Secure Digital Signature Server: Design, Implementation, and Security Analysis

Prashant Gautam – Simone Maccarrone – Reza Almassi

Professor: **GIANLUCA DINI**

Academic Year 2024–2025

Contents

Chapter 1

Introduction

1.1 Background and Motivation

In the rapidly evolving landscape of information technology, the security and integrity of digital communications have become essential requirements for organizations of every scale. One of the foundational pillars of modern cybersecurity is the use of *digital signatures*. A digital signature serves not only to confirm the identity of the sender but also to ensure that the content of a message or document has not been altered in transit. Digital signatures provide three fundamental security properties:

1. **Authenticity**: The assurance that the document originates from a verified source.
2. **Integrity**: The guarantee that the document has not been altered since it was signed.
3. **Non-repudiation**: The impossibility for the signer to later deny having signed the document.

With the proliferation of electronic documents, contracts, and digital workflows, organizations are increasingly reliant on robust mechanisms for managing digital identities and ensuring the trustworthiness of their digital transactions.

Digital signature servers (DSS) have emerged as an essential component for securely managing the digital signing process in enterprise environments. Instead of placing the burden of key management and cryptographic operations on end-users—who may lack the necessary expertise—a DSS centralizes these responsibilities, enforces policy compliance, and enables auditing of signature events.

1.2 Problem Statement

While the cryptographic foundations for digital signatures—such as public-key infrastructure (PKI), RSA, and ECDSA algorithms—are well established, their secure deployment in real-world organizations remains challenging. Major problems include:

- **Key management**: Secure generation, storage, and rotation of cryptographic keys.
- **User authentication**: Reliable verification of user identity prior to any signing operation.
- **Secure communication**: Preventing eavesdropping, replay, and man-in-the-middle attacks during sensitive transactions.
- **Regulatory compliance**: Meeting the standards imposed by laws such as eIDAS (EU), ESIGN (USA), and other frameworks.

A DSS must address these challenges while providing usability, scalability, and flexibility for future improvements.

1.3 Project Objectives

The primary goal of this project is to design and implement a **Digital Signature Server (DSS)** tailored for use within a hypothetical organization, following the requirements set forth by the Cybersecurity course at the University of Pisa. The specific objectives are:

- To design and implement a secure server-side application for generating, storing, and using public/private key pairs for digital signatures.
- To provide robust user registration and authentication workflows, ensuring that only authorized individuals can request digital signatures.
- To enable users to digitally sign arbitrary documents via a secure protocol, guaranteeing both the confidentiality and integrity of communications.
- To store all private keys in encrypted form, minimizing risk in the event of server compromise.
- To fulfill essential cryptographic properties: confidentiality, authenticity, integrity, non-repudiation, and perfect forward secrecy (PFS).

- To deliver a comprehensive documentation, including protocol descriptions, message formats, and security analysis, that can serve as a reference for further academic or professional work.

1.4 Relevance of Digital Signatures in Cybersecurity

Digital signatures are indispensable in securing modern digital interactions. They are employed in diverse scenarios such as secure email (e.g., S/MIME), code signing for software distribution, blockchain transactions, secure document management, and more. As organizations digitize more of their workflows, the need for scalable, user-friendly, and auditable digital signature solutions has grown accordingly.

The adoption of digital signature servers allows organizations to:

- **Reduce risk** by preventing weak or compromised key storage practices.
- **Streamline compliance** with regulations and standards (e.g., GDPR, eIDAS).
- **Centralize auditing** of digital signature events for traceability and investigation.
- **Enable secure remote work** by providing digital signing services over secure channels.

1.5 Overview of the Digital Signature Server (DSS) Project

This project implements a Digital Signature Server with the following core features:

1. **User Registration:** Employees are registered in the DSS system and provided with secure authentication credentials.
2. **Key Generation:** For each user, a public/private key pair is generated by the server; private keys are stored encrypted.

3. **User Authentication:** All user interactions with the server require authentication over a secure channel, using strong password hashing and server certificate verification.

4. **Document Signing:** Authenticated users may request the DSS to sign documents on their behalf, with the server returning the digital signature.

5. **Public Key Retrieval:** Any user may retrieve the public key of another registered user, enabling third-party verification of digital signatures.

6. **Key Deletion:** Users may request the deletion of their key pairs, which is enforced securely and irreversibly by the DSS.

7. **Secure Communication:** All operations occur over an encrypted channel using TLS, configured to ensure PFS, integrity, non-replay, and non-malleability.

The DSS is implemented using modern cryptographic libraries in Python, following best practices for key storage, password handling (hashing with bcrypt), and secure communication (TLS with self-signed certificates for development purposes).

1.6 Structure of This Document

The remainder of this report is organized as follows:

- **Chapter 2: System Architecture and Design Choices**
  This chapter provides an in-depth description of the DSS architecture, including a high-level diagram, the choice of technologies, and the rationale for major design decisions.

- **Chapter 3: Authentication and Communication Protocols**
  This chapter describes the user registration, authentication, and document signing workflows, with precise step-by-step protocol descriptions.

- **Chapter 4: Message Formats**
  This chapter defines the data formats of all exchanged messages, using JSON representations for clarity and precision.

- **Chapter 5: Sequence Diagrams**
  This chapter visualizes the communication protocols using sequence diagrams, making the flow of information between user and server explicit.

- **Chapter 6: Security Analysis**
  Here, the document assesses how the DSS meets its security objectives, identifies potential threats, and justifies the chosen countermeasures.

- **Chapter 7: Future Improvements**
  This chapter suggests potential enhancements, such as hardware security modules, multi-factor authentication, and advanced user management.

- **Chapter 8: Conclusion**
  A summary of the project's achievements, limitations, and potential impact.

This comprehensive documentation is intended to support both academic assessment and practical deployment of digital signature technology in secure organizational environments.

Chapter 2

System Architecture and Design Choices

2.1 Overview of the DSS Architecture

The architecture of the **Digital Signature Server (DSS)** is meticulously designed to prioritize security, modularity, and future extensibility. The DSS is built on a robust client-server paradigm, in which organizational users (employees) interact with a trusted central server through secure channels to perform all operations related to key management and digital signing.

