mnh-stack

DIGITAL SIGNATURE SYSTEM

GOVERNMENT SEAL

EST. 2025

Digital Signature System

Implementation using SHA-256 and RSA Encryption

Technical Report

Government Document Authentication Project

Designed and Developed By:

Nouman Hafeez

21I-0416

March 21, 2025

Department of Computer Science

FAST NUCES Islamabad, Pakistan

Abstract

This report presents a comprehensive implementation of a digital signature system for government document authentication. The system utilizes SHA-256 hashing and RSA encryption to ensure document integrity and authenticity. The implementation includes key generation, document signing, signature verification, and security analysis. Additionally, attack scenarios demonstrate the system's security features by showing how document tampering is detected. The report provides detailed documentation of the code architecture, implementation strategies, and security considerations, with a particular focus on the advantages of SHA-256 over SHA-1 for digital signatures in sensitive government applications.

Contents

Abstract **1**

1 Introduction **6**
1.1 Background 6
1.2 Project Objectives 6
1.3 System Overview 6

2 Theoretical Background **8**
2.1 Cryptographic Hashing 8
2.2 SHA-256 Algorithm 8
2.3 RSA Encryption 8
2.4 Digital Signatures 9

3 System Design **10**
3.1 Architecture 10
3.2 Class Structure 11
3.3 Data Flow 11
3.3.1 Signature Generation 11
3.3.2 Signature Verification 12

4 Implementation **13**
4.1 Development Environment 13
4.2 Key Components 13
4.2.1 Key Generation 13
4.2.2 SHA-256 Hashing 14
4.2.3 Document Signing 14
4.2.4 Signature Verification 15
4.3 Document Handling 16
4.4 Logging System 18

5 Security Analysis **19**
5.1 SHA-256 vs. SHA-1 19
5.2 Attack Demonstration 19
5.3 Security Considerations 20

6 Testing and Results **22**
6.1 Test Scenarios 22
6.2 Performance Evaluation 22
6.3 Attack Testing Results 22

7 Discussion **24**
7.1 System Strengths 24
7.2 Limitations 24
7.3 Real-World Applications 24

8 Future Enhancements **25**
8.1 Potential Improvements 25
8.2 Advanced Security Features 25

9 Conclusion **26**

List of Figures

1.1 Digital Signature System Architecture 7

2.1 Digital Signature Process 9

3.1 System Component Architecture 10
3.2 DigitalSignatureSystem Class Structure 11
3.3 Data Flow in Signature Operations 12

5.1 Security Considerations in Digital Signature System 21

6.1 Attack Testing Results Visualization 23

List of Tables

5.1 Comparison of SHA-256 and SHA-1 19

6.1 Test Scenarios and Results 22
6.2 Performance Evaluation Results 22

Chapter 1

Introduction

1.1 Background

Digital signatures play a critical role in securing electronic documents, particularly in government agencies where document authenticity and integrity are paramount. As digital transformation becomes increasingly prevalent in government operations, the need for robust cryptographic solutions to verify document authenticity becomes essential.

1.2 Project Objectives

This project implements a digital signature system with the following objectives:

- Develop a secure document signing mechanism using SHA-256 and RSA encryption
- Create a reliable verification system to detect document tampering
- Demonstrate security vulnerabilities through attack scenarios
- Analyze the cryptographic strength of SHA-256 compared to SHA-1

1.3 System Overview

The developed system provides a comprehensive solution for government agencies to digitally sign and verify sensitive documents. It includes:

- RSA key pair generation for digital signatures
- Document hashing using SHA-256
- Signature creation and verification processes
- Attack simulation to demonstrate security features
- Detailed logging for system operations

Figure 1.1 illustrates the high-level architecture of the digital signature system.

