FHElect: Un sistema de votación usando Fully Homomorphic Encryption

Fuentes Tievoli D'Elia

December 12, 2025

Abstract

Los procesos electorales son fundamentales para las democracias y la toma de decisiones en organizaciones privadas. Para garantizar su legitimidad, es esencial asegurar propiedades como el anonimato y la integridad. Este trabajo explora el uso de Fully Homomorphic Encryption (FHE) como solución a los desafíos de los sistemas de votación electrónicos. Se analiza como FHE permite el conteo verificable de votos sobre datos cifrados, preservando la privacidad del votante y asegurando la transparencia.

Contents

1 Introducción y Motivación

Los sistemas de votación han evolucionado hacia una mayor transparencia y robustez, pero el aumento de escala y requisitos de seguridad ha elevado sus costos e ineficiencias. Actualmente coexisten dos paradigmas principales.

El **Voto Tradicional** es el estándar en democracias modernas por su confianza histórica y garantía de anonimato físico. No obstante, presenta desventajas significativas: es **ineficiente temporalmente** (recuento lento), **costoso** (logística compleja, ej. elecciones nacionales en Argentina 2025: $230.000 millones [4]) y **centralizado** (vulnerable a errores y fraude de la autoridad).

El **Voto Electrónico** surge para agilizar el proceso y reducir costos logísticos. Sin embargo, introduce nuevos desafíos: falta de **confianza** (el votante no ve su voto físico), riesgos de **centralización** (manipulación por autoridades corruptas) y vulnerabilidad a **ataques** informáticos. Las variantes descentralizadas resuelven la centralización pero plantean otros retos.

2 Propiedades de un Sistema de Votación

Para que un sistema sea legítimo, debe garantizar: **Privacidad** (voto no vinculable al emisor), **Integridad** (resultado inmutable y exacto), **Verificabilidad** (tanto Individual como Universal) y **Resistencia a terceros** (imposibilidad de probar el voto para evitar venta o coerción).

3 Análisis de los Enfoques Actuales

Analizando los enfoques existentes bajo estas propiedades:

El **Voto Tradicional** satisface privacidad y resistencia a terceros (gracias al cuarto oscuro), pero falla en integridad y verificabilidad debido a la dependencia humana y centralizada.

El **Voto Electrónico Centralizado** mejora la eficiencia pero sacrifica la privacidad (el administrador tiene acceso total), la integridad y la verificabilidad. Además, no suele resistir a terceros (capturas de pantalla).

Una implementación **Blockchain "Naive"** (smart contract público) resolvería la integridad y verificabilidad mediante la inmutabilidad de la cadena, pero fallaría catastróficamente en privacidad y resistencia a terceros, ya que los votos serían públicos.

Es necesario un sistema que combine la transparencia de la blockchain con la privacidad del cuarto oscuro.

4 FHE

Como se planteó en la sección anterior, realizar las operaciones necesarias con los datos asociados a los votos sin revelar su contenido es necesario. La solución a este problema es la criptografía homomórfica.

El cifrado homomórfico permite realizar operaciones sobre datos cifrados sin necesidad de descifrarlos previamente. El resultado de estas operaciones, cuando se descifra, coincide con el resultado que se obtendría si las operaciones se hubieran realizado sobre los datos originales.

Si tenemos una función de encriptación H y dos valores x e y, un esquema homomórfico aditivo cumple que:

$$H(x) + H(y) = H(x + y)$$

Con esta propiedad es posible sumar los votos sin necesidad de descifrarlos previamente, preservando la privacidad de los votantes. Existen distintos niveles de homomorfismo. Los esquemas Parcialmente Homomórficos (PHE) permiten realizar solo un tipo de operación (sumas o multiplicaciones, pero no ambas). Fully Homomorphic Encryption (FHE) permite realizar tanto sumas como productos sobre datos cifrados.

5 Implementación

Para implementar un sistema de votación que cumpla con todas las propiedades deseadas, utilizamos la tecnología de Zama, específicamente fhEVM. Esta solución permite ejecutar contratos inteligentes sobre datos cifrados en una EVM compatible.

