA + SHA512 (Very Strong)”

7.4 – Security Validation Matrix

All 6 Algorithm Combinations Validated:

Algorithm	Hash	Security	Strength
RSA 2048	SHA256	✓ Secure	Strong (256-bit equiv.)
RSA 2048	SHA512	✓ Secure	Very Strong (512-bit margin)
RSA 2048	SHA3_256	✓ Secure	Strong (SHA3 latest)
ECDSA P-256	SHA256	✓ Secure	Strong (P-256 + SHA256)
ECDSA P-256	SHA512	✓ Secure	Very Strong (extra margin)
ECDSA P-256	SHA3_256	✓ Secure	Strong (modern standard)

Implementation:

```
def validate_algorithm_security(key_algorithm, hash_algorithm):
    """Validate algorithm combination"""
    security_matrix = {
        'RSA': {
            'SHA256': {'secure': True, 'strength': 'Strong (256-bit)'},
            'SHA512': {'secure': True, 'strength': 'Very Strong (512-bit)'},
            'SHA3_256': {'secure': True, 'strength': 'Strong (256-bit,
SHA3)'}
        },
        'ECDSA': {
            'SHA256': {'secure': True, 'strength': 'Strong (SECP256R1)'},
            'SHA512': {'secure': True, 'strength': 'Very Strong
(SECP256R1)'},
            'SHA3_256': {'secure': True, 'strength': 'Strong (SECP256R1)'}
        }
    }

    if key_algorithm in security_matrix:
        if hash_algorithm in security_matrix[key_algorithm]:
            return security_matrix[key_algorithm][hash_algorithm]

    return {'secure': False, 'strength': 'Unknown'}
```

7.5 – Algorithm Recommendations

For RSA: - “RSA 2048-bit: Suitable for most security needs” – Industry standard, widely supported.

For ECDSA: - "ECDSA SECP256R1: Modern, efficient elliptic curve" – Latest standard, better performance.

For SHA256: - "SHA256: Industry standard, widely supported" – Most compatible, sufficient security.

For SHA512: - "SHA512: Extra security margin, larger output" – Future-proof, maximum security.

For SHA3_256: - "SHA3_256: Latest standard, resistant to length extension" – Newest algorithm, cutting-edge.

User-Facing Recommendations:

When user loads ECDSA + SHA512 keys:

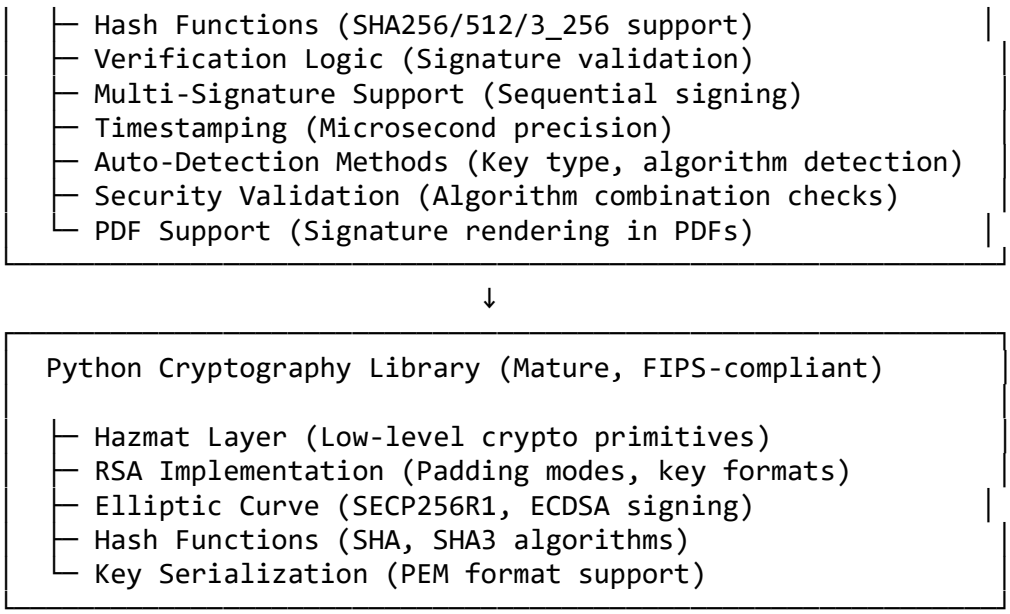
```
✓ Keys loaded successfully!

Auto-detected Algorithm: ECDSA
Security Level: Very Strong (SECP256R1 + SHA512)

Recommendations:
✓ ECDSA SECP256R1: Modern, efficient elliptic curve
✓ SHA512: Extra security margin, larger output
```