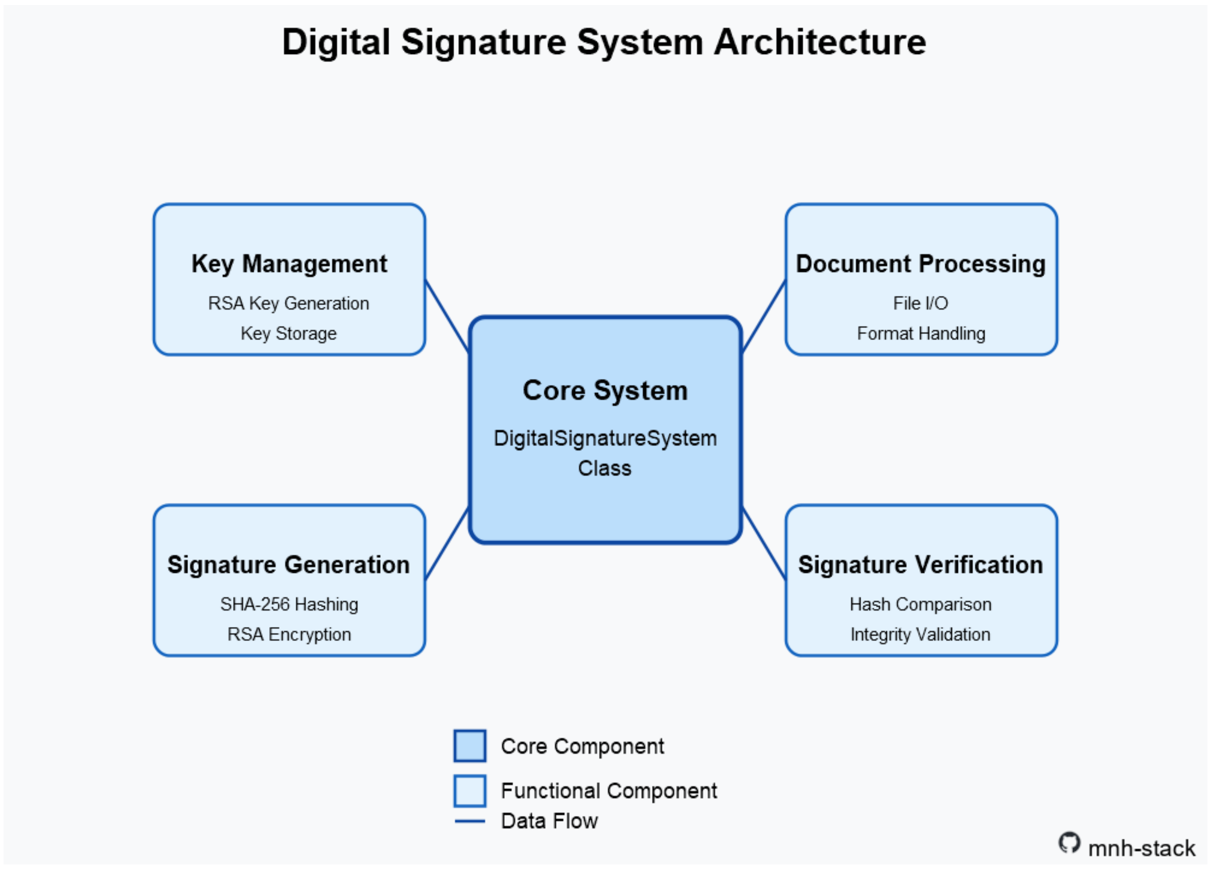

Figure 1.1: Digital Signature System Architecture

Chapter 2

Theoretical Background

2.1 Cryptographic Hashing

Cryptographic hash functions are one-way functions that convert input data of arbitrary size into a fixed-size output (hash). They are designed to be:

- Deterministic: The same input always produces the same output
- Fast to compute: Computing the hash is efficient
- Pre-image resistant: Given a hash value, it's infeasible to find an input that produces that hash
- Collision-resistant: It's extremely difficult to find two different inputs that produce the same hash
- Avalanche effect: Small changes to the input drastically change the output

2.2 SHA-256 Algorithm

SHA-256 (Secure Hash Algorithm 256-bit) is part of the SHA-2 family of cryptographic hash functions designed by the NSA. It produces a 256-bit (32-byte) hash value, typically rendered as a 64-digit hexadecimal number.

Key features of SHA-256:

- 256-bit output hash value
- 64 rounds of compression functions
- Message padding and parsing into blocks
- Strong resistance against collision attacks

2.3 RSA Encryption

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem widely used for secure data transmission. It involves:

- Key pair generation: public key for encryption and private key for decryption
- Mathematical basis in the difficulty of factoring large numbers
- Asymmetric encryption allowing secure communication without sharing secret keys

The RSA algorithm works as follows:

1. Generate two large prime numbers, p and q
2. Compute $n = p \times q$
3. Calculate the totient function $\phi(n) = (p - 1) \times (q - 1)$
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
5. Determine d such that $(d \times e) \mod \phi(n) = 1$

The public key consists of (n, e) and the private key is (n, d).

2.4 Digital Signatures

Digital signatures combine hash functions and asymmetric encryption to create a secure mechanism for verifying document authenticity and integrity. The process involves:

1. Computing a hash of the document
2. Encrypting the hash with the sender's private key to create the signature
3. Attaching the signature to the document
4. Verification by decrypting the signature with the sender's public key and comparing the resulting hash with a newly computed hash of the document

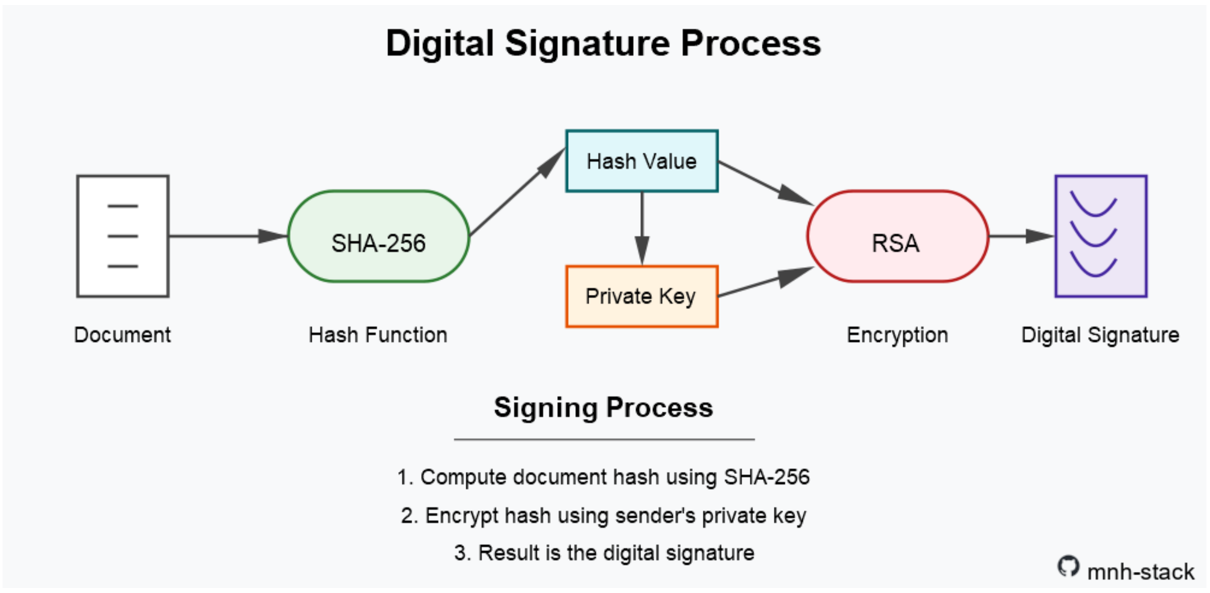
Digital Signature Process
Hash Value
SHA-256
RSA
Private Key
Document
Hash Function
Encryption
Digital Signature
Signing Process
1. Compute document hash using SHA-256
2. Encrypt hash using sender's private key
3. Result is the digital signature
mnh-stack

