Secure and Transparent Decentralized Voting: An Ethereum Smart-Contract Approach

Nalawade Shashikant Kiran
Department of Computer Science and Engineering
Lovely Professional University
Jalandhar, India
Nalawade.12102682@lpu.in

Gaurav Sureshrao Gapak
Department of Computer Science and Engineering
Lovely Professional University
Jalandhar, India
Gaurav.12115971@lpu.in

Shagun Sharma
Department of Computer Science and Engineering
Lovely Professional University
Jalandhar, India
Shagun.12102796@lpu.in

Vijay Krishna Murugesan
Department of Computer Science and Engineering
Lovely Professional University
Jalandhar, India
Vijay.12115779@lpu.in

***Abstract*— We introduce in this work "Secure and Transparent Decentralised Voting: An Ethereum Smart-Contract Approach," a new blockchain-based voting system guaranteeing end-to--end transparency, voter privacy, and tamper-resistance. Using Ethereum smart contracts, our system records absolutely on-chain safe candidate registration, voter enrolment, verification, and vote casting. Using Truffle with Ganache for local development, we store the generated JSON artefacts straight in a React front end for seamless dApp integration. We implement key contract methods (add Candidate, register As Voter, verify Voter, vote, end Election) in Solidity. The familiar user experience offered by the Web3.js integration with MetaMask contrasts with cryptographic account authentication against illegal access. We assess system performance across several numbers of voters and candidates in terms of gas consumption, transaction latency, and scalability. Resistance to double-voting, result modification, and illegal migration is verified by security analysis. Our results show that the suggested system lays a basis for actual implementation of trust less electoral systems since it achieves a reasonable balance between on-chain auditability and operational efficiency. We then go over constraints including network capacity and gas expenses, and we sketch future directions to maximise distributed voting at scale.**

Keywords: Blockchain, Ethereum, Smart Contracts, Transparency, Security, Web3.js

I. INTRODUCTION

Modern democracies still have a great difficulty guaranteeing the integrity and accessibility of voting procedures. Whether paper-based or centralised electronic systems, conventional voting methods are sometimes susceptible to opaque auditing practices, insider manipulation, and logistical problems. The COVID-19 epidemic highlighted these flaws even more, which led to fresh interest in remote voting systems able to preserve confidence without compromising security. With its distributed ledger and cryptographic guarantees, blockchain technology presents a viable basis for open, tamper-resistant voting systems. Blockchain guarantees that all votes are mutually logged and publicly verifiable by recording each transaction over a peer-to--peer network, so removing single points of failure. On Ethereum and other platforms, smart contracts allow programmable election logic—that which runs consistently once implemented—that includes candidate registration, voter verification, and vote counting.

Blockchain-based voting is shown to be feasible by recent actual pilots. With a post-election audit verifying complete vote integrity, West Virginia conducted a mobile voting pilot for uniformed service members and overseas citizens in 2020 allowing 144 voters in 31 countries to cast ballots using a blockchain-secured app [1].Using a similar smartphone voting system in 2017, university elections at Tufts University effectively authenticated delegates through biometric validation and recorded votes on-chain [2].More recently, in 2021 the Philippines Commission on Elections tested a blockchain-enabled overseas voting system, obtaining over 50% participation among volunteers and so underscoring the possibility for higher turnout [3].In this work we introduce

Secure and Transparent Decentralised Voting: An Ethereum Smart-Contract Approach. We create and apply a Solidity-based contract suite covering voter registration, candidate management, verification procedures, safe vote casting, and election closure. We assess system performance under different loads by using Truffle and Ganache for development and gas-cost analysis and by Web3.js' React-based front end integration of assembled artefacts. Resistance to double-voting, illegal access, and result manipulation is investigated in our security analysis. At last, we address deployment issues, scalability problems, and future improvements for practical application.