La arquitectura general del sistema consta de los siguientes componentes:

- **Host Chain**: Es la blockchain donde reside el estado y se ejecutan las transacciones. En nuestro caso, usamos la red de prueba Sepolia con soporte para FHE.
- **Coprocesadores**: Realizan el cómputo pesado sobre los datos cifrados fuera de la cadena (off-chain) para evitar costos de gas prohibitivos.
- **Gateway**: Orquesta la comunicación con los coprocesadores y asegura el consenso sobre los resultados cifrados.

- **KMS (Key Management System)**: Gestiona las claves de descifrado de manera distribuida, asegurando que nadie (ni siquiera los operadores de los nodos) pueda ver los datos en claro sin autorización.

5.1 El Smart Contract

El núcleo de FHElect es un contrato inteligente escrito en Solidity utilizando la librería `FHE.sol` de Zama. El contrato gestiona el estado de la votación, almacena los votos cifrados y realiza el conteo sin revelar los votos individuales.

5.1.1 Estructura de Datos

El contrato mantiene un contador global cifrado (`euint32`) y un mapeo del voto cifrado de cada votante:

```
euint32 private encryptedCount;
mapping(address => ebool) private encryptedVotes;
```

5.1.2 Proceso de Votación

Cuando un usuario emite un voto, este se cifra en el lado del cliente (client-side) junto con una prueba de conocimiento cero (ZKPoK) para asegurar que el cifrado es válido. El contrato recibe este voto cifrado (`externalEbool`) y lo procesa:

```
function vote(externalEbool externalYesOrNo, bytes calldata proof)
    external {
    ebool currentVote = FHE.fromExternal(externalYesOrNo, proof);
    // ... logica de actualizacion ...
    addToCount(eboolToOneOrZero(currentVote));
}
```

Para sumar el voto al contador total, utilizamos operaciones homomórficas. Dado que el voto es un booleano cifrado (`ebool`), primero lo convertimos a un entero 0 o 1 usando `FHE.select`:

```
function eboolToOneOrZero(ebool boolValue) private returns (euint32
    ) {
    return FHE.select(boolValue, encryptedConstantOne,
    encryptedConstantZero);
}
```

5.1.3 Verificabilidad y Privacidad

Para garantizar la verificabilidad individual, el contrato otorga permisos explícitos al votante para que pueda descifrar su propio voto almacenado, utilizando Listas de Control de Acceso (ACL):

```
FHE.allow(currentVote, msg.sender);
```

De esta forma, el votante puede consultar `getMyVote()` y verificar que el sistema guardó correctamente su intención, sin que nadie más pueda acceder a ese dato.

Al finalizar la votación, el dueño del contrato solicita el descifrado del conteo total. Este proceso es asincrónico y requiere la colaboración de la red de KMS para revelar únicamente el resultado final agregados.

6 Vulnerabilidades y Desafíos

6.1 Coerción y Compra de Votos

Una vulnerabilidad importante en los sistemas de votación electrónica es la coerción. Si un votante puede probar cómo votó (por ejemplo, mostrando su pantalla), un atacante podría comprar su voto.

En FHElect, mitigamos esto permitiendo que los usuarios voten múltiples veces, donde solo el último voto es válido. Si un votante es coaccionado, puede emitir el voto que pide el atacante, recibir su "pago", y luego emitir su voto real, sobrescribiendo el anterior. El contrato maneja esto restando el voto anterior del total antes de sumar el nuevo:

```
if (hasVotedBefore) {
    ebool previousVote = encryptedVotes[msg.sender];
    subtractFromCount(eboolToOneOrZero(previousVote));
}
addToCount(eboolToOneOrZero(currentVote));
```

6.2 Análisis de Tráfico

Aunque el contenido del voto es privado, el hecho de que una dirección emita una transacción es información pública en la blockchain. Esto introduce dos vulnerabilidades principales: **Identificación del votante**: Si la dirección está vinculada a una identidad del mundo real, se sabe quién votó y cuándo; y **Detección de voto duplicado**: Un atacante que ha coaccionado a un votante puede monitorear si este envía una segunda transacción para anular el voto coaccionado.

