Advanced Programming of Cryptographic Methods

Project Report

TITOLO

Emanuele Civini, Alessia Pivotto

Academic year 2024/2025

Contents

1 Introduction

This project implements a Certificate Authority (CA) system that manages digital certificates for secure communications. A CA is a trusted entity that issues digital certificates to verify the identity of users, servers, or devices in a network.

The system provides a complete solution for certificate management, including issuing new certificates, checking their validity, renewing them before expiration, and revoking them when necessary.

The implementation focuses on security best practices by using Hardware Security Modules (HSMs) to protect the CA's private key and implementing proper verification procedures.

The system provides user-friendly interfaces through both API endpoints for developers and a web interface for end users. Additionally, it ensures system reliability by handling multiple requests efficiently while maintaining data consistency.

1.1 Key Features

The Certificate Authority system supports the complete certificate lifecycle. For certificate issuance, the system processes certificate requests, verifies the requester's identity, and issues signed digital certificates. It provides certificate validation by checking if certificates are valid, not expired, and haven't been revoked.

The system allows users to renew their certificates before they expire and handles certificate revocation for compromised or no longer needed certificates, maintaining a Certificate Revocation List (CRL). Identity verification ensures that certificate requesters own the private key and control the email address associated with their request.

1.2 System Architecture

The system consists of several integrated components. The backend server is a Go-based API that handles all certificate operations and communicates with the database and HSM. A Next.js frontend application provides an easy-to-use web interface for certificate management.

Data persistence is handled by MongoDB storage for certificate records, user commitments, and system data. Security operations are performed by a Hardware Security Module (HSM) that stores the CA's private key and performs cryptographic operations. An automated email service supports identity verification during certificate requests.

1.3 Report Structure

This report provides a comprehensive overview of the Certificate Authority system implementation. Chapter 2 presents the requirements analysis and system specifications. Chapter 3 details the system design and architecture. Chapter 4 covers implementation details and technology choices, while Chapter 5 discusses security considerations and best practices. Finally, Chapter 6 provides deployment guidance and usage instructions.

Each chapter builds upon the previous one, providing a complete understanding of the Certificate Authority system from concept to deployment.[?]

2 Requirements

This chapter defines the comprehensive set of requirements that guide the design and implementation of our Certificate Authority system. The requirements are categorized into functional requirements, which specify what the system must do, and security requirements, which define how the system must protect itself and its users from various threats and vulnerabilities.

2.1 Functional Requirements

The functional requirements define the core capabilities and operations that the Certificate Authority must provide to fulfill its role in a Public Key Infrastructure. Each requirement is justified based on standard PKI practices and the specific needs of secure certificate management.

2.1.1 FR1: Certificate Issuance

Requirement: The CA must generate and issue digital certificates after verifying the identity of the certificate requester.

Justification: Certificate issuance is the primary function of any Certificate Authority. This process involves receiving Certificate Signing Requests (CSRs) from entities, validating the authenticity of the request, and creating signed digital certificates that bind public keys to verified identities. Without this capability, the CA cannot fulfill its fundamental role as a trusted third party in the PKI ecosystem. The verification step is crucial to prevent unauthorized certificate issuance, which could compromise the entire trust infrastructure.

Implementation Details: The system must support standard CSR formats, perform comprehensive validation of request parameters, and generate certificates compliant with X.509 standards.

2.1.2 FR2: Certificate Revocation

Requirement: Users can request the revocation of any of their certificates at any moment, for example when they suspect a private key leakage.

Justification: Certificate revocation is essential for maintaining the integrity of the PKI when certificates become compromised, are no longer needed, or when the associated private keys are suspected to be leaked. Without a revocation mechanism, compromised certificates would remain valid until their natural expiration, potentially allowing unauthorized access to resources. The ability for certificate holders to initiate revocation ensures that they maintain control over their digital identities and can respond quickly to security incidents.

Implementation Details: The system must provide authenticated revocation requests, requiring proof of private key ownership to prevent malicious revocation attacks by unauthorized parties.

2.1.3 FR3: Certificate Revocation List (CRL) Accessibility

Requirement: The CA must maintain a Certificate Revocation List to track revoked certificates. Anyone must be able to check the status of any certificate by querying the CRL to verify whether a specific certificate has been revoked or not.

Justification: The CRL serves as the authoritative source for certificate revocation status information. Public accessibility is crucial because certificate validation is performed by various parties across the network, not just by the certificate holders themselves. Without a publicly accessible CRL, relying parties would have no way to determine if a certificate has been revoked, potentially accepting invalid certificates and compromising security. This requirement aligns with RFC 5280 standards for certificate validation.

Implementation Details: The CRL must be regularly updated, digitally signed by the CA, and made available through standard protocols such as HTTP or LDAP.

2.1.4 FR4: Identity Verification

Requirement: The CA must implement processes to verify the authenticity of the certificate applicant's email and the ownership of the proper private key. This process simplifies the identity verification typically required by real-world CAs.

Justification: Identity verification is fundamental to establishing trust in the PKI. Without proper verification, malicious actors could obtain certificates for identities they do not control, leading to impersonation attacks and breakdown of trust. Email verification ensures that the applicant controls the email address being certified, while private key ownership verification ensures that only the legitimate key holder can obtain or manage certificates for that key pair. This dual verification approach provides a reasonable balance between security and usability for a simplified CA implementation.

Implementation Details: Email verification can be implemented through challenge-response mechanisms, while private key ownership can be verified through cryptographic challenges requiring digital signatures.