Key architectural components:

- **DSS Server:** Implements all critical logic, including user registration, authentication, key management, document signing, and secure communication. Exposes REST-like HTTP endpoints, protected by TLS.
- **Clients:** User applications (or scripts) that authenticate and interact with the server for key and signing services.
- **Secure Channel:** All communications occur over HTTPS, leveraging server-side certificate authentication to ensure confidentiality, integrity, and perfect forward secrecy.
- **Encrypted Key Storage:** The DSS persistently stores each user's private key in encrypted form, bound to their password.

FIGURE

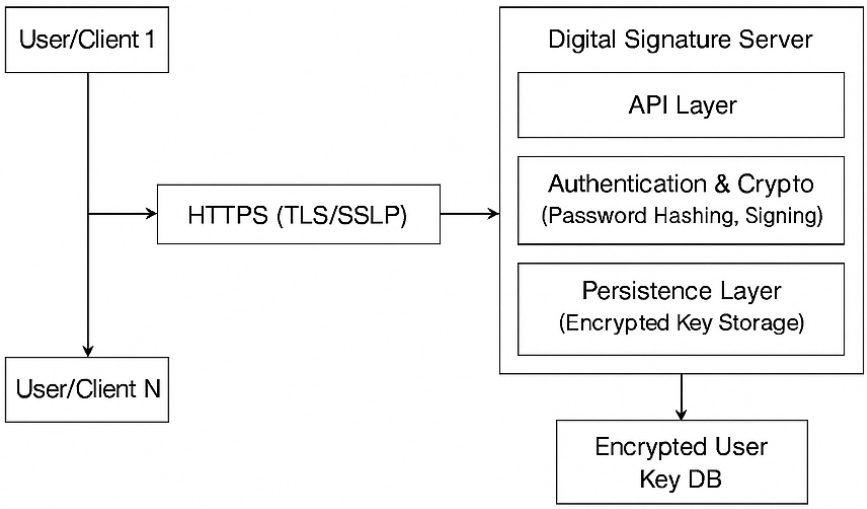

Figure 1: High-Level Architecture of the Digital Signature Server System

Figure 2.1: High-Level Architecture of the Digital Signature Server System

2.2 Technology Choices and Rationale

- **Programming Language: Python 3** is used for rapid prototyping, mature cryptographic libraries, and clear code structure.
- **Web Framework: Flask** enables a lightweight yet extensible API layer for all DSS operations.
- **Cryptography:** The **cryptography** package provides robust, industry-standard primitives for RSA key generation, encryption, and digital signatures.

- **Password Hashing: bcrypt** protects passwords using a strong, slow hash that is resistant to brute-force and rainbow-table attacks.
- **Persistent Storage:** A Python **`shelve`** database is utilized for initial development; it can be replaced with more advanced solutions for production deployments.
- **TLS/SSL:** The server uses a self-signed certificate for development. In production, a certificate from a trusted CA is mandatory to avoid MITM vulnerabilities.

2.3 Layered Structure and Core Components

The DSS server software is structured into several logical layers:

- **API Layer:** Implements HTTPS endpoints for all DSS operations.
- **Authentication Layer:** Manages secure user credential validation and session initiation.
- **Cryptographic Layer:** Responsible for RSA key generation, digital signing, and private key encryption/decryption.
- **Persistence Layer:** Ensures secure storage and retrieval of user data, public/private keys, and hashed passwords.

Example of API Layer Code:

Listing 2.1: User Registration Endpoint

```
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    if not username or not password:
        return jsonify({'error': 'Username␣and␣password␣
           are␣required.'}), 400

    with shelve.open(DB_FILE) as db:
        if username in db:
            return jsonify({'error': 'User␣already␣
               exists.'}), 409

```

```
        # Hash password
        pw_hash = bcrypt.hashpw(password.encode(),
            bcrypt.gensalt())

        # Generate key pair
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        public_key = private_key.public_key()
```

2.4 Design Decisions for Security

2.4.1 Private Key Storage and Encryption

All private keys are stored encrypted with a key derived from the user's password, ensuring that even if the database is compromised, the actual private keys remain protected.

Example Code for Private Key Encryption:

Listing 2.2: Private Key Encryption

```
private_bytes = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.
        BestAvailableEncryption(password.encode())
)
```

2.4.2 Password Handling

User passwords are **never stored in plaintext**. The registration process employs bcrypt to hash passwords, storing only the hashed version in the database. Authentication compares the bcrypt hash of the submitted password with the stored hash.

Password Hashing and Database Storage Example:

Listing 2.3: Password Hashing and Storage

```
pw_hash = bcrypt.hashpw(password.encode(), bcrypt.
    gensalt())
db[username] = {
```

```
    'pw_hash': base64.b64encode(pw_hash).decode(),
    'private_key': base64.b64encode(private_bytes).
        decode(),
    'public_key': base64.b64encode(public_bytes).decode
        (),
}
```

2.4.3 Secure Communication: TLS with Perfect Forward Secrecy

All API communication between clients and the DSS is performed over HTTPS, with a dedicated certificate and enforced secure ciphers. In production, ECDHE or similar suites are required for **Perfect Forward Secrecy (PFS)**.

Example Server Startup Code:

Listing 2.4: Flask Server with SSL

```
if __name__ == '__main__':
    app.run(
        host='0.0.0.0',
        port=5000,
        ssl_context=('server-cert.pem', 'server-key.pem'
            )
    )
```

2.5 Rationale for Chosen Technologies

- **Python cryptography library:** Ensures use of safe, modern cryptographic primitives, including PKCS#8 private key encryption.
- **bcrypt:** Designed for password hashing, including salt and work factor, making offline brute-force attacks infeasible.
- **shelve:** Eases prototyping and development, with future-proofing for migration to more advanced or encrypted backends.
- **Flask:** Simple and flexible for developing RESTful APIs and prototyping secure server behavior.