Chapter 8 – Implementation: Digital Signature Authentication System

8.1 – System Architecture

Two-Module Design:

```
┌─────────────────────────────────────────────────────────┐
│  digital_signature_gui.py (User Interface - Tkinter)    │
│                                                         │
│  ├─ Key Management (Generate/Load/Save)                 │
│  ├─ Algorithm Selection (RSA/ECDSA, SHA256/512/3)       │
│  ├─ Operation Modes (Message/File/Multi-Sign)           │
│  ├─ Metadata Entry (Signer info for compliance)         │
│  ├─ Auto-Detection (Detect algorithms automatically)    │
│  └─ GUI Updates (Comboboxes, buttons, messages)         │
└─────────────────────────────────────────────────────────┘
                            ↓
┌─────────────────────────────────────────────────────────┐
│  digital_signature_auth.py (Core Authentication Logic)  │
│                                                         │
│  ├─ Key Generation (RSA 2048, ECDSA P-256)              │
│  ├─ Signing Algorithms (RSA/ECDSA with PSS/ECDSA padding)  │
```



8.2 – Supported Cryptographic Algorithms

Key Algorithms: - RSA: 2048-bit keys with PKCS#8 serialization - ECDSA: SECP256R1 (P-256) curve

Hash Algorithms: - SHA256: 256-bit output (standard) - SHA512: 512-bit output (high security) - SHA3_256: 256-bit output (latest standard, Keccak-based)

Padding Schemes: - RSA: PSS (Probabilistic Signature Scheme) with MGF1 - ECDSA: ECDSA with deterministic nonce (RFC 6979 compatible)

8.3 – Operation Modes

Mode 1: Message Signing - Sign/verify text messages - Simple hash → sign workflow - Immediate verification in GUI

Mode 2: File Signing - Sign entire files (any type) - Metadata stored in JSON - Verifiable with original file

Mode 3: Multi-Signature - Multiple signers on same document - Preserves all signatures with timestamps - File hash consistency across signers - Auto-detection for each signature

8.4 – Key Management

Key Generation Process:

```
def generate_key_pair(self, algorithm='RSA'):
    if algorithm == 'RSA':
        self.private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )
```

```
    elif algorithm == 'ECDSA':
        self.private_key = ec.generate_private_key(ec.SECP256R1())

    self.public_key = self.private_key.public_key()
```

Key Storage: - PEM format (industry standard) - PKCS#8 for private keys - SubjectPublicKeyInfo for public keys - No encryption on saved keys (user responsibility)

Key Loading: - Automatic type detection (RSA or ECDSA) - Validation of key format - Automatic GUI updates

8.5 – Signature Data Persistence

Storage Format: JSON

```
{
  "signature_id": "550e8400-e29b-41d4-a716-446655440000",
  "signature": "base64_encoded_signature_data...",
  "file_hash": "a1b2c3d4e5f6...",
  "file_name": "document.pdf",
  "timestamp": "2025-12-15T10:15:30.123456Z",
  "algorithm": "ECDSA",
  "hash_algorithm": "SHA512",
  "signer_name": "Alice Johnson",
  "organization": "Legal Dept",
  "email": "alice@company.com",
  "reason": "Contract Approval",
  "location": "New York"
}
```

Advantages of JSON: - Human-readable (easy debugging/auditing) - Language-independent (portable) - Extensible (easy to add fields) - Standards-compliant (web-compatible)

8.6 – Security Considerations

Threats and Mitigations:

Threat	Risk	Mitigation
Private key theft	Signature forgery	Use hardware security modules
Weak randomness	Predictable signatures	Use Python's `os.urandom()`
Hash collision	Document swap attack	Use SHA256+ (proven collision-resistant)
Timestamp replay	Fake signing date	Validate timestamp against external source
Algorithm downgrade	Weaker security	All combinations validated as secure

Best Practices:

1. **Key Protection**: Store private keys securely, consider HSM for production.
2. **Backup**: Maintain encrypted backups of private keys.
3. **Revocation**: Implement key revocation if compromise suspected.
4. **Audit**: Log all signing operations with user, time, document.
5. **Compliance**: Document algorithms used for regulatory compliance.