2.1.5 FR5: Public Key Publishing

Requirement: The CA must publish the issued certificates and its own public key so that users can validate certificate authenticity.

Justification: Public availability of certificates and the CA's public key is essential for the certificate validation process. Relying parties need access to both the certificates they want to validate and the CA's public key to verify the certificate signatures. Without this public accessibility, the certificates would be useless for their intended purpose of enabling secure communications. The CA's public key serves as the trust anchor for the entire PKI, and its availability is crucial for establishing the chain of trust.

Implementation Details: The system must provide standard interfaces for certificate retrieval and maintain a publicly accessible repository of issued certificates and CA public key information.

2.1.6 FR6: Certificate Renewal

Requirement: The CA must allow renewal of certificates before they expire, ensuring continuity of trust. This process must include identity verification challenges and verification that the certificate has not been revoked.

Justification: Certificate renewal is critical for maintaining continuous service availability and trust relationships. Certificates have limited lifespans for security reasons, but the underlying trust relationships often need to persist beyond individual certificate validity periods. Renewal allows for the seamless transition from expiring certificates to new ones without disrupting established trust relationships. The verification requirements ensure that only legitimate certificate holders can renew their certificates and that revoked certificates cannot be renewed, maintaining security integrity.

Implementation Details: The renewal process must verify the current certificate's validity status, perform identity verification equivalent to initial issuance, and ensure proper overlap periods to prevent service interruptions.

2.1.7 FR7: Cryptographic Algorithm Support

Requirement: The CA must support RSA and ECDSA cryptographic algorithms for certificate operations.

Justification: Supporting multiple cryptographic algorithms ensures compatibility with diverse client requirements and provides flexibility for different security and performance needs. RSA remains widely deployed in legacy systems and provides well-understood security properties, while ECDSA offers better performance and smaller key sizes for equivalent security levels. Supporting both algorithms ensures that the CA can serve a broad range of clients while accommodating both current and emerging cryptographic preferences.

Implementation Details: The system must handle key generation, signature creation, and verification for both RSA (minimum 2048-bit keys) and ECDSA (P-256 curve) algorithms.

2.2 Security Requirements

Security requirements define the protective measures and security properties that the Certificate Authority must maintain to ensure the integrity, confidentiality, and availability of the PKI services. These requirements address various threat models and attack vectors that could compromise the CA's operations.

2.2.1 SR1: Certificate Authenticity

Requirement: A certificate that contains a valid and correct signature from the CA must be considered authentic and trustworthy.

Justification: Certificate authenticity forms the foundation of trust in the PKI system. The CA's digital signature on a certificate serves as the cryptographic proof that the certificate was issued by the legitimate CA and has not been tampered with. This requirement ensures that relying parties can confidently trust certificates that pass signature verification, enabling secure communications. Without reliable authenticity verification, the entire PKI system would be vulnerable to certificate forgery and impersonation attacks.

Implementation Details: The system must use cryptographically strong signature algorithms, maintain secure signing procedures, and ensure that signature verification mechanisms are readily available to all relying parties.

2.2.2 SR2: Certificate Validity Verification

Requirement: A certificate that is expired or appears in the CRL must be considered invalid, even if it contains an authentic signature from the CA.

Justification: Certificate validity encompasses more than just authenticity; it also includes temporal validity and revocation status. Expired certificates should not be trusted because they may represent outdated information or compromised keys that have exceeded their intended lifespan. Similarly, revoked certificates must be rejected regardless of their authentic signatures because they have been explicitly invalidated due to compromise or other security concerns. This requirement prevents the acceptance of certificates that may pose security risks despite being technically authentic.

Implementation Details: All certificate validation processes must include expiration date checking and CRL consultation, with clear rejection of certificates that fail either test.

2.2.3 SR3: Secure Key Management

Requirement: The private key of the CA must be stored securely using Hardware Security Modules (HSMs) and protected against tampering, unauthorized access, and extraction.

Justification: The CA's private key is the most critical security asset in the entire PKI system. If this key is compromised, attackers could forge certificates for any identity, completely undermining the trust infrastructure. Traditional software-based key storage is vulnerable to various attacks, including malware, insider threats, and system compromises. HSMs provide tamper-resistant hardware protection that ensures the private key never exists in plaintext outside the secure hardware boundary and that all cryptographic operations are performed within the protected environment.

Implementation Details: The system must integrate with cloud-based HSM services, ensure all signing operations occur within the HSM, implement secure authentication for HSM access, and maintain audit trails of all key usage.

2.2.4 SR4: Authentication and Authorization

Requirement: All certificate management operations must be properly authenticated and authorized to prevent unauthorized access and malicious operations.

Justification: Without proper authentication and authorization controls, malicious actors could perform unauthorized certificate operations such as requesting certificates for identities they don't control, revoking legitimate certificates, or accessing sensitive certificate information. These controls ensure that only authorized parties can perform specific operations and that all actions are traceable to authenticated identities.

Implementation Details: The system must implement multi-factor authentication where appropriate, role-based access controls, and comprehensive audit logging of all administrative actions.

2.2.5 SR5: Data Integrity and Confidentiality

Requirement: All certificate data, configuration information, and audit logs must be protected against unauthorized modification and inappropriate disclosure.

Justification: Data integrity ensures that certificate information remains accurate and trustworthy throughout its lifecycle. Unauthorized modifications could lead to invalid certificates being accepted or valid certificates being rejected. Confidentiality protections prevent sensitive information from being disclosed to unauthorized parties, which could facilitate attacks or privacy violations.