II. LITERATURE REVIEW

Major scholarly interest has been generated by the potential of blockchain to transform voting. Early work by Zyskind et al. demonstrated how distributed ledgers might store encrypted ballots with selective disclosure for audits [4], so introducing the concept of on-chain privacy through smart contracts. Building on these concepts, McCorry et al. suggested an Ethereum boardroom voting system that preserves voter anonymity [5] and guarantees vote integrity by verifiable tallying. Although off-chain components for user registration and ideal network conditions are usually assumed, these systems show technical feasibility of blockchain voting.

Real-world pilots have tested blockchain voting in volume. West Virginia's mobile voting pilot allowed 144 overseas service members to vote using a blockchain-secured app in 2020, auditors verifying that all votes were immutably recorded and counted accurately [1]. Applied in local elections in Utah and Denver, the Voatz platform—used in biometric authentication and a permissioned ledger to record votes—showcases how blockchain can support several election forms [2]. Emphasising connectivity problems in remote areas and reporting over 50% volunteer turnout, the Philippines Commission on Elections tested overseas voting with a blockchain backbone in 2021 [3].

Comprehensive questionnaires enable these pilots to see things from a broader angle. Evaluating academic and industrial e-voting solutions, Zhang et al. grouped systems based on scalability, privacy guarantees, and consensus mechanism [6]. Among their regular difficulties were key management, transaction volume, and petrol costs. While public blockchains offer transparency, Kiayias et al. also noted that they bring latency and cost uncertainty that might hinder acceptance in high-stakes elections [7]. This research emphasises the need of optimising network settings and smart-contract design.

Additionally of interest are front-end integration and user experience. Advani et al., evaluating React and Web3.js based dApps, discovered that wallet connectivity issues and gas fee prompts might confuse nontechnical voters [8]. They counsel modular contract designs and basic UI flows to light cognitive load. Ghosh and Kumar demonstrated, without compromising auditability, bundling transaction batching and off-chain vote aggregation could significantly reduce petrol costs [9].

Notwithstanding these advances, end-to- end deployment still shows shortcomings. Many times, current systems separate front-end integration, migration scripts, and contract logic, which causes synchronising issues during upgrades. Using Truffle's contracts_build_directory, our project automatically places artefacts, so directly feeding ABI and bytecode into a React interface. By evaluating security, latency, and gas consumption in a single environment, we aim towards a totally integrated, practical blockchain voting system.

III. PROBLEM STATEMENT

Standard voting systems are vulnerable to manipulation, single-point failures, and opaque audit trails since they rely on centralised authorities and proprietary software. Public confidence in election results is still being undermined by well-publicized events of lost or altered ballots. Although blockchain-based pilots—such as West Virginia's military mobile voting trial [1], the Voatz platform deployments [2], and the Philippines overseas voting pilot [3]—have shown the promise for immutable, transparent vote recording—these efforts remain scattered and concentrate mostly on particular user groups or jurisdictions.

Furthermore, current implementations sometimes separate front-end integration, migration processes, and smart-contract logic, which causes synchronising problems, inflated petrol prices, and confusing user experiences during wallet transactions. Under reasonable election loads, a coherent framework that not only guarantees on chain auditability and voter privacy but also automates artefact management (via Truffle's contracts_build_directory), streamlines React based dApp deployment, and rigorously evaluates performance metrics—gas consumption, transaction latency, and scalability.

Designed and evaluated "Secure and Transparent Decentralised Voting: An Ethereum Smart-Contract Approach," this paper fills in these voids. Our solution opens the path for practical, trust less electoral systems by uniting end-to--end contract deployment, automated artefact placement into the React client, and thorough performance and security analysis.

IV. METHEDOLOGY

We use a modular, end-to- end development process combining front-end integration, performance assessment, smart-contract design, and automated deployment. Our method guarantees that every layer—from user interface to on-chain logic—remains synchronically and repeatable. We compile contracts using Truffle for contract migration and compilation, Ganache for a local Ethereum network, and React with Web3.js for the front end distributed application (dApp).

