Software Requirements Specification (SRS) of 24-25J-136

24-25J-136

Secure E-Voting Device Using Cryptographic Techniques

Student Details:

Hussain M.R.S	Rafeek A.M	Mushtaq M.B.M	Perera U.L.S.A
IT21361654	IT21318252	IT21303920	IT21278976

Supervised by – Mr. Kavinga Yapa Abeywaradana

BSc (Hons) in Information Technology specialized in Cyber Security

Department of Information Technology Sri Lanka Institute of Information Technology

Date of Submission: December 08, 2024

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to provide a detailed Software Requirements Specification (SRS) for a Secure E-Voting Device. This device is designed for use at polling stations where voters cast their votes electronically. The system incorporates advanced security measures to ensure the privacy, integrity, and transparency of the voting process. Key components include:

- Homomorphic Encryption (HE): Ensures votes remain confidential even during computations.
- Multi-Party Computation (MPC): Aggregates votes securely without revealing individual inputs.
- Zero-Knowledge Proofs (ZKP): Verifies computations without exposing sensitive data.
- Dynamic Taint Analysis: Detects unauthorized data flows in real-time.
- Blockchain Integration: Provides immutable and transparent vote storage.
- Secure APIs with SSL/TLS: Ensures encrypted communication between the device and backend servers.

1.2 Scope

This SRS document focuses on the functional, non-functional, and design requirements of the Secure E-Voting Device deployed at polling stations. The device is responsible for:

- Capturing Votes: Accepts votes through a secure user interface.
- Encrypting Votes: Uses homomorphic encryption to maintain vote confidentiality.
- Transmitting Data: Sends encrypted votes to a backend server via secure APIs with SSL/TLS.
- Verifying Computations: Utilizes Zero-Knowledge Proofs (ZKP) to verify vote integrity.
- Preventing Data Breaches: Implements Dynamic Taint Analysis to monitor data flows.
- Storing Votes Securely: Records votes immutably on a blockchain to prevent tampering.
- Auditing: Provides comprehensive logging and audit trails for transparency.

1.3 Definitions, Acronyms, and Abbreviations

Term/Acronym	Definition
Homomorphic Encryption (HE)	Encryption that allows computations on encrypted data without decryption.
Multi-Party Computation (MPC)	Secure computation method involving multiple parties with private inputs.
Zero-Knowledge Proof (ZKP)	Proves the correctness of a computation without revealing underlying data.
Dynamic Taint Analysis	Technique for tracking and analyzing data flows to detect unauthorized use.
Blockchain	Decentralized ledger for secure, transparent, and immutable vote storage.
SSL/TLS	Protocols for securing communication over a network.
API	Interface for interacting with external systems and services.
CLI	Command-Line Interface for system interactions.
CFG (Control Flow Graph)	Graphical representation of program execution paths.

1.4 Overview

This document outlines the Software Requirements Specification (SRS) for the Secure E-Voting Device. The system ensures that votes are securely captured, encrypted, transmitted, and stored. The key focus areas include:

System Architecture: Overview of the system components and their interactions.

User Interfaces: Description of interfaces for voters and administrators.

Hardware and Software Interfaces: Detailed requirements for the hardware and software components.

Security Measures: Techniques used to ensure privacy, integrity, and transparency.

Performance and Design Constraints: Requirements for latency, scalability, and security compliance.

2. Overall Description

2.1 Product Perspective

The Secure E-Voting Device enhances traditional polling station voting by providing a secure, electronic method for casting votes. The device addresses issues such as vote privacy, data integrity, and auditability through the integration of modern cryptographic techniques and security measures.

Key Differences from Traditional Systems:

Privacy Protection: Votes are encrypted using Homomorphic Encryption before transmission.

Secure Aggregation: Multi-Party Computation (MPC) allows vote counting without exposing individual votes.

Transparency: Blockchain ensures that votes are recorded immutably and can be verified.

Data Flow Monitoring: Dynamic Taint Analysis tracks data flows to prevent unauthorized access or tampering.

Verification: Zero-Knowledge Proofs (ZKP) verify computations while maintaining data confidentiality.

2.1.1 Comparison with Existing E-Voting Systems

a. Traditional E-Voting Systems

Traditional e-voting systems, whether paper-based electronic machines or remote online voting platforms, generally suffer from several inherent weaknesses that compromise the privacy, security, integrity, and transparency of the voting process. These systems often rely on straightforward encryption techniques, centralized vote aggregation, and basic audit mechanisms, which expose them to various vulnerabilities:

Lack of End-to-End Privacy:
Most traditional systems encrypt votes only during transmission. Once the votes are received by a central server, they are decrypted for aggregation and counting. This approach introduces risks where malicious insiders or attackers who compromise the server can access individual votes, thus breaching voter privacy.

Centralized Vote Aggregation:
Traditional systems perform vote counting on a single server or a small group of trusted servers. This centralized approach creates a single point of failure and increases the risk of manipulation or fraud during the tallying process. A compromised aggregation server can alter vote counts without detection.

Limited Verifiability and Transparency:
In traditional systems, verifying that votes are accurately recorded, counted, and stored is challenging. Voters typically have no way of independently verifying that their votes were included in the final tally. Audit logs, if present, are often limited in detail and can be tampered with by malicious actors.

Static Security Analysis:
Traditional e-voting systems lack real-time security monitoring. They rely on pre-deployment security checks and static code analysis, making them ill-equipped to detect and prevent runtime anomalies or unauthorized data flows during the voting process.

b. Improvements in the Proposed Secure E-Voting Device

The proposed Secure E-Voting Device significantly enhances traditional e-voting systems by incorporating modern cryptographic techniques and dynamic security measures. Key improvements include:

Homomorphic Encryption (HE)

Enhanced Privacy Protection:
The Secure E-Voting Device uses Homomorphic Encryption to ensure that votes remain encrypted during the entire process, including aggregation and tallying. Unlike traditional systems, votes are never decrypted, even during counting. This guarantees that voter privacy is maintained throughout the process, and no single entity can access individual votes.