Implementation Details: The system must implement database encryption, secure communication protocols, access controls, and integrity checking mechanisms for all stored data.

2.2.6 SR6: Availability and Resilience

Requirement: The CA services must maintain high availability and resilience against various failure modes and attack scenarios.

Justification: CA unavailability can disrupt certificate validation processes across the entire PKI, potentially preventing legitimate users from accessing services or causing applications to reject valid certificates. High availability ensures that critical PKI services remain accessible when needed, maintaining trust and usability of the infrastructure.

Implementation Details: The system must implement redundancy, failover mechanisms, backup procedures, and monitoring to detect and respond to availability threats.

These requirements collectively define a comprehensive security posture that addresses the primary threats and vulnerabilities associated with Certificate Authority operations while ensuring the functional capabilities necessary for effective PKI services.

3 Technical Details

This chapter provides comprehensive technical details about the Certificate Authority implementation, covering the system architecture, implementation technologies, and detailed code structure. The information presented here enables a thorough understanding of the technical decisions and implementation approaches used in this project.

3.1 Architecture

The Certificate Authority system follows a microservices-based architecture designed for scalability, security, and maintainability. The system is composed of several distinct modules that work together to provide comprehensive PKI services.

3.1.1 High-Level Architecture

The system architecture consists of four primary components:

1. **CA Backend Server**: A Go-based REST API server that handles all certificate operations, including issuance, revocation, and validation.
2. **Web User Interface**: A Next.js-based frontend application that provides an intuitive interface for certificate management operations.
3. **Database Layer**: MongoDB database for persistent storage of certificates, revocation lists, and audit information.
4. **Hardware Security Module (HSM)**: Cloud-based HSM integration for secure cryptographic operations and private key protection.

3.1.2 Component Interaction

The architecture follows a layered approach with clear separation of concerns:

- **Presentation Layer**: The web UI communicates with the CA backend through RESTful API calls, handling user interactions and data visualization.
- **Business Logic Layer**: The CA server implements all PKI business logic, including certificate lifecycle management, validation rules, and identity verification processes.
- **Data Persistence Layer**: MongoDB provides reliable storage for certificates, CRL entries, and operational logs.
- **Security Layer**: HSM integration ensures that all cryptographic operations involving the CA's private key are performed within secure hardware boundaries.

3.1.3 Security Architecture

The security architecture implements defense-in-depth principles:

- **HSM Integration**: All CA private key operations are performed within the HSM, ensuring the key never exists in plaintext outside secure hardware.
- **Authentication Mechanisms**: Multi-layered authentication including email verification and private key ownership proof.

3.1.4 Deployment Architecture

The system is containerized using Docker for consistent deployment across different environments:

- Each major component runs in its own Docker container
- Docker Compose orchestrates multi-container deployment
- Environment-specific configurations through environment variables
- Persistent storage volumes for database data

3.2 Implementation

This section details the technical implementation aspects, including programming languages, frameworks, libraries, and key implementation decisions.

3.2.1 Backend Implementation

Programming Language: Go (Golang) 1.21+

Justification: Go was chosen for the backend implementation due to its performance characteristics, strong standard library for cryptographic operations, robust concurrency support, and extensive ecosystem for web service development.

Key Libraries and Frameworks:

- **Go net/http**: HTTP router and URL matcher for building REST APIs with support for middleware and route variables.
- **MongoDB Driver**: Official Go driver for MongoDB providing high-performance database operations with built-in connection pooling.
- **AWS SDK for Go**: Integration with AWS KMS for HSM operations, providing secure key management and cryptographic operations.
- **Go Standard Crypto Libraries**: Extensive use of crypto/x509, crypto/rsa, crypto/ecdsa, and crypto/rand for certificate operations and cryptographic functions.
- **Email**: Email verification and notification capabilities using Go's Resend package.

3.2.2 Frontend Implementation

Framework: Next.js 15 with React 19

Justification: Next.js provides server-side rendering capabilities, excellent developer experience, and robust production optimizations. React's component-based architecture enables maintainable and reusable UI components.

Key Technologies:

- **Tailwind CSS 4.0**: Utility-first CSS framework for rapid UI development with consistent design systems and responsive layouts.
- **Web Crypto API**: Browser-native cryptographic operations for client-side key generation, signing, and certificate validation without requiring external libraries.
- **Next.js App Router**: Modern routing system with nested layouts and server components for optimal performance.
- **React Hooks**: State management and lifecycle handling using modern React patterns including useState, useEffect, and custom hooks.

3.2.3 Database Implementation

Database: MongoDB 7.0+

Justification: MongoDB's document-oriented structure is well-suited for storing certificate data with varying fields and complex nested structures. Its built-in indexing and query capabilities support efficient certificate lookups and CRL operations.

Key Features Utilized:

- **Document Storage**: Flexible schema for certificate metadata, CRL entries, and audit logs
- **Indexing**: Optimized queries for certificate serial numbers, email addresses, and revocation status

3.2.4 HSM Integration

HSM Provider: AWS KMS (Key Management Service)

Implementation Approach:

- **Customer Master Keys (CMK)**: Dedicated CMK for CA operations with hardware-level protection
- **AWS SDK Integration**: Programmatic access to KMS operations through official AWS SDK
- **Signing Operations**: All certificate signing performed via KMS API calls
- **Key Rotation**: Support for automatic key rotation policies
- **Audit Logging**: Comprehensive logging of all HSM operations through AWS CloudTrail

3.2.5 Containerization and Deployment

Container Technology: Docker with Docker Compose

Container Configuration:

- **Multi-stage Builds**: Optimized Docker images with separate build and runtime stages
- **Environment Variables**: Externalized configuration for different deployment environments
- **Health Checks**: Container health monitoring and automatic restart capabilities
- **Volume Management**: Persistent storage for database data and certificates

3.3 Code Structure

This section provides a detailed description of the project's code organization, explaining the purpose and functionality of each component.