Para mitigar estos riesgos, proponemos dos estrategias complementarias:

6.2.1 Relayers y Mixnets

Para romper el vínculo entre la identidad del votante y la transacción de voto, se pueden utilizar intermediarios conocidos como **Relayers**. En lugar de que el votante

envíe la transacción directamente, firma un mensaje fuera de la cadena (off-chain) utilizando estándares como EIP-712. Este mensaje firmado es enviado al relayer, quien lo empaqueta en una transacción y paga el gas necesario para enviarlo a la blockchain.

Sin embargo, si se utiliza un solo relayer centralizado, este podría censurar transacciones o colaborar con un atacante. Para descentralizar esto y ocultar el origen de la comunicación (dirección IP), se podrían usar **Mixnets** (como Nym o Tor) o **Ring Signatures** (como en Monero). Estas tecnologías permiten que el mensaje llegue al relayer de manera anónima, dificultando rastrear el origen físico del voto.

6.2.2 Inyección de Ruido (Dummy Votes)

Para evitar que un atacante sepa si un usuario está votando realmente o simplemente interactuando con el contrato para confundir, se puede implementar la inyección de tráfico falso o "ruido".

El cliente puede enviar transacciones aleatorias que, desde el punto de vista de un observador externo (incluyendo al atacante), son indistinguibles de un voto real en términos de firma de la función llamada, costo de gas y tamaño de los datos cifrados. Una idea de implementación es la siguiente: el usuario envía un flag booleano cifrado (`ebool isDummy`) junto con su voto. El contrato podría utilizar este flag para condicionalmente sumar 0 al conteo total si `isDummy` es verdadero, o sumar el voto si es falso. De esta manera, el estado interno del conteo no cambia, pero un observador no puede distinguir entre un voto real y uno dummy, protegiendo así al votante contra el análisis de tráfico temporal. Es importante que se haga una suma con 0 al contador total para que las operaciones no difieran en el gas.

7 Extensiones

El diseño de FHElect es extensible a esquemas de votación más complejos.

7.1 Voto con Múltiples Candidatos

Para elecciones con más de dos opciones, en lugar de un booleano, el usuario envía el ID del candidato cifrado (`encryptedCandidateId`). El contrato itera sobre todos los candidatos y suma 1 al contador del candidato cuyo ID coincide con el voto.

7.2 Voto Rankeado

Implementamos también un sistema de voto preferencial, donde el votante selecciona k candidatos y se les asignan puntajes decrecientes. El contrato recibe un array de

votos cifrados y suma el puntaje correspondiente a cada candidato seleccionado, todo de manera cifrada y verificable.

8 Conclusiones

Este trabajo muestra que la tecnología de Fully Homomorphic Encryption (FHE) es una solución viable y potente para resolver el trilema de privacidad, integridad y verificabilidad en los sistemas de votación electrónica. A través de FHElect, logramos un sistema donde:

1. La privacidad es absoluta frente a observadores y administradores.
2. El conteo es íntegro y verificable universalmente.
3. Cada votante puede auditar su propio voto.

Si bien existen limitaciones en términos de resistencia a la coerción (entre otras vulnerabilidades), las soluciones propuestas y la las mejoras en rendimiento y seguridad de los sistemas de criptografía homomórfica sugieren un futuro prometedor para sistemas de votación descentralizados.

References

[1] *Going from bad to worse: from internet voting to blockchain voting.* MIT DigitalCurrency Initiative.

[2] Zama AI. *FHEVM Whitepaper.*

[3] *Towards Secure Electronic Voting: a Survey on E-Voting Systems and Attacks.* ResearchGate.

[4] La Nación. *El costo de la elección será de por lo menos de 230 mil millones.* 18/10/2025.