2.6 Security Properties Ensured

- **Confidentiality**: Achieved through TLS encryption and private key protection.
- **Integrity**: Guaranteed by TLS transport and the use of cryptographic digital signatures.
- **Authenticity**: Enforced by server certificate validation and password-based authentication.
- **Non-repudiation**: Digital signatures ensure cryptographic proof of authorship.
- **Perfect Forward Secrecy (PFS)**: Ensured by configuring ECDHE ciphersuites in a production setup.

2.7 Extensibility and Modularity

The DSS is structured for maximum flexibility and ease of extension:

- **Easily Extensible:** New endpoints, authentication mechanisms (e.g., multi-factor authentication), or advanced key management policies can be incorporated with minimal disruption.
- **Modular:** Each layer—API, authentication, cryptography, and persistence—can be independently enhanced or replaced as the system evolves.
- **Auditable:** All security-critical operations are centralized and can be logged for compliance and forensic analysis.

2.8 Conclusion

The design of the Digital Signature Server is firmly rooted in the principles of secure software engineering and cryptographic best practices. Through careful separation of concerns, adoption of strong cryptographic primitives, and a focus on modularity and future extensibility, the DSS stands as a robust solution for organizational digital signature requirements.

Chapter 3

Protocols and Message Formats

3.1 Overview

This chapter details the communication protocols and message formats employed by the Digital Signature Server (DSS). Every interaction between a user and the DSS follows a secure, structured protocol to guarantee **confidentiality**, **integrity**, and **authenticity** of the exchanged information.

3.2 Communication Principles

All messages are transmitted as JSON objects over HTTPS, ensuring that all data in transit is encrypted using TLS. Endpoints strictly follow RESTful design, using standard HTTP verbs (`POST`, `GET`) and status codes to indicate operation outcomes.

3.3 User Registration Protocol

3.3.1 Client Request

A user registers by submitting a JSON request to the DSS server:

Listing 3.1: Sample Registration Request (JSON)

```
POST /register
Content-Type: application/json

{
  "username": "alice",
  "password": "testpass123"
```

```
}
```

3.3.2 Server Processing (Code Snippet)

Upon receiving a registration request, the server:

1. Verifies that the username is unique.
2. Hashes the password with bcrypt.
3. Generates an RSA key pair.
4. Encrypts the private key using the provided password.
5. Stores the credentials and keys in the database.

Listing 3.2: Registration Endpoint - Server Code

```
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    # [Input validation omitted for brevity]
    pw_hash = bcrypt.hashpw(password.encode(), bcrypt.
       gensalt())
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    private_bytes = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.
           BestAvailableEncryption(password.encode())
    )
    # [Public key and DB storage logic omitted]
```

3.3.3 Server Response

On success:

Listing 3.3: Registration Success Response (JSON)

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "message": "User registered and keys generated."
}
```

On error (e.g., user exists):

Listing 3.4: Registration Error Response (JSON)

```
HTTP/1.1 409 Conflict
Content-Type: application/json

{
  "error": "User already exists."
}
```

3.4 User Authentication Protocol

3.4.1 Client Request

Listing 3.5: Login Request (JSON)

```
POST /login
Content-Type: application/json

{
  "username": "alice",
  "password": "testpass123"
}
```

3.4.2 Server Processing (Code Snippet)

The DSS server validates credentials by checking the submitted password against the bcrypt hash stored in the database.

Listing 3.6: Authentication Verification - Server Code

```
@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
```

```
    username = data.get('username')
    password = data.get('password')
    with shelve.open(DB_FILE) as db:
        user = db.get(username)
        stored_pw_hash = base64.b64decode(user['pw_hash'])
        if bcrypt.checkpw(password.encode(), stored_pw_hash):
            return jsonify({'message': 'Login successful.'}), 200
        else:
            return jsonify({'error': 'Invalid password.'}), 401
```

3.4.3 Server Response

Listing 3.7: Authentication Success Response (JSON)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "message": "Login successful."
}
```

3.5 Document Signing Protocol

3.5.1 Client Request

Listing 3.8: Sign Document Request (JSON)

```
POST /sign_doc
Content-Type: application/json

{
  "username": "alice",
  "password": "testpass123",
  "document": "Hello, sign me!"
}
```

3.5.2 Server Processing (Code Snippet)

Listing 3.9: Document Signing - Server Code

```
@app.route('/sign_doc', methods=['POST'])
def sign_doc():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    document = data.get('document')
    # [User authentication code omitted]
    private_key = serialization.load_pem_private_key(
        base64.b64decode(user['private_key']),
        password=password.encode(),
        backend=default_backend()
    )
    signature = private_key.sign(
        document.encode(),
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    signature_b64 = base64.b64encode(signature).decode()
    return jsonify({'signature': signature_b64}), 200
```

3.5.3 Server Response

Listing 3.10: Sign Document Response (JSON)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "signature": "base64encodedsignature=="
}
```

3.6 Public Key Retrieval Protocol

3.6.1 Client Request

Listing 3.11: Get Public Key Request

```
GET /get_public_key?username=alice
```

3.6.2 Server Response

Listing 3.12: Get Public Key Response (JSON)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "username": "alice",
  "public_key": "-----BEGIN␣PUBLIC␣KEY-----\nMIIB...
    IDAQAB\n-----END␣PUBLIC␣KEY-----"
}
```

3.7 Delete Keys Protocol

3.7.1 Client Request

Listing 3.13: Delete Keys Request (JSON)

```
POST /delete_keys
Content-Type: application/json

{
  "username": "alice",
  "password": "testpass123"
}
```

3.7.2 Server Processing (Code Snippet)

Listing 3.14: Delete Keys - Server Code

```
@app.route('/delete_keys', methods=['POST'])
def delete_keys():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    # [User authentication omitted]
```

```
    with shelve.open(DB_FILE, writeback=True) as db:
        del db[username]
    return jsonify({'message': 'Key pair and user data
       deleted.'}), 200
```