Secure Computations on Encrypted Data:
With HE, computations (e.g., vote tallying) can be performed directly on encrypted data. This eliminates the need to expose votes in plaintext, thus reducing the risk of data breaches and insider attacks.

Multi-Party Computation (MPC)

Decentralized and Secure Aggregation:
The system uses Multi-Party Computation (MPC) for secure vote aggregation. Instead of relying on a single server, votes are aggregated across multiple independent parties, each holding a part of the encrypted data. This decentralized approach ensures that no single party can access or manipulate the votes. Even if one party is compromised, the integrity of the tally remains secure.

Fault Tolerance and Security Against Collusion:
MPC ensures that even if some computing parties collude, they cannot decrypt individual votes. This provides a higher level of security compared to centralized aggregation methods.

Zero-Knowledge Proofs (ZKP)

Verifiable Computations:
The integration of Zero-Knowledge Proofs (ZKP) allows the system to verify that vote

aggregation and other computations are performed correctly without revealing any sensitive information. Voters and auditors can be assured that the final tally is accurate without needing to access the underlying votes.

Increased Transparency:
ZKP enables public verification of the computation process, enhancing transparency and trust in the voting system. This addresses the verifiability issues present in traditional e-voting systems.

Dynamic Taint Analysis

Real-Time Security Monitoring:
The Secure E-Voting Device employs Dynamic Taint Analysis to track data flows and detect unauthorized or malicious activities in real time. Unlike traditional systems that rely on static security checks, dynamic taint analysis identifies anomalies as they occur during the voting process.

Prevention of Data Breaches:
By marking user inputs (votes) as tainted and monitoring their flow through the system, any attempt to access or modify sensitive data inappropriately is detected and logged. This proactive approach helps prevent data breaches and ensures the integrity of the voting process.

c. Summary of Improvements

Feature	Traditional E-Voting Systems	Proposed Secure E-Voting Device
Privacy	Encrypted during transmission only	Encrypted end-to-end with Homomorphic Encryption
Aggregation	Centralized vote counting	Decentralized aggregation with Multi-Party Computation
Verifiability	Limited or no verifiable computation	Verifiable with Zero-Knowledge Proofs
Security Monitoring	Static security analysis	Real-time monitoring with Dynamic Taint Analysis
Transparency	Limited audit capabilities	Immutable blockchain storage and detailed audit logs

2.1.2 System Interfaces

Operating Systems: The device runs on a secure, embedded Linux-based operating system.

APIs: The device uses secure APIs for communication with backend servers.

SSL/TLS: All data transmitted through the APIs is encrypted using SSL/TLS.

Blockchain Network: Votes are stored and verified through a blockchain network (e.g., Ethereum or Hyperledger).

The below diagram illustrates the system architecture

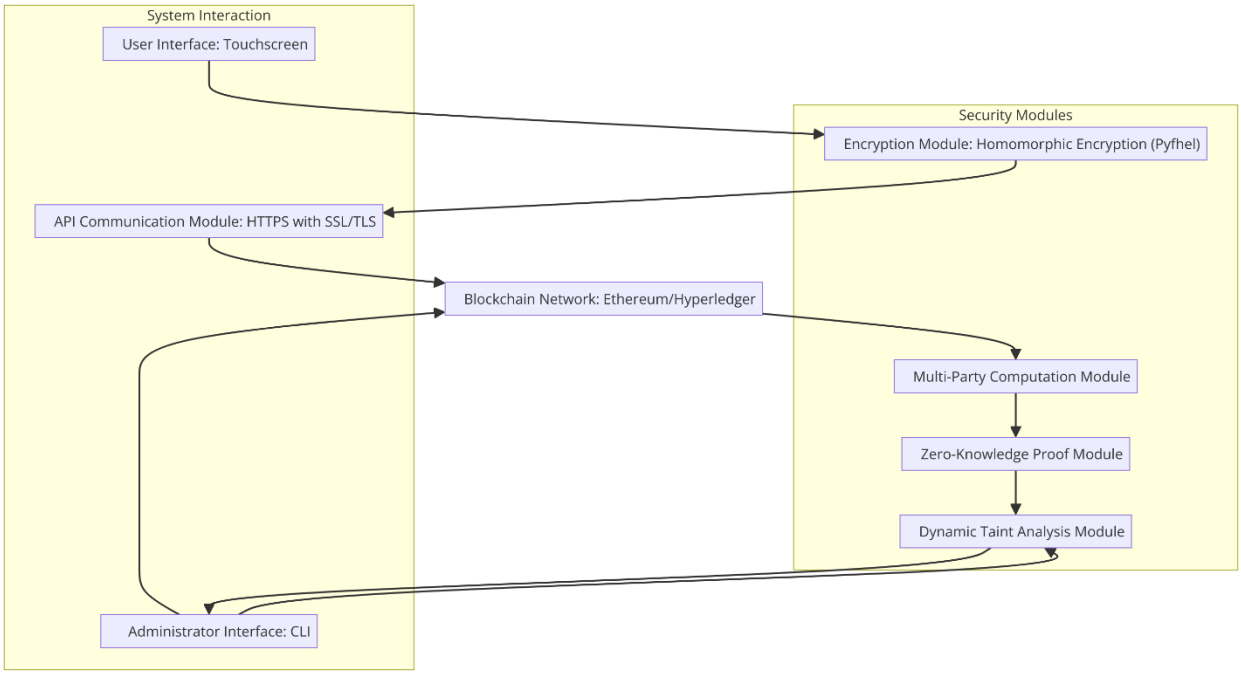

Figure 1: Comprehensive system architecture

2.1.3 User Interfaces

Voter Interface:

Touchscreen Display: Allows voters to select their candidate securely.

