

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	1
2	Propiedades de un Sistema de Votación	1
3	Análisis de los Enfoques Actuales	1
4	FHE	2
5	Implementación	2
5.1	El Smart Contract	3
5.1.1	Estructura de Datos	3
5.1.2	Proceso de Votación	3
5.1.3	Verificabilidad y Privacidad	4
6	Vulnerabilidades y Desafíos	4
6.1	Coerción y Compra de Votos	4
6.2	Análisis de Tráfico	4
6.2.1	Relayers y Mixnets	5
6.2.2	Inyección de Ruido (Dummy Votes)	5
7	Extensiones	5
7.1	Voto con Múltiples Candidatos	5
7.2	Voto Rankeado	6
8	Conclusiones	6
A	Código de Smart Contracts	8
A.1	FHEVoter.sol	8
A.2	FHERankedVoter.sol	12

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 inefficiencias. 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 descierra, 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.
- **Coprocadores:** 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:

```
1 euint32 private encryptedCount;
2 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:

```
1 function vote(externalEbool externalYesOrNo, bytes calldata proof)
2   external {
3     ebool currentVote = FHE.fromExternal(externalYesOrNo, proof);
4     // ... logica de actualizacion ...
5     addToCount(eboolToOneOrZero(currentVote));
6 }
```

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`:

```
1 function eboolToOneOrZero(ebool boolValue) private returns (euint32)
2   {
3     return FHE.select(boolValue, encryptedConstantOne,
4       encryptedConstantZero);
5 }
```

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):

```
1 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:

```
1 if (hasVotedBefore) {  
2     ebool previousVote = encryptedVotes[msg.sender];  
3     subtractFromCount(eboolToOneOrZero(previousVote));  
4 }  
5 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 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

```
1 // SPDX-License-Identifier: BSD-3-Clause-Clear
2 pragma solidity ^0.8.24;
3
4 import {FHE, euint32, externalEuint32, ebool, externalEbool} from "
5     @fhevm/solidity/lib/FHE.sol";
6 import {SepoliaConfig} from "@fhevm/solidity/config/ZamaConfig.sol"
7     ";
8
9
10 // inherits from SepoliaConfig to enable fhEVM support
11 contract FHEVoter is SepoliaConfig {
12     struct VoterState {
13         uint256 lastElectionId;
14         ebool encryptedVote;
15     }
16
17     euint32 private encryptedCount;
18     euint32 private encryptedConstantOne;
19     euint32 private encryptedConstantZero;
20     mapping(address => VoterState) private voterStates;
21
22     uint32 private clearCount; // =0 by default. It is only used
23     when the owner calls "decryptCount"
24     address private owner;
25     bool private isVotingOpen;
26     uint256 private electionId;
27
28     event CountDecrypted(uint32 count);
29     event VotingStarted();
30     event VotingClosed();
31
32     constructor() {
33         owner = msg.sender;
34         encryptedConstantOne = FHE.asEuint32(1);
35         encryptedConstantZero = FHE.asEuint32(0);
36         FHE.allowThis(encryptedConstantOne);
37         FHE.allowThis(encryptedConstantZero);
38         isVotingOpen = true;
39         electionId = 1;
40         emit VotingStarted();
41     }
42
43     modifier onlyWhenVotingOpen() {
44         require(isVotingOpen, "Voting is not open");
45     }
46 }
```

```

41         - ;
42     }
43
44     modifier onlyWhenVotingClosed() {
45         require(!isVotingOpen, "Voting is open");
46         - ;
47     }
48
49     modifier onlyOwner() {
50         require(msg.sender == owner, "Only owner can call this
function");
51         - ;
52     }
53
54     function closeVoting() external onlyWhenVotingOpen onlyOwner {
55         isVotingOpen = false;
56         emit VotingClosed();
57     }
58
59     function startVoting() external onlyOwner {
60         isVotingOpen = true;
61         encryptedCount = FHE.asEuint32(0);
62         FHE.allowThis(encryptedCount);
63         clearCount = 0;
64         electionId++;
65
66         emit VotingStarted();
67     }
68
69     function eboolToOneOrZero(ebool boolValue) private returns (
70     uint32) {
71         return FHE.select(boolValue, encryptedConstantOne,
72     encryptedConstantZero);
73     }
74
75     function subtractFromCount(uint32 valueToSubtract) private {
76         encryptedCount = FHE.sub(encryptedCount, valueToSubtract);
77         FHE.allowThis(encryptedCount);
78     }
79
80     function addToCount(uint32 valueToAdd) private {
81         encryptedCount = FHE.add(encryptedCount, valueToAdd);
82         FHE.allowThis(encryptedCount);
83     }
84
85     function setVote(ebool currentVote, address voter) private {
86         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
90 or 1
91 function vote(externalEbool externalYesOrNo, bytes calldata
92 proof) external onlyWhenVotingOpen {
93     ebool currentVote = FHE.fromExternal(externalYesOrNo, proof
94 );
95     VoterState memory state = voterStates[msg.sender];
96
97     euint32 valueToAdd;
98     if (state.lastElectionId != electionId) {
99         valueToAdd = eboolToOneOrZero(currentVote);
100        // state update happens in setVote
101    } else {
102        // subtract the previous vote from the total, and then
103        // add the new one
104        ebool previousVote = state.encryptedVote;
105        euint32 valueToSubtract = eboolToOneOrZero(previousVote
106 );
107        subtractFromCount(valueToSubtract);
108
109        valueToAdd = eboolToOneOrZero(currentVote);
110    }
111    addToCount(valueToAdd);
112
113    setVote(currentVote, msg.sender);
114
115    function getCount() external view onlyWhenVotingClosed returns
116 (euint32) {
117        return encryptedCount;
118    }
119
120    function getMyVote() external view returns (ebool) {
121        VoterState memory state = voterStates[msg.sender];
122        require(state.lastElectionId == electionId, "You have not
123 voted yet");
124        return state.encryptedVote;
125    }
126
127    function requestDecryption() external onlyWhenVotingClosed
128 onlyOwner {
129        bytes32[] memory cypherTexts = new bytes32[](1);
130        cypherTexts[0] = FHE.toBytes32(encryptedCount);

```

