# File: ./modules/contracts/README.md

# fhEVM Contracts

## Description

fhEVM contracts is a Solidity library for secure smart-contract development using fhEVM and TFHE.

## Getting Started

### Installation

You can import this repo using your package manager.

```
# Using npm
npm install fhevm-contracts

# Using Yarn
yarn add fhevm-contracts

# Using pnpm
pnpm add fhevm-contracts
```

### Simple contract

To write Solidity contracts that use `TFHE` and/or `Gateway`, it is required to set different contract addresses.

Fortunately, the fhevm repo, one of this repo's dependencies, exports config files that can be inherited to simplify the process. The config should be the first to be imported in the order of the inherited contracts.

#### Using the mock network (for testing)

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

import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol";
import { ConfidentialERC20 } from "fhevm-contracts/contracts/token/ERC2

contract MyERC20 is MockZamaFHEVMConfig, ConfidentialERC20 {
  constructor() ConfidentialERC20("MyToken", "MYTOKEN") {
    _unsafeMint(1000000, msg.sender);
  }
}
```

#### Using Sepolia

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

import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so
import { ConfidentialERC20 } from "fhevm-contracts/contracts/token/ERC2

contract MyERC20 is SepoliaZamaFHEVMConfig, ConfidentialERC20 {
  constructor() ConfidentialERC20("MyToken", "MYTOKEN") {
    _unsafeMint(1000000, msg.sender);
  }
}
```

# Available contracts

These Solidity templates include governance-related and token-related contracts.

### Token

- ConfidentialERC20
- ConfidentialERC20Mintable
- ConfidentialERC20WithErrors
- ConfidentialERC20WithErrorsMintable

### Governance

- ConfidentialERC20Votes
- ConfidentialGovernorAlpha

### Utils

- EncryptedErrors

### Contributing

There are two ways to contribute to the Zama fhEVM contracts:

- Open issues to report bugs and typos, or to suggest new ideas.
- Request to become an official contributor by emailing hello@zama.ai.

Becoming an approved contributor involves signing our Contributor License Agreement (CLA). Only approved contributors can send pull requests, so please make sure to get in touch before you do.

### License

> [!CAUTION] Smart contracts are a nascent technology that carry a high level of technical risk and uncertainty. You are solely responsible for any use of the fhEVM Contracts and you assume all risks associated with any such use.

This software is distributed under the **BSD-3-Clause-Clear** license. If you have any question about the license, please contact us at hello@zama.ai.

# File: ./modules/contracts/contracts/token/ERC20/IConfidentialERC20.sol

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

import "fhevm/lib/TFHE.sol";

/**

- @title IConfidentialERC20
- @notice Interface that defines ERC20-like tokens with encrypted balances. */ *interface IConfidentialERC20 {* /*
  - @notice Emitted when the allowance of a `spender` for an `owner` is set by
  - a call to {approve}. */ event Approval(address indexed owner, address indexed spender, uint256 placeholder);

  /**

  - @notice Emitted when tokens are moved from one account (`from`) to
  - another (`to`).
  - Last argument is either a default placeholder, typically equal to max(uint256), in case of
  - a ConfidentialERC20 without error handling, or an errorId in case of encrypted error handling. */ event Transfer(address indexed from, address indexed to, uint256 errorId);

  /**

  - @notice Sets the `encryptedAmount` as the allowance of `spender` over the caller's tokens. */ function approve(address spender, einput encryptedAmount, bytes calldata inputProof) external returns (bool);

  /**

  - @notice Sets the `amount` as the allowance of `spender` over the caller's tokens. */ function approve(address spender, euint64 amount) external returns (bool);

  /**

  - @notice Transfers an encrypted amount from the message sender address to the `to` address. */ function transfer(address to, einput encryptedAmount, bytes calldata inputProof) external returns (bool);

  /**

  - @notice Transfers an amount from the message sender address to the `to` address. */ function transfer(address to, euint64 amount) external returns (bool);

  /**

  - @notice Transfers `amount` tokens using the caller's allowance. */ function transferFrom(address from, address to, euint64 amount) external returns (bool);

/**

- @notice Transfers `encryptedAmount` tokens using the caller's allowance. */ function transferFrom( address from, address to, einput encryptedAmount, bytes calldata inputProof ) external returns (bool);

/**

- @notice Returns the remaining number of tokens that `spender` is allowed to spend on behalf of the caller. */ function allowance(address owner, address spender) external view returns (euint64);

/**

- @notice Returns the balance handle of the caller. */ function balanceOf(address wallet) external view returns (euint64);

/**

- @notice Returns the number of decimals. */ function decimals() external view returns (uint8);

/**

- @notice Returns the name of the token. */ function name() external view returns (string memory);

/**

- @notice Returns the symbol of the token, usually a shorter version of the name. */ function symbol() external view returns (string memory);

/**

- @notice Returns the total supply of the token. */ function totalSupply() external view returns (uint64); }

# File: ./modules/contracts/contracts/token/ERC20/ConfidentialERC20.sol

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

import "fhevm/lib/TFHE.sol";

import { IERC20Errors } from "@openzeppelin/contracts/interfaces/draft-IERC6093.sol"; import { IConfidentialERC20 } from "./IConfidentialERC20.sol"; import { TFHEErrors } from "../../utils/TFHEErrors.sol";

/**

- @title ConfidentialERC20
- @notice This contract implements an encrypted ERC20-like token with confidential balances using

- Zama's FHE (Fully Homomorphic Encryption) library.
- @dev It supports standard ERC20 functions such as transferring tokens, minting,
- and setting allowances, but uses encrypted data types.
- The total supply is not encrypted.

*/ abstract contract ConfidentialERC20 is IConfidentialERC20, IERC20Errors, TFHEErrors { /// @notice used as a placehoder in Approval and Transfer events to comply with the official EIP20 uint256 internal constant _PLACEHOLDER = type(uint256).max; /// @notice Total supply. uint64 internal _totalSupply;

```
/// @notice Name.
string internal _name;

/// @notice Symbol.
string internal _symbol;

/// @notice A mapping from `account` address to an encrypted `balance`.
mapping(address account => euint64 balance) internal _balances;

/// @notice A mapping of the form mapping(account => mapping(spender =>
mapping(address account => mapping(address spender => euint64 allowance

/**
 * @param name_ Name of the token.
 * @param symbol_ Symbol.
 */
constructor(string memory name_, string memory symbol_) {
    _name = name_;
    _symbol = symbol_;
}

/**
 * @notice See {IConfidentialERC20-approve}.
 */
function approve(address spender, einput encryptedAmount, bytes calldat
    approve(spender, TFHE.asEuint64(encryptedAmount, inputProof));
    return true;
}

/**
 * @notice See {IConfidentialERC20-approve}.
 */
function approve(address spender, euint64 amount) public virtual return
    _isSenderAllowedForAmount(amount);
    address owner = msg.sender;
    _approve(owner, spender, amount);
    emit Approval(owner, spender, _PLACEHOLDER);
    return true;
}

/**
```

```
     * @notice See {IConfidentialERC20-transfer}.
     */
    function transfer(address to, einput encryptedAmount, bytes calldata in
        transfer(to, TFHE.asEuint64(encryptedAmount, inputProof));
        return true;
    }

    /**
     * @notice See {IConfidentialERC20-transfer}.
     */
    function transfer(address to, euint64 amount) public virtual returns (b
        _isSenderAllowedForAmount(amount);

        /// Make sure the owner has enough tokens.
        ebool canTransfer = TFHE.le(amount, _balances[msg.sender]);
        _transfer(msg.sender, to, amount, canTransfer);
        return true;
    }

    /**
     * @notice See {IConfidentialERC20-transferFrom}.
     */
    function transferFrom(
        address from,
        address to,
        einput encryptedAmount,
        bytes calldata inputProof
    ) public virtual returns (bool) {
        transferFrom(from, to, TFHE.asEuint64(encryptedAmount, inputProof))
        return true;
    }

    /**
     * @notice See {IConfidentialERC20-transferFrom}.
     */
    function transferFrom(address from, address to, euint64 amount) public
        _isSenderAllowedForAmount(amount);
        address spender = msg.sender;
        ebool isTransferable = _updateAllowance(from, spender, amount);
        _transfer(from, to, amount, isTransferable);
        return true;
    }

    /**
     * @notice See {IConfidentialERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual
        return _allowance(owner, spender);
    }

    /**
     * @notice See {IConfidentialERC20-balanceOf}.
     */
```

```
function balanceOf(address account) public view virtual returns (euint6
    return _balances[account];
}

/**
 * @notice See {IConfidentialERC20-decimals}.
 */
function decimals() public view virtual returns (uint8) {
    return 6;
}

/**
 * @notice See {IConfidentialERC20-name}.
 */
function name() public view virtual returns (string memory) {
    return _name;
}

/**
 * @notice See {IConfidentialERC20-symbol}.
 */
function symbol() public view virtual returns (string memory) {
    return _symbol;
}

/**
 * @notice See {IConfidentialERC20-totalSupply}.
 */
function totalSupply() public view virtual returns (uint64) {
    return _totalSupply;
}

function _approve(address owner, address spender, euint64 amount) inter
    if (owner == address(0)) {
        revert ERC20InvalidApprover(owner);
    }

    if (spender == address(0)) {
        revert ERC20InvalidSpender(spender);
    }

    _allowances[owner][spender] = amount;
    TFHE.allowThis(amount);
    TFHE.allow(amount, owner);
    TFHE.allow(amount, spender);
}

/**
 * @dev It does not incorporate any overflow check. It must be implemen
 *      by the function calling it.
 */
function _unsafeMint(address account, uint64 amount) internal virtual {
    _unsafeMintNoEvent(account, amount);
```

```
    emit Transfer(address(0), account, _PLACEHOLDER);
}

/**
 * @dev It does not incorporate any overflow check. It must be implemen
 *      by the function calling it.
 */
function _unsafeMintNoEvent(address account, uint64 amount) internal vi
    euint64 newBalanceAccount = TFHE.add(_balances[account], amount);
    _balances[account] = newBalanceAccount;
    TFHE.allowThis(newBalanceAccount);
    TFHE.allow(newBalanceAccount, account);
}

function _transfer(address from, address to, euint64 amount, ebool isTr
    _transferNoEvent(from, to, amount, isTransferable);
    emit Transfer(from, to, _PLACEHOLDER);
}

function _transferNoEvent(address from, address to, euint64 amount, ebo
    if (from == address(0)) {
        revert ERC20InvalidSender(from);
    }

    if (to == address(0)) {
        revert ERC20InvalidReceiver(to);
    }

    /// Add to the balance of `to` and subract from the balance of `fro
    euint64 transferValue = TFHE.select(isTransferable, amount, TFHE.as
    euint64 newBalanceTo = TFHE.add(_balances[to], transferValue);
    _balances[to] = newBalanceTo;
    TFHE.allowThis(newBalanceTo);
    TFHE.allow(newBalanceTo, to);
    euint64 newBalanceFrom = TFHE.sub(_balances[from], transferValue);
    _balances[from] = newBalanceFrom;
    TFHE.allowThis(newBalanceFrom);
    TFHE.allow(newBalanceFrom, from);
}

function _updateAllowance(address owner, address spender, euint64 amoun
    euint64 currentAllowance = _allowance(owner, spender);
    /// Make sure sure the allowance suffices.
    ebool allowedTransfer = TFHE.le(amount, currentAllowance);
    /// Make sure the owner has enough tokens.
    ebool canTransfer = TFHE.le(amount, _balances[owner]);
    ebool isTransferable = TFHE.and(canTransfer, allowedTransfer);
    _approve(owner, spender, TFHE.select(isTransferable, TFHE.sub(curre
    return isTransferable;
}

function _allowance(address owner, address spender) internal view virtu
    return _allowances[owner][spender];
```

```
}

function _isSenderAllowedForAmount(euint64 amount) internal view virtua
    if (!TFHE.isSenderAllowed(amount)) {
        revert TFHESenderNotAllowed();
    }
}

}
```

# File: ./modules/contracts/contracts/token/ERC20/extensions/ConfidentialERC20WithErrorsMintable.sol

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

import "fhevm/lib/TFHE.sol"; import { Ownable2Step, Ownable } from "@openzeppelin/contracts/access/Ownable2Step.sol";

import { ConfidentialERC20WithErrors } from "./ConfidentialERC20WithErrors.sol";

/**

- @title ConfidentialERC20WithErrorsMintable
- @notice This contract inherits ConfidentialERC20WithErrors.
- @dev It allows an owner to mint tokens. Mint amounts are public. */ *abstract contract ConfidentialERC20WithErrorsMintable is Ownable2Step, ConfidentialERC20WithErrors { /**
    - @notice Emitted when `amount` tokens are minted to one account (`to`). */ event Mint(address indexed to, uint64 amount);

  /**

    - @param name_ Name of the token.
    - @param symbol_ Symbol.
    - @param owner_ Owner address. */ constructor( string memory name_, string memory symbol_, address owner_ ) Ownable(owner_) ConfidentialERC20WithErrors(name_, symbol_) {}

  /**

    - @notice Mint tokens.
    - @param amount Amount of tokens to mint. */ function mint(uint64 amount) public virtual onlyOwner { _unsafeMint(msg.sender, amount); /// @dev Since _totalSupply is not encrypted and _totalSupply >= balances[msg.sender], /// the next line contains an overflow check for the encrypted operation above. _totalSupply = _totalSupply + amount; emit Mint(msg.sender, amount); } }

# File: ./modules/contracts/contracts/token/

# ERC20/extensions/ ConfidentialERC20WithErrors.sol

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

import "fhevm/lib/TFHE.sol"; import { ConfidentialERC20 } from "../ConfidentialERC20.sol"; import { EncryptedErrors } from "../../../utils/EncryptedErrors.sol";

/**

- @title ConfidentialERC20WithErrors
- @notice This contract implements an encrypted ERC20-like token with confidential balances using
- Zama's FHE (Fully Homomorphic Encryption) library.
- @dev It supports standard ERC20 functions such as transferring tokens, minting,
- and setting allowances, but uses encrypted data types.
- The total supply is not encrypted.
- It also supports error handling for encrypted errors.

*/ *abstract contract ConfidentialERC20WithErrors is ConfidentialERC20, EncryptedErrors { /* * @notice Error codes allow tracking (in the storage) whether a transfer worked. * @dev NO_ERROR: the transfer worked as expected * UNSUFFICIENT_BALANCE: the transfer failed because the * from balances were strictly inferior to the amount to transfer. * UNSUFFICIENT_APPROVAL: the transfer failed because the sender allowance * was strictly lower than the amount to transfer. */ enum ErrorCodes { NO_ERROR, UNSUFFICIENT_BALANCE, UNSUFFICIENT_APPROVAL }

```
/**
 * @param name_     Name of the token.
 * @param symbol_   Symbol.
 */
constructor(
    string memory name_,
    string memory symbol_
) ConfidentialERC20(name_, symbol_) EncryptedErrors(uint8(type(ErrorCod

/**
 * @notice See {IConfidentialERC20-transfer}.
 */
function transfer(address to, euint64 amount) public virtual override r
    _isSenderAllowedForAmount(amount);
    /// @dev Check whether the owner has enough tokens.
    ebool canTransfer = TFHE.le(amount, _balances[msg.sender]);
    euint8 errorCode = _errorDefineIfNot(canTransfer, uint8(ErrorCodes.
    _errorSave(errorCode);
    TFHE.allow(errorCode, msg.sender);
    TFHE.allow(errorCode, to);
    _transfer(msg.sender, to, amount, canTransfer);
```

```
    return true;
}

/**
 * @notice See {IConfidentialERC20-transferFrom}.
 */
function transferFrom(address from, address to, euint64 amount) public
    _isSenderAllowedForAmount(amount);
    address spender = msg.sender;
    ebool isTransferable = _updateAllowance(from, spender, amount);
    _transfer(from, to, amount, isTransferable);
    return true;
}

/**
 * @notice            Returns the error for a transfer id.
 * @param transferId  Transfer id. It can read from the `Transfer` even
 * @return errorCode  Encrypted error code.
 */
function getErrorCodeForTransferId(uint256 transferId) public view virt
    errorCode = _errorGetCodeEmitted(transferId);
}

function _transfer(address from, address to, euint64 amount, ebool isTr
    _transferNoEvent(from, to, amount, isTransferable);
    /// @dev It was incremented in _saveError.
    emit Transfer(from, to, _errorGetCounter() - 1);
}

function _updateAllowance(
    address owner,
    address spender,
    euint64 amount
) internal virtual override returns (ebool isTransferable) {
    euint64 currentAllowance = _allowance(owner, spender);
    /// @dev It checks whether the allowance suffices.
    ebool allowedTransfer = TFHE.le(amount, currentAllowance);
    euint8 errorCode = _errorDefineIfNot(allowedTransfer, uint8(ErrorCo
    /// @dev It checks that the owner has enough tokens.
    ebool canTransfer = TFHE.le(amount, _balances[owner]);
    ebool isNotTransferableButIsApproved = TFHE.and(TFHE.not(canTransfe
    errorCode = _errorChangeIf(
        isNotTransferableButIsApproved,
        /// @dev Should indeed check that spender is approved to not le
        ///      on balance of `from` to unauthorized spender via calli
        uint8(ErrorCodes.UNSUFFICIENT_BALANCE),
        errorCode
    );
    _errorSave(errorCode);
    TFHE.allow(errorCode, owner);
    TFHE.allow(errorCode, spender);
    isTransferable = TFHE.and(canTransfer, allowedTransfer);
    _approve(owner, spender, TFHE.select(isTransferable, TFHE.sub(curre
```

}

}

# File: ./modules/contracts/contracts/token/ERC20/extensions/ConfidentialERC20Mintable.sol

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

import "fhevm/lib/TFHE.sol"; import { Ownable2Step, Ownable } from "@openzeppelin/contracts/access/Ownable2Step.sol";

import { ConfidentialERC20 } from "../ConfidentialERC20.sol";