3.7.3 Server Response

Listing 3.15: Delete Keys Response (JSON)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "message": "Key pair and user data deleted."
}
```

3.8 Summary

All message formats are designed for clarity, security, and ease of integration with other systems. By standardizing all requests and responses as JSON, and enforcing TLS transport, the DSS ensures robust, secure, and auditable digital signature workflows for all users.

Chapter 4

Sequence Diagrams

This chapter visualizes the main protocols of the Digital Signature Server (DSS) using application-level sequence diagrams. Each diagram demonstrates the step-by-step interaction between a user/client and the DSS server for a specific operation. These diagrams clarify the security guarantees at each stage and help highlight trust boundaries and message exchanges.

4.1 Legend

- **User**: A registered user or client application.
- **DSS**: The Digital Signature Server.
- **Secure Channel**: All interactions occur over TLS/HTTPS, ensuring confidentiality and integrity.

4.2 User Registration Sequence

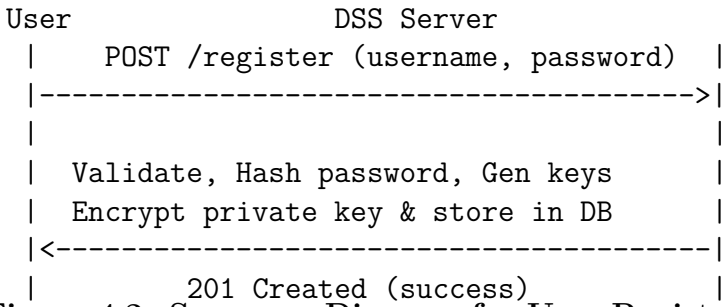

Figure 4.2: Sequence Diagram for User Registration Protocol

Listing 4.1: Relevant Server Logic: User Registration

```
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    # [rest of registration code as above]
```

4.3 User Authentication (Login) Sequence

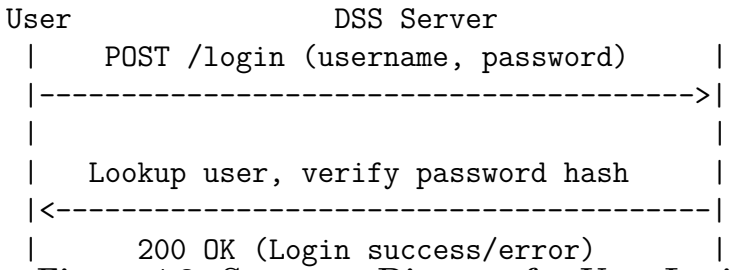

Figure 4.3: Sequence Diagram for User Login Protocol

Listing 4.2: Relevant Server Logic: User Authentication

```
@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    # [rest of login code as above]
```

4.4 Document Signing Sequence

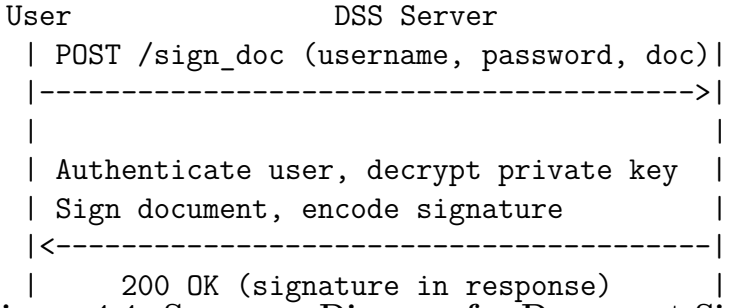

Figure 4.4: Sequence Diagram for Document Signing Protocol

Listing 4.3: Relevant Server Logic: Document Signing

```
@app.route('/sign_doc', methods=['POST'])
def sign_doc():
```

```
    data = request.get_json()
    # [user auth, decrypt private key, sign doc]
```

4.5 Get Public Key Sequence

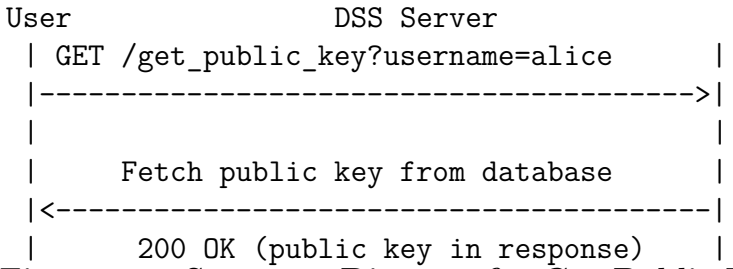

Figure 4.5: Sequence Diagram for Get Public Key Protocol

Listing 4.4: Relevant Server Logic: Public Key Retrieval

```
@app.route('/get_public_key', methods=['GET'])
def get_public_key():
    username = request.args.get('username')
    # [retrieve public key from DB]
```

4.6 Delete Keys Sequence

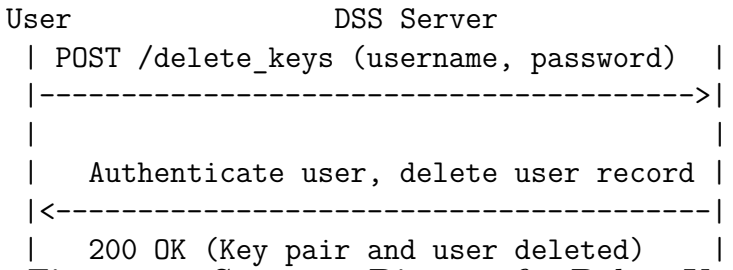

Figure 4.6: Sequence Diagram for Delete Keys Protocol

Listing 4.5: Relevant Server Logic: Delete Keys

```
@app.route('/delete_keys', methods=['POST'])
def delete_keys():
    data = request.get_json()
    # [authenticate user, delete from DB]
```

4.7 How to Generate the Diagrams

The diagrams above can be generated using PlantUML code (shown in comments) or drawn in diagram tools like draw.io, then exported as PDF/PNG for inclusion in this report.

4.8 Summary

Sequence diagrams clarify every protocol's flow, making it straightforward to analyze trust boundaries and confirm the enforcement of security properties in the DSS.

Chapter 5

Security Analysis

This chapter critically examines the security properties of the Digital Signature Server (DSS) and demonstrates how the design choices and implemented protocols ensure robust protection against a wide range of threats. Where relevant, specific code excerpts are referenced to show how these protections are enforced in practice.

5.1 Confidentiality

Confidentiality is achieved by enforcing encrypted communication channels (TLS/HTTPS) between users and the DSS, as well as by securely encrypting all private keys stored on the server.

- **TLS Enforcement:** All API endpoints are accessible exclusively over HTTPS, ensuring that credentials and sensitive data are never transmitted in plaintext.
- **Encrypted Key Storage:** Each private key is encrypted using a key derived from the user's password before storage. Even in the event of a server breach, attackers cannot recover private keys without the corresponding user password.

Listing 5.1: Private Key Encryption in Storage