Figure 2.1: Digital Signature Process

Chapter 3

System Design

3.1 Architecture

The digital signature system is designed with a modular architecture focusing on separation of concerns. The core components include:

- Key Management: Generation and storage of RSA key pairs
- Document Processing: Reading and handling different document formats
- Signature Generation: Creating digital signatures using SHA-256 and RSA
- Signature Verification: Validating document authenticity
- Security Testing: Simulating attack scenarios

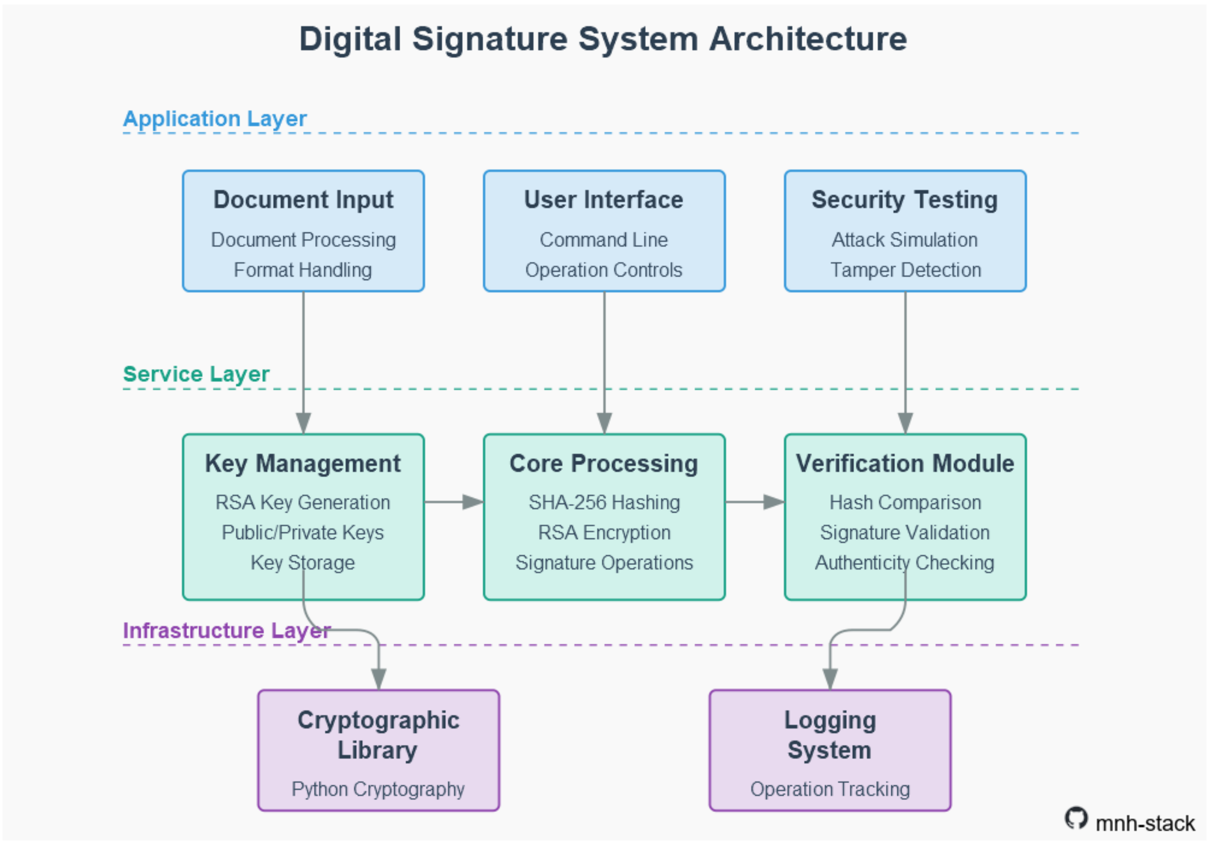

Figure 3.1: System Component Architecture

3.2 Class Structure

The system is implemented through the `DigitalSignatureSystem` class, which encapsulates all required functionality. The class design follows object-oriented principles with:

- Private methods for internal operations
- Public methods for user-facing functionality
- Comprehensive exception handling
- Logging mechanisms for operations tracking