/**

- @title ConfidentialERC20Mintable
- @notice This contract inherits ConfidentialERC20.
- @dev It allows an owner to mint tokens. Mint amounts are public. */ *abstract contract ConfidentialERC20Mintable is Ownable2Step, ConfidentialERC20 { /**
    - @notice Emitted when `amount` tokens are minted to one account (`to`). */ event Mint(address indexed to, uint64 amount);

  /**

    - @param name_ Name of the token.
    - @param symbol_ Symbol.
    - @param owner_ Owner address. */ constructor( string memory name_, string memory symbol_, address owner_ ) Ownable(owner_) ConfidentialERC20(name_, symbol_) {}

  /**

    - @notice Mint tokens.
    - @param amount Amount of tokens to mint. */ function mint(uint64 amount) public virtual onlyOwner { _unsafeMint(msg.sender, amount); /// @dev Since _totalSupply is not encrypted and _totalSupply >= balances[msg.sender], /// the next line contains an overflow check for the encrypted operation above. _totalSupply = _totalSupply + amount; emit Mint(msg.sender, amount); } }

# File: ./modules/contracts/contracts/test/token/ERC20/TestConfidentialERC20Mintable.sol

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

import { ConfidentialERC20Mintable } from "../../../token/ERC20/extensions/ConfidentialERC20Mintable.sol"; import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol";

contract TestConfidentialERC20Mintable is MockZamaFHEVMConfig, ConfidentialERC20Mintable { constructor( string memory name_, string memory symbol_, address owner_ ) ConfidentialERC20Mintable(name_, symbol_, owner_) { // } }

# File: ./modules/contracts/contracts/test/token/ERC20/TestConfidentialERC20WithErrorsMintable.s

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

import { ConfidentialERC20WithErrorsMintable } from "../../../token/ERC20/extensions/ConfidentialERC20WithErrorsMintable.sol"; import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol";

contract TestConfidentialERC20WithErrorsMintable is MockZamaFHEVMConfig, ConfidentialERC20WithErrorsMintable { constructor( string memory name_, string memory symbol_, address owner_ ) ConfidentialERC20WithErrorsMintable(name_, symbol_, owner_) { // } }

# File: ./modules/contracts/contracts/test/utils/TestEncryptedErrors.sol

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

import "fhevm/lib/TFHE.sol"; import { EncryptedErrors } from "../../utils/EncryptedErrors.sol"; import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol";

contract TestEncryptedErrors is MockZamaFHEVMConfig, EncryptedErrors { constructor(uint8 totalNumberErrorCodes_) EncryptedErrors(totalNumberErrorCodes_) { for (uint8 i; i < = totalNumberErrorCodes_; i + + ) { /// @dev It is not possible to access the _errorCodeDefinitions since it is private. TFHE.allow(TFHE.asEuint8(i), msg.sender); } }

```
function errorChangeIf(
    einput encryptedCondition,
    einput encryptedErrorCode,
    bytes calldata inputProof,
    uint8 indexCode
) external returns (euint8 newErrorCode) {
    ebool condition = TFHE.asEbool(encryptedCondition, inputProof);
    euint8 errorCode = TFHE.asEuint8(encryptedErrorCode, inputProof);
    newErrorCode = _errorChangeIf(condition, indexCode, errorCode);
    _errorSave(newErrorCode);
    TFHE.allow(newErrorCode, msg.sender);
}
```

```
function errorChangeIfNot(
    einput encryptedCondition,
    einput encryptedErrorCode,
    bytes calldata inputProof,
    uint8 indexCode
) external returns (euint8 newErrorCode) {
    ebool condition = TFHE.asEbool(encryptedCondition, inputProof);
    euint8 errorCode = TFHE.asEuint8(encryptedErrorCode, inputProof);
    newErrorCode = _errorChangeIfNot(condition, indexCode, errorCode);
    _errorSave(newErrorCode);
    TFHE.allow(newErrorCode, msg.sender);
}

function errorDefineIf(
    einput encryptedCondition,
    bytes calldata inputProof,
    uint8 indexCode
) external returns (euint8 errorCode) {
    ebool condition = TFHE.asEbool(encryptedCondition, inputProof);
    errorCode = _errorDefineIf(condition, indexCode);
    _errorSave(errorCode);
    TFHE.allow(errorCode, msg.sender);
}

function errorDefineIfNot(
    einput encryptedCondition,
    bytes calldata inputProof,
    uint8 indexCode
) external returns (euint8 errorCode) {
    ebool condition = TFHE.asEbool(encryptedCondition, inputProof);
    errorCode = _errorDefineIfNot(condition, indexCode);
    _errorSave(errorCode);
    TFHE.allow(errorCode, msg.sender);
}

function errorGetCodeDefinition(uint8 indexCodeDefinition) external vie
    errorCode = _errorGetCodeDefinition(indexCodeDefinition);
}

function errorGetCodeEmitted(uint256 errorId) external view returns (eu
    errorCode = _errorGetCodeEmitted(errorId);
}

function errorGetCounter() external view returns (uint256 countErrors)
    countErrors = _errorGetCounter();
}

function errorGetNumCodesDefined() external view returns (uint8 totalNu
    totalNumberErrorCodes = _errorGetNumCodesDefined();
}

}
```

# File: ./modules/contracts/contracts/test/governance/TestConfidentialERC20Votes.sol

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

import { ConfidentialERC20Votes } from "../../governance/ConfidentialERC20Votes.sol";
import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol";

contract TestConfidentialERC20Votes is MockZamaFHEVMConfig, ConfidentialERC20Votes { constructor( address owner_, string memory name_, string memory symbol_, string memory version_, uint64 totalSupply_ ) ConfidentialERC20Votes(owner_, name_, symbol_, version_, totalSupply_) { // } }

# File: ./modules/contracts/contracts/test/governance/TestConfidentialGovernorAlpha.sol

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

import { ConfidentialGovernorAlpha } from "../../governance/ConfidentialGovernorAlpha.sol"; import { MockZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.sol"; import { MockZamaGatewayConfig } from "fhevm/config/ZamaGatewayConfig.sol";

contract TestConfidentialGovernorAlpha is MockZamaFHEVMConfig, MockZamaGatewayConfig, ConfidentialGovernorAlpha { constructor( address owner_, address timelock_, address confidentialERC20Votes_, uint256 votingPeriod_, uint256 maxDecryptionDelay_ ) ConfidentialGovernorAlpha(owner_, timelock_, confidentialERC20Votes_, votingPeriod_, maxDecryptionDelay_) { // } }

# File: ./modules/contracts/contracts/utils/EncryptedErrors.sol

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

import "fhevm/lib/TFHE.sol";

/**

- @title EncryptedErrors.
- @notice This abstract contract is used for error handling in the fhEVM.
- `Error codes are encrypted in the constructor inside the `_

- @dev `_errorCodeDefinitions[0]` should always refer to the `NO_ERROR` code, by default. */ abstract contract EncryptedErrors { /// @notice Returned if the error index is invalid. error ErrorIndexInvalid();

  /// @notice Returned if the error index is null. error ErrorIndexIsNull();

  /// @notice Returned if the total number of errors is equal to zero. error TotalNumberErrorCodesEqualToZero();

  /// @notice Total number of error codes. /// @dev Should hold the constant size of the `_errorCodeDefinitions` mapping. uint8 private immutable _TOTAL_NUMBER_ERROR_CODES;

  /// @notice Used to keep track of number of emitted errors. /// @dev Should hold the size of the _errorCodesEmitted mapping. uint256 private _errorCounter;

  /// @notice Mapping of trivially encrypted error codes definitions. /// @dev In storage because solc does not support immutable mapping, neither immutable arrays, yet. mapping(uint8 errorCode = > euint8 encryptedErrorCode) private _errorCodeDefinitions;

  /// @notice Mapping of encrypted error codes emitted. mapping(uint256 errorIndex = > euint8 encryptedErrorCode) private _errorCodesEmitted;

  /**

  - @notice Sets the non-null value for `_TOTAL_NUMBER_ERROR_CODES`
  - corresponding to the total number of
  - @param totalNumberErrorCodes_ Total number of different errors.
  - @dev `totalNumberErrorCodes_` must be non-null
  - (`_errorCodeDefinitions[0]` correspor

  */ constructor(uint8 totalNumberErrorCodes_) { if (totalNumberErrorCodes_ = = 0) { revert TotalNumberErrorCodesEqualToZero(); }

```
for (uint8 i; i <= totalNumberErrorCodes_; i++) {
    euint8 errorCode = TFHE.asEuint8(i);
    _errorCodeDefinitions[i] = errorCode;
    TFHE.allowThis(errorCode);
}

_TOTAL_NUMBER_ERROR_CODES = totalNumberErrorCodes_;
```

  }

  /**

  - @notice Computes an encrypted error code, result will be either a reencryption of
  - `_errorCodeDefinitions[indexCode]` if `cor
  - or of `errorCode` otherwise.

- @param condition Encrypted boolean used in the select operator.
- @param errorCode Selected error code if `condition` encrypts `true`.
- @return newErrorCode New reencrypted error code depending on `condition` value.
- @dev `indexCode` must be below the total number of error codes. */ function _errorChangeIf( ebool condition, uint8 indexCode, euint8 errorCode ) internal virtual returns (euint8 newErrorCode) { if (indexCode > _TOTAL_NUMBER_ERROR_CODES) { revert ErrorIndexInvalid(); }

  newErrorCode = TFHE.select(condition, _errorCodeDefinitions[indexCode], errorCode); }

/**

- @notice Does the opposite of `changeErrorIf`, i.e result will be either a reencryption of
- `_errorCodeDefinitions[indexCode]` if `cor
- or of `errorCode` otherwise.
- @param condition The encrypted boolean used in the `TFHE.select`.
- @param errorCode The selected error code if `condition` encrypts `false`.
- @return newErrorCode New error code depending on `condition` value.
- @dev `indexCode` must be below the total number of error codes. */ function _errorChangeIfNot( ebool condition, uint8 indexCode, euint8 errorCode ) internal virtual returns (euint8 newErrorCode) { if (indexCode > _TOTAL_NUMBER_ERROR_CODES) { revert ErrorIndexInvalid(); }

  newErrorCode = TFHE.select(condition, errorCode, _errorCodeDefinitions[indexCode]); }

/**

- @notice Computes an encrypted error code, result will be either a reencryption of
- `_errorCodeDefinitions[indexCode]` if `conditi
- or of `NO_ERROR` otherwise.
- @param condition Encrypted boolean used in the select operator.
- @param indexCode Index of the selected error code if `condition` encrypts `true`.
- @return errorCode Reencrypted error code depending on `condition` value.
- @dev `indexCode` must be non-null and below the total number of defined error codes. */ function _errorDefineIf(ebool condition, uint8 indexCode) internal virtual returns (euint8 errorCode) { if (indexCode == 0) { revert

ErrorIndexIsNull(); }

if (indexCode > _TOTAL_NUMBER_ERROR_CODES) { revert ErrorIndexInvalid(); }

errorCode = TFHE.select(condition, _errorCodeDefinitions[indexCode], _errorCodeDefinitions[0]); }

/**

- @notice Does the opposite of `defineErrorIf`, i.e result will be either a reencryption of
- `_errorCodeDefinitions[indexCode]` if `condi
- of `NO_ERROR` otherwise.
- @param condition Encrypted boolean used in the select operator.
- @param indexCode Index of the selected error code if `condition` encrypts `false`.
- @return errorCode Reencrypted error code depending on `condition` value.
- @dev `indexCode` must be non-null and below the total number of defined error codes. */ function _errorDefineIfNot(ebool condition, uint8 indexCode) internal virtual returns (euint8 errorCode) { if (indexCode == 0) { revert ErrorIndexIsNull(); }

  if (indexCode > _TOTAL_NUMBER_ERROR_CODES) { revert ErrorIndexInvalid(); }

  errorCode = TFHE.select(condition, _errorCodeDefinitions[0], _errorCodeDefinitions[indexCode]); }

/**

- @notice Saves `errorCode` in storage, in the `_errorCodesEmitted` mapping.
- @param errorCode Encrypted error code to be saved in storage.
- @return errorId The `errorId` key in `_errorCodesEmitted` where `errorCode` is stored. */ function _errorSave(euint8 errorCode) internal virtual returns (uint256 errorId) { errorId = _errorCounter; _errorCounter++; _errorCodesEmitted[errorId] = errorCode;

  TFHE.allowThis(errorCode); }

/**

- @notice Returns the trivially encrypted error code at index `indexCodeDefinition`.
- @param indexCodeDefinition Index of the requested error code definition.
- @return errorCode Encrypted error code located at `indexCodeDefinition` in

`_errorCodeDefinitions`. */ function _errorGetCodeDefinition(uint8 indexCodeDefinition) internal view virtual returns (euint8 errorCode) { if (indexCodeDefinition >= _TOTAL_NUMBER_ERROR_CODES) { revert ErrorIndexInvalid(); }

errorCode = _errorCodeDefinitions[indexCodeDefinition]; }

/**

- @notice Returns the encrypted error code which was stored in `_errorCodesEmitted`
- at key `errorId`.
- @param errorId Requested key stored in the `_errorCodesEmitted` mapping.
- @return errorCode Encrypted error code located at the `errorId` key.
- @dev `errorId` must be a valid id, i.e below the error counter. */ function _errorGetCodeEmitted(uint256 errorId) internal view virtual returns (euint8 errorCode) { if (errorId >= _errorCounter) { revert ErrorIndexInvalid(); }

errorCode = _errorCodesEmitted[errorId]; }

/**

- @notice Returns the total counter of emitted of error codes.
- @return countErrors Number of errors emitted. */ function _errorGetCounter() internal view virtual returns (uint256 countErrors) { countErrors = _errorCounter; }

/**

- @notice Returns the total number of the possible error codes defined.
- @return totalNumberErrorCodes Total number of the different possible error codes. */ function _errorGetNumCodesDefined() internal view virtual returns (uint8 totalNumberErrorCodes) { totalNumberErrorCodes = _TOTAL_NUMBER_ERROR_CODES; } }

# File: ./modules/contracts/contracts/utils/TFHEErrors.sol

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

interface TFHEErrors { /** * @notice Returned when the `sender` is not allowed to access a value. */ error TFHESenderNotAllowed(); }

# File: ./modules/contracts/contracts/governance/ICompoundTimelock.sol

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

/**

- @title ICompoundTimelock */ interface ICompoundTimelock { /// @notice Returned if the delay is below the minimum delay. error DelayBelowMinimumDelay();

  /// @notice Returned if the delay exceeds the maximum delay. error DelayAboveMaximumDelay();

  /// @notice Returned if the transaction's execution reverted. error ExecutionReverted();

  /// @notice Returned if the `msg.sender` is not the admin. error SenderIsNotAdmin();

  /// @notice Returned if the `msg.sender` is not this contract (`CompoundTimelock`). error SenderIsNotTimelock();

  /// @notice Returned if the `msg.sender` is not `pendingAdmin`. error SenderIsNotPendingAdmin();

  /// @notice Returned if the transaction has not been queued. error TransactionNotQueued();

  /// @notice Returned if the transaction has not surpassed the time lock. error TransactionTooEarlyForExecution();

  /// @notice Returned if the estimated execution block does not satisfay the delay. error TransactionTooEarlyForQueuing();

  /// @notice Returned if the transaction is stale (too late for execution). error TransactionTooLateForExecution();

  /// @notice Emitted when there is a change of admin. event NewAdmin(address indexed newAdmin);

  /// @notice Emtited when there is a change of pending admin. event NewPendingAdmin(address indexed newPendingAdmin);

  /// @notice Emitted when there is a new delay set. event NewDelay(uint256 indexed newDelay);

  /// @notice Emitted when the queued transaction is canceled. event CancelTransaction( bytes32 indexed txHash, address indexed target, uint256 value, string signature, bytes data, uint256 eta );

  /// @notice Emitted when the queued transaction is executed. event ExecuteTransaction( bytes32 indexed txHash, address indexed target, uint256 value, string signature, bytes data, uint256 eta );

  /// @notice Emitted when a transaction is queued. event QueueTransaction( bytes32 indexed txHash, address indexed target, uint256 value, string signature, bytes data, uint256 eta );

  /**

  - @notice Returns the delay (in timestamp) for a queued transaction before it can be executed. */ function delay() external view returns (uint256);

/**

- @notice Returns the grace period (in timestamp).
- `The grace period indicates how long a transaction can remai`
- `executed again.`

/ / solhint-disable func-name-mixedcase*/ function GRACE_PERIOD() external view returns (uint256);

/**

- @notice Accept admin role. */ function acceptAdmin() external;

/**

- @notice Returns whether the transactions are queued. */ function queuedTransactions(bytes32 hash) external view returns (bool);

/**

- @notice Queue a transaction.
- @param target Target address to execute the transaction.
- @param signature Function signature to execute.
- @param data The data to include in the transaction.
- @param eta The earliest eta to queue the transaction.
- @return hashTransaction The transaction's hash. */ function queueTransaction( address target, uint256 value, string calldata signature, bytes calldata data, uint256 eta ) external returns (bytes32 hashTransaction);

/**

- @notice Cancel a queued transaction.
- @param target Target address to execute the transaction.
- @param signature Function signature to execute.
- @param data The data to include in the transaction.
- @param eta The earliest eta to queue the transaction. */ function cancelTransaction( address target, uint256 value, string calldata signature, bytes calldata data, uint256 eta ) external;

/**

- @notice Cancel a queued transaction.
- @param target Target address to execute the transaction.
- @param signature Function signature to execute.
- @param data The data to include in the transaction.
- @param eta The earliest eta to queue the transaction.
- @return response The response from the transaction once executed. */ function executeTransaction( address target, uint256 value, string calldata signature, bytes calldata data, uint256 eta ) external payable returns (bytes memory response); }

