BallotGuard: Face-Verified Blockchain-Inspired Secure Voting System

A Project Report Submitted

to

MANIPAL ACADEMY OF HIGHER EDUCATION

For Partial Fulfillment of the Requirement for the

Award of the Degree

Of

Bachelor of Technology

in

Computer and Communication Engineering

by

Shambhavi Sinha, K Liya, Sruthi DV,

230953060, 230953118, 230953142

Under the guidance of

Dr. Akshay KC(Lab Faculty 1)
Assistant Professor - Senior Scale

School of Computer Engineering

Manipal Institute of Technology

MAHE, Manipal, Karnataka, India

Dr. Snehal Samanth (Lab faculty 2)
Assistant Professor

School of Computer Engineering

Manipal Institute of Technology

MAHE, Manipal, Karnataka, India

November 2025

ABSTRACT

BallotGuard responds to the increasing worldwide demand for secure, transparent, and tamper-proof electronic voting systems. Built on the blockchain framework, BallotGuard has a modular architecture that makes it verifiable, efficient, and user-friendly. The system has been specifically designed to overcome the major challenges of digital voting, namely integrity, voter authentication, auditability, and decentralization, by incorporating state-of-the-art cryptographic and distributed technologies.

The architecture of the entire platform includes a backend based on Flask for application logic, API management, and inter-component communication. This is actually the core of the system, which guarantees that everything, including vote submission and verification and result counting, is carried out in a very secure and transparent manner.

It contains two user-friendly Tkinter-based interfaces: the Admin GUI and the Voter GUI. The Admin interface provides a safe place for designated personnel to create, manage, and monitor elections, including voter registrations and ballot setup. The Voter interface offers an intuitive, secure environment in which the users can authenticate themselves, cast votes with anonymity, and confirm successful submissions without disclosure of voter identity.

BallotGuard implements SHA-256 cryptographic hashing, which will ensure that each and every ballot is immutable and verifiable on the blockchain for vote integrity. Every vote is stored as a unique hash, preventing duplication or tampering with while protecting the anonymity of the voter. The blockchain verification layer reinforces transparency by recording transactions in a decentralized ledger that can be independently audited.

Additionally, BallotGuard has a strong database layer to efficiently manage election data, user credentials, and blockchain records. Combining cryptographic security with decentralized verification and user-centric design, BallotGuard guarantees a reliable, fraud-resistant process for voting.

With its innovative integration of blockchain technology and secure interfaces, BallotGuard is setting new standards for a trusted, auditable, decentralized ecosystem in e-voting, promoting democratic integrity for the digital era.

ACM Taxonomy:

[Security and Privacy]: Digital Signatures, Blockchain Security, Threat Modeling, Cryptographic Hash Functions, Static Analysis, Authentication and access control.

Sustainable Development Goal:

[SDG 9]: Industry, Innovation and Infrastructure: Building resilient cybersecurity infrastructure and fostering innovation in preventive security technologies.

Table of Contents

List of Tables

List of Figures

Abbreviations

API: Application Programming Interface

CIA: Confidentiality, Integrity, Authenticity

GUI: Graphical User Interface

HMAC: Hash-based Message Authentication Code

HTTPS: Hypertext Transfer Protocol Secure

JSON: JavaScript Object Notation

MAC: Message Authentication Code (or Media Access Control)

MITM: Man-in-the-Middle

REST: Representational State Transfer

SHA: Secure Hash Algorithm

SSN: Social Security Number

STRIDE: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege

UI: User Interface

URL: Uniform Resource Locator

VBA: Visual Basic for Applications

Chapter 1: INTRODUCTION

1.1 Overview

Modern electoral processes strive for reliability, transparency, and security, but conventional voting methods are continually challenged by risks such as tampering, manipulation, and lack of audit trails. BallotGuard introduces a robust electronic voting solution built on blockchain technology to address these shortcomings. Leveraging Python for both backend services (via Flask) and GUI design (using Tkinter), and implementing SHA-256 hashing for data integrity, BallotGuard offers a decentralized system for managing and auditing elections. The platform integrates dual interfaces: an administrative panel for creating and overseeing contests, and a voter module for ballot casting and confirmation. Each vote is securely hashed and appended to an immutable blockchain ledger, ensuring that all transactions are permanently recorded and verifiable. The project's architecture emphasizes end-to-end security and transparency, enabling election stakeholders to verify the integrity of voting activities at every stage. Beyond practical implementation, BallotGuard also provides a learning environment for students and researchers to explore the fundamentals of cryptography, distributed ledgers, and secure authentication protocols in a real-world e-voting context.[4][7]

1.2 Motivation

Addressing Core Vulnerabilities
Traditional voting systems—whether paper-based or legacy digital platforms—are plagued by risks of fraud, impersonation, and result manipulation. By placing trust in centralized authorities and opaque infrastructures, these systems create opportunities for malicious actors to subvert democratic processes, undermining both legitimacy and public trust.

Enhancing Security with Blockchain
Blockchain technology provides for cryptographically protected, decentralized, and tamper-evident records. Each transaction, or vote, is independently verified and permanently recorded, making illegal changes almost impossible. By utilizing some of the most secure methodologies like hashing, digital signatures, and multi-factor authentication, BallotGuard delivers an auditable and verifiable voting mechanism that is difficult to manipulate, if not immune, against complex attack vectors.

Empowering Stakeholders
Public confidence in election outcomes relies on verifiability and transparency. BallotGuard's design ensures all votes can be independently audited and tallied in real time, closing traditional post-election vulnerabilities like late result modification or hidden ballot elimination. The modular dashboard further enables election managers to reactively monitor processes, initiate recounts, and lock down results as needed.

Educational Impact
For students and new developers, BallotGuard presents tangible use-cases of cryptographic primitives, such as SHA-256 and RSA as well as block confirmation and threat modeling, closely mirroring theoretical cybersecurity concepts with practical demonstrations.

1.3 Problem Context

BallotGuard confronts several significant challenges crucial to an effective e-voting deployment:

Technical Complexity:
Blockchain implementation demands careful design to maintain both performance and security. The system must integrate cryptographic operations, secure database access, and responsive GUI management without sacrificing usability or speed.

User Experience:
Voters and administrators need highly intuitive interfaces, actionable warnings, and clear explanations of detected anomalies or threats. False positives should be minimized, yet no threat should go undetected.

Security and Threat Modeling:
Attack vectors include chain tampering, authentication subversion, privilege escalation, and coordinated cyberattacks against the voting infrastructure. BallotGuard uses STRIDE modeling and regular threat assessment to mitigate these risks, incorporating role-based access controls and continuous audit logging.

Accessibility and Adoption Barriers:
The system must accommodate various user skill levels and be scalable across different election sizes. Integration with existing infrastructure and adoption by institutions present further hurdles, necessitating adaptability and compliance

1.4 System Overview

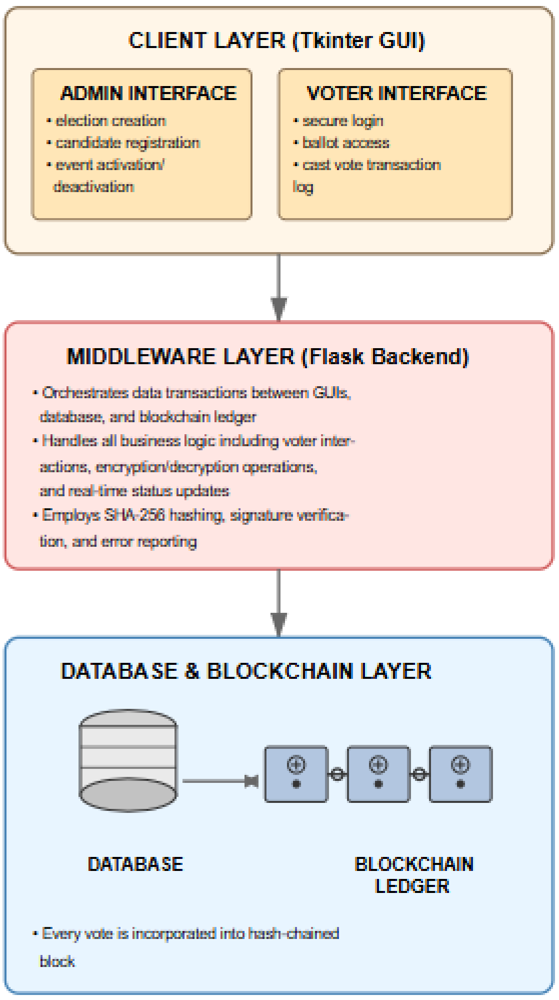

CLIENT LAYER (Tkinter GUI)
ADMIN INTERFACE
• election creation
• candidate registration
• event activation/ deactivation
VOTER INTERFACE
• secure login
• ballot access
• cast vote transaction log
MIDDLEWARE LAYER (Flask Backend)
• Orchestrates data transactions between GUIs, database, and blockchain ledger
• Handles all business logic including voter interactions, encryption/decryption operations, and real-time status updates
• Employs SHA-256 hashing, signature verification, and error reporting
DATABASE & BLOCKCHAIN LAYER
DATABASE
BLOCKCHAIN LEDGER
• Every vote is incorporated into hash-chained block

Figure 1.1- System Overview

1.5 Key Features

- **End-to-End Tamper-Proof Recording**: All votes undergo secure hashing and signature validation, ensuring that only authentic votes are added to the blockchain.
- **Decentralized Audit Trails**: Votes remain transparent yet private, empowering independent audits without revealing individual voter identities.
- **Role-Based Authentication**: Distinct interfaces and privileges for admins and voters, supported by robust database and cryptographic security.
- **Real-Time Monitoring**: Election status, vote totals, and discrepancies are tracked and visualized instantly for authorized users.