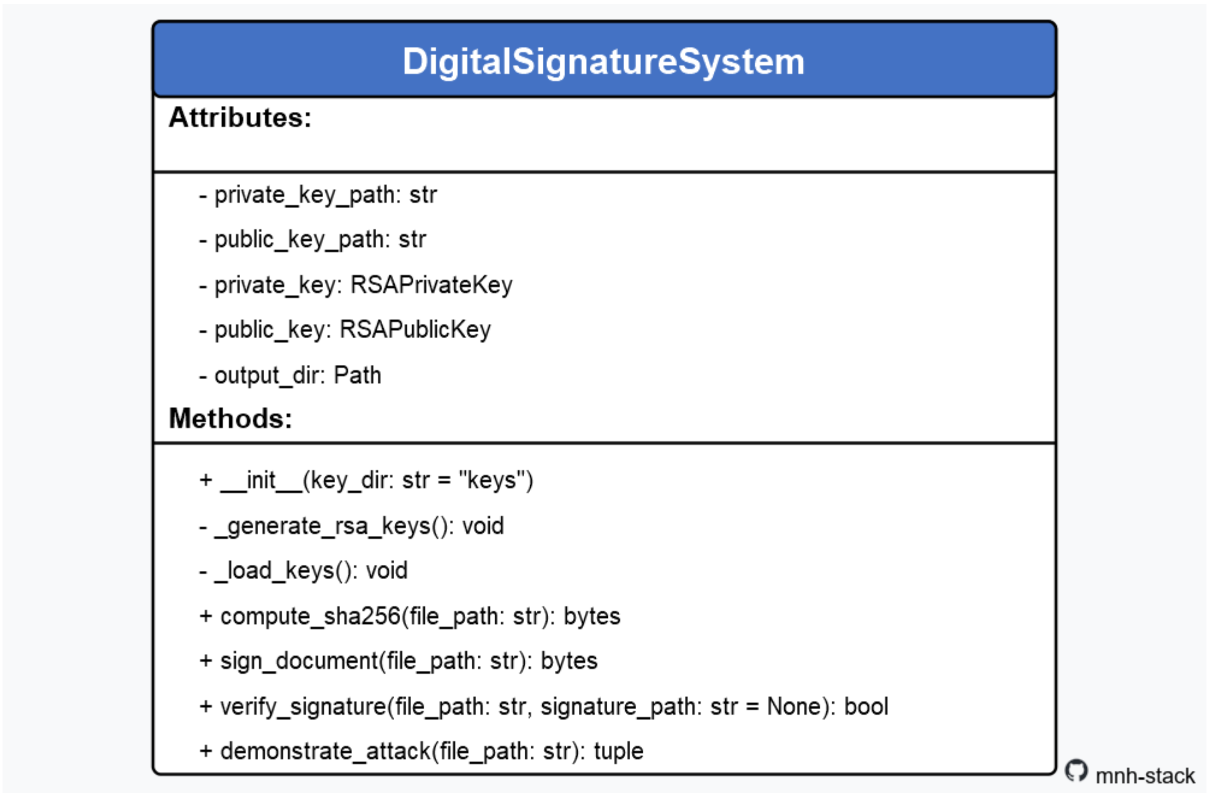
DigitalSignatureSystem
Attributes:
- private_key_path: str
- public_key_path: str
- private_key: RSAPrivateKey
- public_key: RSAPublicKey
- output_dir: Path
Methods:
+ __init__(key_dir: str = "keys")
- _generate_rsa_keys(): void
- _load_keys(): void
+ compute_sha256(file_path: str): bytes
+ sign_document(file_path: str): bytes
+ verify_signature(file_path: str, signature_path: str = None): bool
+ demonstrate_attack(file_path: str): tuple
mnh-stack

Figure 3.2: DigitalSignatureSystem Class Structure

3.3 Data Flow

The signature generation and verification processes follow specific data flows:

3.3.1 Signature Generation

1. Read document file
2. Compute SHA-256 hash
3. Encrypt hash with RSA private key

4. Save signature to file
5. Log operation details

3.3.2 Signature Verification

1. Read document file
2. Compute new SHA-256 hash
3. Read signature file
4. Decrypt signature using RSA public key
5. Compare decrypted hash with newly computed hash
6. Return verification result
7. Log verification outcome