Vote Confirmation:

Displays a summary for voters to verify before final submission.

Administrator Interface:

CLI (Command-Line Interface): For configuring the device, troubleshooting, and exporting logs.

Audit Logs:

Provides detailed logs of voting activities for verification and auditing.

2.1.4 Hardware Interfaces

Processor: Minimum 2 GHz Dual-Core CPU for handling encryption and data transmission.

RAM: Minimum 4 GB (8 GB recommended for better performance).

Storage: 1 GB SSD for encrypted votes, logs, and dependencies.

Network Interface: Ethernet/Wi-Fi module for secure communication with the backend.

Display: Capacitive touchscreen for user interactions.

Power Supply: Uninterruptible Power Supply (UPS) for continuous operation during power outages.

2.1.5 Software Interfaces

Operating System: Embedded windows for secure and stable operation.

Encryption Libraries:

Pyfhel: For homomorphic encryption.

gmpy2: For efficient modular arithmetic.

Blockchain Libraries:

web3.py: For interacting with Ethereum or Hyperledger blockchain networks.

Logging Libraries:

Python logging module: For structured and secure audit logging.

Communication Libraries:

Requests Library: For secure API requests using HTTPS.

2.1.6 Communication Interfaces

Secure API Communication:

Protocol: HTTPS with SSL/TLS encryption.

Authentication: API key or token-based authentication.

Function: Sends encrypted votes to backend servers and receives acknowledgments.

Blockchain Network Communication:

Protocol: JSON-RPC for blockchain interactions.

Function: Submits votes as transactions to the blockchain and retrieves vote verification data.

2.1.7 Memory Constraints

The memory requirements for the Secure E-Voting Device are as follows:

RAM Requirements:

Minimum: 4 GB RAM for basic encryption and vote processing.

Recommended: 8 GB RAM to handle concurrent operations and large datasets efficiently.

Storage Requirements:

Minimum: 1 GB SSD for storing encrypted votes, audit logs, and system dependencies.

Blockchain Data: Each vote submission to the blockchain consumes approximately 1 KB per transaction, which scales based on the number of votes.

Encryption Operations:

Homomorphic encryption tasks typically consume around 100 MB of RAM per vote encryption.

Multi-Party Computation (MPC) operations during vote aggregation may use up to 200 MB of RAM.

Taint Analysis and CFG Visualization:

Dynamic taint analysis consumes up to 100 MB of RAM for medium-sized control flow graphs (CFGs) with up to 50 nodes.

2.1.8 Operations

The Secure E-Voting Device supports the following operations:

1. Vote Submission:

Description: Voters select candidates via a touchscreen interface. The vote is validated, encrypted using homomorphic encryption, and transmitted to the backend server.

Process:

User selects a candidate on the touchscreen.

The vote is validated for correctness.

The vote is encrypted using the Pyfhel library.

The encrypted vote is sent via a secure API (HTTPS with SSL/TLS).

A confirmation message is displayed.

2. Vote Aggregation:

Description: Aggregates encrypted votes securely using Multi-Party Computation (MPC).

Process:

Encrypted votes are sent to different computing parties.

Votes are aggregated using additive secret sharing and modular arithmetic.

The aggregation result is verified using Zero-Knowledge Proofs (ZKP).

3. Audit Logging:

Description: Records detailed logs of all operations for transparency and auditing purposes.

Process:

Every operation (vote submission, encryption, transmission) is logged.

Logs are encrypted and stored securely.

Administrators can export logs via the CLI for auditing.

4. Dynamic Taint Analysis:

Description: Monitors data flows to detect unauthorized access or tampering.

Process:

User inputs are marked as tainted.

Data flows are tracked dynamically using control flow graphs (CFGs).

Anomalies or unauthorized flows trigger alerts for administrators.

5. Blockchain Storage:

Description: Stores votes immutably on a blockchain for integrity and transparency.

Process:

Encrypted votes are submitted as blockchain transactions.

Each vote is recorded with a unique hash and timestamp.

Voters can verify their votes using a unique receipt.

6. Error Handling:

Description: Handles errors gracefully and logs anomalies.

Examples:

Network Errors: Retries vote submission up to three times before logging an error.

Encryption Errors: Displays an error message and prompts the user to retry.

Given Below is expected Data Flow diagram of the system

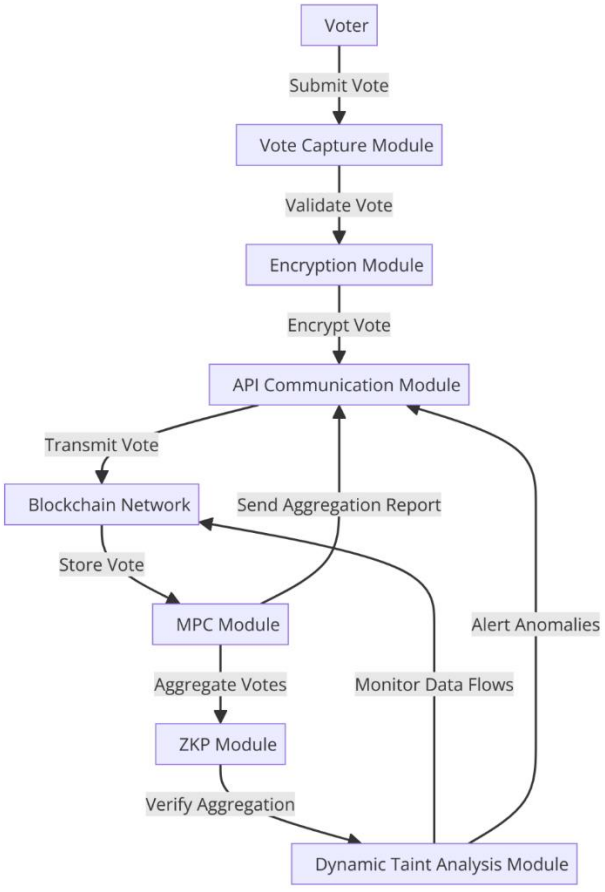

Figure 2: Data Flow Diagram

2.2 Product Functions

The main functions of the Secure E-Voting Device include:

Vote Capture and Validation:

Function: Allows voters to select and validate their votes.