A Código de Smart Contracts

A.1 FHEVoter.sol

```
// SPDX-License-Identifier: BSD-3-Clause-Clear
pragma solidity ^0.8.24;

import {FHE, euint32, externalEuint32, ebool, externalEbool} from "@fhevm/solidity/lib/FHE.sol";
import {SepoliaConfig} from "@fhevm/solidity/config/ZamaConfig.sol";

// inherits from SepoliaConfig to enable fhEVM support
contract FHEVoter is SepoliaConfig {
    struct VoterState {
        uint256 lastElectionId;
        ebool encryptedVote;
    }

    euint32 private encryptedCount;
    euint32 private encryptedConstantOne;
    euint32 private encryptedConstantZero;
    mapping(address => VoterState) private voterStates;

    uint32 private clearCount; // =0 by default. It is only used when the owner calls "decryptCount"
    address private owner;
    bool private isVotingOpen;
    uint256 private electionId;

    event CountDecrypted(uint32 count);
    event VotingStarted();
    event VotingClosed();

    constructor() {
        owner = msg.sender;
        encryptedConstantOne = FHE.asEuint32(1);
        encryptedConstantZero = FHE.asEuint32(0);
        FHE.allowThis(encryptedConstantOne);
        FHE.allowThis(encryptedConstantZero);
        isVotingOpen = true;
        electionId = 1;
        emit VotingStarted();
    }

    modifier onlyWhenVotingOpen() {
        require(isVotingOpen, "Voting is not open");
```

```
        _;
    }

    modifier onlyWhenVotingClosed() {
        require(!isVotingOpen, "Voting is open");
        _;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this function");
        _;
    }

    function closeVoting() external onlyWhenVotingOpen onlyOwner {
        isVotingOpen = false;
        emit VotingClosed();
    }

    function startVoting() external onlyOwner {
        isVotingOpen = true;
        encryptedCount = FHE.asEuint32(0);
        FHE.allowThis(encryptedCount);
        clearCount = 0;
        electionId++;

        emit VotingStarted();
    }

    function eboolToOneOrZero(ebool boolValue) private returns (euint32) {
        return FHE.select(boolValue, encryptedConstantOne, encryptedConstantZero);
    }

    function subtractFromCount(euint32 valueToSubtract) private {
        encryptedCount = FHE.sub(encryptedCount, valueToSubtract);
        FHE.allowThis(encryptedCount);
    }

    function addToCount(euint32 valueToAdd) private {
        encryptedCount = FHE.add(encryptedCount, valueToAdd);
        FHE.allowThis(encryptedCount);
    }

    function setVote(ebool currentVote, address voter) private {
        voterStates[voter] = VoterState(electionId, currentVote);
```

```
85          FHE.allow(currentVote, voter);
86          FHE.allowThis(currentVote);
87      }
88
89      // using a boolean allows us to ensure what we add is always 0
    or 1
90      function vote(externalEbool externalYesOrNo, bytes calldata
    proof) external onlyWhenVotingOpen {
91          ebool currentVote = FHE.fromExternal(externalYesOrNo, proof
    );
92          VoterState memory state = voterStates[msg.sender];
93
94          euint32 valueToAdd;
95          if (state.lastElectionId != electionId) {
96              valueToAdd = eboolToOneOrZero(currentVote);
97              // state update happens in setVote
98          } else {
99              // subtract the previous vote from the total, and then
    add the new one
100             ebool previousVote = state.encryptedVote;
101             euint32 valueToSubtract = eboolToOneOrZero(previousVote
    );
102             subtractFromCount(valueToSubtract);
103
104             valueToAdd = eboolToOneOrZero(currentVote);
105         }
106         addToCount(valueToAdd);
107
108         setVote(currentVote, msg.sender);
109     }
110
111     function getCount() external view onlyWhenVotingClosed returns
    (euint32) {
112         return encryptedCount;
113     }
114
115     function getMyVote() external view returns (ebool) {
116         VoterState memory state = voterStates[msg.sender];
117         require(state.lastElectionId == electionId, "You have not
    voted yet");
118         return state.encryptedVote;
119     }
120
121     function requestDecryption() external onlyWhenVotingClosed
    onlyOwner {
122         bytes32[] memory cypherTexts = new bytes32[](1);
123         cypherTexts[0] = FHE.toBytes32(encryptedCount);
```

```
124         FHE.requestDecryption(
125             // the list of encrypted values we want to publc
    decrypt
126             cypherTexts,
127             // the function selector the FHEVM backend will
    callback with the clear values as arguments
128             this.callbackDecryptSingleUint32.selector
129         );
130     }
131 
132     function getDecryptedCount() external view onlyWhenVotingClosed
     returns (uint32) {
133         return clearCount;
134     }
135 
136     function callbackDecryptSingleUint32(
137         uint256 requestID,
138         bytes memory cleartexts,
139         bytes memory decryptionProof
140     ) external {
141         // The `cleartexts` argument is an ABI encoding of the
    decrypted values associated to the
142         // handles (using `abi.encode`).
143         // ================================
144         //       SECURITY WARNING!        //
145         // ================================
146         // Must call `FHE.checkSignatures(...)` here!
147         //            ------------------------
148         // This callback must only be called by the authorized
    FHEVM backend.
149         // To enforce this, the contract author MUST verify the
    authenticity of the caller
150         // by using the `FHE.checkSignatures` helper. This ensures
    that the provided signatures
151         // match the expected FHEVM backend and prevents
    unauthorized or malicious calls.
152         //
153         // Failing to perform this verification allows anyone to
    invoke this function with
154         // forged values, potentially compromising contract
    integrity.
155         //
156         // The responsibility for signature validation lies
    entirely with the contract author.
157         //
158         // The signatures are included in the `decryptionProof`
    parameter.
```

```
159         //
160         FHE.checkSignatures(requestID, cleartexts, decryptionProof)
    ;
161
162         uint32 decryptedInput = abi.decode(cleartexts, (uint32));
163         clearCount = decryptedInput;
164
165         emit CountDecrypted(clearCount);
166     }
167 }
```