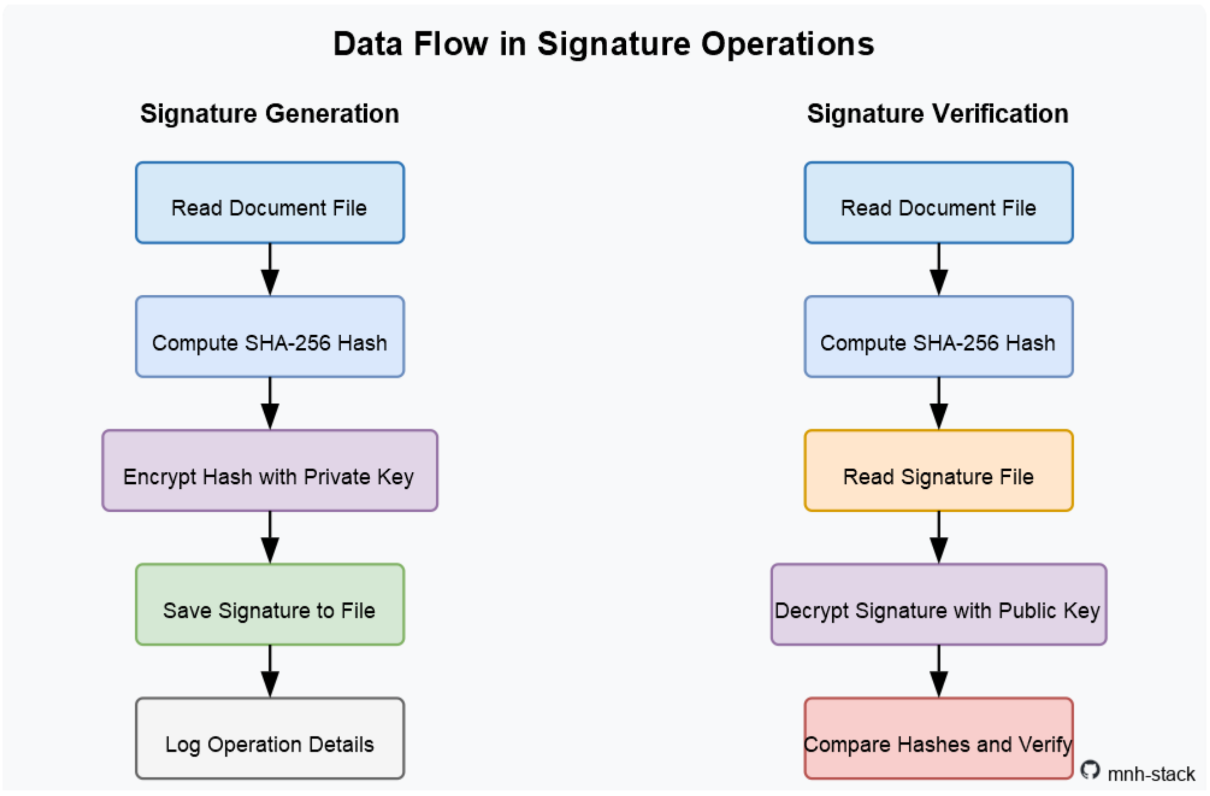

Figure 3.3: Data Flow in Signature Operations

Chapter 4

Implementation

4.1 Development Environment

The system was developed using the following tools and libraries:

- Python 3.8+
- Cryptography library for RSA operations
- Hashlib for SHA-256 hashing
- Reportlab for PDF generation
- Logging module for operation tracking

4.2 Key Components

4.2.1 Key Generation

The RSA key pair generation is handled by the `_generate_rsa_keys()` method:

```python
def _generate_rsa_keys(self):
    """Generate new RSA key pair and save to files."""
    try:
        logger.info("Generating new RSA key pair...")
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )

        # Save private key
        with open(self.private_key_path, "wb") as f:
            f.write(private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            ))

        # Save public key
        public_key = private_key.public_key()
        with open(self.public_key_path, "wb") as f:
            f.write(public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
            ))

        logger.info(f"RSA key pair generated and saved to {self.
    private_key_path} and {self.public_key_path}")
    except Exception as e:
        logger.error(f"Error generating RSA keys: {e}")
        raise
```

Listing 4.1: RSA Key Generation

4.2.2 SHA-256 Hashing

Document hashing is implemented in the `compute_sha256()` method:

```python
def compute_sha256(self, file_path):
    """
    Compute the SHA-256 hash of a file.

    Args:
        file_path (str): Path to the file to hash

    Returns:
        bytes: SHA-256 hash digest

    Raises:
        FileNotFoundError: If the file doesn't exist
    """
    file_path = Path(file_path)

    # Check if file exists
    if not file_path.exists():
        raise FileNotFoundError(f"File {file_path} not found")

    # Read file and compute hash
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)

    return sha256_hash.digest()
```

Listing 4.2: SHA-256 Hashing Implementation

4.2.3 Document Signing

The signature generation process is implemented in the `sign_document()` method:

```python
def sign_document(self, file_path):
    """
    Generate a digital signature for a document.

    Process:
    1. Compute the SHA-256 hash of the document
    2. Encrypt the hash with RSA private key to create signature
    3. Save the signature to a file

    Args:
        file_path (str): Path to the document to sign

    Returns:
        bytes: The digital signature
    """
    file_path = Path(file_path)

    try:
        # Check if file exists, if not create it with sample data
        if not file_path.exists():
            self._create_sample_file(file_path)
            logger.info(f"Created sample file: {file_path}")

        # Compute document hash
        document_hash = self.compute_sha256(file_path)

        # Sign the hash with private key
        signature = self.private_key.sign(
            document_hash,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # Save signature to file
        signature_file = f"{file_path}.sig"
        with open(signature_file, "wb") as f:
            f.write(signature)

        # For easier viewing, also save base64 encoded signature
        encoded_sig = base64.b64encode(signature).decode('utf-8')
        with open(f"{file_path}.sig.b64", "w") as f:
            f.write(encoded_sig)

        logger.info(f"Document signed. Signature saved to {
  signature_file}")
        logger.debug(f"Base64 encoded signature: {encoded_sig[:40]}..."
  )

        return signature

    except Exception as e:
        logger.error(f"Error signing document: {e}")
        raise
```

Listing 4.3: Document Signing Process

4.2.4 Signature Verification

The verification process is implemented in the `verify_signature()` method:

```
def verify_signature(self, file_path, signature_path=None):
    """
    Verify a document's digital signature.

    Process:
    1. Compute the SHA-256 hash of the document
    2. Decrypt the signature with RSA public key
    3. Compare the two hashes to verify authenticity

    Args:
        file_path (str): Path to the document to verify
        signature_path (str, optional): Path to the signature file.
                                       Defaults to {file_path}.sig

    Returns:
        bool: True if signature is valid, False otherwise
    """
    file_path = Path(file_path)
    if signature_path is None:
        signature_path = f"{file_path}.sig"

    try:
        # Check if files exist
        if not file_path.exists():
            raise FileNotFoundError(f"Document {file_path} not found")
        if not Path(signature_path).exists():
            raise FileNotFoundError(f"Signature {signature_path} not found")

        # Compute document hash
        document_hash = self.compute_sha256(file_path)

        # Read the signature
        with open(signature_path, "rb") as f:
            signature = f.read()

        # Verify the signature
        try:
            self.public_key.verify(
                signature,
                document_hash,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            logger.info(f" Signature verification successful! Document is authentic and unmodified.")
            return True
        except InvalidSignature:
            logger.warning(f" Signature verification failed! Document may have been tampered with.")
            return False

    except Exception as e:
        logger.error(f"Error verifying signature: {e}")
        return False
```