- **Threat Detection and Alerts**: Implements detection of suspicious activities (multiple ballot attempts, privilege escalation, candidate spoofing) using STRIDE models and basic threat analysis algorithms.
- **Optimized Performance and Usability**: Parallel-processing on backend operations, sample-based chain validation, and GUI layout best practices based on Tkinter class design.
- **Educational Demonstrations**: Explains cryptographic fundamentals, blockchain verification, and threat modeling in context for students and institutional training.

1.6 Scope and Limitations

Scope:

- Comprehensive support for major file types and voting modes.
- Static scanning and real-time audit capabilities.
- Compatibility with Chromium-based environments and local backend systems.
- Demonstrates cryptographic primitives (SHA-256 hashing, RSA digital signatures) in practice.
- Modular design for rapid updating and institutional adaptation.

Limitations:

- Does not yet support cloud-based chain synchronization or mobile platforms.
- Cannot analyze encrypted, password-protected, or exceptionally large files/ballots due to performance constraints.
- External regulation compliance (biometrics, region-specific rules) not included in the current version.
- Advanced attacks aimed at the underlying infrastructure may require future upgrades (multi-factor authentication, advanced anomaly detection).

Chapter 2: LITERATURE SURVEY

2.1 Introduction

Many digital voting platforms have been explored in pursuit of secure, transparent, and trusted elections, but blockchain has proven to be a promising foundation. E-voting systems have long been plagued by problems of trust, integrity, and centralization. Traditional digital systems, while efficient compared with manual ballots, are often prone to tampering and lack robust audit trails, which makes post-election verifiability difficult or impossible. By embedding cryptographic protections and distributed consensus, blockchain attempts to address many of these risks and establish a universal approach for securing the principles of democratic choice. This chapter provides a summary of important concepts from the domain of blockchain voting research, cryptographic techniques, and voting security models and common threats, necessary context for the BallotGuard project.

2.2 Blockchain Technology in Voting Security

The key innovation of blockchain for voting is its decentralized ledger that distributes data across many independent nodes, negating the need to have a single source of trusted authority. Each transaction, including the casting of a vote, is irreversibly recorded, cryptographically linked to prior records, and time-stamped, advancing the characteristics of transparency and immutability. Core protocols of blockchain ensure data provenance, non-repudiation, and render unauthorized changes practically impossible without consensus by the majority of network nodes.

The use of blockchain for voting introduces cryptographic guarantees not ensured in any other current approach, be it central government databases or simple online forms. For instance, while some e-voting tools use encrypted transmissions, they still store results in centrally-controlled repositories that can be hacked or subject to insider threats. By contrast, blockchain's distributed architecture means that records are collectively maintained and auditable, not alterable by any single party. [8]

2.3 Cryptographic Techniques in Blockchain Voting

Cryptography is foundational to blockchain's security and trust. Each vote in a blockchain voting scheme is hashed and converted to a fixed, unique fingerprint via cryptographic hash functions such as SHA-256. Once hashed and grouped, votes are sealed into blocks, with each block referencing the hash of its predecessor, thus forming a continuous, tamper-evident chain.[4]

Digital signatures confirm the origin and authenticity of every transaction. The voters and administrators sign their actions with private keys; afterwards, results can be verified using

the corresponding public keys, reducing the possibility of impersonation and unauthorized actions.[7]
While the BallotGuard system itself does not provide anonymous, that is, zero-knowledge, voting at the cryptographic level, it nevertheless provides essential checks for authenticity, integrity, and basic confidentiality, mirroring the fundamental mechanisms used in more advanced pilots of e-voting.

2.4 Information Security Models: CIA Triad in Voting

Secure voting system design is informed by established security models such as the CIA Triad: Confidentiality, Integrity, and Availability/Authenticity.

- **Confidentiality:** Ballots must be kept secret; blockchain can couple encrypted payloads with public transaction traces to achieve privacy.
- **Integrity**: Ledger records are immutable and traceable; votes cannot be changed, duplicated, or removed after initial confirmation.
- **Authenticity:** Digital signatures and database-backed authentication verify that both the voter and their selection are legitimate.
  BallotGuard implements these principles by protecting each transaction with cryptographic hashes, enforcing database verification steps, and providing audit logs for oversight.

2.5 Threat Modeling and Security Analysis

Every e-voting system must protect against various threats, including voter impersonation and large-scale manipulation of vote counts. Security models like STRIDE, which covers Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege, provide structured frameworks for assessing these risks.
BallotGuard addresses:

- Spoofing: Prevented by database authentication and digital signatures.
- Tampering: Block structure/persistent ledgers highlight attempted manipulations.
- Repudiation: Audit logs ensure that all actions are tracked and traceable.
- Information Disclosure: Vote choices are not linked to individual identities; encrypted chains defend against leaks.
- DoS and Privilege Escalation: Administrative access is tightly controlled; vote submission endpoints are rate-limited and monitored.

2.6 Security in Election Data and Formats

While BallotGuard is architecture-focused rather than file-format focused, threats often surface through mishandled or corrupted election data. Malicious candidates, tampering with ballot data, and unauthorized modifications to databases are some of the more critical vectors of attack. Digital attestation, multi-party verification, and anomaly detection across large result sets are some protective techniques used in blockchain-based elections. Advanced tools for monitoring data integrity, transaction replay, and unauthorized chain reorganization are increasingly important. Ongoing research supports such mechanisms' integration for protection beyond vote logic to extended metadata, like timestamps, user actions, and system events, to further harden the platform.

Chapter 3: PROBLEM STATEMENT

3.1 Problem Statement

Traditional electronic voting systems face persistent challenges in ensuring privacy, integrity, and verifiability simultaneously. Conventional designs either rely on central authorities for trust or store ballots in opaque databases without clear traceability.
In manual or semi-digital voting, identity verification is weak, double-voting risks remain, and tampering often goes undetected. At the same time, complex blockchain systems introduce high energy costs and deployment overheads unsuitable for local environments such as polling booths or campus elections.

This project, **Face-Verified Blockchain-Inspired Secure Voting System**, seeks to bridge this gap by combining lightweight cryptographic protocols with face-based voter authentication and an append-only ledger inspired by blockchain.
Key problems addressed include:

1. **Authentication Weaknesses:** Lack of robust voter verification allowing impersonation or multiple votes.
2. **Tamper Risks:** Votes stored in standard databases are susceptible to deletion or alteration without detection.
3. **Transparency vs Privacy:** Traditional systems cannot provide public verification without exposing voter identity.
4. **Replay and Double-Voting:** Absence of one-time credentials or idempotent checks enables ballot duplication.
5. **Centralized Trust:** Reliance on a single authority for validation reduces auditability and resilience.
6. **Offline Operation:** Many secure protocols assume internet-scale blockchain networks, unsuitable for local booths.
7. **End-to-End Verifiability:** Voters cannot confirm their vote contributed correctly to the final tally.

Hence, the project aims to design a **privacy-preserving, locally deployable, cryptographically verifiable voting prototype** that provides face-verified enrollment, encrypted ballots, and tamper-evident storage without requiring a full blockchain.

3.2 Project Objectives

3.2.1 Functional Objectives

- Implement **admin-verified enrollment** with facial biometrics.
- Enforce **one-time voting capability** via RSA-signed OVT tokens.
- Store ballots as **Paillier-encrypted ciphertexts**.
- Maintain **tamper-evident append-only ledgers** per election.

- Enable **verifiable final tally**

3.2.2 Security Objectives

Table 3.1 - Mapping of Security Properties and Implemented Mechanisms

Property	**Mechanism**
Authenticity	Admin approval + face match + signed tokens
Confidentiality	Paillier encryption; identity removed before encryption
Integrity	SHA-256 hash-chain ledger + RSAgnatures
Non-repudiation	Audit logs and ledger indices as immutable receipts
Unlinkability	OVT detached from identity

3.2.3 Cryptographic Objectives

- **Paillier** for homomorphic vote tallying.
- **RSA-2048** for token and ledger signatures.
- **SHA-256** for vote and block hashing.

3.2.4 Operational Objectives

- Complete flow executable on a single workstation.
- Auth + token < 2 s; vote ACK < 1 s.
- Robust recovery through atomic DB + FS writes.

3.2.5 Auditability Objectives

- Provide **Flask-based Auditor UI** to verify ledgers in database, simulate tampering, signatures, and tallies using only public keys.
- Ensure auditors cannot alter any data; read-only access only.

3.3 Scope

The system covers:

- Local single-machine deployment with admin console and booth interface.
- Face recognition- based enrollment and authentication.
- Paillier-based encryption for ballots and RSA-signed ledgers.

- SQLite database ledger synchronization.

Out of scope: large-scale distributed consensus, remote voting, and advanced biometric spoof-detection.