# File: ./modules/contracts/contracts/

# governance/ ConfidentialGovernorAlpha.sol

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

import "fhevm/lib/TFHE.sol"; import "fhevm/gateway/GatewayCaller.sol";

import { Ownable2Step, Ownable } from "@openzeppelin/contracts/access/Ownable2Step.sol"; import { IConfidentialERC20Votes } from "./IConfidentialERC20Votes.sol"; import { ICompoundTimelock } from "./ICompoundTimelock.sol";

/**

- @title ConfidentialGovernorAlpha
- @notice This is based on the GovernorAlpha.sol contract written by Compound Labs.
- see: compound-finance/compound-protocol/blob/master/contra
- This decentralized governance system allows users to propo
- The contract is responsible for:
- - Proposal: A new proposal is made to introduce a change.
- - Voting: Users can vote on the proposal, either in favor
- - Quorum: A minimum number of votes (quorum) must be reach
- - Execution: Once a proposal passes, it is executed and ta

*/ abstract contract ConfidentialGovernorAlpha is Ownable2Step, GatewayCaller { /// @notice Returned if proposal contains too many changes. error LengthAboveMaxOperations();

```
/// @notice Returned if the array length is equal to 0.
error LengthIsNull();

/// @notice Returned if array lengths are not equal.
error LengthsDoNotMatch();

/// @notice Returned if the maximum decryption delay is higher than 1 d
error MaxDecryptionDelayTooHigh();

/// @notice Returned if proposal's actions have already been queued.
error ProposalActionsAlreadyQueued();

/// @notice Returned if the proposal state is invalid for this operatio
/// @dev    It is returned for any proposal state not matching the expe
///         state to conduct the operation.
error ProposalStateInvalid();

/// @notice Returned if the proposal's state is active but `block.numbe
```

```
error ProposalStateNotActive();

/// @notice Returned if the proposal state is still active.
error ProposalStateStillActive();

/// @notice Returned if the proposer has another proposal in progress.
error ProposerHasAnotherProposal();

/// @notice Returned if the voter has already cast a vote
///         for this proposal.
error VoterHasAlreadyVoted();

/// @notice Emitted when a proposal is now active.
event ProposalActive(uint256 id);

/// @notice Emitted when a proposal has been canceled.
event ProposalCanceled(uint256 id);

/// @notice Emitted when a new proposal is created.
event ProposalCreated(
    uint256 id,
    address proposer,
    address[] targets,
    uint256[] values,
    string[] signatures,
    bytes[] calldatas,
    uint256 startBlock,
    uint256 endBlock,
    string description
);

/// @notice Emitted when a proposal is defeated either by (1) number of
///         quorum, (2) the number of `for` votes equal or inferior to
event ProposalDefeated(uint256 id);

/// @notice Emitted when a proposal has been executed in the Timelock.
event ProposalExecuted(uint256 id);

/// @notice Emitted when a proposal has been queued in the Timelock.
event ProposalQueued(uint256 id, uint256 eta);

/// @notice Emitted when a proposal has been rejected since the number
/// is lower than the required threshold.
event ProposalRejected(uint256 id);

/// @notice Emitted when a proposal has succeeded since the number of `
///         than quorum and strictly higher than `against` votes.
event ProposalSucceeded(uint256 id);

/// @notice Emitted when a vote has been cast on a proposal.
event VoteCast(address voter, uint256 proposalId);

/**
```

```
 * @notice                                Possible states that a proposal
 * @param Pending                         Proposal does not exist.
 * @param PendingThresholdVerification Proposal is created but token th
 * @param Rejected                        Proposal was rejected as the pro
 * @param Active                          Proposal is active and voters ca
 * @param PendingResults                  Proposal is not active and the r
 * @param Canceled                        Proposal has been canceled by th
 * @param Defeated                        Proposal has been defeated
 *                                        (either not reaching the quorum
 * @param Succeeded                       Proposal has succeeded (`forVote
 * @param Queued                          Proposal has been queued in the
 * @param Expired                         Proposal has expired (@dev This
 * @param Executed                        Proposal has been executed in th
 */
enum ProposalState {
    Pending,
    PendingThresholdVerification,
    Rejected,
    Active,
    PendingResults,
    Canceled,
    Defeated,
    Succeeded,
    Queued,
    Expired,
    Executed
}

/**
 * @param proposer              Proposal creator.
 * @param state                 State of the proposal.
 * @param eta                   The timestamp that the proposal will be
 *                              it is set automatically once the vote s
 * @param targets               The ordered list of target addresses fo
 * @param values                The ordered list of values (i.e. `msg.v
 * @param signatures            The ordered list of function signatures
 * @param calldatas             The ordered list of calldata to be pass
 * @param startBlock            The block at which voting begins: holde
 *                              to this block.
 * @param endBlock              The block at which voting ends: votes m
 * @param forVotes              Current encrypted number of votes for t
 * @param againstVotes          Current encrypted number of votes in op
 * @param forVotesDecrypted     For votes once decrypted by the gateway
 * @param againstVotesDecrypted Against votes once decrypted by the gat
 */

struct Proposal {
    address proposer;
    ProposalState state;
    uint256 eta;
    address[] targets;
    uint256[] values;
    string[] signatures;
```

```
    bytes[] calldatas;
    uint256 startBlock;
    uint256 endBlock;
    euint64 forVotes;
    euint64 againstVotes;
    uint64 forVotesDecrypted;
    uint64 againstVotesDecrypted;
}

/**
 * @param proposer      Proposal creator.
 * @param state         State of the proposal.
 * @param eta           The timestamp when the proposal will be availab
 * @param targets       The ordered list of target addresses for calls
 * @param values        The ordered list of values (i.e. `msg.value`) t
 * @param signatures    The ordered list of function signatures to be c
 * @param calldatas     The ordered list of calldata to be passed to ea
 * @param startBlock    The block at which voting begins: holders must
 * @param endBlock      The block at which voting ends: votes must be c
 * @param forVotes      Number of votes for this proposal once decrypte
 * @param againstVotes  Number of votes in opposition to this proposal
 */
struct ProposalInfo {
    address proposer;
    ProposalState state;
    uint256 eta;
    address[] targets;
    uint256[] values;
    string[] signatures;
    bytes[] calldatas;
    uint256 startBlock;
    uint256 endBlock;
    uint64 forVotes;
    uint64 againstVotes;
}

/**
 * @notice          Ballot receipt record for a voter.
 * @param hasVoted  Whether or not a vote has been cast.
 * @param support   Whether or not the voter supports the proposal.
 * @param votes     The number of votes cast by the voter.
 */
struct Receipt {
    bool hasVoted;
    ebool support;
    euint64 votes;
}

/// @notice The maximum number of actions that can be included in a pro
/// @dev    It is 10 actions per proposal.
uint256 public constant PROPOSAL_MAX_OPERATIONS = 10;

/// @notice The number of votes required for a voter to become a propos
```

```
/// @dev    It is set at 100,000, which is 1% of the total supply of th
uint256 public constant PROPOSAL_THRESHOLD = 100000e6;

/// @notice The number of votes in support of a proposal required in or
///         and for a vote to succeed.
/// @dev    It is set at 400,000, which is 4% of the total supply of th
uint64 public constant QUORUM_VOTES = 400000e6;

/// @notice The delay before voting on a proposal may take place once p
///         It is 1 block.
uint256 public constant VOTING_DELAY = 1;

/// @notice The maximum decryption delay for the Gateway to callback wi
uint256 public immutable MAX_DECRYPTION_DELAY;

/// @notice The duration of voting on a proposal, in blocks
/// @dev    It is recommended to be set at 3 days in blocks
///         (i.e 21,600 for 12-second blocks).
uint256 public immutable VOTING_PERIOD;

/// @notice ConfidentialERC20Votes governance token.
IConfidentialERC20Votes public immutable CONFIDENTIAL_ERC20_VOTES;

/// @notice Compound Timelock.
ICompoundTimelock public immutable TIMELOCK;

/// @notice Constant for zero using TFHE.
/// @dev    Since it is expensive to compute 0, it is stored instead.
///         However, is not possible to define it as constant due to TF
/* solhint-disable var-name-mixedcase*/
euint64 private _EUINT64_ZERO;

/// @notice Constant for PROPOSAL_THRESHOLD using TFHE.
/// @dev    Since it is expensive to compute 0, it is stored instead.
///         However, is not possible to define it as constant due to TF
/* solhint-disable var-name-mixedcase*/
euint64 private _EUINT64_PROPOSAL_THRESHOLD;

/// @notice The total number of proposals made.
///         It includes all proposals, including the ones that
///         were rejected/canceled/defeated.
uint256 public proposalCount;

/// @notice The latest proposal for each proposer.
mapping(address proposer => uint256 proposalId) public latestProposalId

/// @notice Ballot receipt for an account for a proposal id.
mapping(uint256 proposalId => mapping(address => Receipt)) internal _ac

/// @notice The official record of all proposals that have been created
mapping(uint256 proposalId => Proposal proposal) internal _proposals;

/// @notice Returns the proposal id associated with the request id from
```

```
/// @dev    This mapping is used for decryption.
mapping(uint256 requestId => uint256 proposalId) internal _requestIdToP

/**
 * @param owner_                    Owner address.
 * @param timelock_                 Timelock contract.
 * @param confidentialERC20Votes_   ConfidentialERC20Votes token.
 * @param votingPeriod_             Voting period.
 * @dev                            Do not use a small value in product
 *                                 unless for testing purposes. It sho
 *                                 For instance, 3 days would have a v
 * @param maxDecryptionDelay_       Maximum delay for the Gateway to de
 * @dev                            Do not use a small value in product
 *                                 cannot be processed because the blo
 *                                 The current implementation expects
 *                                 value within the delay specified, a
 */
constructor(
    address owner_,
    address timelock_,
    address confidentialERC20Votes_,
    uint256 votingPeriod_,
    uint256 maxDecryptionDelay_
) Ownable(owner_) {
    TIMELOCK = ICompoundTimelock(timelock_);
    CONFIDENTIAL_ERC20_VOTES = IConfidentialERC20Votes(confidentialERC2
    VOTING_PERIOD = votingPeriod_;

    /// @dev The maximum delay is set to 1 day.
    if (maxDecryptionDelay_ > 1 days) {
        revert MaxDecryptionDelayTooHigh();
    }

    MAX_DECRYPTION_DELAY = maxDecryptionDelay_;

    /// @dev Store these constant-like variables in the storage.
    _EUINT64_ZERO = TFHE.asEuint64(0);
    _EUINT64_PROPOSAL_THRESHOLD = TFHE.asEuint64(PROPOSAL_THRESHOLD);

    TFHE.allowThis(_EUINT64_ZERO);
    TFHE.allowThis(_EUINT64_PROPOSAL_THRESHOLD);
}

/**
 * @notice              Cancel the proposal.
 * @param proposalId    Proposal id.
 * @dev                 Only this contract's owner or the proposer can
 *                      In the original GovernorAlpha, the proposer can
 *                      her votes are still above the threshold.
 */
function cancel(uint256 proposalId) public virtual {
    Proposal memory proposal = _proposals[proposalId];
```

```
    if (
        proposal.state == ProposalState.Rejected ||
        proposal.state == ProposalState.Canceled ||
        proposal.state == ProposalState.Defeated ||
        proposal.state == ProposalState.Executed
    ) {
        revert ProposalStateInvalid();
    }

    if (msg.sender != proposal.proposer) {
        _checkOwner();
    }

    /// @dev It is not necessary to cancel the transaction in the timel
    ///      unless the proposal has been queued.
    if (proposal.state == ProposalState.Queued) {
        for (uint256 i = 0; i < proposal.targets.length; i++) {
            TIMELOCK.cancelTransaction(
                proposal.targets[i],
                proposal.values[i],
                proposal.signatures[i],
                proposal.calldatas[i],
                proposal.eta
            );
        }
    }

    _proposals[proposalId].state = ProposalState.Canceled;

    emit ProposalCanceled(proposalId);
}

/**
 * @notice           Cast a vote.
 * @param proposalId Proposal id.
 * @param value      Encrypted value.
 * @param inputProof Input proof.
 */
function castVote(uint256 proposalId, einput value, bytes calldata inpu
    return castVote(proposalId, TFHE.asEbool(value, inputProof));
}

/**
 * @notice           Cast a vote.
 * @param proposalId Proposal id.
 * @param support    Support (true ==> `forVotes`, false ==> `againstVo
 */
function castVote(uint256 proposalId, ebool support) public virtual {
    return _castVote(msg.sender, proposalId, support);
}

/**
 * @notice Execute the proposal id.
```

```
 * @dev    Anyone can execute a proposal once it has been queued and th
 *         delay in the timelock is sufficient.
 */
function execute(uint256 proposalId) public payable virtual {
    Proposal memory proposal = _proposals[proposalId];

    if (proposal.state != ProposalState.Queued) {
        revert ProposalStateInvalid();
    }

    for (uint256 i = 0; i < proposal.targets.length; i++) {
        TIMELOCK.executeTransaction{ value: proposal.values[i] }(
            proposal.targets[i],
            proposal.values[i],
            proposal.signatures[i],
            proposal.calldatas[i],
            proposal.eta
        );
    }

    _proposals[proposalId].state = ProposalState.Executed;

    emit ProposalExecuted(proposalId);
}

/**
 * @notice            Start a new proposal.
 * @param targets     Target addresses.
 * @param values      Values.
 * @param signatures  Signatures.
 * @param calldatas   Calldatas.
 * @param description Plain text description of the proposal.
 * @return proposalId Proposal id.
 */
function propose(
    address[] memory targets,
    uint256[] memory values,
    string[] memory signatures,
    bytes[] memory calldatas,
    string memory description
) public virtual returns (uint256 proposalId) {
    {
        uint256 length = targets.length;

        if (length != values.length || length != signatures.length || l
            revert LengthsDoNotMatch();
        }

        if (length == 0) {
            revert LengthIsNull();
        }

        if (length > PROPOSAL_MAX_OPERATIONS) {
```

```
        revert LengthAboveMaxOperations();
    }
}

uint256 latestProposalId = latestProposalIds[msg.sender];

if (latestProposalId != 0) {
    ProposalState proposerLatestProposalState = _proposals[latestPr

    if (
        proposerLatestProposalState != ProposalState.Rejected &&
        proposerLatestProposalState != ProposalState.Defeated &&
        proposerLatestProposalState != ProposalState.Canceled &&
        proposerLatestProposalState != ProposalState.Executed
    ) {
        revert ProposerHasAnotherProposal();
    }
}

uint256 startBlock = block.number + VOTING_DELAY;
uint256 endBlock = startBlock + VOTING_PERIOD;
uint256 thisProposalId = ++proposalCount;

_proposals[thisProposalId] = Proposal({
    proposer: msg.sender,
    state: ProposalState.PendingThresholdVerification,
    eta: 0,
    targets: targets,
    values: values,
    signatures: signatures,
    calldatas: calldatas,
    startBlock: startBlock,
    endBlock: endBlock,
    forVotes: _EUINT64_ZERO,
    againstVotes: _EUINT64_ZERO,
    forVotesDecrypted: 0,
    againstVotesDecrypted: 0
});

latestProposalIds[msg.sender] = thisProposalId;

emit ProposalCreated(
    thisProposalId,
    msg.sender,
    targets,
    values,
    signatures,
    calldatas,
    startBlock,
    endBlock,
    description
);
```

```
    ebool canPropose = TFHE.lt(
        _EUINT64_PROPOSAL_THRESHOLD,
        CONFIDENTIAL_ERC20_VOTES.getPriorVotesForGovernor(msg.sender, b
    );

    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(canPropose);

    uint256 requestId = Gateway.requestDecryption(
        cts,
        this.callbackInitiateProposal.selector,
        0,
        block.timestamp + MAX_DECRYPTION_DELAY,
        false
    );

    _requestIdToProposalId[requestId] = thisProposalId;

    return thisProposalId;
}

/**
 * @notice            Queue a new proposal.
 * @dev               It can be done only if the proposal has succeeded
 *                    Anyone can queue a proposal.
 * @param proposalId  Proposal id.
 */
function queue(uint256 proposalId) public virtual {
    Proposal memory proposal = _proposals[proposalId];

    if (proposal.state != ProposalState.Succeeded) {
        revert ProposalStateInvalid();
    }

    uint256 eta = block.timestamp + TIMELOCK.delay();

    for (uint256 i = 0; i < proposal.targets.length; i++) {
        _queueOrRevert(proposal.targets[i], proposal.values[i], proposa
    }

    _proposals[proposalId].eta = eta;
    _proposals[proposalId].state = ProposalState.Queued;

    emit ProposalQueued(proposalId, eta);
}

/**
 * @notice            Request the vote results to be decrypted.
 * @dev               Anyone can request the decryption of the vote.
 * @param proposalId  Proposal id.
 */
function requestVoteDecryption(uint256 proposalId) public virtual {
    if (_proposals[proposalId].state != ProposalState.Active) {
```

```
        revert ProposalStateInvalid();
    }

    if (_proposals[proposalId].endBlock >= block.number) {
        revert ProposalStateStillActive();
    }

    uint256[] memory cts = new uint256[](2);
    cts[0] = Gateway.toUint256(_proposals[proposalId].forVotes);
    cts[1] = Gateway.toUint256(_proposals[proposalId].againstVotes);

    uint256 requestId = Gateway.requestDecryption(
        cts,
        this.callbackVoteDecryption.selector,
        0,
        block.timestamp + MAX_DECRYPTION_DELAY,
        false
    );

    _requestIdToProposalId[requestId] = proposalId;
    _proposals[proposalId].state = ProposalState.PendingResults;
}

/**
 * @dev                 Only callable by the gateway.
 * @param requestId     Request id (from the Gateway)
 * @param canInitiate   Whether the proposal can be initiated.
 */
function callbackInitiateProposal(uint256 requestId, bool canInitiate)
    uint256 proposalId = _requestIdToProposalId[requestId];

    if (canInitiate) {
        _proposals[proposalId].state = ProposalState.Active;
        emit ProposalActive(proposalId);
    } else {
        _proposals[proposalId].state = ProposalState.Rejected;
        emit ProposalRejected(proposalId);
    }
}

/**
 * @dev                          Only callable by the gateway.
 *                               If `forVotesDecrypted` == `againstVotes
 * @param forVotesDecrypted     For votes.
 * @param againstVotesDecrypted Against votes.
 */
function callbackVoteDecryption(
    uint256 requestId,
    uint256 forVotesDecrypted,
    uint256 againstVotesDecrypted
) public virtual onlyGateway {
    uint256 proposalId = _requestIdToProposalId[requestId];
```

```
    /// @dev It is safe to downcast since the original values were euin
    _proposals[proposalId].forVotesDecrypted = uint64(forVotesDecrypted
    _proposals[proposalId].againstVotesDecrypted = uint64(againstVotesD

    if (forVotesDecrypted > againstVotesDecrypted && forVotesDecrypted
        _proposals[proposalId].state = ProposalState.Succeeded;
        emit ProposalSucceeded(proposalId);
    } else {
        _proposals[proposalId].state = ProposalState.Defeated;
        emit ProposalDefeated(proposalId);
    }
}

/**
 * @dev Only callable by `owner`.
 */
function acceptTimelockAdmin() public virtual onlyOwner {
    TIMELOCK.acceptAdmin();
}

/**
 * @dev                   Only callable by `owner`.
 * @param newPendingAdmin Address of the new pending admin for the time
 * @param eta             Eta for executing the transaction in the time
 */
function executeSetTimelockPendingAdmin(address newPendingAdmin, uint25
    TIMELOCK.executeTransaction(address(TIMELOCK), 0, "setPendingAdmin(
}

/**
 * @dev                   Only callable by `owner`.
 * @param newPendingAdmin Address of the new pending admin for the time
 * @param eta             Eta for queuing the transaction in the timelo
 */
function queueSetTimelockPendingAdmin(address newPendingAdmin, uint256
    TIMELOCK.queueTransaction(address(TIMELOCK), 0, "setPendingAdmin(ad
}

/**
 * @notice                 Returns proposal information for a proposal
 * @dev                    It returns decrypted `forVotes`/`againstVot
 *                         These are only available after the decrypti
 * @param proposalId       Proposal id.
 * @return proposalInfo    Proposal information.
 */
function getProposalInfo(uint256 proposalId) public view virtual return
    Proposal memory proposal = _proposals[proposalId];
    proposalInfo.proposer = proposal.proposer;
    proposalInfo.state = proposal.state;
    proposalInfo.eta = proposal.eta;
    proposalInfo.targets = proposal.targets;
    proposalInfo.values = proposal.values;
    proposalInfo.signatures = proposal.signatures;
```

```
    proposalInfo.calldatas = proposal.calldatas;
    proposalInfo.startBlock = proposal.startBlock;
    proposalInfo.endBlock = proposal.endBlock;
    proposalInfo.forVotes = proposal.forVotesDecrypted;
    proposalInfo.againstVotes = proposal.againstVotesDecrypted;

    /// The state is adjusted but not closed.
    if (
        (proposalInfo.state == ProposalState.Queued) &&
        (block.timestamp > proposalInfo.eta + TIMELOCK.GRACE_PERIOD())
    ) {
        proposalInfo.state = ProposalState.Expired;
    }
}

/**
 * @notice              Returns the vote receipt information for the ac
 * @param proposalId    Proposal id.
 * @param account       Account address.
 * @return hasVoted     Whether the account has voted.
 * @return support      The support for the account (true ==> vote for,
 * @return votes        The number of votes cast.
 */
function getReceipt(uint256 proposalId, address account) public view vi
    Receipt memory receipt = _accountReceiptForProposalId[proposalId][a
    return (receipt.hasVoted, receipt.support, receipt.votes);
}

function _castVote(address voter, uint256 proposalId, ebool support) in
    Proposal storage proposal = _proposals[proposalId];

    if (proposal.state != ProposalState.Active) {
        revert ProposalStateInvalid();
    }

    if (block.number > proposal.endBlock) {
        revert ProposalStateNotActive();
    }

    Receipt storage receipt = _accountReceiptForProposalId[proposalId][

    if (receipt.hasVoted) {
        revert VoterHasAlreadyVoted();
    }

    euint64 votes = CONFIDENTIAL_ERC20_VOTES.getPriorVotesForGovernor(v
    proposal.forVotes = TFHE.select(support, TFHE.add(proposal.forVotes
    proposal.againstVotes = TFHE.select(support, proposal.againstVotes,

    receipt.hasVoted = true;
    receipt.support = support;
    receipt.votes = votes;
```

```
    TFHE.allowThis(proposal.forVotes);
    TFHE.allowThis(proposal.againstVotes);
    TFHE.allowThis(receipt.support);
    TFHE.allowThis(receipt.votes);
    TFHE.allow(receipt.support, msg.sender);
    TFHE.allow(receipt.votes, msg.sender);

    /// @dev `support` and `votes` are encrypted values.
    ///       There is no need to include them in the event.
    emit VoteCast(voter, proposalId);
}

function _queueOrRevert(
    address target,
    uint256 value,
    string memory signature,
    bytes memory data,
    uint256 eta
) internal virtual {
    if (TIMELOCK.queuedTransactions(keccak256(abi.encode(target, value,
        revert ProposalActionsAlreadyQueued();
    }

    TIMELOCK.queueTransaction(target, value, signature, data, eta);
}

}
```