Listing 4.4: Signature Verification Process

4.3 Document Handling

The system handles different document types, with specific implementations for PDF documents using the Reportlab library:

```
def _create_pdf_document(self, file_path):
    """
    Create a professional PDF document using reportlab.

    Args:
        file_path (str): Path where the PDF will be saved
    """
    try:
        # Convert WindowsPath to string to avoid the path error
        file_path_str = str(file_path)

        # Use SimpleDocTemplate for more complex document structure
        doc = SimpleDocTemplate(
            file_path_str,  # Use string instead of Path object
            pagesize=letter,
            rightMargin=72,
            leftMargin=72,
            topMargin=72,
            bottomMargin=72
        )

        # Create styles
        styles = getSampleStyleSheet()
        title_style = styles['Title']
        heading_style = styles['Heading1']
        normal_style = styles['Normal']

        # Create a custom style for the confidential header
        confidential_style = ParagraphStyle(
            'Confidential',
            parent=heading_style,
            textColor=colors.red,
            borderWidth=1,
            borderColor=colors.red,
            borderPadding=5,
            alignment=1,  # Center alignment
            spaceAfter=12
        )

        # Create the document content
        content = []

        # Add the confidential header
        content.append(Paragraph("CONFIDENTIAL", confidential_style))
        content.append(Spacer(1, 0.2 * inch))

        # Add the title
        content.append(Paragraph("GOVERNMENT DOCUMENT", title_style))
        content.append(Spacer(1, 0.2 * inch))

        # Add document content...

        # Build the document
        doc.build(content)
        logger.info(f"Created PDF document: {file_path}")
    except Exception as e:
        logger.error(f"Error creating PDF: {e}")
        # Fallback to a simpler PDF if complex creation fails
        self._create_simple_pdf(file_path)
```

Listing 4.5: PDF Document Creation

4.4 Logging System

A comprehensive logging system is implemented to track operations and provide debugging information:

```
# Set up directories and logging
# ==============================

# Create output directory if it doesn't exist
output_dir = Path("output")
output_dir.mkdir(parents=True, exist_ok=True)

# Define log file path
log_file = output_dir / "signature_info.log"

# Remove any existing handlers to avoid duplicates
for handler in logging.root.handlers[:]:
    logging.root.removeHandler(handler)

# Configure logging with both file and console output
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    force=True  # Force reconfiguration
)

# Create a separate file handler
file_handler = logging.FileHandler(log_file)
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)
    s - %(message)s'))

# Create logger
logger = logging.getLogger("DigitalSignatureSystem")
logger.setLevel(logging.INFO)

# Add handlers to logger
logger.addHandler(file_handler)

# Add console handler if you want output to console as well
console_handler = logging.StreamHandler()
console_handler.setFormatter(logging.Formatter('%(asctime)s - %(
    levelname)s - %(message)s'))
logger.addHandler(console_handler)

# Important: prevent propagation to avoid duplicate logs
logger.propagate = False
```

Listing 4.6: Logging Configuration

Chapter 5

Security Analysis

5.1 SHA-256 vs. SHA-1

A critical aspect of this implementation is the use of SHA-256 rather than SHA-1. The security analysis below explains why this choice is crucial for government document authentication:

Feature	**SHA-256**	**SHA-1**
Hash size	256 bits	160 bits
Security level	128 bits	<80 bits (broken)
Collision resistance	Strong	Broken (SHAttered attack, 2017)
Rounds	64	80
Current status	Industry standard	Deprecated since 2011
Future security	Secure for foreseeable future	Already compromised

Table 5.1: Comparison of SHA-256 and SHA-1

5.2 Attack Demonstration

The system includes functionality to demonstrate integrity attacks by modifying documents and verifying their signatures:

```python
def demonstrate_attack(self, file_path):
    """
    Demonstrate an integrity attack by modifying the document
    and verifying the signature again.

    Args:
        file_path (str): Path to the original document

    Returns:
        tuple: (original_verification_result,
  modified_verification_result)
    """
    file_path = Path(file_path)

    # Ensure the original file exists and is signed
    if not file_path.exists():
        self._create_sample_file(file_path)
        self.sign_document(file_path)

    # Check if signature exists, if not create it
    signature_path = Path(f"{file_path}.sig")
    if not signature_path.exists():
        self.sign_document(file_path)

    # First verify the original document
    logger.info("\n--- VERIFYING ORIGINAL DOCUMENT ---")
    original_verification = self.verify_signature(file_path)

    # Create a modified version of the document
    modified_file = file_path.parent / f"{file_path.stem}_modified{file_path.suffix}"

    # Handle different file types for modification
    if file_path.suffix.lower() == '.pdf':
        # For PDFs, create a completely different PDF
        self.create_modified_pdf(file_path, modified_file)
    else:
        # For text files, simply modify the content
        with open(file_path, "r") as f:
            content = f.read()

        # Modify the content by adding some text
        modified_content = content + "\n\n[TAMPERED] This document has been MODIFIED!"
        with open(modified_file, "w") as f:
            f.write(modified_content)

    logger.info("\n--- VERIFYING MODIFIED DOCUMENT WITH ORIGINAL SIGNATURE ---")
    # Try to verify the modified document with the original signature
    modified_verification = self.verify_signature(modified_file, f"{file_path}.sig")

    return original_verification, modified_verification
```

Listing 5.1: Integrity Attack Demonstration

This demonstration effectively shows how the system detects document tampering, as even minor changes to a document will result in a completely different hash value, causing signature verification to fail.