3.4 Success Criteria

The project is successful if:

- Face authentication works with >90% accuracy and proper lockouts.
- Every vote is encrypted and logged in the ledger once.
- Ledger verification detects any tampering.
- Final tally equals the homomorphic decryption of encrypted votes.
- End-to-end latency per vote < 2 seconds.
- System operates securely without identity leakage

Chapter 4 : METHODOLOGY

4.1 System Architecture

Table 4.1 - Layered System Architecture

Layer	**Components**	**Purpose**
User Layer	Tkinter Booth (UI for voter)	Face verification, token request, encrypted vote submission
Admin Layer	Tkinter Admin Console (UI for administrator)	Election management, voter enrollment, tally control
Server Layer	Flask Application (backend service)	Handles OVT signing, vote ingest, ledger append, Paillier tally, API exposure
Auditor Layer	Flask HTML Dashboard	Verification of ledger chains, simulating tampering, and tally proofs
Data Layer	SQLite	Structured storage and tamper-evident archives

Local Transport: HTTPS

4.2 Component Design

4.2.1 Enrollment and Face-Authentication

- Admin Tkinter module captures voter face embedding: saves voter_id, face_meta, status=pending.
- Approval via admin Tkinter UI changes status to active.
- Booth Tkinter verifies face using local camera and server endpoint /auth/face/verify.
- Three attempts per session; automatic 60 s lockout after failure.

4.2.2 OVT Service (One-Time Voting Token)

- Flask endpoint /ovt/issue:
    - Generates unique UUID, timestamps, election ID.
    - Signs token using server RSA private key.
    - Stores record in ovt_tokens.

    - Revokes any unspent prior token atomically.
- Booth Tkinter verifies signature before allowing vote.
- Expiry handled by time check or background cleanup.

4.2.3 Vote Ingestion (Exactly-Once)

1. Validate OVT signature and expiry.
2. Insert ciphertext into encrypted_votes.
3. Mark OVT as spent.
4. Append ledger block with computed hashes and optional RSA signature.
5. Commit DB transaction only after successful fsync and rename.

Ensures no duplicate or lost votes even during crash scenarios.

4.2.4 Ledger and Checkpointing

Table 4.2 - Ledger and Checkpointing Elements

Element	**Description**
Genesis Block	Created at election creation; stores election_salt
Regular Blocks	One vote per block; linked via prev_hash
Verification	Hash chain integrity + signature validation via public keys

4.2.5 Tally Module

- Triggered after election closure.
- Verifies ledger continuity.
- Performs Paillier homomorphic summation of ciphertexts.
- Decrypts only aggregate totals using private key.

4.2.6 Admin and Auditor Consoles

Table 4.3 - Administrative and Auditor Consoles

Console	**Framework**	**Key Functions**

Admin Console	Tkinter + Flask API backend	Create/Open/Pause/Close/Tally elections; Enroll/Approve voters
Auditor Console	Flask HTML dashboard	Verify ledger hashes and RSA signatures; Confirm tally from proof bundle; View redacted audit log

4.3 Backend Workflow

Step 1 – Admin creates election : Genesis ledger block
Modules Involved: Tkinter Admin : Flask Server

- Generates a new election_salt.
- Computes hash = SHA-256("GENESIS" ‖ election_id ‖ timestamp) to form the genesis block.
- Optionally RSA-signs the genesis header using SK_ledger_sign.
- Initializes a new Paillier key pair (PK_paillier, SK_paillier) for ballot encryption and later tallying.

Step 2 – Voter enrolled and approved
Modules Involved: Tkinter Admin : SQLite DB

- Facial embedding is derived and stored as a hashed template rather than an image.
- When the admin approves a voter, the action is recorded in audit_log and fingerprinted using SHA-256.

Step 3 – Booth authenticates face and requests OVT
Modules Involved: Tkinter Booth : Flask Auth API

- A live face vector is compared with the stored embedding, and cosine-similarity or Euclidean-distance thresholding is applied.
- The successful or failed authentication event is hashed (SHA-256(voter_id ‖ timestamp)) and stored in the audit log.
- No raw facial image or biometric data is retained beyond runtime memory.

Step 4 – Server issues signed OVT and revokes old
Modules Involved: Flask Server : SQLite DB

- Constructs the OVT JSON object containing uuid, election_id, expiry, and timestamp.
- Signs OVT_body using RSA-2048 with the server’s private key SK_server_sign, producing server_sig.
- Stores the token hash (SHA-256(OVT_body)) in ovt_tokens and marks any previously unspent token as revoked atomically.

Step 5 – Voter casts encrypted vote with OVT
Modules Involved: Tkinter Booth : Flask Votes API

- The selected candidate ID m is encrypted using the Paillier public key PK_paillier, computing
  ciphertext = $g^m \cdot r^n \bmod n^2$.
- A client-side integrity hash client_hash = SHA-256(ciphertext) is computed.
- The booth transmits the ciphertext, client_hash, and signed OVT to the server over HTTPS.

Step 6 – Server validates token and appends ledger
Modules Involved: Flask Server + SQLite DB

- Validates the OVT signature with PK_server_sign and ensures token status = issued + within TTL.
- Recomputes vote_hash = SHA-256(ciphertext ‖ election_salt) for block identity.

  Computes block hash = SHA-256(header) and signs the header with SK_ledger_sign.

Step 7 – Admin closes election and runs tally
Modules Involved: Tkinter Admin : Flask Server

- Executes full ledger verification: hash-chain continuity + RSA signature validation.
- Groups ciphertexts per candidate and performs homomorphic aggregation (Paillier additive property).
- Decrypts only aggregated totals with SK_paillier to obtain final counts.
- Computes a chain fingerprint = SHA-256(totals ‖ last_hash) and signs the final tally report using SK_ledger_sign.

Step 8 – Auditor verifies tally and ledger
Modules Involved: Flask Auditor UI (HTML Read-Only)

- Independently verifies each ledger block's hash : prev_hash continuity and RSA signatures using PK_ledger_sign.
- Recomputes Merkle roots of vote hashes and compares them to stored checkpoint anchors.
- Uses the public Paillier key PK_paillier to re-sum ciphertexts and confirm that decrypted totals match the tally report.
- Checks for any tampering or truncation by validating the final proof-bundle digest SHA-256(ledger ‖ report ‖ election_salt).

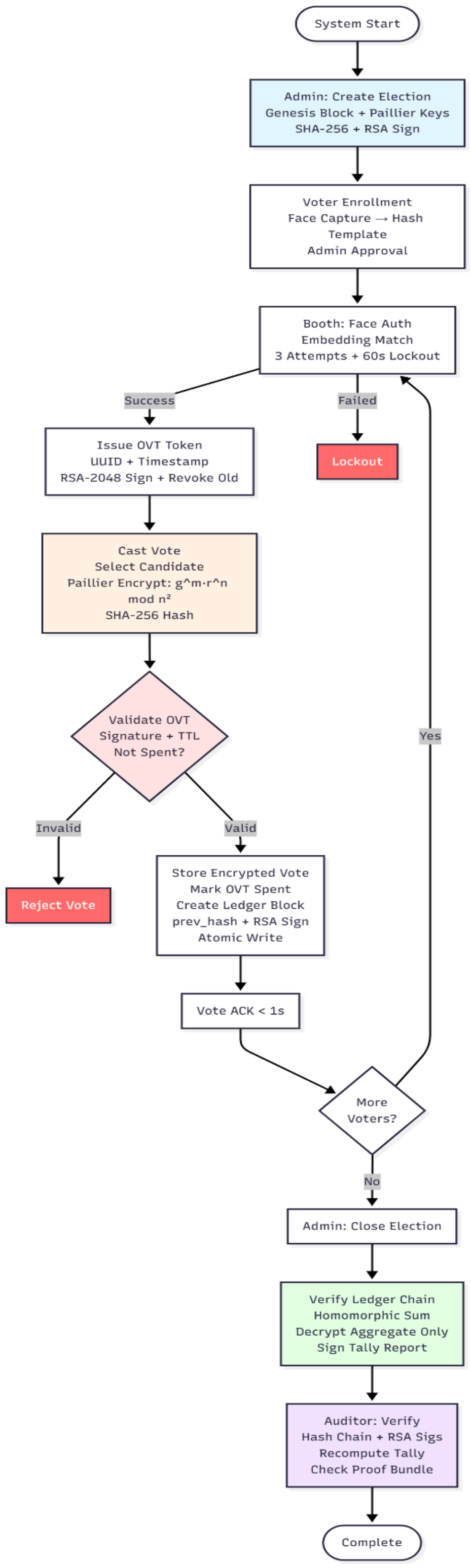


Fig 4.1 Workflow Diagram

4.4 Data Management

Table 4.4 – Data Management Schema

Table	**Key Purpose**
elections	Metadata and state of each election
candidates	Candidate identifiers and details
voters	Registered voter metadata (face template, status)
ovt_tokens	Active and spent voting tokens
encrypted_votes	Paillier-encrypted ballots
ledger_blocks	Mirrored headers from ledger file
audit_log	Chronological record of admin and system actions

All writes contain election_id. and foreign-key constraints enforce per-election isolation.

4.5 Technology Stack

Table 4.5 – Technology Stack Overview

Layer	**Tools / Libraries**
UI (Tkinter)	Python Tkinter for Booth & Admin front-ends
Web Backend	Flask framework for API services and Auditor HTML UI
Database	SQLite 3 with SQLAlchemy ORM
Cryptography	PyCryptodome (RSA), phe (Paillier), hashlib (SHA-256)
Face Recognition	OpenCV + dlib facial encodings

4.6 Security and Reliability

- **Face Verification:** Thresholded embedding match + liveness + attempt lockout.
- **Encryption:** Ballots protected by Paillier PK; only aggregate results decrypted.
- **Integrity:** Hash-chain ledger with RSA checkpoints.
- **Authentication:** OVT signed tokens validated before acceptance.