Listing 1: FHEVoter.sol

A.2 FHERankedVoter.sol

```
1 // SPDX-License-Identifier: BSD-3-Clause-Clear
2 pragma solidity ^0.8.24;
3
4 import {FHE, euint32, externalEuint32, ebool, externalEbool} from "
    @fhevm/solidity/lib/FHE.sol";
5 import {SepoliaConfig} from "@fhevm/solidity/config/ZamaConfig.sol
    ";
6
7 // IDEA: to allow for a voting scheme that has multiple candidates
    (and only one vote per candidate) we can receive
8 // an array of booleans (and a proof that only one of them is true?
     or can we handle this internally with some FHE operations?)
9 // such as `select`
10 // https://claude.ai/share/d065d579-5b33-4c03-a9e7-98586997c237
11 // this has some ideas to implement ZK proofs for the multiple
    votes
12 // we may need to use zk proofs in order to verify that the user
    provided exactly one true value in the array of booleans
13 // what we can do with FHE is get an ebool that (when decrypted) is
     true <=> the sum of all user votes is exactly k
14 // where k is the number of candidates someone can vote (think
    ranked voting systems...)
15 // however, we would need to decrypt that number! And that is not
    very secure... it would give us info about exactly how many
    votes
16 // someone made.
17 // ZK proofs are better. They would allow us to verify that the
    user provided <= k true values in the array of booleans.
18 // without us knowing the exact number of true values in their
    array
19
20 // inherits from SepoliaConfig to enable fhEVM support
```

```
contract FHEVoter is SepoliaConfig {
    struct VoterState {
        uint256 lastElectionId;
        ebool encryptedVote;
    }

    euint32 private encryptedCount;
    euint32 private encryptedConstantOne;
    euint32 private encryptedConstantZero;
    mapping(address => VoterState) private voterStates;

    uint32 private clearCount; // =0 by default. It is only used when the owner calls "decryptCount"
    address private owner;
    bool private isVotingOpen;
    uint256 private electionId;

    event CountDecrypted(uint32 count);
    event VotingStarted();
    event VotingClosed();

    constructor() {
        owner = msg.sender;
        encryptedConstantOne = FHE.asEuint32(1);
        encryptedConstantZero = FHE.asEuint32(0);
        FHE.allowThis(encryptedConstantOne);
        FHE.allowThis(encryptedConstantZero);
        isVotingOpen = true;
        electionId = 1;
        emit VotingStarted();
    }

    modifier onlyWhenVotingOpen() {
        require(isVotingOpen, "Voting is not open");
        _;
    }

    modifier onlyWhenVotingClosed() {
        require(!isVotingOpen, "Voting is open");
        _;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this function");
        _;
    }
```

```
66
67      function closeVoting() external onlyWhenVotingOpen onlyOwner {
68          isVotingOpen = false;
69          emit VotingClosed();
70      }
71
72      function startVoting() external onlyOwner {
73          isVotingOpen = true;
74          encryptedCount = FHE.asEuint32(0);
75          FHE.allowThis(encryptedCount);
76          clearCount = 0;
77          electionId++;
78
79          emit VotingStarted();
80      }
81
82      function eboolToOneOrZero(ebool boolValue) private returns (
    euint32) {
83          return FHE.select(boolValue, encryptedConstantOne,
    encryptedConstantZero);
84      }
85
86      function subtractFromCount(euint32 valueToSubtract) private {
87          encryptedCount = FHE.sub(encryptedCount, valueToSubtract);
88          FHE.allowThis(encryptedCount);
89      }
90
91      function addToCount(euint32 valueToAdd) private {
92          encryptedCount = FHE.add(encryptedCount, valueToAdd);
93          FHE.allowThis(encryptedCount);
94      }
95
96      function setVote(ebool currentVote, address voter) private {
97          voterStates[voter] = VoterState(electionId, currentVote);