3.3.1 Project Root Structure

The project follows a monorepo structure with clear separation between different services:

```
advanced-programming-of-cryptographic-methods/
|-- ca/                 # Backend CA server
|-- ui/                 # Frontend web application
|-- dev-certs/          # Development certificates
|-- report/             # Project documentation
|-- docker-compose.yml  # Container orchestration
|-- mongod.conf         # MongoDB configuration
`-- README.md           # Project overview
```

3.3.2 Backend Code Structure (ca/)

The backend follows Go's standard project layout with clear separation of concerns:

Main Application (cmd/)

- **cmd/server/main.go**: Application entry point that initializes all services (HSM, database, email) and starts the HTTP server. Handles dependency injection and graceful shutdown procedures.

Internal Packages (internal/)

- **internal/config/config.go**: Centralized configuration management using environment variables. Handles HSM and server parameters.
- **internal/db/**: Database abstraction layer
  - **db.go**: Database connection management and initialization
  - **models.go**: Data models for certificates, CRL entries, and logs
- **internal/email/email.go**: Email service implementation for identity verification and notifications.
- **internal/hsm/hsm.go**: Hardware Security Module integration layer. Provides abstraction over AWS KMS operations including key creation, signing, and key management.
- **internal/server/**: HTTP server implementation
  - **server.go**: HTTP server initialization, middleware configuration, and route setup
  - **handlers/**: Request handlers organized by functionality
    - **health.go**: Health check endpoints for monitoring
    - **info.go**: CA information and public key endpoints
    - **utils.go**: Common handler utilities and response formatting
    - **certificate/**: Certificate-specific handlers
      - **handler.go**: Certificate issuance, revocation, and validation endpoints
      - **repository.go**: Database operations for certificate management

Configuration Files

- **go.mod/go.sum**: Go module dependencies and version management
- **Dockerfile**: Multi-stage Docker build configuration for optimized container images

3.3.3 Frontend Code Structure (ui/)

The frontend follows Next.js 13+ app directory structure with modern React patterns:

Application Core (app/)

- **layout.js**: Root layout component with global styles, navigation, and shared UI elements
- **page.js**: Home page component with CA overview and navigation links
- **globals.css**: Global CSS styles including Tailwind CSS imports and custom component styles
- **favicon.ico**: Application favicon

Feature Pages (app/)

- **sign/page.js**: Certificate signing interface with CSR upload, key generation, and email verification workflows
- **certs/page.js**: Certificate viewer and revocation interface with ASN.1 parsing and cryptographic operations
- **crl/page.js**: Certificate Revocation List viewer with pagination and search capabilities
- **commit/page.js**: Certificate commitment and validation interface

Utility Modules (utils/)

- **crypto.js**: Client-side cryptographic utilities including key generation, signing, and certificate validation using Web Crypto API
- **pemUtils.js**: PEM format parsing and manipulation utilities for certificate and key handling

Configuration Files

- **package.json**: Node.js dependencies and build scripts
- **next.config.mjs**: Next.js configuration including build optimizations and deployment settings
- **postcss.config.mjs**: PostCSS configuration for Tailwind CSS processing
- **jsconfig.json**: JavaScript/TypeScript configuration for IDE support
- **Dockerfile**: Frontend container build configuration

Static Assets (public/)

- ***.svg**: UI icons and graphics including Next.js branding and custom icons

3.3.4 Development and Deployment Support

Development Certificates (dev-certs/)

- **root.pem**: CA root certificate for the trust chain. It is generated during the initial setup.

Container Orchestration

- **docker-compose.yml**: Multi-container application orchestration including MongoDB, local KMS, CA server, and frontend services
- **mongod.conf**: MongoDB server configuration including security settings and connection parameters

3.3.5 Code Quality and Maintainability

The codebase implements several best practices for maintainability and reliability:

- **Separation of Concerns**: Clear boundaries between presentation, business logic, and data layers
- **Error Handling**: Comprehensive error handling with appropriate logging and user feedback
- **Configuration Management**: Externalized configuration through environment variables

- **Security Practices**: Input validation, secure defaults, and defense-in-depth implementation
- **Documentation**: Inline code comments and comprehensive API documentation
- **Testing Support**: Structure conducive to unit testing and integration testing

This code structure provides a solid foundation for a production-ready Certificate Authority system while maintaining flexibility for future enhancements and scaling requirements.

4 Security Considerations

This chapter critically examines the security architecture of the implemented Certificate Authority system, focusing on identifying fundamental vulnerabilities and proposing concrete pathways for improvement.

4.1 Use of custom JSON-based OCSP responses

The implementation uses custom JSON-based response formats instead of standard ASN.1 OCSP responses, creating fundamental incompatibility with the broader PKI ecosystem. This decision isolates the CA from existing PKI tools, browsers, and validation libraries that expect RFC 6960 compliant responses.