Details:

Touchscreen interface for candidate selection.

Immediate validation of input to ensure correctness.

Homomorphic Encryption:

Function: Encrypts votes to ensure confidentiality.

Details:

Uses the Pyfhel library for encryption.

Allows computations on encrypted data without decryption.

Secure Vote Transmission:

Function: Transmits encrypted votes to the backend.

Details:

Uses HTTPS with SSL/TLS encryption.

API-based communication for secure data exchange.

Multi-Party Computation (MPC):

Function: Aggregates votes securely without revealing individual inputs.

Details:

Uses additive secret sharing for secure aggregation.

Modular arithmetic for efficient computation.

Zero-Knowledge Proofs (ZKP):

Function: Verifies computations while preserving data privacy.

Details:

zk-SNARKs for efficient proof generation.

Ensures vote aggregation integrity.

Dynamic Taint Analysis:

Function: Monitors data flows to detect anomalies.

Details:

Tracks tainted data dynamically.

Visualizes data flows using CFGs.

Blockchain Storage:

Function: Stores votes immutably for transparency.

Details:

Votes are recorded as blockchain transactions.

Ensures votes cannot be altered or deleted.

Audit Logging:

Function: Maintains logs for transparency and auditing.

Details:

Logs encryption, transmission, and aggregation activities.

Provides logs in a secure and exportable format.

2.3 User Characteristics

Voters:

Skills: Basic computer literacy.

Role: Cast votes using the touchscreen interface.

Expectations: Intuitive interface, clear instructions, and immediate feedback.

Polling Station Administrators:

Skills: Familiarity with system configuration and troubleshooting.

Role: Set up the device, manage operations, and export logs.

Expectations: CLI for system management and diagnostics.

Election Auditors:

Skills: Basic understanding of cryptographic techniques and log analysis.

Role: Verify the integrity of the voting process.

Expectations: Access to detailed audit logs and verification tools.

2.4 Constraints

Latency:

Vote Submission: Must complete within 5 seconds.

Blockchain Confirmation: Votes must be recorded within 30 seconds.

Scalability:

Must handle up to 10,000 votes per polling station during peak hours.

Security Compliance:

Must comply with GDPR, CCPA, and other data protection regulations.

Hardware Limitations:

Operates on a 2 GHz Dual-Core CPU with 4 GB RAM.

2.5 Assumptions and Dependencies

Internet Connectivity:
Reliable internet access for API communication and blockchain interactions.

Cryptographic Libraries:
Libraries like Pyfhel for homomorphic encryption and web3.py for blockchain interactions.

Secure Infrastructure:
Deployment in a secure environment with firewalls and access controls.

Authentication Systems:
Voter authentication handled by polling station officials.

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

a. Voter Interface

The **voter interface** is a user-friendly touchscreen system designed to ensure a seamless and secure voting experience. It provides a clear and intuitive way for voters to select and submit their votes. The **vote selection screen** displays a list of candidates with corresponding buttons for easy selection. Clear instructions guide the voter through the process, minimizing confusion and ensuring accessibility for all users.

Once the voter has made their selection, the interface presents a **confirmation screen** that summarizes their choice. This screen allows the voter to verify their selection and offers the option to either confirm or modify their vote before final submission. This step ensures that voters have full control over their choices and reduces the risk of mistakes.

The system also provides **feedback and notifications** in real-time. Upon successful submission, a confirmation message assures the voter that their vote has been recorded. In the event of errors, such as invalid selections or network issues, the interface displays clear error messages with guidance on how to resolve the issue. This dynamic feedback system helps maintain voter confidence and ensures a smooth voting process.

b. Administrator Interface

The **administrator interface** is a robust Command-Line Interface (CLI) designed to provide polling station officials and system administrators with the tools needed to manage and troubleshoot the Secure E-Voting Device. This interface supports essential tasks for system setup, diagnostics, and maintenance.

During **system setup and configuration**, administrators can use the CLI to set network parameters such as IP addresses and API endpoints. The interface also facilitates the initialization of encryption keys and secure tokens, ensuring that the device is ready for secure operations. These configuration tasks are streamlined to simplify deployment and minimize errors.

For **diagnostics and monitoring**, the CLI offers commands to check the device's status, connectivity, and encryption operations. This helps administrators quickly identify and resolve issues, ensuring the system remains operational during the voting process. Additionally, the CLI supports **audit log management**, allowing administrators to export encrypted logs for auditing purposes. The interface also provides real-time views of log updates, giving administrators the ability to monitor system activities and detect anomalies as they occur.

3.1.2 Hardware Interfaces

The Secure E-Voting Device is designed with robust hardware components to ensure reliable performance, security, and usability during the voting process.

The device is powered by a **2 GHz Dual-Core CPU**, which handles critical tasks such as encryption, data processing, and secure communications. This processing capability ensures the system can efficiently perform homomorphic encryption and transmit votes securely without lag.

To support these operations, the device requires a minimum of **4 GB of RAM**. For optimal performance during peak voting periods, such as large-scale elections, **8 GB of RAM** is recommended. This ensures smooth and concurrent execution of multiple processes, including vote encryption, real-time monitoring, and secure transmission.

The device uses a **1 GB SSD** for storage, which is sufficient for holding encrypted votes, audit logs, and essential system dependencies. The use of SSD storage ensures faster read and write operations, contributing to overall system efficiency.