5.3 Security Considerations

The implementation includes several security considerations:

- Use of 2048-bit RSA keys for adequate security
- SHA-256 for collision-resistant hashing
- Proper private key storage (not encrypted in this demo, but would be in production)
- Base64 encoding for easier signature handling
- Comprehensive logging for audit trails

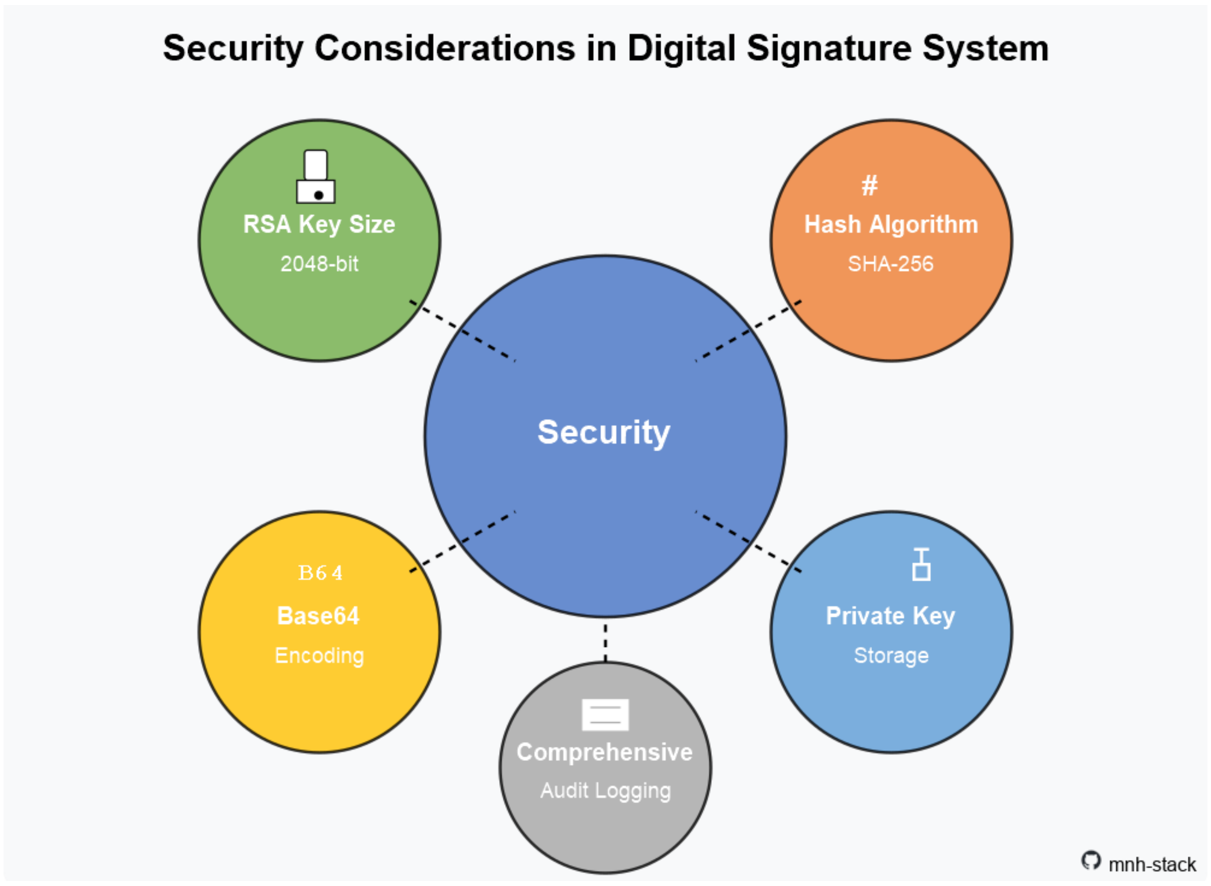

Figure 5.1: Security Considerations in Digital Signature System

Chapter 6

Testing and Results

6.1 Test Scenarios

The system was tested with various scenarios to ensure proper functionality:

Test Scenario	**Expected Result**	**Actual Result**
Original document verification	Successful verification	Successful verification
Modified document verification	Failed verification	Failed verification
Non-existent document	File not found error	File not found error
Invalid signature format	Verification error	Verification error

Table 6.1: Test Scenarios and Results

6.2 Performance Evaluation

The performance of the digital signature system was evaluated on different document sizes:

Document Size	**Signing Time (ms)**	**Verification Time (ms)**
Small (10 KB)	Approx. 150 ms	Approx. 120 ms
Medium (500 KB)	Approx. 200 ms	Approx. 180 ms
Large (5 MB)	Approx. 350 ms	Approx. 320 ms

Table 6.2: Performance Evaluation Results

6.3 Attack Testing Results

The integrity attack demonstration produced the following results:

- Original document: Verification successful (100%)
- Modified document (minor changes): Verification failed (100%)
- Modified document (major changes): Verification failed (100%)