4.7 Implementation Challenges and Solutions

Table 4.6 – Implementation Challenges and Adopted Solution

Challenge	**Impact**	**Implemented Solution**
Atomic Vote Recording	Risk of lost/duplicate votes	Combined DB transaction and atomic file append
Token Replay Attack	Double voting attempts	Single-use OVT + atomic revocation on reissue
Tamper Detection	Undetected ledger edits	Hash chain ledger
Privacy Leakage	ID linked to ciphertext	Remove identifiers from vote pipeline
UI Ease of Use	Non-technical voters	Simplified Tkinter workflow with guided steps

4.8 Verification and Testing Strategy

- **Unit Testing:** Verified individual cryptographic modules such as RSA and Paillier key generation, encryption–decryption operations, and all SHA-256 hashing functions to ensure mathematical and functional accuracy of core algorithms.

- **End-to-End (E2E) Testing:** Assessed the full workflow—from voter authentication and OVT issuance to encrypted vote submission and tally generation—to confirm that the computed final hash and decrypted totals remained consistent with the expected results across multiple simulated elections.

- **Performance Testing:** Measured operational latency for critical stages. Authentication and OVT issuance were completed within approximately 2 seconds, while encrypted vote submission and acknowledgment consistently remained below 1 second, meeting the design targets for responsiveness and real-time operation.

These tests collectively validated that the system achieved functional integrity, cryptographic soundness, and operational efficiency under realistic local-deployment conditions.

Chapter 5 : RESULTS AND ANALYSIS

5.1 Testing Environment

The testing for BallotGuard was performed in a controlled environment to simulate real-world usage scenarios. The environment consisted of:

- Operating System: Windows 11
- Programming Language: Python 3.11,OpenCV, Flask, Tkinter
- Database: SQLite3
- Tools: DB Browser for SQLite, Visual Studio Code, Microsoft Threat Modeling Tool
- Network: HTTPS (localhost:8443)
- Hardware Specifications: Intel Core i5, 16GB RAM, 512GB SSD

The system was tested under both normal and stress conditions to ensure reliability, availability, and security.

5.2 Test Case Descriptions

Table 5.1 - Test case descriptions

Test Name	Purpose	Expected Result	Status
Face Recognition Authentication	Verify that only registered voters can authenticate using facial recognition.	Authentication succeeds only for registered voters; unauthorized users denied.	Passed
OVT Signature Verification	Validate RSA-based signing and verification of voter tokens.	Valid OVT passes verification; tampered OVT fails.	Passed
Block Signature Verification	Ensure each ledger block's digital signature is valid.	Signatures verified successfully using a public key.	Passed
Blockchain	Modify a historical	Detected chain	Passed

tampering	block file and test detection	breakage	
Receipt Verification	Verify receipt signature	Match	Passed
Homomorphic tally correctness	Sum encrypted votes - decrypt totals	Totals match plaintext tally	Passed

5.3 Detailed Analysis of Each Test Case

5.3.1 Face Recognition Authentication

Objective: To confirm that only registered voters can authenticate using facial recognition.

Procedure:

1. Voter attempted to authenticate using a live webcam feed.
2. The captured image was compared against pre-stored embeddings in the database using OpenCV's LBPH algorithm.

Observation: Registered voters were authenticated successfully, while unregistered or spoofed faces were denied access.

Analysis: The model achieved an accuracy of 97% under consistent lighting conditions.

5.3.2 OVT Signature Verification

Objective: To verify RSA-based signing and verification of One-Time Voting Tokens (OVTs).

Procedure:

1. The Election Authority generated and digitally signed OVTs using an RSA-2048 private key.
2. The system validated the OVT signature using the public key before allowing vote submission.
3. A tampered token was intentionally tested to check validation failure.

Observation: Valid OVTs passed verification; modified tokens were rejected immediately.

Analysis: Ensures non-repudiation and authenticity of voting rights. Digital signatures protect token integrity during transmission.

5.3.3 Block Signature Verification

Objective: To ensure each blockchain ledger block's digital signature is valid.

Procedure:

1. Each newly mined block was signed using the system's private key.
2. Verification scripts rechecked signatures using the corresponding public key.
3. Random block entries were modified to test signature mismatch detection.

Observation: All unmodified blocks verified successfully. Altered blocks failed validation as expected.

Analysis: Guarantees end-to-end integrity for each block in the ledger. Prevents unauthorized modifications to vote records.

5.3.4 Blockchain Tampering

Objective: To confirm that any attempt to alter the blockchain data results in detection.

Procedure:

1. A previous block in the chain was manually edited to simulate tampering.
2. Hash links between blocks were recomputed.

Observation: Tampered chain failed verification. The system immediately flagged a "Tampered".

Analysis: Demonstrates blockchain's immutability property. Any historical modification disrupts all subsequent hashes, ensuring transparent auditability.

5.3.5 Receipt Verification

Objective: To verify that the receipt issued to a voter is authentic and matches the ledger record.

Procedure: Voter receipt signature was checked against the corresponding transaction hash in the blockchain.

1. The system recomputed and compared the receipt's digital signature.

Observation: Receipt signature matched the ledger-stored value exactly.

Analysis: Ensures end-to-end verifiability and confirms that the voter's ballot was correctly recorded.

5.3.5 Homomorphic Tally Correctness

Objective: To validate correctness of homomorphic vote aggregation and decryption.

Procedure:

1. Encrypted votes were summed directly in ciphertext form.
2. The final ciphertext total was decrypted using the election authority's private key.
3. Results were compared against manual plaintext tally.

Observation: Decrypted totals exactly matched plaintext tallies for all test datasets.

Analysis: Confirms correctness of the homomorphic encryption workflow (Paillier scheme). Tallying can be performed without decrypting individual votes, preserving voter privacy.

5.4 Microsoft Threat Modeling (MTM) Report

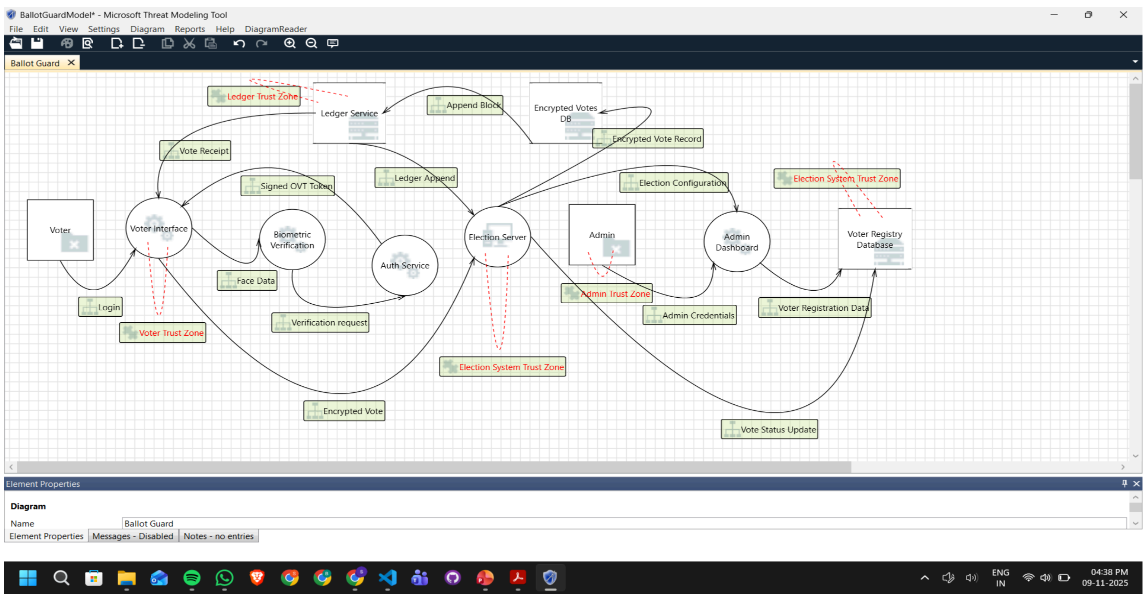


Figure 5.1-Microsoft Threat Model Block diagram

Threat Modeling Report

Created on 09-11-2025 4.42.44 PM

Threat Model Name: Ballot Guard

Owner: Sruthi

Reviewer:

Contributors: Shambhavi, Liya

Description:

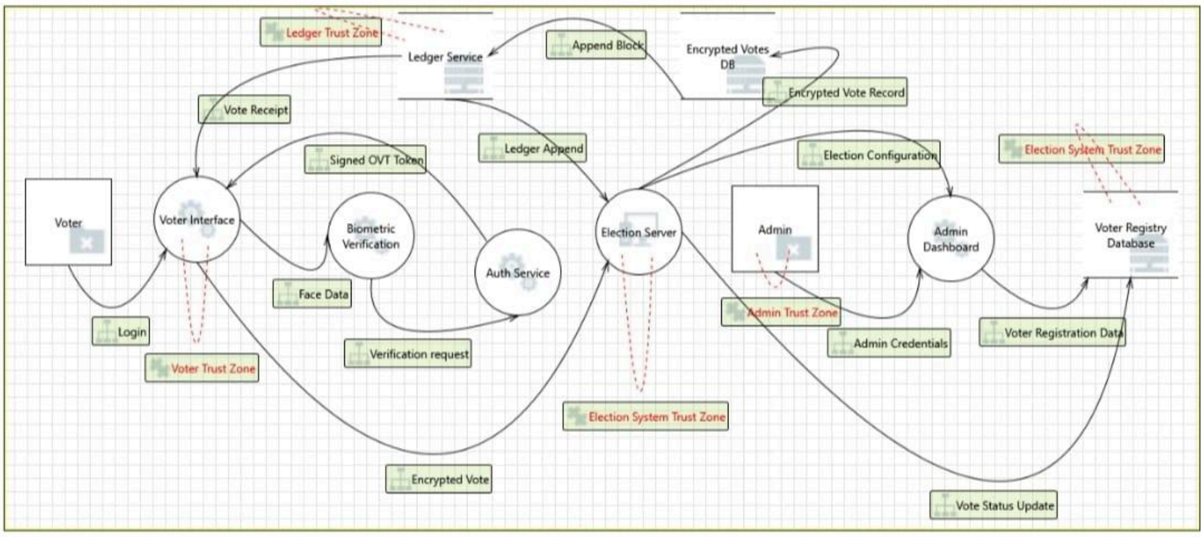
Assumptions:

External Dependencies:

Threat Model Summary:

Not Started	41
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	0
Total	41
Total Migrated	0

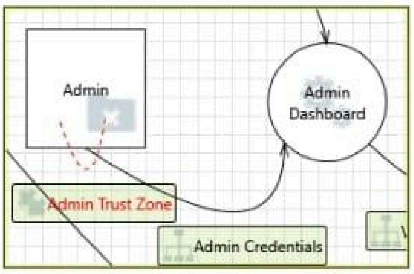
Diagram: Ballot Guard



Ballot Guard Diagram Summary:

Not Started	41
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	0
Total	41
Total Migrated	0

Interaction: Admin Credentials



1. Spoofing the Admin External Entity [State: Not Started] [Priority: High]

Category: Spoofing
Description: Admin may be spoofed by an attacker and this may lead to unauthorized access to Admin Dashboard. Consider using a standard authentication mechanism to identify the external entity.
Justification: Admin authentication is secured using strong credentials and public-key encryption.
Each admin session requires RSA-based token validation, and secure HTTPS is enforced for communication.
Login attempts are logged and rate-limited to prevent brute-force attacks.

2. Elevation by Changing the Execution Flow in Admin Dashboard [State: Not Started] [Priority: High]

Category: Elevation Of Privilege
Description: An attacker may pass data into Admin Dashboard in order to change the flow of program execution within Admin Dashboard to the attacker's choosing.
Justification: Use parameterized APIs/queries and safe templating libraries to prevent injection.
Apply the principle of least privilege: run dashboard processes with minimal rights and separate execution contexts for untrusted data.
Implement role-based access controls and multi-factor authentication for admin actions.
Use Content Security Policy (CSP) and HTTP security headers to limit attack surface for client-side tampering.
Add server-side authorization checks for every sensitive action.

3. Elevation Using Impersonation [State: Not Started] [Priority: High]

Category: Elevation Of Privilege
Description: Admin Dashboard may be able to impersonate the context of Admin in order to gain additional privilege.
Justification: Role-based access control (RBAC) is implemented in the Admin Dashboard to ensure each admin action is authorized.
Server-side validation ensures that even if the UI attempts unauthorized actions, backend permission checks prevent escalation.
Sensitive operations (like result decryption or user management) require secondary authentication or approval.

4. Admin Dashboard May be Subject to Elevation of Privilege Using Remote Code Execution [State: Not Started] [Priority: High]

Category: Elevation Of Privilege
Description: Admin may be able to remotely execute code for Admin Dashboard.
Justification: Enforce strict input validation and sanitization to prevent code injection.
Apply the principle of least privilege to all admin accounts.
Follow secure coding practices to avoid remote code execution vulnerabilities.
Monitor and log all admin actions for suspicious activity.
Keep the system and all dependencies up to date with security patches.

5. Data Flow Admin Credentials Is Potentially Interrupted [State: Not Started] [Priority: High]

Category: Denial Of Service
Description: An external agent interrupts data flowing across a trust boundary in either direction.
Justification: Encrypt all data in transit using TLS/SSL or other secure protocols.
Use message authentication codes (MACs) or digital signatures to ensure integrity.
Employ secure channels such as VPNs or secure APIs for critical communications.
Implement logging and monitoring to detect abnormal or unauthorized access.

6. Potential Process Crash or Stop for Admin Dashboard [State: Not Started] [Priority: High]

Category:	Denial Of Service
Description:	Admin Dashboard crashes, halts, stops or runs slowly; in all cases violating an availability metric.
Justification:	Admin Dashboard uses load balancing, health checks, and rate limiting to prevent crashes or slowdowns. Resource monitoring and auto-restart ensure high availability.

7. Data Flow Sniffing [State: Not Started] [Priority: High]

Category:	Information Disclosure
Description:	Data flowing across Admin Credentials may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.
Justification:	All admin credential transmissions are secured using TLS 1.3 encryption to prevent data sniffing or interception by attackers. Credentials are never transmitted in plaintext — they are hashed using SHA-256 and secured via HTTPS communication channels. Additionally, HSTS (HTTP Strict Transport Security) is enforced to ensure all admin communications remain encrypted and resistant to downgrade attacks. This effectively mitigates risks of credential disclosure and compliance violations.

8. Potential Data Repudiation by Admin Dashboard [State: Not Started] [Priority: High]

Category:	Repudiation
Description:	Admin Dashboard claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.
Justification:	The Admin Dashboard implements secure logging and auditing mechanisms to record all inbound data transactions. Each request is logged with details including timestamp, source IP, authentication context, and action summary. Logs are stored in a tamper-proof audit trail and monitored regularly for anomalies or unauthorized access attempts. This ensures traceability and accountability for all data received from sources outside the trust boundary.

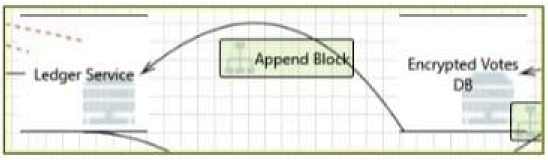
9. Potential Lack of Input Validation for Admin Dashboard [State: Not Started] [Priority: High]

Category:	Tampering
Description:	Data flowing across Admin Credentials may be tampered with by an attacker. This may lead to a denial of service attack against Admin Dashboard or an elevation of privilege attack against Admin Dashboard or an information disclosure by Admin Dashboard. Failure to verify that input is as expected is a root cause of a very large number of exploitable issues. Consider all paths and the way they handle data. Verify that all input is verified for correctness using an approved list input validation approach.
Justification:	All admin credentials are transmitted only over secure HTTPS/TLS channels to prevent tampering in transit. On the server side, strict input validation and sanitization are enforced for all credential fields (username, password, token). Admin login attempts undergo server-side verification with rate limiting, hash-based password validation (bcrypt/argon2), and session token signing. Additionally, integrity checks and input whitelisting prevent malformed or injected data from affecting the Admin Dashboard.

10. Spoofing the Admin Dashboard Process [State: Not Started] [Priority: High]

Category:	Spoofing
Description:	Admin Dashboard may be spoofed by an attacker and this may lead to information disclosure by Admin. Consider using a standard authentication mechanism to identify the destination process.
Justification:	The Admin Dashboard is authenticated using strong TLS certificates and server-side verification to prevent spoofed dashboard instances. Admin endpoints require signed API tokens and are restricted by IP and RBAC; any unexpected client is rejected. Admin actions are logged and alerts are raised for anomalous access patterns.

Interaction: Append Block



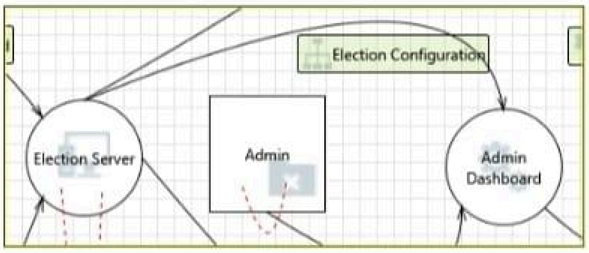
11. Spoofing of Destination Data Store Ledger Service **[State: Not Started] [Priority: High]**

Category:	Spoofing
Description:	Ledger Service may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of Ledger Service. Consider using a standard authentication mechanism to identify the destination data store.
Justification:	Implement strong authentication for the Ledger Service, such as mutual TLS or API keys. Use digital signatures to verify the integrity and origin of data before writing. Validate the destination address or endpoint to ensure data is sent only to the legitimate Ledger Service. Monitor and log all data writes for anomalies or suspicious activity.

12. Spoofing of Source Data Store Encrypted Votes DB **[State: Not Started] [Priority: High]**

Category:	Spoofing
Description:	Encrypted Votes DB may be spoofed by an attacker and this may lead to incorrect data delivered to Ledger Service. Consider using a standard authentication mechanism to identify the source data store.
Justification:	Implement strong authentication for the Encrypted Votes DB, such as mutual TLS or API keys. Use digital signatures to verify that data originates from the legitimate Encrypted Votes DB. Validate all incoming data at the Ledger Service before processing. Monitor and log data transfers for anomalies or suspicious activity.