A **capacitive touchscreen display** with a minimum resolution of **1024x768 pixels** provides a responsive and intuitive interface for voters. This touchscreen allows voters to easily select their candidates and confirm their choices, with clear visuals and accurate touch response enhancing the voting experience.

For secure communication with backend servers, the device is equipped with an **Ethernet/Wi-Fi module**. This module supports encrypted data transmission over HTTPS, ensuring that votes are securely sent to the backend without risk of interception.

To ensure continuous operation even during power disruptions, the device includes an **Uninterruptible Power Supply (UPS)**. The UPS provides backup power, allowing the system to remain operational during short-term power outages and ensuring that no votes are lost or compromised.

3.1.3 Software Interfaces

The Secure E-Voting Device is supported by a suite of software components that ensure secure, efficient, and reliable operation throughout the voting process.

The device runs on **Embedded Windows**, offering a secure, stable, and customizable operating system for managing e-voting functions. Embedded Windows is well-suited for embedded applications, providing robust performance, security features, and the flexibility to meet the specific needs of polling stations.

To implement **homomorphic encryption**, the system uses the **Pyfhel library**, which allows for secure computations on encrypted data without the need for decryption. This ensures that votes remain confidential during the entire process. For performing efficient modular arithmetic required in **Multi-Party Computation (MPC)** operations, the system leverages the **gmpy2**

library, which provides optimized mathematical functions necessary for secure vote aggregation.

For blockchain integration, the system uses **web3.py**, a Python library designed for interacting with blockchain networks such as **Ethereum** or **Hyperledger**. This allows the device to submit encrypted votes as immutable transactions, ensuring transparency and integrity in the voting process.

The system maintains comprehensive audit trails using the **Python logging module**. This library generates structured, encrypted logs that document all critical operations, such as vote submissions, encryption processes, and data transmissions. These logs can be exported and reviewed by administrators and auditors to verify the integrity and security of the system.

For secure communication, the device employs the **Requests library**, which facilitates API interactions over **HTTPS** with **SSL/TLS encryption**. This ensures that all data transmitted between the device and the backend servers is protected against interception and tampering, maintaining the confidentiality and integrity of the votes.

3.1.4 Communication Interfaces

Secure API Communication

The Secure E-Voting Device ensures safe and reliable transmission of encrypted votes to the backend server using a **secure API**. The communication is conducted over **HTTPS with SSL/TLS encryption**, protecting the data from interception and tampering. To enhance security, the system employs **API key-based authentication** to ensure that only authorized devices can communicate with the backend server.

The primary function of the secure API is to **send encrypted votes** to the backend server and **receive acknowledgments and status updates** confirming successful receipt or highlighting any issues. This process ensures the integrity and confidentiality of the voting data throughout transmission. For a more in-depth understanding of how the API communication is implemented, refer to the relevant code snippets in the **Appendices** section.

Blockchain Network Communication

To maintain transparency and immutability, the system interacts with a **blockchain network** to record votes as transactions. The communication with blockchain nodes is carried out using the **JSON-RPC protocol**, which facilitates seamless interaction with blockchain platforms like **Ethereum** or **Hyperledger**.

The primary function of blockchain communication is to **submit encrypted votes** as immutable transactions on the blockchain. Once recorded, these transactions cannot be altered or deleted, ensuring the integrity of the voting data. Additionally, the system can **retrieve vote verification and audit data** from the blockchain, enabling voters and auditors to independently verify that votes have been accurately recorded and counted. View Appendix section for more understanding on the code.

Logging and Auditing

The Secure E-Voting Device maintains comprehensive records of all critical activities through **structured logging and auditing mechanisms**. The system generates logs in **JSON or XML format**, ensuring that the data is organized, easy to parse, and compatible with various auditing tools.

These logs serve as a critical component for maintaining transparency and accountability. They provide detailed records of operations such as vote submissions, encryption processes, and data transmissions. The logs are **encrypted** to protect sensitive information and can be accessed by administrators and auditors for verification purposes. This audit trail ensures that any anomalies or issues can be identified and resolved promptly, contributing to the overall integrity and trustworthiness of the voting process.

.

3.2 Performance Requirements

The Secure E-Voting Device is designed to operate efficiently under strict performance constraints to ensure a seamless voting experience, even during peak usage periods.

Latency

The system ensures minimal delays during critical operations. The entire **vote submission process** — including capturing the vote, encrypting it using homomorphic encryption, and transmitting it securely to the backend server — must be completed within **5 seconds**. Once the vote reaches the backend, it should be recorded on the **blockchain within 30 seconds** to ensure timely and immutable storage.

Throughput

To support large-scale elections, the device is capable of processing up to **500 votes per hour** during peak voting periods. This ensures that high voter turnout can be managed efficiently without delays or bottlenecks.

Response Time

The system prioritizes a responsive user experience. All **user interactions**, such as selecting a candidate on the touchscreen and confirming the vote, should respond within **1 second**. This quick response time minimizes voter frustration and ensures a smooth, intuitive voting process.

Memory Usage

Efficient memory management is critical to maintaining system performance. Each **homomorphic encryption operation** should consume no more than **100 MB of RAM** to ensure the device can handle encryption tasks without performance degradation. Additionally, **blockchain interactions**, such as submitting a vote to the blockchain network, should use no more than **50 MB of RAM** to maintain efficient communication and data handling.

3.3 Design Constraints

The Secure E-Voting Device is designed to meet specific requirements for security, performance, and maintainability, while operating within defined limitations. These constraints ensure reliable operation during elections but also highlight potential challenges to consider.

Security Compliance

The system must adhere to essential **data protection regulations**, including the **General Data Protection Regulation (GDPR)** and the **California Consumer Privacy Act (CCPA)**. These regulations ensure that voter data is handled responsibly, maintaining privacy, integrity, and transparency throughout the voting process.