# File: ./modules/contracts/contracts/governance/IConfidentialERC20Votes.sol

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

import "fhevm/lib/TFHE.sol";

/**

- @title IConfidentialERC20Votes
- @dev The ConfidentialGovernorAlpha relies on this interface. */ *interface IConfidentialERC20Votes { /**
    - @notice Determine the prior number of votes for an account as of a block number.
    - @dev Block number must be a finalized block or else this function will revert.
    - This function can change the state since the c
    - contract.
    - @param account Account address.
    - @param blockNumber The block number to get the vote balance at.
    - @return votes Number of votes the account as of the given block number. */

function getPriorVotesForGovernor(address account, uint256 blockNumber) external returns (euint64 votes); }

# File: ./modules/contracts/contracts/governance/CompoundTimelock.sol

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

import { ICompoundTimelock } from "./ICompoundTimelock.sol";

/**

- @title CompoundTimelock
- @notice This contract allows the admin to set a delay period before executing transactions.

- Transactions must be queued before execution. No transacti
- which offers time to verify the validity of pending transa
- It also has a grace period to allow for transactions
- not to be executed after a specific period following the q

*/ *contract CompoundTimelock is ICompoundTimelock { /** * @notice See {ICompoundTimelock-GRACE_PERIOD}. */* uint256 public constant GRACE_PERIOD = 14 days;

```
/// @notice Minimum delay that can be set in the `setDelay` function.
uint256 public constant MINIMUM_DELAY = 2 days;

/// @notice Maximum delay that can be set in the `setDelay` function.
uint256 public constant MAXIMUM_DELAY = 30 days;

/// @notice Admin address.
address public admin;

/// @notice Pending admin address.
/// @dev    The transer of the admin is a two-step process.
address public pendingAdmin;

/**
 * @notice See {ICompoundTimelock-delay}.
 */
uint256 public delay;

/// @notice Return whether the transaction is queued based on its hash.
mapping(bytes32 hashTransaction => bool isQueued) public queuedTransact

/**
 * @param admin_ Admin address.
 * @param delay_ Delay (in timestamp).
 */
```

```
constructor(address admin_, uint256 delay_) {
    if (delay_ < MINIMUM_DELAY) {
        revert DelayBelowMinimumDelay();
    }

    if (delay_ > MAXIMUM_DELAY) {
        revert DelayAboveMaximumDelay();
    }

    admin = admin_;
    delay = delay_;
}

receive() external payable {}

/**
 * @notice       Set the delay.
 * @dev          This transaction must be queued.
 * @param delay_ Delay (in timestamp).
 */
function setDelay(uint256 delay_) public {
    if (msg.sender != address(this)) {
        revert SenderIsNotTimelock();
    }

    if (delay_ < MINIMUM_DELAY) {
        revert DelayBelowMinimumDelay();
    }

    if (delay_ > MAXIMUM_DELAY) {
        revert DelayAboveMaximumDelay();
    }

    delay = delay_;

    emit NewDelay(delay);
}

/**
 * @notice See {ICompoundTimelock-acceptAdmin}.
 */
function acceptAdmin() public {
    if (msg.sender != pendingAdmin) {
        revert SenderIsNotPendingAdmin();
    }

    admin = msg.sender;
    pendingAdmin = address(0);

    emit NewAdmin(admin);
}

/**
```

```
     * @notice               Set the pending admin.
     * @dev                  This transaction must be queued.
     * @param pendingAdmin_ Pending admin address.
     */
    function setPendingAdmin(address pendingAdmin_) public {
        if (msg.sender != address(this)) {
            revert SenderIsNotTimelock();
        }

        pendingAdmin = pendingAdmin_;

        emit NewPendingAdmin(pendingAdmin);
    }

    /**
     * @notice See {ICompoundTimelock-queueTransaction}.
     */
    function queueTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public returns (bytes32) {
        if (msg.sender != admin) {
            revert SenderIsNotTimelock();
        }

        if (eta < block.timestamp + delay) {
            revert TransactionTooEarlyForQueuing();
        }

        bytes32 txHash = keccak256(abi.encode(target, value, signature, dat
        queuedTransactions[txHash] = true;

        emit QueueTransaction(txHash, target, value, signature, data, eta);
        return txHash;
    }

    /**
     * @notice See {ICompoundTimelock-cancelTransaction}.
     */
    function cancelTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public {
        if (msg.sender != admin) {
            revert SenderIsNotAdmin();
        }
```

```
    bytes32 txHash = keccak256(abi.encode(target, value, signature, dat
    queuedTransactions[txHash] = false;

    emit CancelTransaction(txHash, target, value, signature, data, eta)
}

/**
 * @notice See {ICompoundTimelock-executeTransaction}.
 */
function executeTransaction(
    address target,
    uint256 value,
    string memory signature,
    bytes memory data,
    uint256 eta
) public payable returns (bytes memory) {
    if (msg.sender != admin) {
        revert SenderIsNotAdmin();
    }

    bytes32 txHash = keccak256(abi.encode(target, value, signature, dat
    if (!queuedTransactions[txHash]) {
        revert TransactionNotQueued();
    }
    if (block.timestamp < eta) {
        revert TransactionTooEarlyForExecution();
    }

    if (block.timestamp > eta + GRACE_PERIOD) {
        revert TransactionTooLateForExecution();
    }

    queuedTransactions[txHash] = false;

    bytes memory callData;

    if (bytes(signature).length == 0) {
        callData = data;
    } else {
        callData = abi.encodePacked(bytes4(keccak256(bytes(signature)))
    }

    (bool success, bytes memory returnData) = target.call{ value: value

    if (!success) {
        revert ExecutionReverted();
    }

    emit ExecuteTransaction(txHash, target, value, signature, data, eta

    return returnData;
}
```

}

# File: ./modules/contracts/contracts/governance/ConfidentialERC20Votes.sol

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

import "fhevm/lib/TFHE.sol"; import { Ownable2Step, Ownable } from "@openzeppelin/contracts/access/Ownable2Step.sol"; import { EIP712 } from "@openzeppelin/contracts/utils/cryptography/EIP712.sol"; import { SignatureChecker } from "@openzeppelin/contracts/utils/cryptography/SignatureChecker.sol"; import { ConfidentialERC20 } from "../token/ERC20/ConfidentialERC20.sol"; import { IConfidentialERC20Votes } from "./IConfidentialERC20Votes.sol";

/**

- @title ConfidentialERC20Votes
- @notice This contract inherits ConfidentialERC20, EIP712, and Ownable2Step.
- `This is based on the Comp.sol contract written by Compound`
- `see: compound-finance/compound-protocol/blob/master/contra`
- `It is a governance token used to delegate votes, which can`
- `ConfidentialGovernorAlpha.sol.`
- `It uses encrypted votes to delegate the voting power assoc`
- `with an account's balance.`
- @dev The delegation of votes leaks information about the account's encrypted balance to the `delegatee`. */ abstract contract ConfidentialERC20Votes is IConfidentialERC20Votes, ConfidentialERC20, EIP712, Ownable2Step { /// @notice Returned if the `blockNumber` is higher or equal to the (current) `block.number`. /// @dev It is returned in requests to access votes. error BlockNumberEqualOrHigherThanCurrentBlock();

  /// @notice Returned if the `msg.sender` is not the `governor` contract. error GovernorInvalid();

  /// @notice Returned if the signature has expired. error SignatureExpired();

  /// @notice Returned if the signature's nonce is invalid. error SignatureNonceInvalid();

  /// @notice Returned if the signature's verification has failed. /// @dev See {SignatureChecker} for potential reasons. error SignatureVerificationFail();

  /// @notice Emitted when an `account` (i.e. `delegator`) changes its delegate. event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

  /// @notice Emitted when the governor contract that can reencrypt votes changes. ///

@dev WARNING: it can be set to a malicious contract, which could reencrypt all user votes. event NewGovernor(address indexed governor);

/// @notice Emitted when the account cancels a signature. event NonceIncremented(address account, uint256 newNonce);

/// @notice A checkpoint for marking number of votes from a given block. /// @param fromBlock Block from where the checkpoint applies. /// @param votes Total number of votes for the account power. /// @dev In Compound's implementation, `fromBlock` is defined as uint32 to allow tight-packing. /// However, in this implementations `votes` is uint256-based. /// `fromBlock`'s type is set to uint256, which simplifies the codebase. struct Checkpoint { uint256 fromBlock; euint64 votes; }

/// @notice The EIP-712 typehash for the `Delegation` struct. bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

/// @notice The smart contract that can access encrypted votes. /// @dev The contract is expected to be a governor contract. address public governor;

/// @notice A record of each account's `delegate`. mapping(address account => address delegate) public delegates;

/// @notice A record of states for signing/validating signatures. mapping(address account => uint256 nonce) public nonces;

/// @notice The number of checkpoints for an `account`. mapping(address account => uint32 _checkpoints) public numCheckpoints;

/// @notice A record of votes _checkpoints for an `account` using incremental indices. mapping(address account => mapping(uint32 index => Checkpoint checkpoint)) internal _checkpoints;

/// @notice Constant for zero using TFHE. /// @dev Since it is expensive to compute 0, it is stored instead. /// However, is not possible to define it as constant due to TFHE constraints. /* solhint-disable var-name-mixedcase*/ euint64 private _EUINT64_ZERO;

/**

- @param owner_ Owner address.
- @param name_ Token name.
- @param symbol_ Token symbol.
- @param version_ Version (e.g. "0.1", "1.0").
- @param totalSupply_ Total supply to mint. */ constructor( address owner_, string memory name_, string memory symbol_, string memory version_, uint64 totalSupply_ ) ConfidentialERC20(name_, symbol_) EIP712(name_, version_) Ownable(owner_) { *unsafeMint(owner*, totalSupply_); *totalSupply = totalSupply*;

  /// @dev Define the constant in the storage. _EUINT64_ZERO = TFHE.asEuint64(0); TFHE.allowThis(_EUINT64_ZERO); }

/**

- @notice Delegate votes from `msg.sender` to `delegatee`.
- @param delegatee The address to delegate votes to. */ function delegate(address delegatee) public virtual { return _delegate(msg.sender, delegatee); }

/**

- @notice Delegate votes from signatory to `delegatee`.
- @param delegator The account that delegates its votes. It must be the signer.
- @param delegatee The address to delegate votes to.
- @param nonce The contract state required to match the signature.
- @param expiry The time at which to expire the signature.
- @param signature The signature.
- @dev Signature can be either 64-byte or 65-byte long if it is from an EOA.

- `Else, it must adhere to ERC1271. See {https://eips`

*/ function delegateBySig( address delegator, address delegatee, uint256 nonce, uint256 expiry, bytes memory signature ) public virtual { bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry)); bytes32 digest = keccak256(abi.encodePacked("\x19\x01", _domainSeparatorV4(), structHash));

```
 if (!SignatureChecker.isValidSignatureNow(delegator, digest, signa
     revert SignatureVerificationFail();
 }

 if (nonce != nonces[delegator]++) {
     revert SignatureNonceInvalid();
 }

 if (block.timestamp > expiry) {
     revert SignatureExpired();
 }

 return _delegate(delegator, delegatee);
```

}

/**

- @notice Increment the nonce.