```

124     FHE.requestDecryption(
125         // the list of encrypted values we want to public
126         decrypt
127             cypherTexts,
128             // the function selector the FHEVM backend will
129             callback with the clear values as arguments
130             this.callbackDecryptSingleUint32.selector
131         );
132     }
133
134     function getDecryptedCount() external view onlyWhenVotingClosed
135     returns (uint32) {
136         return clearCount;
137     }
138
139     function callbackDecryptSingleUint32(
140         uint256 requestID,
141         bytes memory cleartexts,
142         bytes memory decryptionProof
143     ) external {
144         // The 'cleartexts' argument is an ABI encoding of the
145         decrypted values associated to the
146         // handles (using 'abi.encode').
147         // =====
148         // SECURITY WARNING!           //
149         // =====
150         // Must call 'FHE.checkSignatures(...)' here!
151         // -----
152         // This callback must only be called by the authorized
153         // FHEVM backend.
154         // To enforce this, the contract author MUST verify the
155         // authenticity of the caller
156         // by using the 'FHE.checkSignatures' helper. This ensures
157         // that the provided signatures
158         // match the expected FHEVM backend and prevents
159         // unauthorized or malicious calls.
160         //
161         // Failing to perform this verification allows anyone to
162         // invoke this function with
163         // forged values, potentially compromising contract
164         // integrity.
165         //
166         // The responsibility for signature validation lies
167         // entirely with the contract author.
168         //
169         // The signatures are included in the 'decryptionProof'
170         // parameter.

```

```

159         //
160         FHE.checkSignatures(requestID, cleartexts, decryptionProof)
161     ;
162
163     uint32 decryptedInput = abi.decode(cleartexts, (uint32));
164     clearCount = decryptedInput;
165
166     emit CountDecrypted(clearCount);
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 "
5     @fhevm/solidity/lib/FHE.sol";
6 import {SepoliaConfig} from "@fhevm/solidity/config/ZamaConfig.sol"
7
8 // IDEA: to allow for a voting scheme that has multiple candidates
9 // (and only one vote per candidate) we can receive
10 // an array of booleans (and a proof that only one of them is true?
11 // or can we handle this internally with some FHE operations?)
12 // such as 'select'
13 // https://claude.ai/share/d065d579-5b33-4c03-a9e7-98586997c237
14 // this has some ideas to implement ZK proofs for the multiple
15 // votes
16 // we may need to use zk proofs in order to verify that the user
17 // provided exactly one true value in the array of booleans
18 // what we can do with FHE is get an ebool that (when decrypted) is
19 // true <=> the sum of all user votes is exactly k
20 // where k is the number of candidates someone can vote (think
21 // ranked voting systems...)
22 // however, we would need to decrypt that number! And that is not
23 // very secure... it would give us info about exactly how many
24 // votes
25 // someone made.
26 // ZK proofs are better. They would allow us to verify that the
27 // user provided <= k true values in the array of booleans.
28 // without us knowing the exact number of true values in their
29 // array
30
31 // inherits from SepoliaConfig to enable fhEVM support

```