Encryption Standards

To protect sensitive data, the system employs industry-standard encryption techniques. Data stored on the device is encrypted using **AES-256**, a robust encryption algorithm that ensures data confidentiality at rest. During data transmission, the system uses **SSL/TLS protocols** to encrypt communications between the device and backend servers, preventing unauthorized interception and tampering.

Hardware Limitations

The Secure E-Voting Device is designed to operate efficiently within defined hardware constraints. The device requires a minimum of a **2 GHz Dual-Core CPU** to handle encryption, data processing, and secure communications. Additionally, it operates with a minimum of **4 GB RAM** to support vote capture, encryption, and real-time monitoring without performance degradation. Performance may degrade if the hardware specifications are not met.

Modular Architecture

The system follows a **modular architecture** to facilitate ease of updates, maintenance, and scalability. Each core function is encapsulated within its own module, allowing for independent development and testing. The key modules include:

- **Vote Capture Module**: Manages the user interface and vote selection process.
- **Encryption Module**: Handles homomorphic encryption to secure votes.
- **Communication Module**: Manages secure API communication and blockchain interactions.
- **Logging and Auditing Module**: Records system activities and generates encrypted audit logs.

This modular design ensures that updates to one component do not disrupt the overall system, enhancing maintainability and flexibility.

Scalability

To accommodate varying election sizes and increasing voter turnout, the system is designed for **scalability**. It supports integration with additional **blockchain nodes** and **backend servers** to handle higher voting loads. This ensures that the system remains responsive and efficient, even during large-scale elections with thousands of concurrent voters.

3.3.1 Limitations

The following are potential limitations of the Secure E-Voting Device:

Scalability Limits:

The system is designed to handle up to **500 votes per hour per device**. During large-scale elections with high voter turnout, deploying multiple devices at each polling station is necessary to avoid delays.

Power Dependency:

The device relies on an **Uninterruptible Power Supply (UPS)** to ensure continuous operation. Extended power outages beyond the UPS capacity may disrupt the voting process.

Network Dependency:

The system requires a stable internet connection for secure transmission of votes and interaction with the blockchain network. **Network failures** or poor connectivity may delay vote submission and storage.

Hardware Constraints:

The device operates with a minimum of **2 GHz Dual-Core CPU** and **4 GB RAM**. Performance may degrade if the hardware specifications are not met.

Blockchain Latency:

Recording votes on a blockchain may introduce latency, especially during network congestion. Votes are typically confirmed within **30 seconds**, but delays may occur under high blockchain load.

3.4 Software System Attributes

3.4.1 Reliability

The system must operate continuously during the voting period without crashes or data loss.

Uptime Requirement: 99.9% uptime during elections.

Failover Mechanism: The system should automatically switch to a backup server in case of failure.

3.4.2 Availability

The device must remain available and operational throughout the entire voting period.

Backup Power Supply: UPS ensures availability during power outages.

3.4.3 Security

Data Encryption:

All votes must be encrypted using homomorphic encryption before transmission.

Secure Transmission:

Data must be transmitted via HTTPS with SSL/TLS encryption.

Audit Logs:

Maintain encrypted logs for all operations to support transparency and verification.

3.4.4 Maintainability

Modular Design:

Components such as vote capture, encryption, and communication modules should be easily upgradable.

Documentation:

Comprehensive documentation for administrators and developers for system setup, troubleshooting, and maintenance.

3.4.5 Performance

The system must meet the specified latency and throughput requirements under normal and peak load conditions.

3.5 Traceability Matrix

The following table maps the **functional and non-functional requirements** to their respective system components or functions to ensure all requirements are addressed.

Requirement ID	Requirement Description	System Component/Function
FR-1	The system must allow voters to cast votes via a touchscreen interface.	**Vote Capture Module**
FR-2	The system must encrypt votes using homomorphic encryption.	**Encryption Module (Pyfhel)**
FR-3	The system must securely transmit encrypted votes to the backend.	**API Communication Module (HTTPS)**
FR-4	The system must aggregate votes using MPC.	**MPC Module**
FR-5	The system must verify computations using Zero-Knowledge Proofs.	**ZKP Module**
FR-6	The system must store votes immutably on a blockchain.	**Blockchain Network**
FR-7	The system must log all activities for auditing purposes.	**Audit Logging Module**
NFR-1	The system must process votes within 5 seconds.	**Encryption & Communication Modules**
NFR-2	The system must handle 500 votes per hour during peak load.	**Vote Capture & Transmission Modules**

Requirement ID	Requirement Description	System Component/Function
NFR-3	The system must comply with GDPR and CCPA regulations.	**Security & Compliance Measures**
NFR-4	The system must detect unauthorized data flows in real time.	**Dynamic Taint Analysis Module**

4. Supporting Information

APPENDIX A: Figures

A.1 Figure 1: Comprehensive system architecture

A.2 Figure 2: Data Flow Diagram

APPENDIX B: ZKP and Block Chain Integration Snippets

B.1 The following code snippets illustrate the generation and verification of ZKP proofs:

Proof Generation:

```python
# ZKP Proof Generation
def generate_zk_snark_proof(vote_data, secret_key):
    combined_data = f"{vote_data}:{secret_key}"
    proof = sha256(combined_data.encode()).hexdigest()
    return proof
```

Proof Verification:

```python
# ZKP Proof Verification
def verify_zk_snark_proof(vote_data, secret_key, proof):
    combined_data = f"{vote_data}:{secret_key}"
    expected_proof = sha256(combined_data.encode()).hexdigest()
    return expected_proof == proof
```

B.2 Blockchain Code Snippets

The following code snippet demonstrates how votes are stored on the blockchain:

```python
def add_block(self, voter_id, proof):
    """
    Adds a new block to the blockchain.
    Args:
        voter_id (str): Unique voter ID.
        proof (str): Cryptographic proof to validate the vote.
    """
    block = {
        "voter_id": voter_id,
        "proof": proof,
        "status": "Processing",  # Default status
        "previous_hash": self.chain[-1]["hash"] if self.chain else "0",
    }
    block["hash"] = sha256(f"{block}".encode()).hexdigest()  # Compute block hash
    self.chain.append(block)
```