The custom format prevents integration with Certificate Transparency systems, blocks browser-based certificate validation, and requires custom client implementations for every system. Organizations must build custom integration layers, significantly increasing deployment complexity and limiting practical utility.

Resolution requires implementing full RFC 6960 compliance with ASN.1 DER encoding while maintaining backward compatibility during transition. This affects the entire client-server protocol and requires careful coordination to avoid breaking existing integrations.

4.2 Missing certificate extensions

The certificate generation process lacks essential X.509 extensions required for proper PKI operation. Missing Authority Key Identifier and Subject Key Identifier extensions create problems for certificate chain validation and path building. The absence of Certificate Policies extensions prevents policy-based validation required in enterprise environments.

The revocation infrastructure suffers from similar gaps. Missing CRL Distribution Points prevent automated revocation checking, while absent Authority Information Access extensions block automatic issuer certificate retrieval. These omissions force relying parties to implement custom discovery mechanisms, significantly increasing deployment complexity.

Resolution requires developing a comprehensive extension framework providing full RFC 5280 compliance, including configurable policy engines, name constraints support, and complete certificate path validation logic.

4.3 Side channel attacks and information leakage

The system exhibits concerning information disclosure patterns that could provide attackers with valuable intelligence about internal architecture and state. Timing variations in cryptographic operations represent the most serious vulnerability, with signature verification showing different execution times based on validity and HSM state.

Database queries reveal certificate existence through timing patterns, while nonce lookup mechanisms use standard hash operations with predictable performance characteristics. Error message patterns provide another disclosure channel, allowing attackers to distinguish between different failure conditions and map internal system architecture.

Resolution requires implementing constant-time operations, normalizing response timing through artificial delays, and sanitizing error responses to provide uniform feedback regardless of underlying failure conditions.

4.4 HSM as a single point of failure

The current architecture presents a critical vulnerability through its reliance on a single HSM instance for all cryptographic operations. This design creates a single point of failure that could render the entire CA infrastructure inoperable if the HSM becomes unavailable due to hardware failure, network connectivity issues, or maintenance requirements.

The centralized HSM approach means that all certificate signing operations, key generation, and cryptographic validations depend on a single cryptographic device. Any disruption to HSM availability immediately halts all CA operations, preventing certificate issuance, revocation, and status verification. This design violates fundamental principles of high-availability system architecture and creates unacceptable operational risks for production environments.

Furthermore, the HSM represents a single point of compromise where an attacker who gains access to the HSM could potentially sign malicious certificates, extract private key material, or manipulate the cryptographic operations that form the foundation of the CA's trust model. The lack of key distribution or threshold cryptography means that the security of the entire PKI infrastructure depends entirely on the security of a single device.

Production PKI environments require HSM redundancy through clustered HSM deployments, threshold cryptography that distributes signing operations across multiple HSMs, automated failover mechanisms that maintain service continuity during HSM maintenance or failure, and comprehensive HSM monitoring with real-time availability checking. These enhancements transform the HSM from a single point of failure into a resilient cryptographic infrastructure capable of maintaining operations under various failure scenarios.

4.5 Inadequate key validation and security enforcement

The current public key validation performs only basic ASN.1 format checking, missing critical opportunities to detect weak keys, inappropriate parameters, or compromised cryptographic material before certificate issuance. This superficial approach creates risks of issuing certificates for cryptographically weak keys, deprecated curve parameters, or keys below current security standards.

Enhanced validation requires comprehensive key strength analysis following NIST guidelines, verification of elliptic curve parameters against approved standards, checking against databases of known compromised keys, and enforcement of evolving minimum security requirements.

4.6 Incomplete revocation infrastructure

The revocation mechanism lacks critical features for production environments. The absence of revocation reason codes prevents proper classification of revocation events, hampering incident response capabilities. The system provides no mechanism for CA-initiated revocation, creating problems when certificates need revocation due to external threats but certificate holders are unavailable.

Required enhancements include revocation reason code support, CA-initiated revocation capabilities, emergency revocation procedures, bulk revocation mechanisms, and automated notification systems for timely communication of revocation events.

4.7 Lack of post-quantum cryptography support

The current implementation lacks post-quantum cryptography support, creating a significant long-term security vulnerability as quantum computing technology advances. The system relies entirely on ECDSA-SHA256, which will become vulnerable when cryptographically relevant quantum computers emerge, potentially within the next 10-15 years.

This limitation stems from the use of local-hsm, an emulated version of AWS KMS that provides a simplified HSM interface for development purposes. Unlike production AWS KMS, local-hsm does not support post-quantum algorithms such as CRYSTALS-Dilithium or CRYSTALS-Kyber, preventing the implementation of quantum-resistant cryptography within the current development environment.

The absence of post-quantum support means the CA cannot generate hybrid certificates that provide both current ECDSA compatibility and quantum resistance. This architectural limitation prevents preparation for the inevitable quantum transition and leaves the system vulnerable to future quantum attacks on elliptic curve cryptography.

Addressing this vulnerability requires migrating from local-hsm to production AWS KMS that supports post-quantum algorithms, implementing hybrid signature schemes that combine classical and post-quantum cryptography, and establishing a transition timeline for quantum-resistant certificate deployment. This migration represents a fundamental architectural change that affects all cryptographic operations and requires careful planning to maintain compatibility during the transition period.

5 Known Limitations

This chapter provides a comprehensive analysis of the known limitations of the implemented Certificate Authority system. Understanding these limitations is crucial for proper deployment, risk assessment, and future development planning. The limitations are categorized by their nature and impact on the system's functionality and security.