98          FHE.allow(currentVote, voter);
99          FHE.allowThis(currentVote);
100     }
101
102     // using a boolean allows us to ensure what we add is always 0
    or 1
103     function vote(externalEbool externalYesOrNo, bytes calldata
    proof) external onlyWhenVotingOpen {
104         ebool currentVote = FHE.fromExternal(externalYesOrNo, proof
    );
105         VoterState memory state = voterStates[msg.sender];
106
107         euint32 valueToAdd;
```

```
108         if (state.lastElectionId != electionId) {
109             valueToAdd = eboolToOneOrZero(currentVote);
110             // state update happens in setVote
111         } else {
112             // subtract the previous vote from the total, and then
    add the new one
113             ebool previousVote = state.encryptedVote;
114             euint32 valueToSubtract = eboolToOneOrZero(previousVote
    );
115             subtractFromCount(valueToSubtract);
116 
117             valueToAdd = eboolToOneOrZero(currentVote);
118         }
119         addToCount(valueToAdd);
120 
121         setVote(currentVote, msg.sender);
122     }
123 
124     function getCount() external view onlyWhenVotingClosed returns
    (euint32) {
125         return encryptedCount;
126     }
127 
128     function getMyVote() external view returns (ebool) {
129         VoterState memory state = voterStates[msg.sender];
130         require(state.lastElectionId == electionId, "You have not
    voted yet");
131         return state.encryptedVote;
132     }
133 
134     function requestDecryption() external onlyWhenVotingClosed
    onlyOwner {
135         bytes32[] memory cypherTexts = new bytes32[](1);
136         cypherTexts[0] = FHE.toBytes32(encryptedCount);
137         FHE.requestDecryption(
138             // the list of encrypted values we want to publicly
    decrypt
139             cypherTexts,
140             // the function selector the FHEVM backend will
    callback with the clear values as arguments
141             this.callbackDecryptSingleUint32.selector
142         );
143     }
144 
145     function getDecryptedCount() external view onlyWhenVotingClosed
     returns (uint32) {
146         return clearCount;
```

```
147     }
148
149     function callbackDecryptSingleUint32(
150         uint256 requestID,
151         bytes memory cleartexts,
152         bytes memory decryptionProof
153     ) external {
154         // The ‘cleartexts‘ argument is an ABI encoding of the
    decrypted values associated to the
155         // handles (using ‘abi.encode‘).
156         // ================================
157         //      SECURITY WARNING!         //
158         // ================================
159         // Must call ‘FHE.checkSignatures(...)‘ here!
160         //             ------------------------
161         // This callback must only be called by the authorized
    FHEVM backend.
162         // To enforce this, the contract author MUST verify the
    authenticity of the caller
163         // by using the ‘FHE.checkSignatures‘ helper. This ensures
    that the provided signatures
164         // match the expected FHEVM backend and prevents
    unauthorized or malicious calls.
165         //
166         // Failing to perform this verification allows anyone to
    invoke this function with
167         // forged values, potentially compromising contract
    integrity.
168         //
169         // The responsibility for signature validation lies
    entirely with the contract author.
170         //
171         // The signatures are included in the ‘decryptionProof‘
    parameter.
172         //
173         FHE.checkSignatures(requestID, cleartexts, decryptionProof)
    ;
174
175         uint32 decryptedInput = abi.decode(cleartexts, (uint32));
176         clearCount = decryptedInput;
177
178         emit CountDecrypted(clearCount);
179     }
180 }
```

Listing 2: FHERankedVoter.sol