```
private_bytes = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.
       BestAvailableEncryption(password.encode())
)
```

5.2 Integrity

Integrity is preserved through the use of cryptographically secure digital signatures and by enforcing TLS transport for all data in transit.

- **TLS Protection:** Data exchanged between users and the DSS cannot be altered or injected by attackers without detection, due to message authentication codes in TLS.
- **Digital Signatures:** Signed documents include a cryptographic signature that detects any modification of document content.

5.3 Authenticity

Authenticity is guaranteed by strong user authentication and server certificate verification.

- **User Authentication:** Each sensitive operation (sign, delete keys) requires username and password. Passwords are hashed with bcrypt, defending against brute-force and rainbow table attacks.
- **Server Authentication:** Users verify the server's TLS certificate, preventing man-in-the-middle attacks.

Listing 5.2: Password Hash Verification on Login

```
1 if bcrypt.checkpw(password.encode(), stored_pw_hash):
2     return jsonify({'message': 'Login␣successful.'}),
          200
3 else:
4     return jsonify({'error': 'Invalid␣password.'}), 401
```

5.4 Non-repudiation

Non-repudiation is achieved by leveraging cryptographic digital signatures, which serve as irrefutable proof that a particular user authorized a document.

- **Digital Signature Generation:** Only the owner of the private key (i.e., only the user who can authenticate and decrypt it) can create a valid signature.
- **Auditability:** All operations can be logged for forensic analysis and compliance.

5.5 Perfect Forward Secrecy (PFS)

Perfect Forward Secrecy (PFS) is ensured (in production deployments) by requiring modern ECDHE-based cipher suites for all TLS connections. This guarantees that even if long-term server keys are compromised, past session data remains secure.

5.6 Threat Model and Mitigations

1. **Brute-force and Password Guessing Attacks:**
   - Mitigated by bcrypt password hashing, which uses salt and a configurable work factor to slow down attacks.
2. **Server Compromise:**
   - Mitigated by encrypting all private keys at rest; keys cannot be used without user passwords.
3. **Replay Attacks:**
   - Prevented by relying on the anti-replay protections inherent in TLS sessions.
4. **Man-in-the-middle Attacks:**
   - Prevented by TLS certificate verification on the client side.
5. **Insider Attacks:**
   - Private keys are never accessible in plaintext by server admins; only the user can decrypt and use their key.

5.7 Limitations and Assumptions

- **Password Security:** The system's security ultimately depends on users choosing strong, unique passwords.
- **Self-signed Certificates:** For development, self-signed certificates are used; for production, CA-signed certificates must be enforced.
- **Session Management:** This implementation does not use persistent sessions or tokens; multi-factor authentication can be considered as a future improvement.

5.8 Summary

The design and implementation of the DSS employ state-of-the-art cryptographic protections and best practices to secure user identities, signing keys, and documents. While certain features could be enhanced further (see the next chapter), the system robustly defends against the most significant threats in practical digital signature services.

Chapter 6

Certification Authority Integration and Certificate-Based Workflow

6.1 Motivation and Overview

To bring the Digital Signature Server (DSS) in line with modern public key infrastructure (PKI) best practices, an additional certificate-based workflow has been introduced. This extension allows user public keys to be certified by a trusted Certification Authority (CA), replacing direct key exchange with X.509 certificates and enabling standards-based signature verification and trust management. **Both the classic public key workflow and this enhanced CA-based workflow are fully supported and can coexist within the same system.**

6.2 Certification Authority (CA) Setup

A simple CA was established using OpenSSL. The following commands generated the CA's private key and self-signed certificate:

Listing 6.1: CA Key and Certificate Creation

```
openssl genrsa -out ca-key.pem 2048
openssl req -x509 -new -nodes -key ca-key.pem -sha256 -
   days 3650 -out ca-cert.pem
```

These files are stored in a dedicated directory (e.g., `/dss-project/ca`) and are required for all certificate signing and verification operations.

6.3 Certificate-Based API Workflow

The certificate-based process involves several new endpoints and commands. Below, each step is described in detail.

6.3.1 1. User Registration

The user registers using the classic API:

Listing 6.2: User Registration API

```
POST /register
{
  "username": "bob",
  "password": "bobpass123"
}
```

6.3.2 2. Generate and Download a Certificate Signing Request (CSR)

The DSS generates a CSR for the user, which can be retrieved as follows:

Listing 6.3: Obtain User CSR

```
GET /get_csr?username=bob&password=bobpass123
```

Example command:

```
curl -k "https://localhost:5000/get_csr?username=bob&
   password=bobpass123" | jq -r .csr > bob.csr
```

6.3.3 3. Sign the CSR with the CA

The CA operator signs the user's CSR, issuing a user certificate:

Listing 6.4: CA Signs CSR to Issue Certificate

```
openssl x509 -req -in bob.csr -CA ca-cert.pem -CAkey ca-
   key.pem -CAcreateserial -out bob-cert.pem -days 365 -
   sha256
```

6.3.4 4. Upload the Signed Certificate to DSS

The signed certificate is uploaded to DSS and linked to the user:

Listing 6.5: Upload Signed Certificate API

```
POST /upload_certificate
{
  "username": "bob",
  "certificate": "-----BEGIN␣CERTIFICATE-----\n...
     certificate␣content...\n-----END␣CERTIFICATE-----"
}
```

Example shell command (using Bash):