5.1 Functional Limitations

5.1.1 Limited Identity Verification

Limitation: The current implementation uses simplified identity verification mechanisms compared to commercial Certificate Authorities.

Description: The system only verifies email ownership and private key possession, lacking the comprehensive identity validation processes used by production CAs. Real-world CAs typically perform:

- Document-based identity verification (government IDs, business registrations)
- Domain validation for web certificates
- Extended validation procedures for high-assurance certificates
- Physical presence verification for certain certificate types

Impact: This limitation reduces the trust level that can be placed in issued certificates and makes the system unsuitable for high-stakes applications requiring strong identity assurance.

Mitigation: For production use, additional verification layers would need to be implemented, including integration with official identity verification services and document validation systems.

5.1.2 Simplified Certificate Profiles

Limitation: The system implements basic X.509 certificate profiles without the full range of extensions and policies used in production environments.

Description: The current implementation lacks support for:

- Certificate policy extensions
- Authority Information Access (AIA) extensions
- Subject Alternative Names (SAN) for multiple identities
- Key usage constraints and extended key usage (TODO: Check)
- Certificate transparency integration
- Custom extension fields for specific use cases

Impact: Generated certificates may not be fully compatible with all applications and use cases that require specific certificate extensions or policy compliance.

Mitigation: Future development should include configurable certificate profiles and support for standard X.509 extensions based on intended use cases.

5.1.3 Limited Revocation Mechanisms

Limitation: The system only implements Certificate Revocation Lists (CRL) without support for more modern revocation mechanisms.

Description: Missing revocation features include:

- Online Certificate Status Protocol (OCSP) support
- OCSP stapling capabilities
- Delta CRLs for efficient updates
- Revocation reason codes and detailed status information
- Automatic revocation based on compromise detection

Impact: Revocation checking may be slower and less efficient than modern alternatives, potentially affecting application performance and user experience.

Mitigation: OCSP implementation should be prioritized for production deployment to provide real-time revocation status checking.

5.2 Security Limitations

5.2.1 Simplified Trust Model

Limitation: The system implements a simplified trust model without the hierarchical certificate chains used in production PKI deployments.

Description: Current limitations include:

- Single-level CA hierarchy (no intermediate CAs)
- No support for cross-certification
- Limited policy enforcement mechanisms
- No separation of duties for different certificate types
- Absence of certificate path validation complexity

Impact: The simplified trust model may not scale to complex organizational structures or meet enterprise security requirements that demand hierarchical trust relationships.

Mitigation: Future versions should support intermediate CAs and configurable trust hierarchies to match real-world deployment scenarios.

5.2.2 Key Management Limitations

Limitation: While HSM integration provides secure key storage, the system lacks advanced key management features found in enterprise solutions.

Description: Current key management limitations include: (TODO: Check)

- No automatic key rotation policies
- No support for key escrow or split knowledge
- Absence of hardware attestation mechanisms
- Limited integration with external key management systems

Impact: These limitations may affect long-term key security and disaster recovery capabilities in production environments.

Mitigation: Implementation of comprehensive key lifecycle management policies and procedures would enhance the security posture.

5.3 Scalability and Performance Limitations

5.3.1 Database Scalability

Limitation: The current MongoDB implementation may face scalability challenges under high certificate volume scenarios.

Description: Potential scalability issues include:

- Single database instance without sharding
- Limited optimization for high-throughput certificate operations
- No built-in caching mechanisms for frequently accessed data
- Absence of read replicas for load distribution

Impact: Performance may degrade significantly under high load, potentially affecting service availability and response times.

Mitigation: Database clustering, caching layers, and performance optimization would be necessary for high-volume production deployment.

5.3.2 HSM Performance Constraints

Limitation: Cloud HSM operations introduce latency that may limit certificate issuance throughput.

Description: HSM-related performance limitations include: (TODO: Check)

- Network latency for each signing operation
- Limited concurrent HSM operations
- No local caching of non-sensitive operations
- Dependency on external HSM service availability
- Potential cost implications for high-volume operations

Impact: Certificate issuance rates may be constrained by HSM performance, potentially creating bottlenecks during peak usage periods.

Mitigation: Implementation of operation batching, local caching where appropriate, and HSM capacity planning would improve performance.

5.4 Operational Limitations

5.4.1 Backup and Disaster Recovery

Limitation: The current implementation lacks comprehensive backup and disaster recovery procedures.

Description: Current limitations include:

- No automated backup procedures for certificate data
- Limited disaster recovery testing and procedures
- Absence of geographic redundancy
- No point-in-time recovery capabilities
- Limited business continuity planning

Impact: System failures could result in significant downtime and potential data loss, affecting service availability and trust.

Mitigation: Development of comprehensive backup strategies and disaster recovery procedures would be essential for production deployment.

5.5 User Experience Limitations

5.5.1 Limited User Interface Features

Limitation: The web interface provides basic functionality but lacks advanced features expected in modern certificate management systems.

Description: UI limitations include:

- Basic certificate lifecycle management
- Limited search and filtering capabilities
- No bulk operations support
- Absence of advanced certificate analytics
- Limited customization options
- No mobile-responsive design optimization

Impact: User productivity may be limited, and the system may not meet the usability expectations of modern certificate management workflows.

Mitigation: UI enhancement with advanced features and improved user experience design would increase system adoption and efficiency.