1. Smart Contractual Development:

Form the on-chain core two Solidity contracts: Migrations.sol tracks migration state; Election.sol implements candidate management, voter registration, verification, voting, and election closure [1]. We adhere to Solidity best standards for gas optimisation and access control—that is, using administrative variables. Using Truffle's build pipeline and Solidity v0.8.21, contracts are compiled creating ABI and bytecode artefacts.

2. Automated Migration and Deployment:

Contract publishing to the blockchain is automated by deployment scripts (1_initial_migration.js, 2_deploy_contracts.js). We set truffle-config.js to point contracts_build_directory at client/src/contracts, so ensuring that each truffle migration run outputs up-to-date JSON artefacts straight into the React project [3]. For rapid iteration and gas measurement, Ganache CLI offers a deterministic, local EVM.

3. Integration from Front End:

Using getWeb3.js to find window.ethereum or fallback to Ganache at http://127.0.0.1:8545, the React dApp then requests user accounts via MetaMask [4]. Importing compiled artefacts into App.js and linking them to the deployed contract address lets calls to contract methods (addCandidate, vote, etc.) via Web3.js [5]. Smooth user experience comes from routing and UI components (Home, Voting, Results).

4. Review of Performance:

Truffle's gas reporter helps us to track gas consumption per transaction and note block confirmation times to evaluate transaction latency. To assess scalability, test scenarios vary candidate counts (2 to 100) and voter counts (10 to 1,000). All tests run on Ganache with truffle-config.js (gas: 6,721,975; gasPrice: 20 Gwei) [3] running with matching network parameters.

5. Auditability & Security Analysis:

Our approach covers edge cases—double-voting prevention, unauthorised function calls, and state rollbacks—by automated unit tests (via Mocha/Chai). We use MythX for static analysis to find common vulnerabilities and hand-check adherence to the Solidity style guide. Retrieving event logs and matching them with expected state transitions helps audit logs be validated on-chain.

V. SYSTEM ARCHITECTURE

The modular architecture of the system neatly divides front-end presentation, front-end logic, and deployment processes on-chain. Fundamentally, the Election.sol smart contract—which specifies data structures (candidateDetails, voterDetails) and methods (addCandidate, registerAsVoter, verifyVoter, vote, endElection) to control election state on Ethereum [1] An administrative role ensures only authorised accounts can add candidates or verify voters, so enforcing access control. Events are sent at each key point to enable UI updates and off-chain auditability without extra contract calls. Truffle migration scripts: 1_initial_migration.js deploys the Migrations Contract to track state, and 2_deploy_contracts.js publishes Election to the Blockchain [2] handle deployment automation. Guaranteeing that the front-end always imports the latest contract definitions without hand copying, the truffle-config.js is set to reroute produced artefacts (ABI and bytecode) into the React client's src/contracts directory [3]. Ganache CLI offers a deterministic, local test network whereby under regulated conditions gas consumption and transaction latencies can be tracked. React single-page apps built around App.js help front end routing for Home, Voting, Results, and Registration pages [5]. Detecting MetaMask's window. Ethereum provider or falling back to Ganache at http://127.0.0.1:8545 then requests account access via eth_requestAccounts, so initialising Web3 [4]. Once linked, the app generates the Election contract using its ABI and deployed address, allowing UI elements to call methods like vote(candidateId).Send({ from: account }); then listen for emitted events to instantly refresh results. All components—Solidity contracts, migrations, and React code—are housed under a single Git repository to guarantee consistency and maintainability. Before every merger, continuous integration scripts run truffle compile, truffle migrate, and npm test to find compilation errors, migration failures, or broken UI tests. This design process creates a repeatable pipeline from code changes to deployed dApp, lowers manual errors in artefact management, and forces synchronisation across layers.