- @dev This function enables the sender to cancel a signature. */ function incrementNonce() public virtual { uint256 currentNonce = nonces[msg.sender]; nonces[msg.sender] = ++currentNonce;

  emit NonceIncremented(msg.sender, currentNonce); }

/**

- @notice See {IConfidentialERC20Votes-getPriorVotesForGovernor}. */ function getPriorVotesForGovernor(address account, uint256 blockNumber) public virtual returns (euint64 votes) { if (msg.sender != governor) { revert GovernorInvalid(); }

if (blockNumber >= block.number) { revert BlockNumberEqualOrHigherThanCurrentBlock(); }

votes = _getPriorVote(account, blockNumber); TFHE.allow(votes, msg.sender); }

/**

- @notice Get current votes of account.
- @param account Account address
- @return votes Current (encrypted) votes. */ function getCurrentVotes(address account) public view virtual returns (euint64 votes) { uint32 nCheckpoints = numCheckpoints[account]; if (nCheckpoints > 0) { votes = _checkpoints[account][nCheckpoints - 1].votes; } }

/**

- @notice Get the prior number of votes for an account as of a block number.
- @dev Block number must be a finalized block or else this function will revert.
- @param account Account address.
- @param blockNumber The block number to get the vote balance at.
- @return votes Number of votes the account as of the given block. */ function getPriorVotes(address account, uint256 blockNumber) public view virtual returns (euint64 votes) { if (blockNumber >= block.number) { revert BlockNumberEqualOrHigherThanCurrentBlock(); }

return _getPriorVote(account, blockNumber); }

/**

- @notice Set a governor contract.
- @param newGovernor New governor contract that can reencrypt/access votes. */ function setGovernor(address newGovernor) public virtual onlyOwner { governor = newGovernor; emit NewGovernor(newGovernor); }

function _delegate(address delegator, address delegatee) internal virtual { address currentDelegate = delegates[delegator]; euint64 delegatorBalance = _balances[delegator]; TFHE.allowThis(delegatorBalance); TFHE.allow(delegatorBalance, msg.sender); delegates[delegator] = delegatee;

```
 emit DelegateChanged(delegator, currentDelegate, delegatee);
 _moveDelegates(currentDelegate, delegatee, delegatorBalance);
```

}

function _getPriorVote(address account, uint256 blockNumber) internal view returns (euint64 votes) { uint32 nCheckpoints = numCheckpoints[account];

```
 if (nCheckpoints == 0) {
     /// If there is no checkpoint for the `account`, return encryp
     /// @dev It will not be possible to reencrypt it by the `accou
     votes = _EUINT64_ZERO;
 } else if (_checkpoints[account][nCheckpoints - 1].fromBlock <= bl
```

```
        /// First, check the most recent balance.
        votes = _checkpoints[account][nCheckpoints - 1].votes;
    } else if (_checkpoints[account][0].fromBlock > blockNumber) {
        /// Then, check if there is zero balance.
        /// @dev It will not be possible to reencrypt it by the `accou
        votes = _EUINT64_ZERO;
    } else {
        /// Else, search for the voting power at the `blockNumber`.
        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            /// Ceil to avoid overflow.
            uint32 center = upper - (upper - lower) / 2;
            Checkpoint memory cp = _checkpoints[account][center];

            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        votes = _checkpoints[account][lower].votes;
    }

}
```

function _moveDelegates(address srcRep, address dstRep, euint64 amount) internal virtual { if (srcRep != dstRep) { if (srcRep != address(0)) { uint32 srcRepNum = numCheckpoints[srcRep]; euint64 srcRepOld = srcRepNum > 0 ? _checkpoints[srcRep][srcRepNum - 1].votes : _EUINT64_ZERO; euint64 srcRepNew = TFHE.sub(srcRepOld, amount); /// srcRepOld - amount; _writeCheckpoint(srcRep, srcRepNum, srcRepNew); }

```
        if (dstRep != address(0)) {
            uint32 dstRepNum = numCheckpoints[dstRep];
            euint64 dstRepOld = dstRepNum > 0 ? _checkpoints[dstRep][d
            euint64 dstRepNew = TFHE.add(dstRepOld, amount); /// dstRe
            _writeCheckpoint(dstRep, dstRepNum, dstRepNew);
        }
    }

}
```

/// @dev Original restrictions to transfer from/to address(0) are removed since they /// are inherited. function _transfer(address from, address to, euint64 amount, ebool isTransferable) internal virtual override { super._transfer(from, to, amount, isTransferable); _moveDelegates(delegates[from], delegates[to], amount); }

function _writeCheckpoint(address delegatee, uint32 nCheckpoints, euint64 newVotes) internal virtual { if (nCheckpoints > 0 && _checkpoints[delegatee][nCheckpoints - 1].fromBlock == block.number) { _checkpoints[delegatee][nCheckpoints - 1].votes = newVotes; } else { _checkpoints[delegatee][nCheckpoints] =

Checkpoint(block.number, newVotes); numCheckpoints[delegatee] = nCheckpoints + 1; }

```
 TFHE.allowThis(newVotes);
 TFHE.allow(newVotes, delegatee);
```

}}

# File: ./modules/contracts/tasks/taskGatewayRelayer.ts

import { exec as oldExec } from "child_process"; import dotenv from "dotenv"; import fs from "fs"; import { task, types } from "hardhat/config"; import type { TaskArguments } from "hardhat/types"; import path from "path"; import { promisify } from "util";

const exec = promisify(oldExec);

const getCoin = async (address: string) => { const containerName = process.env["TEST_CONTAINER_NAME"] || "fhevm"; const response = await exec(`docker exec -i ${containerName} faucet ${address} | grep height`); const res = JSON.parse(response.stdout); if (res.raw_log.match("account sequence mismatch")) await getCoin(address); };

task("task:computeGatewayAddress") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployerAddress = new ethers.Wallet(taskArguments.privateKey).address; const gatewayContractAddressPrecomputed = ethers.getCreateAddress({ from: deployerAddress, nonce: 1, // deployer is supposed to have nonce 0 when deploying GatewayContract (0 nonce for implementation, +1 for UUPS) }); const envFilePath = path.join(__dirname, "../node_modules/fhevm/gateway/.env.gateway"); const content = `GATEWAY_CONTRACT_PREDEPLOY_ADDRESS=${gatewayContractAddressPrecomputed}`; try { fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log("gatewayContractAddress written to node_modules/fhevm/gateway/.env.gateway successfully!"); } catch (err) { console.error("Failed to write to node_modules/fhevm/gateway/.env.gateway:", err); }

```
const solidityTemplate = `// SPDX-License-Identifier: BSD-3-Clause-Clea
```

pragma solidity ^0.8.24;

address constant GATEWAY_CONTRACT_PREDEPLOY_ADDRESS = ${gatewayContractAddressPrecomputed}; `;

```
try {
  fs.writeFileSync("./node_modules/fhevm/gateway/lib/GatewayContractAdd
    encoding: "utf8",
    flag: "w",
  });
  console.log("node_modules/fhevm/gateway/lib/GatewayContractAddress.so
} catch (error) {
  console.error("Failed to write node_modules/fhevm/gateway/lib/Gateway
}
```

});

task("task:addRelayer") .addParam("privateKey", "The owner private key") .addParam("gatewayAddress", "The GatewayContract address") .addParam("relayerAddress", "The relayer address") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const codeAtAddress = await ethers.provider.getCode(taskArguments.gatewayAddress); if (codeAtAddress === "0x") { throw Error(`${taskArguments.gatewayAddress} is not a smart contract`); } const owner = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const gateway = await ethers.getContractAt("GatewayContract", taskArguments.gatewayAddress, owner); const tx = await gateway.addRelayer(taskArguments.relayerAddress); const rcpt = await tx.wait(); if (rcpt!.status === 1) { console.log(`Account ${taskArguments.relayerAddress} was succesfully added as an gateway relayer`); } else { console.log("Adding relayer failed"); } });

task("task:removeRelayer") .addParam("privateKey", "The owner private key") .addParam("gatewayAddress", "The GatewayContract address") .addParam("relayerAddress", "The relayer address") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const codeAtAddress = await ethers.provider.getCode(taskArguments.gatewayAddress); if (codeAtAddress === "0x") { throw Error(`${taskArguments.gatewayAddress} is not a smart contract`); } const owner = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const gateway = await ethers.getContractAt("GatewayContract", taskArguments.gatewayAddress, owner); const tx = await gateway.removeRelayer(taskArguments.relayerAddress); const rcpt = await tx.wait(); if (rcpt!.status === 1) { console.log(`Account ${taskArguments.relayerAddress} was succesfully removed from authorized relayers`); } else { console.log("Removing relayer failed"); } });

task("task:launchFhevm") .addOptionalParam("skipGetCoin", "Skip calling getCoin()", false, types.boolean) .addOptionalParam("useAddress", "Use address instead of privte key for the Gateway Relayer", false, types.boolean) .setAction(async function (taskArgs, hre) { const privKeyDeployer = process.env.PRIVATE_KEY_GATEWAY_DEPLOYER; const deployerAddress = new hre.ethers.Wallet(privKeyDeployer!).address; let relayerAddress; if (!taskArgs.useAddress) { const privKeyRelayer = process.env.PRIVATE_KEY_GATEWAY_RELAYER; relayerAddress = new hre.ethers.Wallet(privKeyRelayer!).address; } else { relayerAddress = process.env.ADDRESS_GATEWAY_RELAYER; } if (!taskArgs.skipGetCoin) { if (hre.network.name === "hardhat") { const bal = "0x1000000000000000000000000000000000000000"; const p1 = hre.network.provider.send("hardhat_setBalance", [deployerAddress, bal]); const p2 = hre.network.provider.send("hardhat_setBalance", [relayerAddress, bal]); await Promise.all([p1, p2]); } else { const p1 = getCoin(deployerAddress); const p2 = getCoin(relayerAddress); await Promise.all([p1, p2]); await new Promise((res) => setTimeout(res, 5000)); // wait 5 seconds } } await hre.run("task:deployGateway", { privateKey: privKeyDeployer, ownerAddress: deployerAddress });

```
const parsedEnv = dotenv.parse(fs.readFileSync("node_modules/fhevm/gate
const gatewayContractAddress = parsedEnv.GATEWAY_CONTRACT_PREDEPLOY_ADD

await hre.run("task:addRelayer", {
  privateKey: privKeyDeployer,
  gatewayAddress: gatewayContractAddress,
  relayerAddress: relayerAddress,
});
```

});

task("task:getBalances").setAction(async function (taskArgs, hre) { const privKeyDeployer = process.env.PRIVATE_KEY_GATEWAY_DEPLOYER; const privKeyRelayer = process.env.PRIVATE_KEY_GATEWAY_RELAYER; const deployerAddress = new hre.ethers.Wallet(privKeyDeployer!).address; const relayerAddress = new hre.ethers.Wallet(privKeyRelayer!).address; console.log(await hre.ethers.provider.getBalance(deployerAddress)); console.log(await hre.ethers.provider.getBalance(relayerAddress)); });

task("task:faucetToPrivate") .addParam("privateKey", "The receiver private key") .setAction(async function (taskArgs, hre) { const receiverAddress = new hre.ethers.Wallet(taskArgs.privateKey).address;

```
if (hre.network.name === "hardhat") {
  const bal = "0x1000000000000000000000000000000000000000";
  await hre.network.provider.send("hardhat_setBalance", [receiverAddres
} else {
  await getCoin(receiverAddress);
  await new Promise((res) => setTimeout(res, 5000)); // wait 5 seconds
}
```

});

task("task:faucetToAddress") .addParam("address", "The receiver address") .setAction(async function (taskArgs, hre) { const receiverAddress = taskArgs.address;

```
if (hre.network.name === "hardhat") {
  const bal = "0x1000000000000000000000000000000000000000";
  await hre.network.provider.send("hardhat_setBalance", [receiverAddres
} else {
  await getCoin(receiverAddress);
  await new Promise((res) => setTimeout(res, 5000)); // wait 5 seconds
}
```

});

# File: ./modules/contracts/tasks/taskDeploy.ts

import dotenv from "dotenv"; import fs from "fs"; import { task, types } from "hardhat/config"; import type { TaskArguments } from "hardhat/types";

task("task:deployGateway") .addParam("privateKey", "The deployer private key") .addParam("ownerAddress", "The owner address") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory("GatewayContract", deployer); const Gateway = await upgrades.deployProxy(factory, [taskArguments.ownerAddress], { initializer: "initialize", kind: "uups", }); await Gateway.waitForDeployment(); const GatewayContractAddress = await Gateway.getAddress(); const envConfig = dotenv.parse(fs.readFileSync("node_modules/fhevm/gateway/.env.gateway")); if (GatewayContractAddress !==

envConfig.GATEWAY_CONTRACT_PREDEPLOY_ADDRESS) { throw new Error( `The nonce of the deployer account is not null. Please use another deployer private key or relaunch a clean instance of the fhEVM`, ); } console.log("GatewayContract was deployed at address: ", GatewayContractAddress); });

task("task:deployACL") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory("fhevmTemp/contracts/ACL.sol:ACL", deployer); const acl = await upgrades.deployProxy(factory, [deployer.address], { initializer: "initialize", kind: "uups" }); await acl.waitForDeployment(); const address = await acl.getAddress(); const envConfigAcl = dotenv.parse(fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.acl")); if (address !== envConfigAcl.ACL_CONTRACT_ADDRESS) { throw new Error( `The nonce of the deployer account is not correct. Please relaunch a clean instance of the fhEVM`, ); } console.log("ACL was deployed at address:", address); });

task("task:deployTFHEExecutor") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory( "fhevmTemp/contracts/TFHEExecutor.events.sol:TFHEExecutor", deployer, ); const exec = await upgrades.deployProxy(factory, [deployer.address], { initializer: "initialize", kind: "uups" }); await exec.waitForDeployment(); const address = await exec.getAddress(); const envConfig = dotenv.parse(fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.exec")); if (address !== envConfig.TFHE_EXECUTOR_CONTRACT_ADDRESS) { throw new Error( `The nonce of the deployer account is not correct. Please relaunch a clean instance of the fhEVM`, ); } console.log("TFHEExecutor was deployed at address:", address); });

task("task:deployKMSVerifier") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory("fhevmTemp/contracts/KMSVerifier.sol:KMSVerifier", deployer); const kms = await upgrades.deployProxy(factory, [deployer.address], { initializer: "initialize", kind: "uups" }); await kms.waitForDeployment(); const address = await kms.getAddress(); const envConfig = dotenv.parse(fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.kmsverifier")); if (address !== envConfig.KMS_VERIFIER_CONTRACT_ADDRESS) { throw new Error( `The nonce of the deployer account is not correct. Please relaunch a clean instance of the fhEVM`, ); } console.log("KMSVerifier was deployed at address:", address); });

task("task:deployInputVerifier") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); let factory; if (process.env.IS_COPROCESSOR === "true") { factory = await ethers.getContractFactory( "fhevmTemp/contracts/InputVerifier.coprocessor.sol:InputVerifier", deployer, ); } else { factory = await ethers.getContractFactory("fhevmTemp/contracts/InputVerifier.native.sol:InputVerifier", deployer); } const kms = await upgrades.deployProxy(factory, [deployer.address], { initializer: "initialize", kind: "uups" }); await kms.waitForDeployment(); const address = await kms.getAddress(); const envConfig = dotenv.parse(fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.inputverifier")); if (address !== envConfig.INPUT_VERIFIER_CONTRACT_ADDRESS) { throw new Error( `The nonce of

the deployer account is not correct. Please relaunch a clean instance of the fhEVM, ); } console.log("InputVerifier was deployed at address:", address); });

task("task:deployFHEPayment") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers, upgrades }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory("fhevmTemp/contracts/FHEPayment.sol:FHEPayment", deployer); const payment = await upgrades.deployProxy(factory, [deployer.address], { initializer: "initialize", kind: "uups", }); await payment.waitForDeployment(); const address = await payment.getAddress(); const envConfig = dotenv.parse(fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.fhepayment")); if (address !== envConfig.FHE_PAYMENT_CONTRACT_ADDRESS) { throw new Error( The nonce of the deployer account is not correct. Please relaunch a clean instance of the fhEVM, ); } console.log("FHEPayment was deployed at address:", address); });

task("task:addSigners") .addParam("privateKey", "The deployer private key") .addParam("numSigners", "Number of KMS signers to add") .addOptionalParam( "useAddress", "Use addresses instead of private keys env variables for kms signers", false, types.boolean, ) .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployer = new ethers.Wallet(taskArguments.privateKey).connect(ethers.provider); const factory = await ethers.getContractFactory("fhevmTemp/contracts/KMSVerifier.sol:KMSVerifier", deployer); const kmsAdd = dotenv.parse( fs.readFileSync("node_modules/fhevm-core-contracts/addresses/.env.kmsverifier"), ).KMS_VERIFIER_CONTRACT_ADDRESS; const kmsVerifier = await factory.attach(kmsAdd); for (let idx = 0; idx < taskArguments.numSigners; idx++) { if (!taskArguments.useAddress) { const privKeySigner = process.env[PRIVATE_KEY_KMS_SIGNER_${idx}]; const kmsSigner = new ethers.Wallet(privKeySigner).connect(ethers.provider); const tx = await kmsVerifier.addSigner(kmsSigner.address); await tx.wait(); console.log(KMS signer no ${idx} (${kmsSigner.address}) was added to KMSVerifier contract); } else { const kmsSignerAddress = process.env[ADDRESS_KMS_SIGNER_${idx}]; const tx = await kmsVerifier.addSigner(kmsSignerAddress); await tx.wait(); console.log(KMS signer no${idx} (${kmsSignerAddress}) was added to KMSVerifier contract); } } });

# File: ./modules/contracts/tasks/taskTFHE.ts

import fs from "fs"; import { task, types } from "hardhat/config"; import type { TaskArguments } from "hardhat/types"; import path from "path";

task("task:computeACLAddress") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployer = new ethers.Wallet(taskArguments.privateKey).address; const aclAddress = ethers.getCreateAddress({ from: deployer, nonce: 1, // using nonce of 1 for the ACL contract (0 for original implementation, +1 for proxy) }); const envFilePath = path.join(__dirname, "../node_modules/fhevm-core-contracts/addresses/.env.acl"); const content = ACL_CONTRACT_ADDRESS=${aclAddress}\n; try { fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log(ACL address ${aclAddress} written successfully!); } catch (err) { console.error("Failed to write ACL address:", err); }

```
const solidityTemplate = `// SPDX-License-Identifier: BSD-3-Clause-Clea
```

pragma solidity ^0.8.24;

address constant aclAdd = ${aclAddress};\n`;

```
try {
  fs.writeFileSync("./node_modules/fhevm-core-contracts/addresses/ACLAd
    encoding: "utf8",
    flag: "w",
  });
  console.log("./node_modules/fhevm-core-contracts/addresses/ACLAddress
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}
```