5.6 Compliance and Standards Limitations

5.6.1 Standards Compliance

Limitation: While the system implements basic X.509 standards, it lacks full compliance with advanced PKI standards and best practices.

Description: Standards compliance gaps include: (TODO: Check)

- Partial RFC 5280 compliance
- No Common Criteria evaluation
- Limited FIPS 140-2 compliance verification
- Absence of WebTrust or similar audit frameworks
- No compliance with specific industry standards (e.g., CA/Browser Forum requirements)

Impact: The system may not be suitable for environments requiring specific compliance certifications or industry-standard validation.

Mitigation: Comprehensive standards compliance assessment and implementation would be necessary for regulated industry deployment.

5.7 Future Enhancement Recommendations

To address these limitations, future development should prioritize:

1. Implementation of comprehensive identity verification mechanisms
2. Development of hierarchical CA support and intermediate certificates
3. Integration of OCSP and modern revocation mechanisms
4. Enhancement of audit logging and compliance features
5. Performance optimization and scalability improvements
6. Development of comprehensive operational monitoring
7. Implementation of disaster recovery and business continuity procedures
8. UI/UX improvements and mobile responsiveness
9. API enhancement with enterprise integration features
10. Standards compliance assessment and implementation

Understanding these limitations is essential for making informed decisions about deployment scenarios, risk assessment, and future development priorities. While these limitations exist, the current implementation provides a solid foundation for a Certificate Authority system that can be enhanced to meet more demanding requirements as needed.

6 Instructions for Installation and Execution

This chapter provides comprehensive instructions for installing, configuring, and executing the Certificate Authority system. These instructions are designed to enable readers to successfully deploy and test the system in their own environments.

6.1 System Prerequisites

6.1.1 Hardware Requirements

Minimum Requirements:

- CPU: 2 cores
- RAM: 4 GB
- Storage: 10 GB free disk space
- Network: Stable broadband internet connection

Recommended Requirements:

- CPU: 4+ cores
- RAM: 8+ GB
- Storage: 20+ GB free disk space
- Network: Stable broadband internet connection

6.1.2 Software Prerequisites

The following software must be installed on the target system:

Required Software:

- **Operating System**: Linux, macOS, or Windows with WSL2
- **Docker Engine**: Version 20.10 or higher
- **Docker Compose**: Version 2.0 or higher
- **Git**: Version 2.30 or higher (for source code retrieval)

6.2 Installation Instructions

6.2.1 Docker Installation

For Ubuntu/Debian Linux:

```
# Update package index
sudo apt update

# Install Docker
sudo apt install -y docker.io docker-compose-plugin

# Add user to docker group
sudo usermod -aG docker $USER

# Restart shell or logout/login to apply group changes
newgrp docker

# Verify installation
docker --version
docker compose version
```

For macOS:

1. Download Docker Desktop from `https://docker.com/products/docker-desktop`
2. Install the application following the provided installer
3. Launch Docker Desktop and ensure it's running
4. Verify installation in terminal:

```
docker --version
docker compose version
```

For Windows:

1. Install WSL2 following Microsoft's official documentation
2. Download Docker Desktop for Windows
3. Install with WSL2 backend enabled
4. Verify installation in WSL2 terminal or PowerShell

6.2.2 Source Code Acquisition

Option 1: Git Clone (Recommended)

```
# Clone the repository
git clone [repository-url]
cd advanced-programming-of-cryptographic-methods

# Verify directory structure
ls -la
```

Option 2: Direct Download

1. Download the project archive from the provided source
2. Extract to desired directory
3. Navigate to project root directory

6.3 Configuration Setup

6.3.1 Environment Configuration

Create a `.env` file in the project root directory with the following configuration:

```
# MongoDB Configuration
MONGO_USERNAME=admin
MONGO_PASSWORD=securepassword123

# AWS/HSM Configuration
AWS_REGION=us-east-1
AWS_ACCESS_KEY_ID=test
AWS_SECRET_ACCESS_KEY=test

# Email Service Configuration (Resend.com)
RESEND_API_KEY=your_resend_api_key_here
RESEND_FROM=noreply@yourdomain.com
```

Important Configuration Notes:

- **MongoDB Credentials**: Use strong passwords for production deployment
- **AWS Credentials**: For development, the test values work with local KMS
- **Email Service**: Sign up at resend.com for API key (required for email verification)
- **Domain Configuration**: Replace `yourdomain.com` with your actual domain

6.3.2 Email Service Setup

Resend.com Setup (Required for Email Verification):

1. Visit `https://resend.com` and create an account
2. Navigate to API Keys section in dashboard
3. Create a new API key with appropriate permissions
4. Add your domain for email sending verification
5. Update `RESEND_API_KEY` and `RESEND_FROM` in `.env` file

Alternative Email Providers: If using different email service, modify the email configuration in: `ca/internal/email/email.go`

6.4 System Execution

6.4.1 Starting the Complete System

Initial Startup:

```
# Navigate to project directory
cd advanced-programming-of-cryptographic-methods

# Build and start all services
docker compose up --build

# Alternative: Run in background
docker compose up --build -d
```

Expected Output: The system will start the following services:

- **MongoDB**: Database service on port 27017
- **Local KMS**: HSM simulation service on port 8080
- **Backend**: CA server on port 5000
- **Frontend**: Web interface on port 3000

6.4.2 Service Verification

Check Service Status:

```
# View running containers
docker compose ps

# View service logs
docker compose logs backend
docker compose logs frontend
docker compose logs mongo
docker compose logs local-kms
```