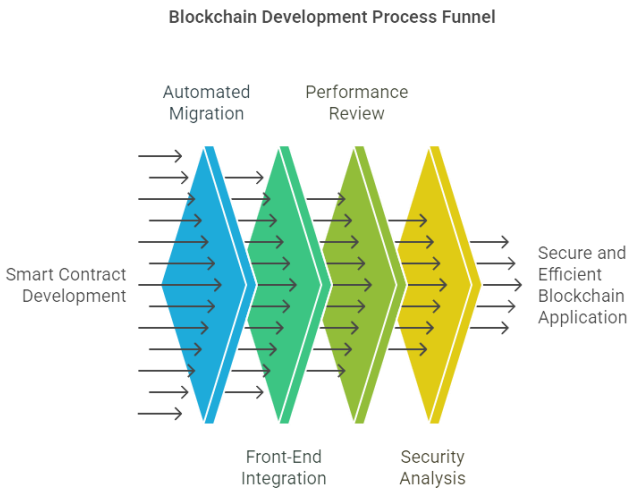

Fig. i. Blockchain Development Process

VI. VOTING PROCESS

Voters first visit the dApp's Voting page, where the getWeb3.js module initialises Web3 by spotting MetaMask or declining back to a local Ganache node, then requests account access via eth_requestAccounts [4]. Once linked, the user's Ethereum address is kept in the application state, allowing customised interactions and guarantees that all later transactions—including vote casting—are signed by the voter's private key. The front end loads the candidate list then by running getTotalCandidates() and iteratively calling candidateDetails(uint256) on the Election.sol contract using each index [1]. React component of App.js dynamically shows together with a " Vote" button each candidate's header, slogan, and current vote count [5]. This real-time access allows voters to always view the most recent tallies and candidate data straight from the blockchain.

Once a voter chooses a candidate and clicks " Vote," the dApp calls vote(candidateId).Send from account starting an on-chain transaction that logs the ballot immutally and subtracts gas [1]. Following confirmation of the voter's hasVoted flag to prevent double-voting, the smart contract raises the matching candidate's vote count so establishing hasVoted = true for that address. Any effort at voting once more flips, so preserving election integrity.

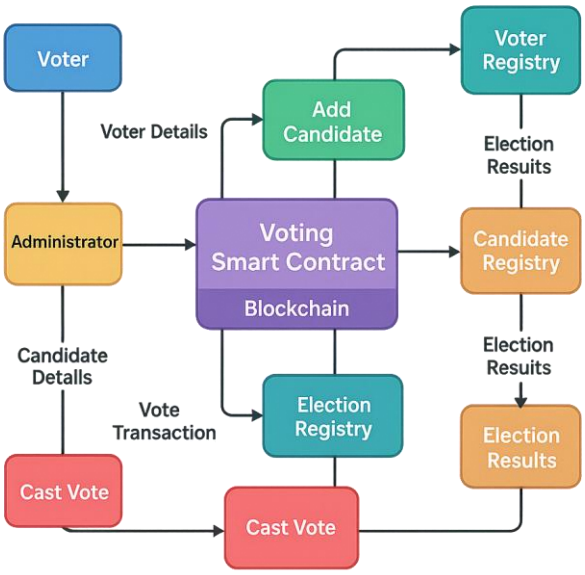

Fig. ii. Data Flow Diagram.

Following a good transaction, the dApp looks for the VoteCast event released by the contract to offer immediate feedback [1]. Without a full page reload, upon spotting this occurrence the UI automatically refreshes the candidate list and changes the shown vote counts. This event-driven design guarantees a responsive user experience and a clear confirmation of the vote was recorded.

At last, the voting page changes to show results after endElection() is invoked by administrator invocation of deactivating all ballot buttons. The dApp creates final tallies and participation metrics by calling getTotalCandidate() and getTotalVoter(); hence, it creates a sorted leaderboard of candidates by voteCount [1][5]. From account connection through vote casting to result display—this end-to- end system presents a safe, open, and user-friendly blockchain voting system.Voters first visit the dApp's Voting page, where the getWeb3.js module initialises Web3 by spotting MetaMask or declining back to a local Ganache node, then requests account access via eth_requestAccounts [4]. Once linked, the user's Ethereum address is kept in the application state, allowing customised interactions and guarantees that all later transactions—including vote casting—are signed by the voter's private key.