});

task("task:computeTFHEExecutorAddress") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployer = new ethers.Wallet(taskArguments.privateKey).address; const execAddress = ethers.getCreateAddress({ from: deployer, nonce: 3, // using nonce of 3 for the TFHEExecutor contract (2 for original implementation, +1 for proxy) }); const envFilePath = path.join(__dirname, "../node_modules/fhevm-core-contracts/addresses/.env.exec"); const content = `TFHE_EXECUTOR_CONTRACT_ADDRESS=${execAddress}\n`; try { fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log(`TFHEExecutor address ${execAddress} written successfully!`); } catch (err) { console.error("Failed to write TFHEExecutor address:", err); }

```
const solidityTemplateCoprocessor = `// SPDX-License-Identifier: BSD-3-
```

pragma solidity ^0.8.24;

address constant tfheExecutorAdd = ${execAddress};\n`;

```
try {
  fs.writeFileSync(
    "./node_modules/fhevm-core-contracts/addresses/TFHEExecutorAddress.
    solidityTemplateCoprocessor,
    { encoding: "utf8", flag: "w" },
  );
  console.log("./node_modules/fhevm-core-contracts/addresses/TFHEExecut
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}
```

});

task("task:computeKMSVerifierAddress") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployer = new ethers.Wallet(taskArguments.privateKey).address; const kmsVerfierAddress = ethers.getCreateAddress({ from: deployer, nonce: 5, // using nonce of 5 for the KMSVerifier contract (4 for original implementation, +1 for proxy) }); const envFilePath = path.join(__dirname, "../node_modules/fhevm-core-contracts/addresses/.env.kmsverifier"); const content = `KMS_VERIFIER_CONTRACT_ADDRESS=${kmsVerfierAddress}\n`; try {

fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log(`KMSVerifier address ${kmsVerfierAddress} written successfully!`); } catch (err) { console.error("Failed to write KMSVerifier address:", err); }

```
const solidityTemplate = `// SPDX-License-Identifier: BSD-3-Clause-Clea
```

pragma solidity ^0.8.24;

address constant kmsVerifierAdd = ${kmsVerfierAddress};\n`;

```
try {
  fs.writeFileSync("./node_modules/fhevm-core-contracts/addresses/KMSVe
    encoding: "utf8",
    flag: "w",
  });
  console.log("./node_modules/fhevm-core-contracts/addresses/KMSVerifie
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}
```

});

task("task:computeInputVerifierAddress") .addParam("privateKey", "The deployer private key") .addOptionalParam( "useAddress", "Use addresses instead of private key env variable for coprocessor", false, types.boolean, ) .setAction(async function (taskArguments: TaskArguments, { ethers }) { // this script also compute the coprocessor address from its private key const deployer = new ethers.Wallet(taskArguments.privateKey).address; const inputVerfierAddress = ethers.getCreateAddress({ from: deployer, nonce: 7, // using nonce of 7 for the InputVerifier contract (6 for original implementation, +1 for proxy) }); const envFilePath = path.join(_dirname, "../node_modules/fhevm-core-contracts/addresses/.env.inputverifier"); const content = `INPUT_VERIFIER_CONTRACT_ADDRESS=${inputVerfierAddress}\n`; try { fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log(`InputVerifier address ${inputVerfierAddress} written successfully!`); } catch (err) { console.error("Failed to write InputVerifier address:", err); }

```
const solidityTemplate = `// SPDX-License-Identifier: BSD-3-Clause-Clea
```

pragma solidity ^0.8.24;

address constant inputVerifierAdd = ${inputVerfierAddress};\n`;

```
try {
  fs.writeFileSync("./node_modules/fhevm-core-contracts/addresses/Input
    encoding: "utf8",
    flag: "w",
  });
  console.log(
    "./node_modules/fhevm-core-contracts/addresses/InputVerifierAddress
  );
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}
let coprocAddress;
if (!taskArguments.useAddress) {
```

```
  coprocAddress = new ethers.Wallet(process.env.PRIVATE_KEY_COPROCESSOR
} else {
  coprocAddress = process.env.ADDRESS_COPROCESSOR_ACCOUNT;
}
const envFilePath2 = path.join(__dirname, "../node_modules/fhevm-core-c
const content2 = `COPROCESSOR_ADDRESS=${coprocAddress}\n`;
try {
  fs.writeFileSync(envFilePath2, content2, { flag: "w" });
  console.log(`Coprocessor address ${coprocAddress} written successfull
} catch (err) {
  console.error("Failed to write InputVerifier address:", err);
}

const solidityTemplate2 = `// SPDX-License-Identifier: BSD-3-Clause-Cle
```

pragma solidity ^0.8.24;

address constant coprocessorAdd = ${coprocAddress};\n`;

```
try {
  fs.writeFileSync("./node_modules/fhevm-core-contracts/addresses/Copro
    encoding: "utf8",
    flag: "w",
  });
  console.log("./node_modules/fhevm-core-contracts/addresses/Coprocesso
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}
```

});

task("task:computeFHEPaymentAddress") .addParam("privateKey", "The deployer private key") .setAction(async function (taskArguments: TaskArguments, { ethers }) { const deployer = new ethers.Wallet(taskArguments.privateKey).address; const fhePaymentAddress = ethers.getCreateAddress({ from: deployer, nonce: 9, // using nonce of 9 for the FHEPayment contract (8 for original implementation, +1 for proxy) }); const envFilePath = path.join(__dirname, "../node_modules/fhevm-core-contracts/addresses/.env.fhepayment"); const content = `FHE_PAYMENT_CONTRACT_ADDRESS=${fhePaymentAddress}\n`; try { fs.writeFileSync(envFilePath, content, { flag: "w" }); console.log(`FHEPayment address ${fhePaymentAddress} written successfully!`); } catch (err) { console.error("Failed to write FHEPayment address:", err); }

```
const solidityTemplate = `// SPDX-License-Identifier: BSD-3-Clause-Clea
```

pragma solidity ^0.8.24;

address constant fhePaymentAdd = ${fhePaymentAddress};\n`;

```
try {
  fs.writeFileSync("./node_modules/fhevm-core-contracts/addresses/FHEPa
    encoding: "utf8",
    flag: "w",
  });
  console.log("./node_modules/fhevm-core-contracts/addresses/FHEPayment
```

```
} catch (error) {
  console.error("Failed to write ./node_modules/fhevm-core-contracts/ad
}

});
```

# File: ./modules/contracts/tasks/accounts.ts

import { task } from "hardhat/config";

task("accounts", "Prints the list of accounts", async (_taskArgs, hre) = > { const accounts = await hre.ethers.getSigners();

for (const account of accounts) { console.info(account.address); } });

# File: ./modules/contracts/tasks/getEthereumAddress.ts

import dotenv from "dotenv"; import { task } from "hardhat/config"; import { HardhatRuntimeEnvironment } from "hardhat/types";

dotenv.config();

const getEthereumAddress = (index: number = 0) = > async (_taskArgs: unknown, hre: HardhatRuntimeEnvironment) = > { const { ethers } = hre; const words = process.env.MNEMONIC!; const mnemonic = ethers.Mnemonic.fromPhrase(words); if (! mnemonic) { throw new Error("No MNEMONIC in .env file"); } const wallet = ethers.HDNodeWallet.fromMnemonic(mnemonic, `m/44'/60'/0'/0`); console.log(wallet.deriveChild(index).address); };

task( "task:getEthereumAddress", "Gets the first address derived from a mnemonic phrase defined in .env", getEthereumAddress(0), );

const accounts = ["Alice", "Bob", "Carol", "Dave", "Eve"];

accounts.forEach((name, index) = > { task( `task:getEthereumAddress${name}`, "Gets the first address derived from a mnemonic phrase defined in .env", getEthereumAddress(index), ); });

# File: ./modules/contracts/test/confidentialERC20/ConfidentialERC20.fixture.ts

import { ethers } from "hardhat";

import type { IConfidentialERC20, TestConfidentialERC20Mintable } from "../../types";

```
import { reencryptEuint64 } from "../reencrypt"; import { Signers } from "../signers"; import { FhevmInstances } from "../types";

export async function deployConfidentialERC20Fixture( signers: Signers, name: string, symbol: string, owner: string, ): Promise { const contractFactory = await ethers.getContractFactory("TestConfidentialERC20Mintable"); const contract = await contractFactory .connect(signers[owner as keyof Signers]) .deploy(name, symbol, signers[owner as keyof Signers].address); await contract.waitForDeployment(); return contract; }

export async function reencryptAllowance( signers: Signers, instances: FhevmInstances, account: string, spender: string, token: IConfidentialERC20, tokenAddress: string, ): Promise { const allowanceHandle = await token.allowance(signers[account as keyof Signers], signers[spender as keyof Signers]); const allowance = await reencryptEuint64(signers, instances, account, allowanceHandle, tokenAddress); return allowance; }

export async function reencryptBalance( signers: Signers, instances: FhevmInstances, account: string, token: IConfidentialERC20, tokenAddress: string, ): Promise { const balanceHandle = await token.balanceOf(signers[account as keyof Signers]); const balance = await reencryptEuint64(signers, instances, account, balanceHandle, tokenAddress); return balance; }
```

# File: ./modules/contracts/test/confidentialERC20/ConfidentialERC20WithErrors.test.ts

```
import { expect } from "chai";

import { createInstances } from "../instance"; import { getSigners, initSigners } from "../signers"; import { reencryptAllowance, reencryptBalance } from "./ConfidentialERC20.fixture"; import { checkErrorCode, deployConfidentialERC20WithErrorsFixture } from "./ConfidentialERC20WithErrors.fixture";

describe("ConfidentialERC20WithErrors", function () { // @dev The placeholder is type(uint256).max --> 2**256 - 1. const PLACEHOLDER = 2n ** 256n - 1n;

before(async function () { await initSigners(2); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20WithErrorsFixture(this.signers, "Naraggara", "NARA", "alice"); this.confidentialERC20Address = await contract.getAddress(); this.confidentialERC20 = contract; this.instances = await createInstances(this.signers); });

it("post-deployment state", async function () { expect(await this.confidentialERC20.totalSupply()).to.equal(0); expect(await this.confidentialERC20.name()).to.equal("Naraggara"); expect(await this.confidentialERC20.symbol()).to.equal("NARA"); expect(await this.confidentialERC20.decimals()).to.be.eq(BigInt(6)); });

it("should mint the contract", async function () { const mintAmount = 1000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await
```

expect(tx).to.emit(this.confidentialERC20, "Mint").withArgs(this.signers.alice, mintAmount);

```
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount);

expect(await this.confidentialERC20.totalSupply()).to.equal(mintAmount)
```

});

it("should transfer tokens between two users", async function () { const mintAmount = 10_000; const transferAmount = 1337; const expectedTransferId = 0n;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedTransferAmount.handles[0], encry

await expect(tx)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, expectedTransferId);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount - transferAmount);

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
```

```
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount);

// Check the error code matches no error
expect(
  await checkErrorCode(
    this.signers,
    this.instances,
    "alice",
    expectedTransferId,
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal("NO_ERROR");

// Check that both the from/to address can read the error code
expect(
  await checkErrorCode(
    this.signers,
    this.instances,
    "bob",
    expectedTransferId,
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal("NO_ERROR");
```

});

it("should not transfer tokens between two users if transfer amount is higher than balance", async function () { // @dev There is no transfer done since the mint amount is smaller than the transfer // amount. const mintAmount = 1000; const transferAmount = 1337; const expectedTransferId = 0n;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

await expect(tx)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, expectedTransferId);

// Decrypt Alice's balance
```

```
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount);

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(0);

// Check that the error code matches if balance is not sufficient
expect(
  await checkErrorCode(
    this.signers,
    this.instances,
    "bob",
    expectedTransferId,
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal("UNSUFFICIENT_BALANCE");
```

});

it("should be able to transferFrom only if allowance is sufficient", async function () { // @dev There is no transfer done since the mint amount is smaller than the transfer // amount. const mintAmount = 10_000; const transferAmount = 1337;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const inputAlice = this.instances.alice.createEncryptedInput(
  this.confidentialERC20Address,
  this.signers.alice.address,
);
inputAlice.add64(transferAmount);
const encryptedAllowanceAmount = await inputAlice.encrypt();

tx = await this.confidentialERC20["approve(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedAllowanceAmount.handles[0],
  encryptedAllowanceAmount.inputProof,
```

```
);

await expect(tx)
  .to.emit(this.confidentialERC20, "Approval")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// @dev The allowance amount is set to be equal to the transfer amount.
expect(
  await reencryptAllowance(
    this.signers,
    this.instances,
    "alice",
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount);

const expectedTransferId1 = 0n;

const inputBob1 = this.instances.bob.createEncryptedInput(this.confiden
inputBob1.add64(transferAmount + 1); // above allowance so next tx shou
const encryptedTransferAmount = await inputBob1.encrypt();

const tx2 = await this.confidentialERC20
  .connect(this.signers.bob)
  [
    "transferFrom(address,address,bytes32,bytes)"
  ](this.signers.alice.address, this.signers.bob.address, encryptedTran

await expect(tx2)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, expectedTransferId1);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount); // check that transfer did not happen, as expec

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
```

```
  ),
).to.equal(0); // check that transfer did not happen, as expected

// Check that the error code matches if approval is not sufficient
expect(
  await checkErrorCode(
    this.signers,
    this.instances,
    "bob",
    expectedTransferId1,
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal("UNSUFFICIENT_APPROVAL");

const expectedTransferId2 = 1n;

const inputBob2 = this.instances.bob.createEncryptedInput(this.confiden
inputBob2.add64(transferAmount); // below allowance so next tx should s
const encryptedTransferAmount2 = await inputBob2.encrypt();

const tx3 = await await this.confidentialERC20
  .connect(this.signers.bob)
  [
    "transferFrom(address,address,bytes32,bytes)"
  ](this.signers.alice.address, this.signers.bob.address, encryptedTran

await expect(tx3)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, expectedTransferId2);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount - transferAmount); // check that transfer did hap

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount); // check that transfer did happen this time
```

```
// Verify Alice's allowance is 0
expect(
  await reencryptAllowance(
    this.signers,
    this.instances,
    "alice",
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(0);

// Check that the error code matches if there is no error
expect(
  await checkErrorCode(
    this.signers,
    this.instances,
    "bob",
    expectedTransferId2,
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal("NO_ERROR");
```

});

it("should not be able to read the allowance if not spender/owner after initialization", async function () { const amount = 10_000;

```
const inputAlice = this.instances.alice.createEncryptedInput(
  this.confidentialERC20Address,
  this.signers.alice.address,
);
inputAlice.add64(amount);
const encryptedAllowanceAmount = await inputAlice.encrypt();

const tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedAllowanceAmount.handles[0], encr

await tx.wait();

const allowanceHandleAlice = await this.confidentialERC20.allowance(thi

const { publicKey: publicKeyCarol, privateKey: privateKeyCarol } = this
const eip712Carol = this.instances.carol.createEIP712(publicKeyCarol, t
const signatureCarol = await this.signers.carol.signTypedData(
  eip712Carol.domain,
  { Reencrypt: eip712Carol.types.Reencrypt },
  eip712Carol.message,
);
```

```
await expect(
  this.instances.bob.reencrypt(
    allowanceHandleAlice,
    privateKeyCarol,
    publicKeyCarol,
    signatureCarol.replace("0x", ""),
    this.confidentialERC20Address,
    this.signers.carol.address,
  ),
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

});
```

it("should not be able to read the balance if not user after initialization", async function () { // Mint is used to initialize the balanceOf(alice) const amount = 10_000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(amount); await tx.wait();

```
const balanceHandleAlice = await this.confidentialERC20.balanceOf(this.

const { publicKey: publicKeyBob, privateKey: privateKeyBob } = this.ins
const eip712Bob = this.instances.bob.createEIP712(publicKeyBob, this.co
const signatureBob = await this.signers.bob.signTypedData(
  eip712Bob.domain,
  { Reencrypt: eip712Bob.types.Reencrypt },
  eip712Bob.message,
);

await expect(
  this.instances.bob.reencrypt(
    balanceHandleAlice,
    privateKeyBob,
    publicKeyBob,
    signatureBob.replace("0x", ""),
    this.confidentialERC20Address,
    this.signers.bob.address,
  ),
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

});
```

it("spender cannot be null address", async function () { const NULL_ADDRESS = "0x0000000000000000000000000000000000000000"; const mintAmount = 100_000; const transferAmount = 50_000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await tx.wait();

```
const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

await expect(
  this.confidentialERC20
    .connect(this.signers.alice)
    [
```

```
      "approve(address,bytes32,bytes)"
    ](NULL_ADDRESS, encryptedTransferAmount.handles[0], encryptedTransf
).to.be.revertedWithCustomError(this.confidentialERC20, "ERC20InvalidSp

});
```

it("receiver cannot be null address", async function () { const NULL_ADDRESS = "0x0000000000000000000000000000000000000000"; const mintAmount = 100_000; const transferAmount = 50_000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await tx.wait();

```
const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

await expect(
  this.confidentialERC20
    .connect(this.signers.alice)
    [
      "transfer(address,bytes32,bytes)"
    ](NULL_ADDRESS, encryptedTransferAmount.handles[0], encryptedTransf
).to.be.revertedWithCustomError(this.confidentialERC20, "ERC20InvalidRe

});
```

it("sender who is not allowed cannot transfer using a handle from another account", async function () { const mintAmount = 100_000; const transferAmount = 50_000; let tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await tx.wait();

```
const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedTransferAmount.handles[0], enc

await tx.wait();

const balanceHandleAlice = await this.confidentialERC20.balanceOf(this.

await expect(
  this.confidentialERC20.connect(this.signers.bob).transfer(this.signer
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA

});
```

it("sender who is not allowed cannot transferFrom using a handle from another account", async function () { const mintAmount = 100_000; const transferAmount = 50_000;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();
```

```
let input = this.instances.alice.createEncryptedInput(this.confidential
input.add64(mintAmount);
const encryptedAllowanceAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedAllowanceAmount.handles[0], en

input = this.instances.carol.createEncryptedInput(this.confidentialERC2
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.carol)
  [
    "transferFrom(address,address,bytes32,bytes)"
  ](this.signers.alice.address, this.signers.carol.address, encryptedTr

const allowanceHandleAlice = await this.confidentialERC20.allowance(
  this.signers.alice.address,
  this.signers.carol.address,
);

await expect(
  this.confidentialERC20
    .connect(this.signers.bob)
    .transferFrom(this.signers.alice.address, this.signers.bob.address,
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA

});
```

it("cannot reencrypt errors if the account is not a participant of the transfer", async function () { const mintAmount = 10_000; const transferAmount = 1337; const expectedTransferId = 0;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedTransferAmount.handles[0], encry

await expect(tx)
  .to.emit(this.confidentialERC20, "Transfer")
```

```
  .withArgs(this.signers.alice, this.signers.bob, expectedTransferId);

const errorCodeHandle = await this.confidentialERC20.getErrorCodeForTra

const { publicKey: publicKeyCarol, privateKey: privateKeyCarol } = this
const eip712Carol = this.instances.carol.createEIP712(publicKeyCarol, t
const signatureCarol = await this.signers.carol.signTypedData(
  eip712Carol.domain,
  { Reencrypt: eip712Carol.types.Reencrypt },
  eip712Carol.message,
);

await expect(
  this.instances.bob.reencrypt(
    errorCodeHandle,
    privateKeyCarol,
    publicKeyCarol,
    signatureCarol.replace("0x", ""),
    this.confidentialERC20Address,
    this.signers.carol.address,
  ),
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

});
```

it("sender who is not allowed cannot approve using a handle from another account", async function () { const amount = 100_000; const input = this.instances.alice.createEncryptedInput(this.confidentialERC20Address, this.signers.alice.address); input.add64(amount); const encryptedAllowanceAmount = await input.encrypt();

```
const tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedAllowanceAmount.handles[0], en

await tx.wait();

const allowanceHandleAlice = await this.confidentialERC20.allowance(
  this.signers.alice.address,
  this.signers.carol.address,
);

await expect(
  this.confidentialERC20.connect(this.signers.bob).approve(this.signers
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA

});
```

it("ConfidentialERC20WithErrorsMintable - only owner can mint", async function () { await expect(this.confidentialERC20.connect(this.signers.bob).mint(1)).to.be.revertedWithCustomErro this.confidentialERC20, "OwnableUnauthorizedAccount", ); }); });

# File: ./modules/contracts/test/confidentialERC20/ConfidentialERC20WithErrors.fixture.ts

import { ethers } from "hardhat";

import type { TestConfidentialERC20WithErrorsMintable } from "../../types"; import { reencryptEuint8 } from "../reencrypt"; import { Signers } from "../signers"; import { FhevmInstances } from "../types";

export async function deployConfidentialERC20WithErrorsFixture( signers: Signers, name: string, symbol: string, owner: string, ): Promise { const contractFactory = await ethers.getContractFactory("TestConfidentialERC20WithErrorsMintable"); const contract = await contractFactory .connect(signers[owner as keyof Signers]) .deploy(name, symbol, signers[owner as keyof Signers].address); await contract.waitForDeployment(); return contract; }

export async function checkErrorCode( signers: Signers, instances: FhevmInstances, account: string, transferId: bigint, token: TestConfidentialERC20WithErrorsMintable, tokenAddress: string, ): Promise { const errorCodeHandle = await token.getErrorCodeForTransferId(transferId); const errorCode = await reencryptEuint8(signers, instances, account, errorCodeHandle, tokenAddress); switch (errorCode) { case 0n: { return "NO_ERROR"; } case 1n: { return "UNSUFFICIENT_BALANCE"; } case 2n: { return "UNSUFFICIENT_APPROVAL"; } default: { throw "Error code is invalid"; } } }

# File: ./modules/contracts/test/confidentialERC20/ConfidentialERC20.test.ts

import { expect } from "chai";

import { createInstances } from "../instance"; import { getSigners, initSigners } from "../signers"; import { deployConfidentialERC20Fixture, reencryptAllowance, reencryptBalance } from "./ConfidentialERC20.fixture";

describe("ConfidentialERC20", function () { // @dev The placeholder is type(uint256).max --> 2**256 - 1. const PLACEHOLDER = 2n ** 256n - 1n;

before(async function () { await initSigners(2); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20Fixture(this.signers, "Naraggara", "NARA", "alice"); this.confidentialERC20Address = await contract.getAddress(); this.confidentialERC20 = contract; this.instances = await createInstances(this.signers); });

it("post-deployment state", async function () { expect(await this.confidentialERC20.totalSupply()).to.equal(0); expect(await

this.confidentialERC20.name()).to.equal("Naraggara"); expect(await this.confidentialERC20.symbol()).to.equal("NARA"); expect(await this.confidentialERC20.decimals()).to.be.eq(BigInt(6)); });

it("should mint the contract", async function () { const mintAmount = 1000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await expect(tx).to.emit(this.confidentialERC20, "Mint").withArgs(this.signers.alice, mintAmount);

```
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount);

expect(await this.confidentialERC20.totalSupply()).to.equal(mintAmount)
```

});

it("should transfer tokens between two users", async function () { const mintAmount = 10_000; const transferAmount = 1337;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedTransferAmount.handles[0], encry

await expect(tx)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount - transferAmount);

// Decrypt Bob's balance
```

```
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount);
```

});

it("should not transfer tokens between two users if transfer amount is higher than balance", async function () { // @dev There is no transfer done since the mint amount is smaller than the transfer // amount. const mintAmount = 1000; const transferAmount = 1337;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

// @dev There is no error-handling in this version of ConfidentialERC20
await expect(tx)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount);

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
```

```
  ),
).to.equal(0);

});
```

it("should be able to transferFrom only if allowance is sufficient", async function () { // @dev There is no transfer done since the mint amount is smaller than the transfer // amount. const mintAmount = 10_000; const transferAmount = 1337;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

const inputAlice = this.instances.alice.createEncryptedInput(
  this.confidentialERC20Address,
  this.signers.alice.address,
);
inputAlice.add64(transferAmount);
const encryptedAllowanceAmount = await inputAlice.encrypt();

tx = await this.confidentialERC20["approve(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedAllowanceAmount.handles[0],
  encryptedAllowanceAmount.inputProof,
);

await expect(tx)
  .to.emit(this.confidentialERC20, "Approval")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// @dev The allowance amount is set to be equal to the transfer amount.
expect(
  await reencryptAllowance(
    this.signers,
    this.instances,
    "alice",
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount);

const bobErc20 = this.confidentialERC20.connect(this.signers.bob);
const inputBob1 = this.instances.bob.createEncryptedInput(this.confiden
inputBob1.add64(transferAmount + 1); // above allowance so next tx shou
const encryptedTransferAmount = await inputBob1.encrypt();

const tx2 = await bobErc20["transferFrom(address,address,bytes32,bytes)
  this.signers.alice.address,
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);
```

```
await expect(tx2)
  .to.emit(this.confidentialERC20, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount); // check that transfer did not happen, as expec

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(0); // check that transfer did not happen, as expected

const inputBob2 = this.instances.bob.createEncryptedInput(this.confiden
inputBob2.add64(transferAmount); // below allowance so next tx should s
const encryptedTransferAmount2 = await inputBob2.encrypt();

const tx3 = await bobErc20["transferFrom(address,address,bytes32,bytes)
  this.signers.alice.address,
  this.signers.bob.address,
  encryptedTransferAmount2.handles[0],
  encryptedTransferAmount2.inputProof,
);
await tx3.wait();

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(mintAmount - transferAmount); // check that transfer did hap

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
```

```
    this.instances,
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(transferAmount); // check that transfer did happen this time

// Verify Alice's allowance is 0
expect(
  await reencryptAllowance(
    this.signers,
    this.instances,
    "alice",
    "bob",
    this.confidentialERC20,
    this.confidentialERC20Address,
  ),
).to.equal(0);
```

});

it("should not be able to read the allowance if not spender/owner after initialization", async function () { const amount = 10_000;

```
const inputAlice = this.instances.alice.createEncryptedInput(
  this.confidentialERC20Address,
  this.signers.alice.address,
);
inputAlice.add64(amount);
const encryptedAllowanceAmount = await inputAlice.encrypt();

const tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedAllowanceAmount.handles[0], encr

await tx.wait();

const allowanceHandleAlice = await this.confidentialERC20.allowance(thi

const { publicKey: publicKeyCarol, privateKey: privateKeyCarol } = this
const eip712Carol = this.instances.carol.createEIP712(publicKeyCarol, t
const signatureCarol = await this.signers.carol.signTypedData(
  eip712Carol.domain,
  { Reencrypt: eip712Carol.types.Reencrypt },
  eip712Carol.message,
);

await expect(
  this.instances.bob.reencrypt(
    allowanceHandleAlice,
    privateKeyCarol,
```

```
    publicKeyCarol,
    signatureCarol.replace("0x", ""),
    this.confidentialERC20Address,
    this.signers.carol.address,
  ),
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

});
```

it("should not be able to read the balance if not user after initialization", async function () { // Mint is used to initialize the balanceOf(alice) const amount = 10_000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(amount); await tx.wait();

```
const balanceHandleAlice = await this.confidentialERC20.balanceOf(this.

const { publicKey: publicKeyBob, privateKey: privateKeyBob } = this.ins
const eip712Bob = this.instances.bob.createEIP712(publicKeyBob, this.co
const signatureBob = await this.signers.bob.signTypedData(
  eip712Bob.domain,
  { Reencrypt: eip712Bob.types.Reencrypt },
  eip712Bob.message,
);

await expect(
  this.instances.bob.reencrypt(
    balanceHandleAlice,
    privateKeyBob,
    publicKeyBob,
    signatureBob.replace("0x", ""),
    this.confidentialERC20Address,
    this.signers.bob.address,
  ),
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

});
```

it("receiver cannot be null address", async function () { const NULL_ADDRESS = "0x0000000000000000000000000000000000000000"; const mintAmount = 100_000; const transferAmount = 50_000; const tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await tx.wait();

```
const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

await expect(
  this.confidentialERC20
    .connect(this.signers.alice)
    [
      "transfer(address,bytes32,bytes)"
    ](NULL_ADDRESS, encryptedTransferAmount.handles[0], encryptedTransf
).to.be.revertedWithCustomError(this.confidentialERC20, "ERC20InvalidRe

});
```

it("sender who is not allowed cannot transfer using a handle from another account", async function () { const mintAmount = 100_000; const transferAmount = 50_000; let tx = await this.confidentialERC20.connect(this.signers.alice).mint(mintAmount); await tx.wait();

```
const input = this.instances.alice.createEncryptedInput(this.confidenti
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedTransferAmount.handles[0], enc

await tx.wait();

const balanceHandleAlice = await this.confidentialERC20.balanceOf(this.

await expect(
  this.confidentialERC20.connect(this.signers.bob).transfer(this.signer
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA
```

});

it("sender who is not allowed cannot transferFrom using a handle from another account", async function () { const mintAmount = 100_000; const transferAmount = 50_000;

```
let tx = await this.confidentialERC20.connect(this.signers.alice).mint(
await tx.wait();

let input = this.instances.alice.createEncryptedInput(this.confidential
input.add64(mintAmount);
const encryptedAllowanceAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedAllowanceAmount.handles[0], en

input = this.instances.carol.createEncryptedInput(this.confidentialERC2
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20
  .connect(this.signers.carol)
  [
    "transferFrom(address,address,bytes32,bytes)"
  ](this.signers.alice.address, this.signers.carol.address, encryptedTr

const allowanceHandleAlice = await this.confidentialERC20.allowance(
  this.signers.alice.address,
  this.signers.carol.address,
);
```

```
await expect(
  this.confidentialERC20
    .connect(this.signers.bob)
    .transferFrom(this.signers.alice.address, this.signers.bob.address,
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA

});
```

it("sender who is not allowed cannot approve using a handle from another account", async function () { const amount = 100_000; const input = this.instances.alice.createEncryptedInput(this.confidentialERC20Address, this.signers.alice.address); input.add64(amount); const encryptedAllowanceAmount = await input.encrypt();

```
const tx = await this.confidentialERC20
  .connect(this.signers.alice)
  [
    "approve(address,bytes32,bytes)"
  ](this.signers.carol.address, encryptedAllowanceAmount.handles[0], en

await tx.wait();

const allowanceHandleAlice = await this.confidentialERC20.allowance(
  this.signers.alice.address,
  this.signers.carol.address,
);

await expect(
  this.confidentialERC20.connect(this.signers.bob).approve(this.signers
).to.be.revertedWithCustomError(this.confidentialERC20, "TFHESenderNotA

});
```

it("ConfidentialERC20Mintable - only owner can mint", async function () { await expect(this.confidentialERC20.connect(this.signers.bob).mint(1)).to.be.revertedWithCustomErro this.confidentialERC20, "OwnableUnauthorizedAccount", ); }); });

# File: ./modules/contracts/test/governance/ConfidentialGovernorAlpha.fixture.ts

import { ethers } from "hardhat";

import type { CompoundTimelock, TestConfidentialGovernorAlpha } from "../../types"; import { reencryptEbool, reencryptEuint64 } from "../reencrypt"; import { Signers, getSigners } from "../signers"; import { FhevmInstances } from "../types";

export async function deployTimelockFixture(admin: string): Promise { const signers = await getSigners(); const timelockFactory = await ethers.getContractFactory("CompoundTimelock"); const timelock = await

timelockFactory.connect(signers.alice).deploy(admin, 60 * 60 * 24 * 2); await timelock.waitForDeployment(); return timelock; }

export async function deployConfidentialGovernorAlphaFixture( signers: Signers, confidentialERC20VotesAddress: string, timelockAddress: string, ): Promise { // @dev We use 5 only for testing purpose. // DO NOT use this value in production. const votingPeriod = 5; // @dev We use 5 minutes for the maximum decryption delay (from the Gateway). const maxDecryptionDelay = 60 * 5; const governorFactory = await ethers.getContractFactory("TestConfidentialGovernorAlpha"); const governor = await governorFactory .connect(signers.alice) .deploy(signers.alice.address, timelockAddress, confidentialERC20VotesAddress, votingPeriod, maxDecryptionDelay); await governor.waitForDeployment(); return governor; }

export async function reencryptVoteReceipt( signers: Signers, instances: FhevmInstances, proposalId: bigint, account: string, governor: TestConfidentialGovernorAlpha, governorAddress: string, ): Promise<[boolean, boolean, bigint]> { const [hasVoted, supportHandle, voteHandle] = await governor.getReceipt( proposalId, signers[account as keyof Signers].address, ); const support = await reencryptEbool(signers, instances, account, supportHandle, governorAddress); const vote = await reencryptEuint64(signers, instances, account, voteHandle, governorAddress);

return [hasVoted, support, vote]; }

# File: ./modules/contracts/test/governance/CompoundTimelock.test.ts

import { expect } from "chai"; import { ethers, network } from "hardhat";

import { getSigners, initSigners } from "../signers"; import { deployTimelockFixture } from "./ConfidentialGovernorAlpha.fixture";

describe("CompoundTimelock", function () { before(async function () { await initSigners(3); this.signers = await getSigners(); });

beforeEach(async function () { this.timelock = await deployTimelockFixture(this.signers.alice.address); });

it("non-timelock account could not call setPendingAdmin", async function () { await expect(this.timelock.setPendingAdmin(this.signers.bob)).to.be.revertedWithCustomError( this.timelock, "SenderIsNotTimelock", ); });

it("non-timelock account could not call setDelay", async function () { await expect(this.timelock.setDelay(60 * 60 * 24 * 3)).to.be.revertedWithCustomError( this.timelock, "SenderIsNotTimelock", ); });

it("setDelay could only be called with a delay between MINIMUM_DELAY and MAXIMUM_DELAY", async function () { const latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 60 * 60 * 24 * 2 + 60; const timeLockAdd = await this.timelock.getAddress(); const callData1 = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 1]); // below MINIMUM_DELAY const callData2 = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"],

[60 * 60 * 24 * 40]); // above MAXIMUM_DELAY const callData3 = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 20]); // OK

```
const tx1 = await this.timelock.queueTransaction(timeLockAdd, 0, "setDe
await tx1.wait();
const tx2 = await this.timelock.queueTransaction(timeLockAdd, 0, "setDe
await tx2.wait();
const tx3 = await this.timelock.queueTransaction(timeLockAdd, 0, "setDe
await tx3.wait();

if (network.name === "hardhat") {
  // hardhat cheatcodes are available only in mocked mode
  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);
  await expect(
    this.timelock.executeTransaction(timeLockAdd, 0, "setDelay(uint256)
  ).to.be.revertedWithCustomError(this.timelock, "ExecutionReverted");
  await expect(
    this.timelock.executeTransaction(timeLockAdd, 0, "setDelay(uint256)
  ).to.be.revertedWithCustomError(this.timelock, "ExecutionReverted");
  await this.timelock.executeTransaction(timeLockAdd, 0, "setDelay(uint
  expect(await this.timelock.delay()).to.equal(60 * 60 * 24 * 20);
}

});
```

it("only admin could cancel queued transaction", async function () { const latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 60 * 60 * 24 * 2 + 60; const timeLockAdd = await this.timelock.getAddress(); const callData = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 20]); // OK

```
let tx = await this.timelock.queueTransaction(timeLockAdd, 0, "setDelay
await tx.wait();

await expect(
  this.timelock.connect(this.signers.bob).cancelTransaction(timeLockAdd
).to.be.revertedWithCustomError(this.timelock, "SenderIsNotAdmin");

tx = await this.timelock.cancelTransaction(timeLockAdd, 0, "setDelay(ui
await tx.wait();

if (network.name === "hardhat") {
  // hardhat cheatcodes are available only in mocked mode
  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);
  await expect(
    this.timelock.executeTransaction(timeLockAdd, 0, "setDelay(uint256)
  ).to.be.revertedWithCustomError(this.timelock, "TransactionNotQueued"
}

});
```

it("only admin could queue transaction, only if it satisfies the delay", async function () { const latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 60 * 60 *