```
CERT_CONTENT=$(awk 'NF␣{sub(/\r/,␣"");␣printf␣"%s\\n",$0
   ;}' bob-cert.pem)
curl -k -X POST https://localhost:5000/
   upload_certificate \
  -H "Content-Type:␣application/json" \
  -d "{\"username\":\"bob\",␣\"certificate\":\"
     $CERT_CONTENT\"}"
```

6.3.5 5. Retrieve a User's Certificate

The DSS now serves the signed certificate to requesting clients, not just a raw public key:

Listing 6.6: Get User Certificate API

```
GET /get_certificate?username=bob
```

Example command:

```
curl -k "https://localhost:5000/get_certificate?username
   =bob" | jq -r .certificate > bob-cert.pem
```

6.3.6 6. Signature Creation and Verification

Signing proceeds as before, using the classic API:

Listing 6.7: Sign Document API

```
POST /sign_doc
{
  "username": "bob",
  "password": "bobpass123",
```

```
  "document": "This is a document signed by Bob."
}
```

The response is a base64-encoded signature. To verify it, a client should use both Bob's certificate and the CA certificate:

Listing 6.8: Python: Verify Certificate and Signature

```python
from cryptography import x509
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
import base64

with open("/home/iot_ubuntu_intel/dss-project/ca/bob-cert.pem", "rb") as f:
    bob_cert = x509.load_pem_x509_certificate(f.read())
with open("/home/iot_ubuntu_intel/dss-project/ca/ca-cert.pem", "rb") as f:
    ca_cert = x509.load_pem_x509_certificate(f.read())

# 1. Verify certificate chain
ca_cert.public_key().verify(
    bob_cert.signature,
    bob_cert.tbs_certificate_bytes,
    padding.PKCS1v15(),
    bob_cert.signature_hash_algorithm
)

# 2. Verify document signature
public_key = bob_cert.public_key()
signature = base64.b64decode("...signature...")
document = "...document string..."
public_key.verify(
    signature,
    document.encode(),
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

Backward Compatibility

The DSS supports both the classic public key API and the new certificate-based API in parallel. Existing endpoints, for example:

- `/get_public_key`

remain unchanged, ensuring that legacy integrations continue to function as before. New clients are encouraged to use the certificate-based endpoints, which provide enhanced trust and allow for CA validation as described above.

6.4 Security Benefits

The use of a Certification Authority ensures:

- Only trusted, validated users receive signed certificates.
- Public key distribution is no longer vulnerable to man-in-the-middle attacks.
- Signature verification can now enforce true identity, revocation, and compliance with PKI standards.

6.5 Summary

By integrating a Certification Authority into the DSS workflow, the system now offers a complete, standards-compliant PKI workflow for digital signatures. This enhancement provides a higher level of trust, interoperability, and auditability, while keeping classic operations available for backward compatibility.

Chapter 7

API Documentation

This chapter documents all HTTP API endpoints provided by the Digital Signature Server (DSS). Each endpoint is described with its HTTP method, path, purpose, input parameters, response format, and its corresponding CRUD (Create, Read, Update, Delete) operation, highlighted with color for clarity.

7.1 Legend: CRUD Color Codes

- **Create**: Creation of new resources (e.g., users, key pairs).
- **Read**: Retrieval of existing resources or information.
- **Update**: Modification of existing resources (rare in this API).
- **Delete**: Removal of resources or data.

7.2 API Endpoints

7.2.1 POST /register — User Registration (Create)

Purpose: Registers a new user and generates a key pair. **Input (JSON):**

```
{
  "username": "bob",
  "password": "bobpass123"
}
```

Response (Success):

```
201 Created
{
  "message": "User registered and keys generated."
}
```

Response (User Exists/Error):

```
409 Conflict
{
  "error": "User already exists."
}
```

Details: - Hashes the password using bcrypt. - Generates RSA key pair. - Encrypts private key with password. - Stores all data in the database. - No user can register with a duplicate username.

7.2.2 GET /get_public_key — Retrieve User Public Key (Read)

Purpose: Retrieves the PEM-encoded public key for a specified user (classic API). **Input:** Query parameter `username`.

```
GET /get_public_key?username=bob
```

Response (Success):

```
200 OK
{
  "username": "bob",
  "public_key": "-----BEGIN PUBLIC KEY-----\n...\n-----
    END PUBLIC KEY-----"
}
```

Response (Not Found):

```
404 Not Found
{
  "error": "User not found."
}
```

Details: - This endpoint remains for backward compatibility. - Returns the user's public key (not certificate).

7.2.3 POST /login — User Login (Read/Authentication)

Purpose: Authenticates a user by verifying their password. **Input (JSON):**

```
{
  "username": "bob",
  "password": "bobpass123"
}
```

Response (Success):

```
200 OK
{
  "message": "Login successful."
}
```

Response (Failure):

```
401 Unauthorized
{
  "error": "Invalid password."
}
```

Details: - Checks the bcrypt password hash stored for the user. - No session/token is created (stateless authentication per request).

7.2.4 POST /sign_doc — Sign Document (Read/Action)

Purpose: Authenticates the user, decrypts their private key, and returns a digital signature for a document. **Input (JSON):**

```
{
  "username": "bob",
  "password": "bobpass123",
  "document": "This is a document signed by Bob."
}
```

Response (Success):

```
200 OK
{
  "signature": "base64encodedsignature=="
}
```

Details: - Authenticates user and decrypts their private key using password. - Signs the provided document using RSA-PSS with SHA-256. - Returns the digital signature (base64-encoded). - Signature can be verified using user's public key or X.509 certificate.

7.2.5 POST /delete_keys — Delete User Keys and Data (Delete)

Purpose: Authenticates the user and permanently deletes their key pair and all stored data. **Input (JSON):**

```
{
  "username": "bob",
  "password": "bobpass123"
}
```

Response (Success):

```
200 OK
{
  "message": "Key pair and user data deleted."
}
```

Details: - Authenticates the user before deletion. - After deletion, the user cannot sign documents or retrieve keys. - Irreversible; re-registration is required for future use.