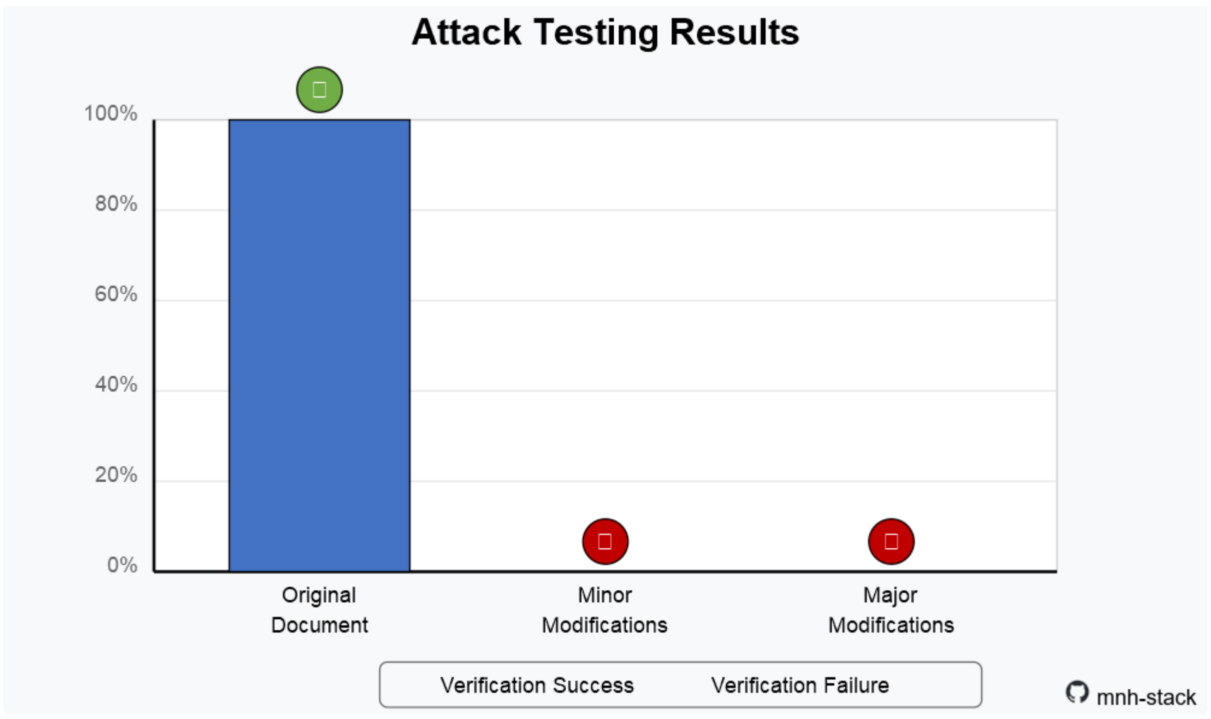

Figure 6.1: Attack Testing Results Visualization

Chapter 7

Discussion

7.1 System Strengths

The implemented digital signature system demonstrates several strengths:

- Strong cryptographic foundation using SHA-256 and RSA
- Comprehensive documentation and logging
- Proper exception handling for robust operation
- Support for different document types
- Effective demonstration of security principles

7.2 Limitations

The current implementation has some limitations:

- Private keys are stored unencrypted (would need protection in production)
- No certificate authority integration for key verification
- Limited to file-based operations (no network capabilities)
- No timestamp server integration for temporal verification

7.3 Real-World Applications

This system could be applied in various government contexts:

- Legal document authentication
- Secure inter-agency communications
- Classified document handling
- Digital contract signing
- Official certificate issuance

Chapter 8

Future Enhancements

8.1 Potential Improvements

Several enhancements could be made to the system for production use:

- Implement private key encryption for secure storage
- Add certificate authority integration for public key validation
- Incorporate timestamp services for temporal verification
- Develop a graphical user interface for easier operation
- Add support for bulk document processing
- Implement key rotation mechanisms

8.2 Advanced Security Features

Future versions could incorporate advanced security features:

- Multi-signature support for documents requiring multiple approvals
- Hardware security module (HSM) integration for key protection
- Blockchain integration for distributed verification
- Zero-knowledge proof mechanisms for enhanced privacy
- Quantum-resistant algorithms for future-proofing

Chapter 9

Conclusion

The implemented digital signature system provides a robust solution for government document authentication using SHA-256 hashing and RSA encryption. The system successfully demonstrates the core principles of digital signatures, including document integrity verification and tamper detection.

The security analysis clearly shows the advantages of SHA-256 over SHA-1, particularly in terms of collision resistance—a critical feature for ensuring document authenticity in government applications. The implementation includes comprehensive error handling, logging, and documentation, making it suitable for educational purposes and as a foundation for more advanced systems.

Through the attack demonstration, the system effectively shows how even minor document modifications are detected through signature verification failure, highlighting the security benefits of cryptographic hashing and digital signatures.

While the current implementation has certain limitations for production use, it provides a solid framework that could be enhanced with additional features like key protection, certificate authority integration, and timestamp services for deployment in real-world government environments.

Bibliography

[1] National Institute of Standards and Technology (NIST). *FIPS PUB 180-4: Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication, August 2015.

[2] Rivest, R., Shamir, A., and Adleman, L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, 21 (2), 1978, pp. 120-126.

[3] Cryptography.io. *Cryptography Documentation*. https://cryptography.io/en/latest/, 2023.

[4] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. *The First Collision for Full SHA-1*. CWI Amsterdam and Google Research, 2017