B.3 Sample Vote Casting and Verification Process

Vote Casting:

The voter selects a candidate and generates a ZKP proof.

```
1. Cast a Vote
2. Exit
Enter your choice: 1
Welcome to the Voting System!
Enter your Government-issued ID Number: 2001XXXXXXXX
Enter your Voter ID: ID0001
Please cast your vote:
Options: A or B
Enter your choice: A
Your unique ZKP code is: ac724dfce7273abe0b9315189ec5ceaf90fdc32305a5dbad84f23008f22453b9
Your vote has been casted successfully and saved!
```

The proof and anonymized metadata are submitted to the blockchain.

```
(env) PS C:\Users\Esclipse\Desktop\Uni\Research\Project> python viewBlock.py
Blockchain Data:
{'voter_id': 'ID0001', 'proof': 'ac724dfce7273abe0b9315189ec5ceaf90fdc32305a5dbad84f23008f22453b9', 'status': 'Processing', 'previous_hash': '0', 'hash': '23d0801b7cf423f68881ecf61
8d59e9af61650e704ee83c4205f73a4c9de07c1'}
```

Vote Verification:

The voter inputs their unique ZKP code on the verification page.

```
(env) PS C:\Users\Esclipse\Desktop\Uni\Research\Project> python checkStatus.py
>>
Vote Status Checker
Enter your ZKP code to check your vote status: ac724dfce7273abe0b9315189ec5ceaf90fdc32305a5dbad84f23008f22453b9
Your vote status is: Processing
```

The system queries the blockchain to verify the vote status.

B.4 Secure SSL connection using Flask

```python
@app.route("/cast-vote", methods=["POST"])
def cast_vote():
    """
    HTTPS endpoint to cast a vote and store it in the blockchain.
    Request:
        JSON payload with "voter_id", "id_number", and "vote".
    Response:
        JSON object with success or error message.
    """
    blockchain = load_blockchain()
    data = request.json

    # Extract fields from the request
    voter_id = data.get("voter_id")
    id_number = data.get("id_number")
    vote = data.get("vote")

    # Validate input
    if not voter_id or not id_number or not vote:
        return jsonify({"error": "Missing 'voter_id', 'id_number', or 'vote'"}), 400

    # Check for duplicate voting
    for block in blockchain:
        if block["voter_id"] == voter_id:
            return jsonify({"error": "Duplicate voting detected! You have already voted."}), 403

    # Generate zk-SNARK proof
    secret_key = f"{voter_id}:{id_number}"
    zk_snark_proof = generate_zk_snark_proof(vote, secret_key)

    # Add the vote to the blockchain
    block = {
        "voter_id": voter_id,
        "proof": zk_snark_proof,
        "status": "Processing",
        "previous_hash": blockchain[-1]["hash"] if blockchain else "0"
    }
    block["hash"] = sha256(f"{block}".encode()).hexdigest()
    blockchain.append(block)
```

APPENDIX C: Real Time Monitoring Snippets

C.1 TaintTracker (Essential)

The TaintTracker class is responsible for tracking and marking data as tainted and providing methods to check taint status and the source of taint.

Functions:

mark_as_tainted(data, source="unknown") (Essential)

Marks the given data as tainted and associates it with a source.

is_tainted(data) (Essential)

Checks whether the given data is tainted.

get_taint_source(data) (Essential)

Retrieves the source of the taint for the given data.

```python
class TaintTracker:
    def __init__(self):
        self.tainted_data = {}  # Dictionary to track tainted variables and their sources

    def mark_as_tainted(self, data, source="unknown"):
        """Mark data as tainted by wrapping it in a TaintedData object."""
        if not isinstance(data, TaintedData):
            tainted_data = TaintedData(data, source)
        else:
            tainted_data = data  # If already tainted, retain original
        self.tainted_data[id(tainted_data)] = tainted_data
        return tainted_data

    def is_tainted(self, data):
        """Check if data is tainted."""
        return id(data) in self.tainted_data or isinstance(data, TaintedData)

    def get_taint_source(self, data):
        """Retrieve the source of the taint."""
        if isinstance(data, TaintedData):
            return data.source
        return "untainted"
```

C.2. TaintedData (Essential)

The TaintedData class is a wrapper for data that allows taint metadata to be associated with it and propagates the taint through data transformations.

Functions:

__init__(data, source) (Essential)

Initializes the TaintedData object with the given data and taint source.

__getattr__(attr) (Essential)

Allows wrapped data to access original string methods while propagating taint.

__repr__() (Desirable)

Provides a developer-friendly string representation of the TaintedData object.

```python
class TaintedData:
    def __init__(self, data, source):
        """Wrap the original data with taint metadata."""
        self.data = data
        self.source = source

    def __str__(self):
        return str(self.data)

    def __repr__(self):
        return f"TaintedData({repr(self.data)}, source={repr(self.source)})"

    def __getattr__(self, attr):
        """Allow wrapped data to access original methods (e.g., .upper(), .lower())."""
        original_attr = getattr(self.data, attr)
        if callable(original_attr):
            def wrapper(*args, **kwargs):
                result = original_attr(*args, **kwargs)
                if isinstance(result, str):
                    return TaintedData(result, self.source)
                return result
            return wrapper
        return original_attr
```

C.3. DataFlow (Essential)

The DataFlow class simulates the flow of data through the system, performing validation and interacting with the TaintTracker.