7.2.6 GET /get_csr — Generate CSR for User (Read)

Purpose: Provides a Certificate Signing Request (CSR) for the authenticated user, which can be signed by a Certification Authority (CA). **Input:** Query parameters `username` and `password`.

```
GET /get_csr?username=bob&password=bobpass123
```

Response (Success):

```
200 OK
{
  "csr": "-----BEGIN CERTIFICATE REQUEST-----\n...\n
    -----END CERTIFICATE REQUEST-----"
}
```

Details: - Authenticates the user and generates a CSR with the user's identity. - The CSR is used for CA-based certificate issuance.

7.2.7 POST /upload_certificate — Store User Certificate (Update)

Purpose: Stores a user's X.509 certificate signed by a CA for later retrieval.
Input (JSON):

```
{
  "username": "bob",
  "certificate": "-----BEGIN␣CERTIFICATE-----\n...\n
    -----END␣CERTIFICATE-----"
}
```

Response (Success):

```
200 OK
{
  "message": "Certificate␣uploaded␣and␣stored."
}
```

Details: - Associates the uploaded certificate with the user's record. - Certificate must be valid PEM format, signed by your CA.

7.2.8 GET /get_certificate — Retrieve User Certificate (Read)

Purpose: Retrieves the user's X.509 certificate (PEM), which can be verified with the CA's public key. **Input:** Query parameter `username`.

```
GET /get_certificate?username=bob
```

Response (Success):

```
200 OK
{
  "certificate": "-----BEGIN␣CERTIFICATE-----\n...\n
    -----END␣CERTIFICATE-----"
}
```

Details: - The returned certificate is signed by the CA. - Clients should use the CA certificate to validate authenticity.

7.2.9 GET / — Server Root (Read/Info)

Purpose: Simple informational endpoint confirming the DSS server is running. **Input:** None.

Response:

```
Hello, this is the Digital Signature Server!
```

Details: - Can be used as a health check endpoint. - No authentication required.

7.3 Summary Table of All Endpoints

Endpoint	Method	CRUD Type	Description
/register	POST	Create	Register new user
/get_public_key	GET	Read	Get user's public key
/login	POST	Read	Authenticate user
/sign_doc	POST	Read	Sign a document
/delete_keys	POST	Delete	Delete user keys/data
/get_csr	GET	Read	Get certificate request
/upload_certificate	POST	Update	Store certificate
/get_certificate	GET	Read	Get user's certificate
/	GET	Read	Server health/info

7.4 General Notes and Security

All endpoints except '/' require secure HTTPS and, where applicable, authentication via username and password. Sensitive operations are protected against unauthorized access, and all data is stored securely as described in previous chapters.

Chapter 8

Future Improvements

While the current Digital Signature Server (DSS) implementation now supports both classic public key workflows and an advanced X.509 certificate-based PKI integration, there remain significant opportunities to further enhance security, usability, compliance, and enterprise scalability.

8.1 Hybrid Trust and API Evolution

- **API Versioning and Dual Support:** Maintain backward compatibility while providing a clear migration path towards certificate-only workflows. Explicit versioning (e.g., `/v2/register`, `/v2/get_certificate`) will allow new features without disrupting legacy clients.
- **Seamless Migration:** Develop tools to help existing users upgrade from classic key-based accounts to certificate-backed identities with minimal disruption.

8.2 Advanced Authentication Mechanisms

- **Multi-factor Authentication (MFA):** Combine certificates or private keys with second factors such as OTP, app push, or biometrics for higher assurance.
- **Adaptive and Risk-based Authentication:** Analyze access patterns and context to trigger additional verification for high-risk actions (e.g., from new devices or locations).
- **Password Policy Enforcement:** Apply strict password requirements,

automatic expiration, and blacklist checks to minimize credential compromise risk.

8.3 Enhanced Key and Certificate Management

- **Hardware Security Modules (HSM):** Integrate HSMs to generate, store, and use CA and user keys, ensuring that private keys never leave tamper-resistant environments.
- **Key and Certificate Rotation:** Automate periodic renewal and rotation of both user and CA keys/certificates, supporting certificate expiry and revocation.
- **Automated CA Integration:** Replace manual certificate signing with automated CA endpoints, supporting certificate enrollment protocols such as ACME or SCEP.
- **Certificate Revocation:** Implement Certificate Revocation Lists (CRL) or Online Certificate Status Protocol (OCSP) to handle lost or compromised credentials.

8.4 Auditability, Monitoring, and Compliance

- **Comprehensive Logging:** Record every certificate issuance, signing, and deletion event, including API caller, IP, and timestamp.
- **Tamper-evident Audit Trails:** Use hash-chained or blockchain-backed logs to ensure audit trail integrity.
- **Regulatory Compliance:** Evolve the DSS to meet standards such as eIDAS, GDPR, or FIPS, including consent, retention, and right-to-erasure controls.
- **External Auditing:** Facilitate third-party penetration tests and cryptographic audits of both classic and CA-based workflows.

8.5 API Security and Access Control

- **Token-based and Certificate-based Authentication:** Migrate from password-per-request to using JWT, OAuth2 tokens, or even mutual TLS with client certificates.
- **Fine-grained Authorization:** Implement user roles, delegation, and per-endpoint access control policies.
- **API Rate Limiting and Abuse Prevention:** Enforce per-user and per-IP request limits to prevent brute-force or DoS attacks.

8.6 Scalability and Reliability

- **Containerization and Orchestration:** Package the DSS and CA as Docker containers for deployment on Kubernetes or similar platforms, enabling horizontal scaling.
- **Distributed, Encrypted Databases:** Move beyond file-based storage to scalable, encrypted databases (e.g., PostgreSQL with Transparent Data Encryption or HashiCorp Vault).
- **Geo-Redundancy and Backup:** Automate backups and enable recovery across geographic locations for high availability.

8.7 User Experience and Accessibility

- **Unified Web Interface:** Develop a web dashboard for managing users, certificates, signing requests, and audit logs, with both classic and certificate-based flows.
- **Localization and Accessibility:** Support internationalization, timezone awareness, and accessibility standards (e.g., WCAG 2.1).
- **Self-service Certificate Management:** Allow users to view, renew, or revoke their certificates and download CA roots.