```

21 contract FHEVoter is SepoliaConfig {
22     struct VoterState {
23         uint256 lastElectionId;
24         ebool encryptedVote;
25     }
26
27     euint32 private encryptedCount;
28     euint32 private encryptedConstantOne;
29     euint32 private encryptedConstantZero;
30     mapping(address => VoterState) private voterStates;
31
32     uint32 private clearCount; // =0 by default. It is only used
33     when the owner calls "decryptCount"
34     address private owner;
35     bool private isVotingOpen;
36     uint256 private electionId;
37
38     event CountDecrypted(uint32 count);
39     event VotingStarted();
40     event VotingClosed();
41
42     constructor() {
43         owner = msg.sender;
44         encryptedConstantOne = FHE.asEuint32(1);
45         encryptedConstantZero = FHE.asEuint32(0);
46         FHE.allowThis(encryptedConstantOne);
47         FHE.allowThis(encryptedConstantZero);
48         isVotingOpen = true;
49         electionId = 1;
50         emit VotingStarted();
51     }
52
53     modifier onlyWhenVotingOpen() {
54         require(isVotingOpen, "Voting is not open");
55         _;
56     }
57
58     modifier onlyWhenVotingClosed() {
59         require(!isVotingOpen, "Voting is open");
60         _;
61     }
62     modifier onlyOwner() {
63         require(msg.sender == owner, "Only owner can call this
64         function");
65         _;
66     }

```

```

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 (
83         euint32) {
84         return FHE.select(boolValue, encryptedConstantOne,
85         encryptedConstantZero);
86     }
87
88     function subtractFromCount(euint32 valueToSubtract) private {
89         encryptedCount = FHE.sub(encryptedCount, valueToSubtract);
90         FHE.allowThis(encryptedCount);
91     }
92
93     function addToCount(euint32 valueToAdd) private {
94         encryptedCount = FHE.add(encryptedCount, valueToAdd);
95         FHE.allowThis(encryptedCount);
96     }
97
98     function setVote(ebool currentVote, address voter) private {
99         voterStates[voter] = VoterState(electionId, currentVote);
100        FHE.allow(currentVote, voter);
101        FHE.allowThis(currentVote);
102    }
103
104    // using a boolean allows us to ensure what we add is always 0
105    // or 1
106    function vote(ebool externalYesOrNo, bytes calldata
107      proof) external onlyWhenVotingOpen {
108        ebool currentVote = FHE.fromExternal(externalYesOrNo, proof
109    );
110        VoterState memory state = voterStates[msg.sender];
111
112        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
113         // add the new one
114         ebool previousVote = state.encryptedVote;
115         euint32 valueToSubtract = eboolToOneOrZero(previousVote
116 );
117         subtractFromCount(valueToSubtract);
118
119         valueToAdd = eboolToOneOrZero(currentVote);
120     }
121     addToCount(valueToAdd);
122
123     setVote(currentVote, msg.sender);
124 }
125
126 function getCount() external view onlyWhenVotingClosed returns
127 (euint32) {
128     return encryptedCount;
129 }
130
131 function getMyVote() external view returns (ebool) {
132     VoterState memory state = voterStates[msg.sender];
133     require(state.lastElectionId == electionId, "You have not
134 voted yet");
135     return state.encryptedVote;
136 }
137
138 function requestDecryption() external onlyWhenVotingClosed
139 onlyOwner {
140     bytes32[] memory cypherTexts = new bytes32[](1);
141     cypherTexts[0] = FHE.toBytes32(encryptedCount);
142     FHE.requestDecryption(
143         // the list of encrypted values we want to publicly
144         decrypt
145         cypherTexts,
146         // the function selector the FHEVM backend will
147         callback with the clear values as arguments
148         this.callbackDecryptSingleUint32.selector
149     );
150 }
151
152 function getDecryptedCount() external view onlyWhenVotingClosed
153 returns (uint32) {
154     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
155         // decrypted values associated to the
156         // handles (using 'abi.encode').
157         // =====
158         // SECURITY WARNING!           //
159         // =====
160         // Must call 'FHE.checkSignatures(...)' here!
161         // -----
162         // This callback must only be called by the authorized
163         // FHEVM backend.
164         // To enforce this, the contract author MUST verify the
165         // authenticity of the caller
166         // by using the 'FHE.checkSignatures' helper. This ensures
167         // that the provided signatures
168         // match the expected FHEVM backend and prevents
169         // unauthorized or malicious calls.
170         //
171         // Failing to perform this verification allows anyone to
172         // invoke this function with
173         // forged values, potentially compromising contract
174         // integrity.
175         //
176         // The responsibility for signature validation lies
177         // entirely with the contract author.
178         //
179         // The signatures are included in the 'decryptionProof'
180         // parameter.
181         //
182         FHE.checkSignatures(requestID, cleartexts, decryptionProof)
183     ;
184
185         uint32 decryptedInput = abi.decode(cleartexts, (uint32));
186         clearCount = decryptedInput;
187
188         emit CountDecrypted(clearCount);
189     }
190 }
```

Listing 2: FHERankedVoter.sol