Health Check Endpoints:

```
# Backend health check
curl http://localhost:5000/v1/health

# Frontend accessibility
curl http://localhost:3000
```

6.4.3 First-Time Setup

During the initial startup, the system automatically:

1. Creates a new root key pair in the HSM
2. Generates the root certificate for the CA
3. Initializes the database schema
4. Sets up necessary indexes

To Reset the System:

```
# Stop all services
docker compose down

# Remove HSM data for clean restart
docker container rm local-kms

# Remove database data (optional)
docker volume rm advanced-programming-of-cryptographic-methods_mongo-data

# Restart system
docker compose up --build
```

6.5 System Access and Usage

6.5.1 Web Interface Access

Primary Interface:

- URL: `http://localhost:3000`
- Description: Main web interface for certificate management

Available Pages:

- **Home**: `http://localhost:3000` - System overview and navigation
- **Certificate Signing**: `http://localhost:3000/sign` - Request new certificates
- **Certificate Viewer**: `http://localhost:3000/certs` - View and revoke certificates
- **CRL Viewer**: `http://localhost:3000/crl` - View revoked certificates
- **Certificate Commitment**: `http://localhost:3000/commit` - Validate certificates

6.5.2 API Access

Base URL: `http://localhost:5000`

Key Endpoints:

- `GET /v1/health` - System health status
- `GET /v1/info/pk` - CA information and public key
- `PUT /v1/identity` - Commit identity information (email and public key)
- `PUT /v1/certificate` - Verify proof of identity and generate certificate
- `GET /v1/certificate/:serial/status` - Get the status of a certificate'
- `POST /v1/certificate/:serial/renew` -Renew an existing certificate
- `POST /v1/certificate/:serial/revoke` - Revoke an existing certificate
- `GET /v1/crl` - Certificate revocation list

Example API Usage:

```
# Get CA information
curl http://localhost:5000/v1/info

# Check system health
curl http://localhost:5000/v1/health

# Get CRL
curl http://localhost:5000/v1/crl
```

6.6 Testing and Validation

6.6.1 Basic Functionality Testing

Test Certificate Workflow:

1. Access web interface at `http://localhost:3000`
2. Navigate to `/sign` page
3. Generate a new key pair using the interface
4. Enter a valid email address for verification
5. Submit the certificate signing request
6. Check email for verification link and complete verification
7. Verify certificate appears in the system

Test Certificate Revocation:

1. Navigate to `/certs` page
2. Upload or paste a certificate for viewing
3. Use the revocation interface to revoke the certificate
4. Verify the certificate appears in the CRL at `/crl`

6.6.2 API Testing

Health Check Test:

```
curl -X GET http://localhost:5000/v1/health
# Expected: {"status":"healthy"}
```

CA Information Test:

```
curl -X GET http://localhost:5000/v1/info
# Expected: JSON with CA certificate and public key
```

6.7 Troubleshooting

6.7.1 Common Issues and Solutions

Port Conflicts:

- **Issue**: Ports 3000, 5000, 8080, or 27017 already in use
- **Solution**: Modify port mappings in `docker-compose.yml`
- **Example**: Change `"3000:3000"` to `"3001:3000"`

Email Service Issues:

- **Issue**: Email verification not working
- **Solution**: Verify `RESEND_API_KEY` is correct and domain is verified
- **Alternative**: Check email service logs: `docker compose logs backend`

Database Connection Issues:

- **Issue**: Backend cannot connect to MongoDB
- **Solution**: Verify MongoDB container is running and credentials are correct
- **Check**: `docker compose logs mongo`

HSM Service Issues:

- **Issue**: Backend cannot connect to HSM
- **Solution**: Verify local-kms container is running
- **Reset**: Remove HSM container and restart

6.7.2 Log Analysis

Viewing Detailed Logs:

```
# All services
docker compose logs -f

# Specific service
docker compose logs -f backend

# With timestamps
docker compose logs -t backend
```

Debug Mode: For additional debugging, set environment variables:

```
# In .env file
DEBUG=true
LOG_LEVEL=debug
```

6.8 Production Deployment Considerations

6.8.1 Security Hardening

For Production Use:

- Replace development certificates with production certificates
- Use real AWS KMS instead of local KMS simulation
- Configure proper firewall rules and access controls
- Enable SSL/TLS for all communications
- Implement proper backup and disaster recovery procedures
- Set up monitoring and alerting systems

6.8.2 Performance Optimization

Recommended Optimizations:

- Configure MongoDB replica sets for high availability
- Implement load balancing for multiple backend instances
- Set up CDN for frontend asset delivery
- Configure caching layers for improved performance
- Implement database indexing for query optimization

6.9 Support and Additional Resources

6.9.1 Documentation References

- **Docker Documentation**: `https://docs.docker.com`
- **MongoDB Documentation**: `https://docs.mongodb.com`
- **AWS KMS Documentation**: `https://docs.aws.amazon.com/kms`
- **Go Documentation**: `https://golang.org/doc`
- **Next.js Documentation**: `https://nextjs.org/docs`

6.9.2 Contact Information

For technical support or questions regarding this implementation:

- **Emanuele Civini**: emanuele.civini@studenti.unitn.it
- **Alessia Pivotto**: alessia.pivotto@studenti.unitn.it

This comprehensive guide provides all necessary information to successfully install, configure, and execute the Certificate Authority system. Following these instructions should result in a fully functional PKI implementation suitable for development, testing, and educational purposes.