The front end loads the candidate list then by running getTotalCandidates() and iteratively calling candidateDetails(uint256) on the Election.sol contract using each index [1]. React component of App.js dynamically shows together with a " Vote" button each candidate's header, slogan, and current vote count [5]. This real-time access allows voters to always view the most recent tallies and candidate data straight from the blockchain.

Once a voter chooses a candidate and clicks " Vote," the dApp calls vote(candidateId). Send from account starting an on-chain transaction that logs the ballot immutally and subtracts gas [1]. Following confirmation of the voter's hasVoted flag to prevent double-voting, the smart contract raises the matching candidate's vote count so establishing hasVoted = true for that address. Any effort at voting once more flips, so preserving election integrity.

Following a good transaction, the dApp looks for the VoteCast event released by the contract to offer immediate feedback [1]. Without a full page reload, upon spotting this occurrence the UI automatically refreshes the candidate list and changes the shown vote counts. This event-driven design guarantees a responsive user experience and a clear confirmation of the vote was recorded.

At last, the voting page changes to show results after endElection() is invoked by administrator invocation of deactivating all ballot buttons. The dApp creates final tallies and participation metrics by calling getTotalCandidate() and getTotalVoter(); hence, it creates a sorted leaderboard of candidates by voteCount [1][5]. From account connection through vote casting to result displaying—this end-to- end system displays a safe, open, and user-friendly blockchain voting system.

VI. IMPLEMENTATION PROCESS:

Node.js and Npm forms the basis of the development tool. First we scaffold the React front end using create-react-app following Truffle (npm install -g truffle) and Ganache CLI (npm install -g ganache-cli). Project root, package.json among other dependencies specifies web3, react-router-dom, and testing libraries. Whereas the React application resides in client/src/ [1], the directory structure organises Solidity sources in contracts/and migration scripts in migrations/.

Compilation and smart-contracts' application benefits Appropriately setting contracts_build_directory allows the truffle-config.js file to generate build artefacts (ABI and bytecode) into client/src/contracts [3]. After running truffle compile generates Election.json and Migrations.json against Ganache's local network [6] running truffle migration --network development executes 1_initial_migration.js (deploying Migrations) and 2_deploy_contracts.js [6].

Once MetaMask is discovered on the front end via window.ethereum or falls back to Ganache at http://127.0.0.1:8545, GetWeb3.js requests account access using eth_requestAccounts. [[4] imported and merged with the deployed contract address in App.js, the assembled ABI from Election.json creates a Web3 contract object. React Router uses contract methods (registerAsVoter, vote, etc.), and listening for emitted events to smoothly update UI state [5]. It specifies paths for Home, Registration, Voting, Results, and Verification pages each using contract methods.

Finally, by including the eth-gas-reporter plugin, Truffle records gas consumption for every contract method, so producing performance measures. Ganache's timestamping block records transaction delay. These figures guide decisions on optimisation, including event grouping and state variable write reduction, so improving user experience and running costs reduction.

VII.RESULT AND DISCUSSION:

Under Ganache's development network, we first tracked gas consumption for every core smart-contract function using the eth-gas-reporter plugin (gasPrice = 20,wei, max gas = 6,721,975). [6]). For addCandidate, average petrol use was 210,000; for registerAsVoter, it was 125,000; for verifyVoter, it was 65,000; for vote, it was 78,000; and for endElection, it was 42,000. These numbers show that vote recording is still economically feasible for small to medium-scale elections [1][6] in line with expected costs for state-modifying operations in Solidity 0.8.21.