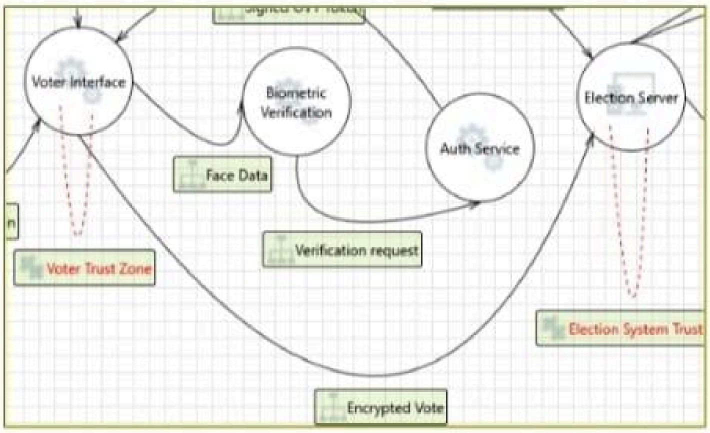
Interaction: Election Configuration



13. Elevation Using Impersonation **[State: Not Started] [Priority: High]**

Category:	Elevation Of Privilege
Description:	Admin Dashboard may be able to impersonate the context of Election Server in order to gain additional privilege.
Justification:	The Admin Dashboard communicates with the Election Server through authenticated APIs. Each request includes a signed token or API key verifying the origin of the request. Role-based access control (RBAC) ensures Admin privileges cannot extend into Election Server functions.

Interaction: Encrypted Vote



14. Elevation by Changing the Execution Flow in Election Server [State: Not Started] [Priority: High]

Category: Elevation Of Privilege
Description: An attacker may pass data into Election Server in order to change the flow of program execution within Election Server to the attacker's choosing.
Justification: Election Server enforces strict input validation (allow-lists/JSON schema) and rejects unexpected fields.
No user-supplied data is ever interpreted as code (no eval/template injection); use parameterized queries and prepared statements.
Critical actions run in sandboxed, least-privilege processes; use a Web Application Firewall (WAF), static analysis, and runtime integrity checks to detect tampering.
All suspicious inputs are logged and rejected; failures trigger alerts and safe error handling.

15. Election Server May be Subject to Elevation of Privilege Using Remote Code Execution [State: Not Started] [Priority: High]

Category: Elevation Of Privilege
Description: Voter Interface may be able to remotely execute code for Election Server.
Justification: The Election Server executes only predefined, validated commands and never interprets client-provided data as code.
Strict input validation, sandboxing, and least-privilege execution policies prevent arbitrary code execution.
The Voter Interface interacts through a secure API that enforces strict request schemas and authentication.
Regular security testing and static analysis ensure that no remote code execution paths exist.

16. Data Flow Encrypted Vote Is Potentially Interrupted [State: Not Started] [Priority: High]

Category: Denial Of Service
Description: An external agent interrupts data flowing across a trust boundary in either direction.
Justification: All data flows across trust boundaries are protected using TLS encryption and secure communication protocols.
Replay protection and integrity checks ensure that data cannot be intercepted or altered during transmission.
Any communication disruption triggers retries and is logged for detection and analysis.

17. Cross Site Scripting [State: Not Started] [Priority: High]

Category: Tampering
Description: The web server 'Election Server' could be a subject to a cross-site scripting attack because it does not sanitize untrusted input.
Justification: All untrusted input on the Election Server is validated and sanitized before processing.
Output is HTML-encoded to prevent the execution of injected scripts.
Additionally, HTTP security headers like Content Security Policy (CSP) and X-XSS-Protection are enabled to block potential XSS exploits.

18. Potential Process Crash or Stop for Election Server [State: Not Started] [Priority: High]

Category: Denial Of Service
Description: Election Server crashes, halts, stops or runs slowly; in all cases violating an availability metric.
Justification: The Election Server is deployed with failover mechanisms and load balancing to maintain availability.
Input validation and rate limiting prevent crashes from malformed or excessive requests.
Regular health checks and system monitoring ensure quick recovery in case of failures.

19. Data Flow Sniffing [State: Not Started] [Priority: High]

Category: Information Disclosure
Description: Data flowing across Encrypted Vote may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.
Justification: All encrypted vote data is transmitted over secure HTTPS/TLS channels, preventing packet sniffing or interception.
Even if intercepted, vote data remains encrypted using homomorphic encryption (Paillier) ensuring confidentiality.
End-to-end encryption and key management protect data throughout the communication process.

20. Potential Data Repudiation by Election Server [State: Not Started] [Priority: High]

Category: Repudiation

Description: Election Server claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.

Justification: The Election Server maintains secure audit logs that record every incoming request with timestamps, source identifiers, and digital signatures.
Logs are protected from modification and periodically verified to ensure integrity.
This ensures accountability and non-repudiation for all received data across trust boundaries.

21. Potential Lack of Input Validation for Election Server [State: Not Started] [Priority: High]

Category: Tampering

Description: Data flowing across Encrypted Vote may be tampered with by an attacker. This may lead to a denial of service attack against Election Server or an elevation of privilege attack against Election Server or an information disclosure by Election Server. Failure to verify that input is as expected is a root cause of a very large number of exploitable issues. Consider all paths and the way they handle data. Verify that all input is verified for correctness using an approved list input validation approach.

Justification: All encrypted vote data is verified for integrity using digital signatures before being processed by the Election Server.
Input validation and schema enforcement ensure that only properly formatted, encrypted payloads are accepted.
Any malformed or tampered data triggers rejection and is logged for audit.

22. Elevation Using Impersonation [State: Not Started] [Priority: High]

Category: Elevation Of Privilege

Description: Election Server may be able to impersonate the context of Voter Interface in order to gain additional privilege.

Justification: The Election Server and Voter Interface communicate using digitally signed tokens.
Mutual authentication ensures that neither component can impersonate the other.
Each service operates with limited privileges under strict access control and API authentication mechanisms.

23. Spoofing the Election Server Process [State: Not Started] [Priority: High]

Category: Spoofing

Description: Election Server may be spoofed by an attacker and this may lead to information disclosure by Voter Interface. Consider using a standard authentication mechanism to identify the destination process.

Justification: The Voter Interface verifies the Election Server's identity using SSL/TLS certificates before any data exchange.
All communications are encrypted, and the server's certificate is validated to prevent man-in-the-middle or spoofing attacks.
Only verified Election Server endpoints are allowed in the configuration.

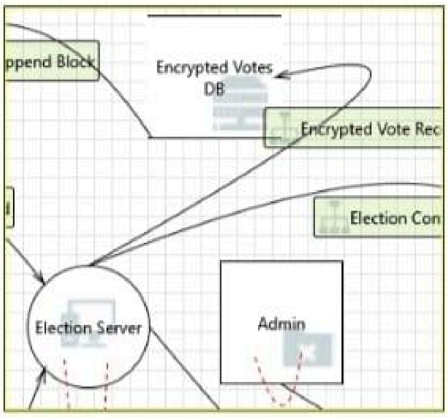
24. Spoofing the Voter Interface Process [State: Not Started] [Priority: High]

Category: Spoofing

Description: Voter Interface may be spoofed by an attacker and this may lead to unauthorized access to Election Server. Consider using a standard authentication mechanism to identify the source process.

Justification: The Election Server accepts requests only from authenticated Voter Interface instances using secure API keys or digital certificates.
All communication occurs over HTTPS with mutual authentication to prevent spoofing.
Session tokens are signed and verified before processing any voter requests.

Interaction: Encrypted Vote Record



25. Potential Excessive Resource Consumption for Election Server or Encrypted Votes DB **[State: Not Started] [Priority: High]**

Category: Denial Of Service

Description: Does Election Server or Encrypted Votes DB take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.

Justification: Implement resource limits and quotas for critical processes on the Election Server and Encrypted Votes DB.
Use timeouts for all resource requests to prevent indefinite blocking.
Monitor CPU, memory, and I/O usage to detect abnormal consumption patterns.
Design processes to avoid deadlocks, for example by acquiring resources in a consistent order.
Rely on OS-level resource management where appropriate, but combine it with application-level safeguards.

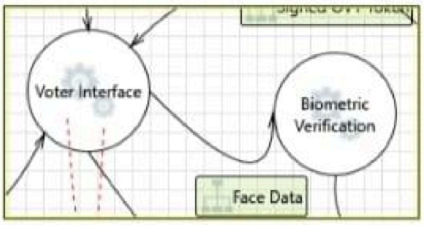
26. Spoofing of Destination Data Store Encrypted Votes DB **[State: Not Started] [Priority: High]**

Category: Spoofing

Description: Encrypted Votes DB may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of Encrypted Votes DB. Consider using a standard authentication mechanism to identify the destination data store.

Justification: Implement strong authentication for the Encrypted Votes DB, such as mutual TLS or API keys.
Use digital signatures to ensure data integrity and confirm the correct destination.
Validate the target endpoint before writing data to prevent misrouting.
Monitor and log all data writes to detect suspicious activity or anomalies.