8.7 – PDF Signature Support

PDF Signature Workflow:

```
def add_pdf_signature_footer(file_path, output_path, signature_data):
    """Add signature appearance to PDF"""
    # 1. Read original PDF
    pdf_reader = PdfReader(file_path)
    pdf_writer = PdfWriter()

    # 2. Create signature appearance text
    appearance = f"""
    DIGITALLY SIGNED
    By: {signature_data['signer_name']}
    Date: {signature_data['timestamp']}
    Reason: {signature_data['reason']}
    """

    # 3. Render appearance using ReportLab
    packet = io.BytesIO()
    can = canvas.Canvas(packet, pagesize=letter)
    can.drawString(50, 50, appearance)
    can.save()

    # 4. Add to last page of PDF
    overlay = PdfReader(packet)
    pdf_writer.add_page(pdf_reader.pages[0])
    pdf_writer.pages[0].merge_page(overlay.pages[0])

    # 5. Write output
    with open(output_path, 'wb') as f:
        pdf_writer.write(f)
```

8.8 – Future Enhancements

Proposed Features:

1. **Hardware Security Module (HSM) Integration**: Use smart cards/USB tokens for key storage.

2. **Timestamp Authority (TSA) Integration**: Use external trusted timestamp service (RFC 3161).
3. **Certificate Authority (CA) Integration**: Validate signer certificates against trusted CAs.
4. **Post-Quantum Cryptography**: Support lattice-based algorithms (when standardized).
5. **Batch Signing**: Sign multiple documents in one operation.
6. **Biometric Authentication**: Add fingerprint/face recognition for key access.
7. **Blockchain Integration**: Anchor signatures to blockchain for immutability.
8. **Advanced Formats**: Support XAdES, CAdES, CMS standards.

Conclusion

Digital signatures represent one of the most important innovations in cybersecurity and legal compliance. By combining:

- **Cryptographic mathematics** (RSA, ECDSA, hash functions)
- **Public-key infrastructure** (key generation and management)
- **Audit trail requirements** (timestamps, metadata, signer information)
- **Automatic intelligence** (algorithm detection and validation)

…we create a system that is simultaneously:

- **Secure**: Based on proven mathematical principles and industry standards
- **Usable**: Automates complexity, detects algorithms, prevents errors
- **Compliant**: Meets regulatory requirements (eIDAS, ESIGN, UETA)
- **Auditable**: Complete trail of who signed what and when
- **Future-proof**: Supports multiple algorithms, easily extensible

The implementation demonstrated in this project showcases how theoretical cryptographic concepts translate into practical, production-ready systems that solve real-world business challenges.

Bibliography

1. **RFC 3447** – PKCS #1: RSA Cryptography Specifications Version 2.1
2. **RFC 5280** – Internet X.509 Public Key Infrastructure Certificate and CRL Profile
3. **RFC 5652** – Cryptographic Message Syntax (CMS)
4. **RFC 3161** – Time-Stamp Protocol (TSP)
5. **RFC 6979** – Deterministic ECDSA
6. **FIPS 180-4** – Secure Hash Standard (SHS)
7. **FIPS 202** – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions
8. **FIPS 186-4** – Digital Signature Standard (DSS)
9. **SEC 2: Recommended Elliptic Curve Domain Parameters** – Standards for Efficient Cryptography Group
10. **ETSI TS 102 900** – XAdES (XML Advanced Electronic Signatures)
11. **eIDAS Regulation (EU 910/2014)** – Advanced Electronic Signatures and Qualified Electronic Signatures
12. **NIST SP 800-56B** – Recommendation for Pair-Wise Key Establishment Schemes
13. **NIST SP 800-175B** – Guideline for Using PKIX (RFC 5280)
14. **Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A.** – Handbook of Applied Cryptography (1997)
15. **Stallings, W.** – Cryptography and Network Security: Principles and Practice (6th Edition)
16. **Python Cryptography Library Documentation** – https://cryptography.io/