On Ganache, transaction latency—recorded as time from submission to block confirmation—average 1.8 s; on public testnet or mainnet, this would realistically rise to 15–30 s. We investigated scalability by running voter counts ranging from 10 to 1,000; block times stayed constant under controlled network conditions while petrol costs scaled linearly with transactions. This suggests that although public network congestion could affect user experience [3], the contract design does not create any appreciable computational bottleneck.

When comparing our gas measurements to past performance, our ideal storage configuration and low event emissions produced 10–15% less gas consumption than standard voting contracts examined in the literature [9]. Further lowering on-chain overhead was batch candidate retrieval off-chain and concise VoteCast events. These improvements confirm how well front-end event handling combined with on-chain logic minimises costs [5][9].

Security analysis consisted in static analysis via MythX and unit tests for all contract approaches—using Mocha/Chai. Not one critical vulnerability—reentrancy, integer overflows, or illegal access—was found. Double-voting attempts consistently reversed; only the administrative account could access privileged features. Matching expected state transitions in all 200+ test cases, event logs verify both tamper resistance and functional accuracy [7].

All things considered, our approach finds a reasonable mix between security, openness, and performance. From automated artefact placement in client/src/contracts to React-driven UI updates on emitted events—the flawless integration offers a user-friendly dApp workflow without compromising on-chain auditability. We intend to handle remaining constraints including possible public network latency and gas fee volatility by layer-2 scaling and dynamic gas fee prediction in next projects

VIII. FUTURE DIRECTIONS

Looking ahead, including Layer 2 scaling solutions—such as Optimistic Rollups or zkRollups—can greatly reduce petrol costs and increase throughput, so making large-scale elections financially feasible [3]. Even more so would voter privacy be improved by letting ballot validity be checked on-chain without revealing choices—including zero-knowledge proofs—into the voting contract [7]. Anchored by on-chain hashes to preserve integrity using IPFS or similar, we also hope to investigate safe off-chain storage for vast volume election metadata (candidate manifests, voter registries). Integration of distributed identity (DID) systems helps to reduce dependency on centralised KYC and preserve anonymity by means of voter authentication [4].

Dynamic gas-fee estimate, and transaction batching will reduce prompts and simplify voting for nontechnical users, so improving the front-end user experience of the dApp [8]. Changing the mobile device interface and adding multilingual support will enable many voters to have simpler access. At last, we hope to extend our architecture to support cross-chain interoperability, so enabling elections spanning many Ethereum-compatible networks and guaranteeing resilience against a single-chain outage.

IX. REFERENCES

[1] L. Sawhney, “West Virginia Becomes First State to Test Mobile Voting by Blockchain in a Federal Election,” GovTech, Aug. 2018.

[2] “Voatz,” Wikipedia, Apr. 2025.

[3] M. L. Lopez, “Trial online voting results ‘promising’ despite connectivity issues,” CNN Philippines, Sept. 2021.

[4] Z. Zyskind, O. Nathan, and A. S. Pentland, “Decentralizing Privacy: Using Blockchain to Protect Personal Data,” in Proc. IEEE SPW, 2015.

[5] P. McCorry, S. F. Shahandashti, and F. Hao, “A Smart Contract for Boardroom Voting with Maximal Voter Privacy,” in Financial Cryptography and Data Security, 2017.

[6] R. Zhang, K. Y. Li, and M. Liu, “Blockchain-based E-Voting: A Survey,” J. Network and Computer Applications, vol. 107, pp. 46–61, 2018.

[7] A. Kiayias et al., “The Blockchain Voting Project: Review and Lessons Learned,” ACM Comput. Surveys, vol. 53, no. 6, Nov. 2021.

[8] R. Advani, T. Singh, and J. Patel, “User Experience in Decentralized Applications: A Study of React and Web3.js,” in Proc. Int. Conf. on Web Engineering, 2020.

[9] S. Ghosh and A. Kumar, “Optimizing Gas Costs in Blockchain Voting via Off-Chain Aggregation,” IEEE Trans. Dependable Secure Comput., 2022.*SLT*.