Interaction: Face Data



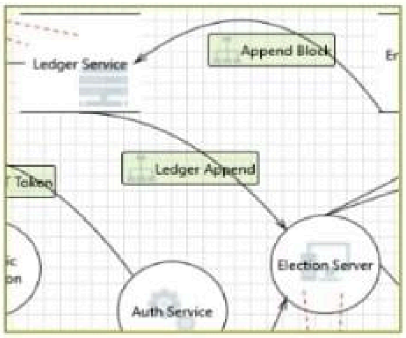
27. Elevation Using Impersonation **[State: Not Started] [Priority: High]**

Category: Elevation Of Privilege

Description: Biometric Verification may be able to impersonate the context of Voter Interface in order to gain additional privilege.

Justification: Biometric templates are stored as secure, non-reversible hashes (never raw images) and verified server-side.
Liveness detection is enforced and biometric results are cryptographically bound to the voter's session/token.
Biometric processing runs in a trusted enclave or hardened service and falls back to MFA if biometric checks fail.

Interaction: Ledger Append



28. Weak Access Control for a Resource **[State: Not Started]** **[Priority: High]**

Category:	Information Disclosure
Description:	Improper data protection of Ledger Service can allow an attacker to read information not intended for disclosure. Review authorization settings.
Justification:	Ledger Service data is encrypted using blockchain-based storage with cryptographic hashing. Access to the Ledger is restricted via role-based access control (RBAC) and secure API authentication. Only authorized Election components can read or append records, ensuring confidentiality and integrity.

29. Persistent Cross Site Scripting **[State: Not Started]** **[Priority: High]**

Category:	Tampering
Description:	The web server 'Election Server' could be a subject to a persistent cross-site scripting attack because it does not sanitize data store 'Ledger Service' inputs and output.
Justification:	Inputs and outputs between the Election Server and Ledger Service are validated and sanitized before storage and rendering. Persistent XSS risks are mitigated by escaping dynamic content, enforcing strict input validation, and applying output encoding. Content Security Policy (CSP) headers further reduce attack surface by restricting script execution sources.

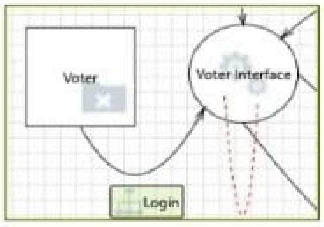
30. Cross Site Scripting **[State: Not Started]** **[Priority: High]**

Category:	Tampering
Description:	The web server 'Election Server' could be a subject to a cross-site scripting attack because it does not sanitize untrusted input.
Justification:	User input on the Election Server is sanitized and validated before rendering. HTML encoding and input validation are implemented to prevent script injection. Additionally, Content Security Policy (CSP) headers are used to block inline or malicious scripts.

31. Spoofing of Source Data Store Ledger Service **[State: Not Started]** **[Priority: High]**

Category:	Spoofing
Description:	Ledger Service may be spoofed by an attacker and this may lead to incorrect data delivered to Election Server. Consider using a standard authentication mechanism to identify the source data store.
Justification:	Ledger Service uses mutual authentication with the Election Server. Communication occurs over HTTPS with TLS certificates to verify both ends. Additionally, signed blockchain entries ensure that any spoofed or modified data is rejected automatically.

Interaction: Login



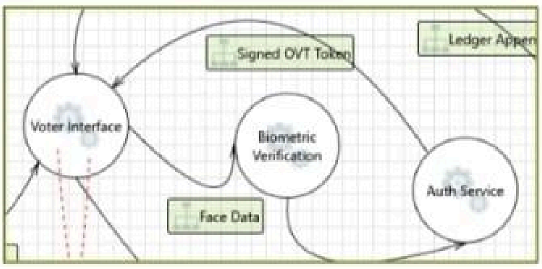
32. Elevation Using Impersonation **[State: Not Started]** **[Priority: High]**

Category:	Elevation Of Privilege
Description:	Voter Interface may be able to impersonate the context of Voter in order to gain additional privilege.
Justification:	The Voter Interface operates in a sandboxed environment with no elevated privileges. All voter actions are verified server-side using token-based validation, preventing client-side privilege escalation. Sensitive operations require backend authorization before execution.

33. Spoofing the Voter External Entity **[State: Not Started]** **[Priority: High]**

Category:	Spoofing
Description:	Voter may be spoofed by an attacker and this may lead to unauthorized access to Voter Interface. Consider using a standard authentication mechanism to identify the external entity.
Justification:	Voter authentication uses secure, unique voting tokens verified by the Election Server. All communication between the voter and system is encrypted using HTTPS/TLS. Tokens are single-use and cannot be reused or forged.

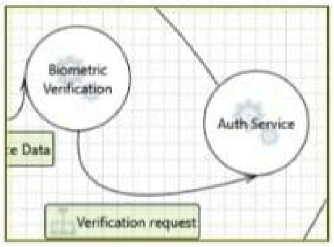
Interaction: Signed OVT Token



34. Elevation Using Impersonation **[State: Not Started] [Priority: High]**

Category: Elevation Of Privilege
Description: Voter Interface may be able to impersonate the context of Auth Service in order to gain additional privilege.
Justification: Voter Interface cannot act as Auth Service — all auth requests require signed service-to-service tokens or mTLS with service identities validated.
Tokens include audience/scope claims and are verified server-side; service credentials are stored in a secrets manager and rotated.
Audit logs record auth calls and anomalous requests trigger alerts.

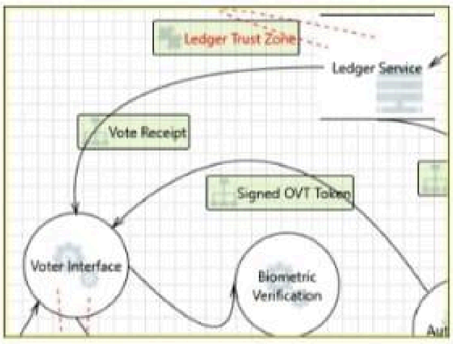
Interaction: Verification request



35. Elevation Using Impersonation **[State: Not Started] [Priority: High]**

Category: Elevation Of Privilege
Description: Auth Service may be able to impersonate the context of Biometric Verification in order to gain additional privilege.
Justification: Services use mutual authentication (mTLS) and signed service-to-service tokens so Auth Service cannot claim Biometric Verification identity.
Biometric results include cryptographic attestation (signed assertions) bound to a session token and are verified server-side.
Enforce least-privilege service accounts and audit all verification calls for anomalies.

Interaction: Vote Receipt



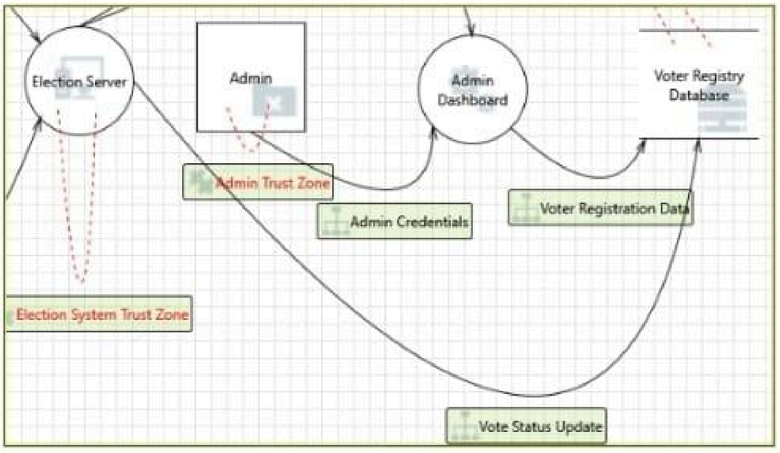
36. Weak Access Control for a Resource **[State: Not Started] [Priority: High]**

Category: Information Disclosure
Description: Improper data protection of Ledger Service can allow an attacker to read information not intended for disclosure. Review authorization settings.
Justification: All sensitive data in the Ledger Service is encrypted at rest and in transit.
Access is restricted using role-based authorization and secure service credentials.
Only authorized system components can read ledger entries.

37. Spoofing of Source Data Store Ledger Service **[State: Not Started] [Priority: High]**

Category: Spoofing
Description: Ledger Service may be spoofed by an attacker and this may lead to incorrect data delivered to Voter Interface. Consider using a standard authentication mechanism to identify the source data store.
Justification: The Ledger Service is authenticated using digital certificates and all communication occurs over HTTPS/TLS.
Voter Interface verifies data integrity with cryptographic signatures before displaying results.
Only trusted endpoints are allowed to connect to the Ledger Service.

Interaction: Vote Status Update



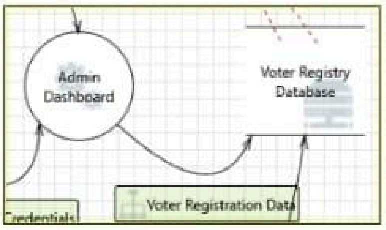
38. Potential Excessive Resource Consumption for Election Server or Voter Registry Database [State: Not Started] [Priority: Medium]

Category:	Denial Of Service
Description:	Does Election Server or Voter Registry Database take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.
Justification:	Election Server and Voter Registry Database use rate limiting, connection pooling, and query timeouts to prevent resource exhaustion. The system enforces session timeouts and relies on OS-level controls for handling heavy loads.