Functions:

__init__(tracker) (Essential)

Initializes the DataFlow object with user input marked as tainted.

validate() (Essential)

Validates the data to ensure it is free from malicious patterns.

traverse_cfg() (Essential)

Simulates the traversal of data through the Control Flow Graph (CFG) and logs the path taken. Detects anomalies if data reaches processing or storage without proper validation.

```python
class DataFlow:
    def __init__(self, tracker):
        user_input = input("Enter your data: ")  # Prompt user for input
        self.data = tracker.mark_as_tainted(user_input, source="user_input")
        self.is_validated = False

    def validate(self):
        """Sanitize the data and check for invalid patterns."""
        if isinstance(self.data, TaintedData):
            # Check if the input is non-empty
            if len(self.data.data.strip()) == 0:
                print("Validation failed: Input is empty.")
                return False

            # Check for malicious patterns (e.g., script tags, SQL keywords)
            malicious_patterns = re.compile(r"<.*?>|(\b(SELECT|DROP|INSERT|DELETE|UPDATE|SCRIPT)\b)", re.IGNORECASE)
            if malicious_patterns.search(self.data.data):
                print("Validation failed: Malformed or malicious input detected.")
                return False

            # If all checks pass, mark as validated
            self.is_validated = True
            self.data = self.data.data  # Remove taint after validation
        return self.is_validated
```

C.4. ControlFlowGraph (Desirable)

The ControlFlowGraph class represents the control flow of data through different stages (e.g., Input, Validation, Processing, Storage) using a directed graph.

Functions:

add_node(node, description) (Desirable)

Adds a new node to the control flow graph.

add_edge(start_node, end_node, condition) (Desirable)

Adds an edge between two nodes with an associated condition.

```python
# Initialize the Control Flow Graph
cfg = nx.DiGraph()
cfg.add_node("Input", description="User input is received")
cfg.add_node("Validation", description="Input is validated")
cfg.add_node("Processing", description="Validated input is processed")
cfg.add_node("Storage", description="Processed data is stored")
cfg.add_edge("Input", "Validation", condition="Input received")
cfg.add_edge("Validation", "Processing", condition="Input validated")
cfg.add_edge("Processing", "Storage", condition="Input processed")
```

C.5. TaintFlowVisualizer (Optional)

Description:

The TaintFlowVisualizer class handles the visualization of taint propagation using libraries like NetworkX and Matplotlib.

Functions:

visualize_taint_flow(cfg, path_log, tracker, data_flow, validation_passed) (Desirable)

Visualizes the control flow graph with annotations for taint status.

```
def visualize_taint_flow(cfg, path_log, tracker, data_flow, validation_passed):
    """Visualize the CFG with taint flow annotations."""
    annotated_cfg = cfg.copy()
    for node in annotated_cfg.nodes:
        if node in path_log:
            # Mark node status based on validation and taint
            if node == "Validation" and not validation_passed:
                annotated_cfg.nodes[node]['status'] = "Tainted"
            elif tracker.is_tainted(data_flow.data):
                                                    us'] = "Tainted"
            else
                (variable) annotated_cfg: Any
                annotated_cfg.nodes[node]['status'] = "Sanitized"
        else:
            annotated_cfg.nodes[node]['status'] = "Skipped"

    # Define node colors
    color_map = []
    for node in annotated_cfg.nodes(data=True):
        status = node[1].get('status', 'Skipped')
        if status == "Tainted":
            color_map.append("red")
        elif status == "Sanitized":
            color_map.append("green")
        else:
            color_map.append("gray")

    # Draw the graph
    pos = nx.spring_layout(annotated_cfg)
    nx.draw(
        annotated_cfg,
        pos,
        with_labels=True,
        node_color=color_map,
        node_size=2000,
        font_size=10,
```

```
# Draw the graph
pos = nx.spring_layout(annotated_cfg)
nx.draw(
    annotated_cfg,
    (function) node_color: list
    node_color=color_map,
    node_size=2000,
    font_size=10,
    font_color="white"
)
edge_labels = nx.get_edge_attributes(cfg, "condition")
nx.draw_networkx_edge_labels(annotated_cfg, pos, edge_labels=edge_labels, font_size = 8)

plt.title("Taint Flow Visualization")
plt.show()
```

References

1. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable Zero Knowledge via Cycles of Elliptic Curves," in *Proceedings of the 33rd Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT 2014)*, 2014, pp. 276–294.

2. D. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

3. V. Buterin, "A Next-Generation Smart Contract and Decentralized Application Platform," *Ethereum Whitepaper*, 2014.

4. S. Panja and B. Roy, "A Secure End-to-End Verifiable E-Voting System Using Zero-Knowledge Proof and Blockchain," in *Advances in Cyber Security: Principles, Techniques, and Applications*, 2021, pp. 109–128.

5. "Z-Voting: A Zero-Knowledge Based Confidential Voting on Blockchain," in *Proceedings of the IEEE International Conference on Blockchain Technology and Applications*, 2023.

6. S. Heiberg and J. Willemson, "Modeling Threats of a Voting Method," in *Handbook of Research on E-Government Readiness for Information and Service Exchange: Utilizing Progressive Information Communication Technologies*, IGI Global, 2014, pp. 167–188. doi: 10.4018/978-1-4666-5820-2.ch007.

7. J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

8. K. Hough and J. Bell, “A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, Article 26, pp. 1–43, Dec. 2021. doi: 10.1145/3485464.

9. R. Cramer, I. Damgård, and J. B. Nielsen, *Secure Multiparty Computation and Secret Sharing*. Cambridge, U.K.: Cambridge University Press, 2015.

10. Pyfhel Contributors, *Pyfhel: A Python Library for Homomorphic Encryption*, 2023.

11. D. Escudero, “An Introduction to Secret-Sharing-Based MPC,” *Cryptology ePrint Archive*, Paper 2023/023, 2023.

12. *Homomorphic Encryption Standardization*. 2023.

13. *Pyfhel Documentation*. 2023.

14. *Python 3.9 Documentation*. 2023.

15. *SHA-256 Specification*. 2023.