8.8 Example: Automated CA Workflow (Pseudo-code)

To further automate and secure certificate issuance, a dedicated endpoint can accept CSRs and return signed certificates directly, rather than requiring manual OpenSSL commands:

Listing 8.1: Automated CA Signing Endpoint (Pseudo-code)

```
@app.route('/ca/sign', methods=['POST'])
def sign_csr():
    data = request.get_json()
    csr_pem = data.get('csr')
    # [Parse CSR, validate requester identity]
    # Sign CSR with CA key
    certificate_pem = ca_sign_function(csr_pem)
    return jsonify({'certificate': certificate_pem}),
       200
```

This can integrate with ACME clients, or be triggered as part of user registration or renewal.

8.9 Research and Development Directions

- **Post-Quantum Security:** Evaluate and pilot support for quantum-resistant signature algorithms (e.g., Dilithium, Falcon, or SPHINCS+).
- **Decentralized Identity (DID):** Investigate integration with decentralized identity frameworks and verifiable credentials for privacy-preserving authentication.
- **Federation and SSO:** Allow users to authenticate using trusted external identity providers (e.g., SAML, OpenID Connect) and receive organizational certificates.
- **Privacy-preserving Analytics:** Use advanced analytics and machine learning for anomaly detection while preserving user privacy.

8.10 Summary

The recent introduction of a Certification Authority and support for X.509 certificates greatly extends the trust, interoperability, and compliance of

the DSS. By following the improvements outlined above—ranging from authentication and API modernization to scalable infrastructure and future-proof cryptography—the DSS can continue to evolve as a secure, standards-compliant, and enterprise-ready digital signature solution.

Chapter 9

Conclusion

The development of the Digital Signature Server (DSS) for the Cybersecurity course has resulted in a secure, standards-driven, and extensible platform for digital signatures in modern organizations. This project has advanced beyond basic cryptographic key management to support a full public key infrastructure (PKI) model, including a Certification Authority (CA) and X.509 certificate management, while retaining backward compatibility with classic public key workflows.

9.1 Major Achievements

- **Secure Key and Certificate Management:** The DSS now supports both encrypted private key storage and robust certificate lifecycle management. User keys are never stored in plaintext, and all sensitive operations use strong, industry-standard cryptography.
- **Flexible, Dual-Mode APIs:** The system supports both classic endpoints (e.g., `/get_public_key`) and new, CA-integrated endpoints (e.g., `/get_csr`, `/upload_certificate`, `/get_certificate`), enabling seamless transition to certificate-backed security.
- **Full Certification Authority Workflow:** Users can generate Certificate Signing Requests (CSRs), obtain CA-signed certificates, and leverage these for verifiable, trusted signatures, supporting organization-wide trust anchors and secure third-party verification.
- **Comprehensive Protocol Documentation:** Every DSS protocol—registration, login, document signing, key and certificate retrieval, and key/certificate deletion—has been fully specified, diagrammed, and documented, facilitating integration and audit.

- **Defense-in-Depth and Compliance:** TLS/HTTPS for all communication, careful error handling, and certificate-based authentication together ensure layered security and regulatory readiness.
- **Extensible and Maintainable Codebase:** Modular Python design and detailed documentation allow straightforward future expansion (such as hardware security modules, multi-factor authentication, or API versioning).

9.2 Key Lessons Learned

- **Security is Multi-layered:** Combining cryptography, PKI, and good software engineering multiplies overall trust. Secure system design is ongoing and iterative.
- **Interoperability and Standards Matter:** Supporting X.509 certificates, CA signing, and clear API contracts enables organizational and third-party adoption with confidence.
- **Usability and Compatibility:** By maintaining both classic and PKI endpoints, the DSS enables gradual migration and robust backward compatibility.
- **Auditability is Essential:** Sequence diagrams, code documentation, and clear message formats support audit and compliance, vital for any critical security system.

9.3 Example: Certificate-based Signature Verification

A highlight of this project is PKI-enabled end-to-end verification. Recipients can verify not only a document signature, but also the identity of the signer via the CA. The following Python code (used in the final workflow) demonstrates how:

Listing 9.1: Certificate-based Signature and Chain Verification

```
1 from cryptography import x509
2 from cryptography.hazmat.primitives.asymmetric import
     padding
3 from cryptography.hazmat.primitives import hashes
4 import base64
```

```
5
6  # Load Bob's certificate and CA certificate
7  with open("bob-cert.pem", "rb") as f:
8      bob_cert = x509.load_pem_x509_certificate(f.read())
9  with open("ca-cert.pem", "rb") as f:
10     ca_cert = x509.load_pem_x509_certificate(f.read())
11
12 # 1. Verify that Bob's certificate is signed by the CA
13 ca_cert.public_key().verify(
14     bob_cert.signature,
15     bob_cert.tbs_certificate_bytes,
16     padding.PKCS1v15(),
17     bob_cert.signature_hash_algorithm
18 )
19
20 # 2. Verify Bob's signature on the document
21 public_key = bob_cert.public_key()
22 signature = base64.b64decode("...signature...")
23 document = "...document␣string..."
24 public_key.verify(
25     signature,
26     document.encode(),
27     padding.PSS(
28         mgf=padding.MGF1(hashes.SHA256()),
29         salt_length=padding.PSS.MAX_LENGTH
30     ),
31     hashes.SHA256()
32 )
```

This process ensures signatures are trusted, verifiable, and compliant with industry best practices.

9.4 Final Reflections

The enhanced DSS provides a robust foundation for secure digital signature operations in real-world organizations, supporting both classic and PKI workflows. Its modular, well-documented architecture enables easy adaptation to future security and compliance needs, such as multi-factor authentication, hardware-backed keys, or post-quantum cryptography.

9.5 Future Outlook

As digital transformation accelerates and security threats evolve, platforms like the DSS—grounded in open standards, PKI, and layered security—will play a key role in protecting organizational integrity, trust, and compliance. Ongoing research into certificate automation, decentralized identities, and advanced cryptography will only further expand the system's relevance and security posture.