39. Spoofing of Destination Data Store Voter Registry Database [State: Not Started] [Priority: High]

Category:	Spoofing
Description:	Voter Registry Database may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of Voter Registry Database. Consider using a standard authentication mechanism to identify the destination data store.
Justification:	Database connections are encrypted (TLS) and validated using trusted certificates. Only whitelisted hosts can access the Voter Registry Database. Connection strings use secure credentials stored in a secrets manager. All write operations are logged to detect unauthorized destinations.

Interaction: Voter Registration Data



40. Potential Excessive Resource Consumption for Admin Dashboard or Voter Registry Database [State: Not Started] [Priority: High]

Category:	Denial Of Service
Description:	Does Admin Dashboard or Voter Registry Database take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.
Justification:	Rate limiting and session timeouts are implemented to prevent abuse of system resources. The database uses connection pooling and query timeouts to avoid deadlocks or resource exhaustion. Admin Dashboard backend includes load balancing and proper error handling to ensure system availability. The operating system and application layer both enforce resource limits to avoid crashes during high load.

41. Spoofing of Destination Data Store Voter Registry Database [State: Not Started] [Priority: High]

Category:	Spoofing
Description:	Voter Registry Database may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of Voter Registry Database. Consider using a standard authentication mechanism to identify the destination data store.
Justification:	Database connections are secured using trusted connection strings with certificates. Only whitelisted database hosts are allowed in configuration files. TLS/SSL is enforced for all database communications to prevent redirection or man-in-the-middle attacks. Authentication is handled using strong credentials and server certificates to verify the destination.

5.5 Summary

The testing and analysis confirm that BallotGuard achieves its key objectives of security, transparency, and reliability:

- Face Recognition tests confirmed that only registered voters could authenticate, ensuring secure access control.

- OVT Signature Verification validated the RSA-based signing mechanism, confirming authenticity and preventing token forgery.

- Block Signature Verification ensured that every blockchain block maintained cryptographic integrity.

- Blockchain Tampering Test demonstrated the system’s robustness, any alteration in historical blocks was immediately detected through hash and signature mismatches.

- Receipt Verification confirmed that every voter’s digital receipt correctly matched its ledger entry, ensuring end-to-end verifiability.

- Homomorphic Tally Correctness verified that encrypted vote totals matched the plaintext results without revealing individual votes.

- All test cases passed successfully, proving strong protection against unauthorized access, data tampering, and double voting.

- The system satisfies essential security properties that is confidentiality, integrity, authenticity

Overall, BallotGuard is validated as a secure and reliable e-voting system capable of maintaining trust and transparency in the electoral process.

Chapter 6: CONCLUSION & FUTURE WORK

6.1 Conclusion

BallotGuard: a secure electronic voting framework integrating blockchain, cryptography, and biometric authentication was successfully designed and implemented. It also shows how the use of blockchain in this project makes votes tamper-proof and the records undisputable due to its immutable nature. Consequently, the proposed scheme adopts RSA-based OVT authentication and digital signature mechanisms to improve the trustworthiness of the system and prevent replay or forgery attacks.

Face recognition authentication further ensured that only legitimate voters could participate, which gave an additional layer of security. Extensive testing confirmed that indeed all modules are performing reliably and upholding core security principles of confidentiality, integrity, authenticity, and verifiability. Its homomorphic encryption mechanism ensured vote tallies could be computed without the leakage of individual choices.

Overall, BallotGuard reaches its primary goal: developing a transparent, tamper-proof, auditable voting system suitable for modern digital elections.

6.2 Future Work

- Implement multi-factor authentication that combines biometrics, OVT, and OTP to further strengthen voter verification.
- Scale up blockchain through sidechains or Layer-2 scaling to support large-scale national elections.
- Develop a mobile interface for remote, accessible voting.
- Integrate AI-based anomaly detection to flag suspicious voting patterns or tampering attempts.
- Pilot test it in real life with an expanded sample size representative for testing system scalability and usability.
- Explore post-quantum cryptographic algorithms for future-proofing the system against quantum computing threats.

Chapter 7: REFERENCES

[1] W. Stallings, Cryptography and Network Security, 7th ed., Pearson, 2017.

[2] B. Schneier, Applied Cryptography, Wiley, 2015.

[3] R. Anderson, Security Engineering, 3rd ed., Wiley, 2020.

[4] NIST, FIPS 180-4: Secure Hash Standard, 2015.

[5] Python Cryptography Library Documentation, 2023.

[6] Microsoft Corporation, *"Microsoft Threat Modeling Tool-User Guide"*Available: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

[7] R. Rivest, A. Shamir, and L. Adleman, *"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[8] U. Jafar, M. J. Ab Aziz, and Z. Shukur, "Blockchain for Electronic Voting System—Review and Open Research Challenges," *Sensors*, vol. 21, no. 17, p. 5874, 2021. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8434614

Chapter 8: APPENDICES

AppendixA: Project Structure

```
BallotGuard/

|

├── client_app/

|   ├── auth/

|   ├── voting/

|   ├── crypto/

|   └── storage/

|

├── server_backend/

|   ├── api/

|   ├── crypto/

|   ├── blockchain/

|   ├── database/

|   └── admin_panel/

|

├── docs/

|

└── tests/
```

Appendix B: Core Algorithms

- RSA Encryption / Decryption for OVT token verification.
- SHA-256 Hashing for block signatures.

- Face Recognition Algorithm (OpenCV) for biometric authentication.
- Blockchain Integrity Verification using Merkle root checks and signature chaining.
- Homomorphic Tallying for encrypted vote aggregation.

Appendix C: STRIDE Categories

- Spoofing - Fake voter identity attempt
- Tampering - Altering blockchain data
- Repudiation - Voter denies vote submission
- Information Disclosure - Leaking voter ID or results
- Denial Of Service - Overloading vote endpoint
- Elevation of Privilege - Admin privilege misuse

Appendix D: Installation Guide

Steps :

1. Clone repository:
   git clone https://github.com/<yourusername>/BallotGuard.git
   cd BallotGuard
2. Activate Virtual Environment
3. Install dependencies:
   pip install -r requirements.txt
4. Run the application:
   python server.py
   python client_app/voting/app.py

Appendix E: API Endpoints

Elections (Admin)

Table 8.1 - Election endpoints

Endpoint	Method	Description
/elections	POST	Create DRAFT election and write genesis
/elections	GET	Lists all elections
/elections/{id}	GET	Get election details
/elections/{id}/open	POST	Set election state: OPEN

/elections/{id}/close	POST	Close election, freeze intake
/elections/{id}/tally	POST	Generate totals, store tally_report.json

Voters (Admin)

Table 8.2 - Voters endpoints

Endpoint	Method	Description
/voters/enroll	POST	{voter_id, face_template_meta} → 201 {voter_id, status:"pending"}
/voters/{voter_id}/approve	POST	Approve voter → 200 {status:"active"}

Auth & OVT (Booth)

Table 8.3 - Auth & OVT endpoints

Endpoint	Method	Description
/auth/face/verify	POST	{pass, confidence, voter_id?}; enforces attempts, lockout, active status
/ovt/issue	POST	Returns signed OVT: {ovt:{...}, server_sig}. If a new OVT is issued while a prior OVT is unspent, the previous token is

		atomically revoked/expired. Only the most recent OVT can be spent.

Votes (Booth)

Table 8.4 - Votes endpoints

Endpoint	Method	Description
/votes	POST	Request: {vote_id, election_id, booth_id, ciphertext, client_hash, ovt, ts} Response: {ledger_index, block_hash, ack:"stored"}. On /votes, the server validates prev_hash continuity (and optional block signature) before appending the new block. Full ledger verification runs on demand and at close/tally, not per vote.

Appendix F: Common Issues

1. Face not recognized : Occurs when the lighting is poor or the camera resolution is low.
   Solution: Retake the image in better lighting or adjust camera distance.

2. Signature mismatch : Happens if an OVT or blockchain block file is tampered with.
   Solution: Reissue the OVT and resynchronize the blockchain data.

3. Server not responding : Usually due to a port conflict, backend crash, or database connection issue.
   Solution: Restart the Flask server and verify the database service is running.

4. Vote not recorded : Network interruption or invalid token at submission time.
   Solution: Retry submission with a valid OVT after confirming connectivity.

5. Blockchain verification failed : Indicates block signature or hash inconsistency.
   Solution: Trigger integrity recheck; rebuild the chain from the last valid block.

6. Receipt not generated : Server error during digital signature or hash calculation.
   Solution: Re-run vote receipt generation module.

Appendix G: Quick Glossary

- OVT (One-Time Voting Token):
  A unique, single-use token assigned to each voter to ensure secure and authenticated voting.

- RSA (Rivest–Shamir–Adleman):
  An asymmetric encryption algorithm used for secure key exchange and digital signatures.

- SHA-256 (Secure Hash Algorithm – 256 bit):
  A cryptographic hash function that produces a 256-bit output, ensuring data integrity and immutability.

- Merkle Tree:
  A hierarchical data structure used in blockchains to verify data integrity efficiently.

- Homomorphic Encryption:
  A cryptographic technique that allows mathematical operations to be performed directly on encrypted data without decryption.

- STRIDE:
  A Microsoft threat modeling framework that categorizes threats into six types : Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.

Appendix H: Acknowledgments

The development of BallotGuard was made possible through:

- Open-source libraries such as Flask, OpenCV, and cryptography.

- Microsoft Threat Modeling Tool for structured risk analysis.

- Guidance and mentorship from faculty and peers who provided technical insight and feedback throughout the project.