24 * 2 + 60; const expiryTooShort = block!.timestamp + 60 * 60 * 24 * 1 + 60; const timeLockAdd = await this.timelock.getAddress(); const callData = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 20]); // OK

```
// Bob is not the admin.
await expect(
  this.timelock.connect(this.signers.bob).queueTransaction(timeLockAdd,
).to.be.revertedWithCustomError(this.timelock, "SenderIsNotTimelock");

// The expiry is too short.
await expect(
  this.timelock
    .connect(this.signers.alice)
    .queueTransaction(timeLockAdd, 0, "setDelay(uint256)", callData, ex
).to.be.revertedWithCustomError(this.timelock, "TransactionTooEarlyForQ

const tx = await this.timelock
  .connect(this.signers.alice)
  .queueTransaction(timeLockAdd, 0, "setDelay(uint256)", callData, expi
await tx.wait();

});
```

it("only admin could execute transaction, only before grace period", async function () { const latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 60 * 60 * 24 * 2 + 60; const timeLockAdd = await this.timelock.getAddress(); const callData = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 20]); // OK const tx = await this.timelock.queueTransaction(timeLockAdd, 0, "setDelay(uint256)", callData, expiry); await tx.wait();

```
if (network.name === "hardhat") {
  // hardhat cheatcodes are available only in mocked mode
  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);
  await expect(
    this.timelock
      .connect(this.signers.bob)
      .executeTransaction(timeLockAdd, 0, "setDelay(uint256)", callData
  ).to.be.revertedWithCustomError(this.timelock, "SenderIsNotAdmin");

  const idSnapshot = await ethers.provider.send("evm_snapshot");
  await ethers.provider.send("evm_increaseTime", ["0xffffff"]);
  await expect(
    this.timelock.executeTransaction(timeLockAdd, 0, "setDelay(uint256)
  ).to.be.revertedWithCustomError(this.timelock, "TransactionTooLateFor

  await ethers.provider.send("evm_revert", [idSnapshot]); // roll back
  const tx2 = await this.timelock.executeTransaction(timeLockAdd, 0, "s
  await tx2.wait();
  expect(await this.timelock.delay()).to.equal(60 * 60 * 24 * 20);
}

});
```

it("if signature string is empty, calldata must append the signature", async function () { const latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 60 * 60 * 24 * 2 + 60; const timeLockAdd = await this.timelock.getAddress(); const functionSig = ethers.FunctionFragment.getSelector("setDelay", ["uint256"]); const callData = ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [60 * 60 * 24 * 20]); // OK

```
const tx = await this.timelock.queueTransaction(timeLockAdd, 0, "", fun
await tx.wait();

if (network.name === "hardhat") {
  // hardhat cheatcodes are available only in mocked mode
  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);
  const tx2 = await this.timelock.executeTransaction(timeLockAdd, 0, ""
  await tx2.wait();
  expect(await this.timelock.delay()).to.equal(60 * 60 * 24 * 20);
}
```

});

it("could not deploy timelock contract if delay is below 2 days or above 31 days", async function () { const timelockFactory = await ethers.getContractFactory("CompoundTimelock");

```
if (network.name === "hardhat") {
  await expect(
    timelockFactory.connect(this.signers.alice).deploy(this.signers.ali
  ).to.be.revertedWithCustomError(this.timelock, "DelayBelowMinimumDela

  await expect(
    timelockFactory.connect(this.signers.alice).deploy(this.signers.ali
  ).to.be.revertedWithCustomError(this.timelock, "DelayAboveMaximumDela
}
```

}); });

# File: ./modules/contracts/test/governance/DelegateBySig.ts

import { HardhatEthersSigner } from "@nomicfoundation/hardhat-ethers/signers"; import { ethers } from "hardhat"; import { Address } from "hardhat-deploy/types";

import type { ConfidentialERC20Votes } from "../../types";

/** *

- @param _signer Signer from ethers.
- @param _delegatee Delegatee address.
- @param _confidentialERC20Votes ConfidentialERC20Votes token.
- @param _nonce Nonce to sign.
- @param _expiry Expiry timestamp.
- @returns The signature. */ export const delegateBySig = async ( _signer:

HardhatEthersSigner, _delegatee: Address, _confidentialERC20Votes: ConfidentialERC20Votes, _nonce: number, *expiry: number, ): Promise = > { const confidentialERC20VotesAddress* = await *confidentialERC20Votes.getAddress(); const delegatee = delegatee; const nonce = nonce; const expiry* = _expiry; const network = await ethers.provider.getNetwork(); const chainId = network.chainId;

const domain = { name: await *confidentialERC20Votes.name(), version: "1.0", chainId: chainId, verifyingContract: confidentialERC20VotesAddress*, };

// Delegation(address delegatee,uint256 nonce,uint256 expiry)

const types = { Delegation: [ { name: "delegatee", type: "address", }, { name: "nonce", type: "uint256", }, { name: "expiry", type: "uint256", }, ], };

const message = { delegatee: delegatee_, nonce: nonce_, expiry: expiry_, };

const signature = await _signer.signTypedData(domain, types, message); return signature; };

# File: ./modules/contracts/test/governance/ConfidentialERC20Votes.fixture.ts

import { parseUnits } from "ethers"; import { ethers } from "hardhat";

import type { TestConfidentialERC20Votes } from "../../types"; import { reencryptEuint64 } from "../reencrypt"; import { Signers } from "../signers"; import { FhevmInstances } from "../types";

export async function deployConfidentialERC20Votes(signers: Signers): Promise { const contractFactory = await ethers.getContractFactory("TestConfidentialERC20Votes"); const contract = await contractFactory .connect(signers.alice) .deploy(signers.alice.address, "CompoundZama", "CONFIDENTIAL_ERC20_VOTES", "1.0", parseUnits("10000000", 6)); await contract.waitForDeployment(); return contract; }

export async function transferTokensAndDelegate( signers: Signers, instances: FhevmInstances, transferAmount: bigint, account: string, delegate: string, confidentialERC20Votes: TestConfidentialERC20Votes, confidentialERC20VotesAddress: string, ): Promise { const input = instances.alice.createEncryptedInput(confidentialERC20VotesAddress, signers.alice.address); input.add64(transferAmount); const encryptedTransferAmount = await input.encrypt();

let tx = await confidentialERC20Votes .connect(signers.alice) [ "transfer(address,bytes32,bytes)" ](signers[account as keyof Signers], encryptedTransferAmount.handles[0], encryptedTransferAmount.inputProof); await tx.wait();

tx = await confidentialERC20Votes .connect(signers[account as keyof Signers]) .delegate(signers[delegate as keyof Signers].address); await tx.wait(); }

export async function reencryptCurrentVotes( signers: Signers, instances: FhevmInstances, account: string, confidentialERC20Votes: TestConfidentialERC20Votes, confidentialERC20VotesAddress: string, ): Promise { const voteHandle = await

confidentialERC20Votes.getCurrentVotes(signers[account as keyof Signers].address); const vote = await reencryptEuint64(signers, instances, account, voteHandle, confidentialERC20VotesAddress); return vote; }

export async function reencryptPriorVotes( signers: Signers, instances: FhevmInstances, account: string, blockNumber: number, confidentialERC20Votes: TestConfidentialERC20Votes, confidentialERC20VotesAddress: string, ): Promise { const voteHandle = await confidentialERC20Votes.getPriorVotes(signers[account as keyof Signers].address, blockNumber); const vote = await reencryptEuint64(signers, instances, account, voteHandle, confidentialERC20VotesAddress); return vote; }

# File: ./modules/contracts/test/governance/ConfidentialGovernorAlpha.test.ts

import { expect } from "chai"; import { parseUnits } from "ethers"; import { ethers, network } from "hardhat";

import { awaitAllDecryptionResults } from "../asyncDecrypt"; import { createInstances } from "../instance"; import { getSigners, initSigners } from "../signers"; import { mineNBlocks } from "../utils"; import { deployConfidentialERC20Votes, transferTokensAndDelegate } from "./ConfidentialERC20Votes.fixture"; import { deployConfidentialGovernorAlphaFixture, deployTimelockFixture, reencryptVoteReceipt, } from "./ConfidentialGovernorAlpha.fixture";

describe("ConfidentialGovernorAlpha", function () { before(async function () { await initSigners(4); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20Votes(this.signers); this.confidentialERC20Votes = contract; this.confidentialERC20VotesAddress = await contract.getAddress(); this.instances = await createInstances(this.signers);

```
const precomputedGovernorAddress = ethers.getCreateAddress({
  from: this.signers.alice.address,
  nonce: (await this.signers.alice.getNonce()) + 1,
});

const timelock = await deployTimelockFixture(precomputedGovernorAddress
this.timelock = timelock;
this.timelockAddress = await timelock.getAddress();

const governor = await deployConfidentialGovernorAlphaFixture(
  this.signers,
  this.confidentialERC20VotesAddress,
  this.timelockAddress,
);
this.governor = governor;
this.governorAddress = await governor.getAddress();

const tx = await this.confidentialERC20Votes.setGovernor(this.governorA
await tx.wait();
```

```
this.VOTING_DELAY = await this.governor.VOTING_DELAY();
this.VOTING_PERIOD = await this.governor.VOTING_PERIOD();
this.TIMELOCK_DELAY = await this.timelock.delay();

});
```

it("can propose a vote that becomes active if votes match the token threshold", async function () { const transferAmount = parseUnits(String(500_000), 6); const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description";

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

const blockNumber = BigInt(await ethers.provider.getBlockNumber());

const tx = await this.governor
  .connect(this.signers.bob)
  .propose(targets, values, signatures, calldatas, description);

await expect(tx)
  .to.emit(this.governor, "ProposalCreated")
  .withArgs(
    1n,
    this.signers.bob.address,
    targets,
    values,
    signatures,
    calldatas,
    blockNumber + this.VOTING_DELAY + 1n, // @dev We add one since the
    blockNumber + this.VOTING_DELAY + this.VOTING_PERIOD + 1n,
    description,
  );

const proposalId = await this.governor.latestProposalIds(this.signers.b
let proposalInfo = await this.governor.getProposalInfo(proposalId);

// @dev .to.eql is used to compare array elements
expect(proposalInfo.proposer).to.equal(this.signers.bob.address);
expect(proposalInfo.targets).to.eql(targets);
expect(proposalInfo.signatures).to.eql(signatures);
expect(proposalInfo.calldatas).to.eql(calldatas);
// 1 ==> PendingThresholdVerification
```

```
expect(proposalInfo.state).to.equal(1);

await awaitAllDecryptionResults();

proposalInfo = await this.governor.getProposalInfo(proposalId);
// 3 ==> Active
expect(proposalInfo.state).to.equal(3);

});
```

it("anyone can propose a vote but it is rejected if votes are below the token threshold", async function () { const transferAmount = (await this.governor.PROPOSAL_THRESHOLD()) - 1n; const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description";

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

const tx = await this.governor
  .connect(this.signers.bob)
  .propose(targets, values, signatures, calldatas, description);
await tx.wait();

const proposalId = await this.governor.latestProposalIds(this.signers.b
let proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.proposer).to.equal(this.signers.bob.address);

// 1 ==> PendingThresholdVerification
expect(proposalInfo.state).to.equal(1);
await awaitAllDecryptionResults();

proposalInfo = await this.governor.getProposalInfo(proposalId);

await awaitAllDecryptionResults();

// 2 ==> Rejected
expect(proposalInfo.state).to.equal(2);

});
```

it("multiple users can vote and the vote succeeds if forVotes > quorum", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = parseUnits(String(200_000), 6);

```
// Bob and Carol receive 200k tokens and delegate to themselves.
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "carol",
  "carol",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob and Carol vote for
let input = this.instances.bob.createEncryptedInput(this.governorAddres
input.addBool(true);
let encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles

await expect(tx).to.emit(this.governor, "VoteCast").withArgs(
  this.signers.bob,
  1n, // @dev proposalId
);

input = this.instances.carol.createEncryptedInput(this.governorAddress,
input.addBool(true);
encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.carol)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles

await expect(tx).to.emit(this.governor, "VoteCast").withArgs(
  this.signers.carol,
  1n, // @dev proposalId
```

```
);

// Bob/Carol can reeencrypt his/her receipt
let [hasVoted, support, votes] = await reencryptVoteReceipt(
  this.signers,
  this.instances,
  proposalId,
  "bob",
  this.governor,
  this.governorAddress,
);

expect(hasVoted).to.be.eq(true);
expect(support).to.be.eq(true);
expect(votes).to.be.eq(transferAmount);

[hasVoted, support, votes] = await reencryptVoteReceipt(
  this.signers,
  this.instances,
  proposalId,
  "carol",
  this.governor,
  this.governorAddress,
);

expect(hasVoted).to.be.eq(true);
expect(support).to.be.eq(true);
expect(votes).to.be.eq(transferAmount);

// Mine blocks
await mineNBlocks(3);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();

let proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(parseUnits(String(0), 6));
expect(proposalInfo.againstVotes).to.be.eq(parseUnits(String(0), 6));
// 4 ==> Succeeded
expect(proposalInfo.state).to.equal(4);

// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();
proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(transferAmount * 2n);
expect(proposalInfo.againstVotes).to.be.eq(parseUnits(String(0), 6));
// 7 ==> Succeeded
expect(proposalInfo.state).to.equal(7);

const block = await ethers.provider.getBlock(await ethers.provider.getB
let nextBlockTimestamp: BigInt;
```

```
if (block === null) {
  throw "Block is null. Check RPC config.";
} else {
  nextBlockTimestamp = BigInt(block.timestamp) + BigInt(30);
}

await ethers.provider.send("evm_setNextBlockTimestamp", [nextBlockTimes

// QUEUING
tx = await this.governor.queue(proposalId);
await expect(tx)
  .to.emit(this.governor, "ProposalQueued")
  .withArgs(
    1n, // @dev proposalId,
    nextBlockTimestamp + this.TIMELOCK_DELAY,
  );

proposalInfo = await this.governor.getProposalInfo(proposalId);
// 8 ==> Queued
expect(proposalInfo.state).to.equal(8);
const eta = proposalInfo.eta;
expect(eta).to.equal(nextBlockTimestamp + this.TIMELOCK_DELAY);

// EXECUTE
await ethers.provider.send("evm_setNextBlockTimestamp", [eta.toString()
tx = await this.governor.execute(proposalId);
await expect(tx).to.emit(this.governor, "ProposalExecuted").withArgs(
  1n, // @dev proposalId
);

proposalInfo = await this.governor.getProposalInfo(proposalId);
// 10 ==> Executed
expect(proposalInfo.state).to.equal(10);
```

});

it("vote is defeated if forVotes < quorum", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = (await this.governor.QUORUM_VOTES()) - 1n;

```
// Bob receives enough to create a proposal but not enough to match the
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);
```

```
// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Bob reeencrypts his receipt
const [hasVoted, support, votes] = await reencryptVoteReceipt(
  this.signers,
  this.instances,
  proposalId,
  "bob",
  this.governor,
  this.governorAddress,
);

expect(hasVoted).to.be.eq(true);
expect(support).to.be.eq(true);
expect(votes).to.be.eq(transferAmount);

// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);

await tx.wait();
let proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(parseUnits(String(0), 6));
expect(proposalInfo.againstVotes).to.be.eq(parseUnits(String(0), 6));
// 4 ==> Succeeded
expect(proposalInfo.state).to.equal(4);

// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();
proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(transferAmount);
expect(proposalInfo.againstVotes).to.be.eq(parseUnits(String(0), 6));

// 6 ==> Defeated
expect(proposalInfo.state).to.equal(6);
```

});

it("vote is rejected if forVotes < = againstVotes", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmountFor = parseUnits(String(500_000), 6); const transferAmountAgainst = transferAmountFor;

```
// Bob and Carol receive 200k tokens and delegate to themselves.
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmountFor,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmountAgainst,
  "carol",
  "carol",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob votes for but Carol votes against
let input = this.instances.bob.createEncryptedInput(this.governorAddres
input.addBool(true);
let encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

input = this.instances.carol.createEncryptedInput(this.governorAddress,
input.addBool(false);
encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.carol)
```

```
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Bob/Carol can reeencrypt his/her receipt
let [hasVoted, support, votes] = await reencryptVoteReceipt(
  this.signers,
  this.instances,
  proposalId,
  "bob",
  this.governor,
  this.governorAddress,
);

expect(hasVoted).to.be.eq(true);
expect(support).to.be.eq(true);
expect(votes).to.be.eq(transferAmountFor);

[hasVoted, support, votes] = await reencryptVoteReceipt(
  this.signers,
  this.instances,
  proposalId,
  "carol",
  this.governor,
  this.governorAddress,
);

expect(hasVoted).to.be.eq(true);
expect(support).to.be.eq(false);
expect(votes).to.be.eq(transferAmountAgainst);

// Mine blocks
await mineNBlocks(3);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();
let proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(parseUnits(String(0), 6));
expect(proposalInfo.againstVotes).to.be.eq(parseUnits(String(0), 6));
// 4 ==> Succeeded
expect(proposalInfo.state).to.equal(4);

// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();
proposalInfo = await this.governor.getProposalInfo(proposalId);
expect(proposalInfo.forVotes).to.be.eq(transferAmountFor);
expect(proposalInfo.againstVotes).to.be.eq(transferAmountAgainst);
// 6 ==> Defeated
expect(proposalInfo.state).to.equal(6);

});
```

it("only owner could queue setTimelockPendingAdmin then execute it, and then

acceptTimelockAdmin", async function () { const block = await ethers.provider.getBlock(await ethers.provider.getBlockNumber()); let expiry;

```
if (block === null) {
  throw "Block is null. Check RPC config.";
} else {
  expiry = BigInt(block.timestamp) + this.TIMELOCK_DELAY + 1n;
}

const tx = await this.governor.queueSetTimelockPendingAdmin(this.signer
await tx.wait();

if (network.name === "hardhat") {
  // hardhat cheatcodes are available only in mocked mode
  await expect(
    this.governor.executeSetTimelockPendingAdmin(this.signers.bob, expi
  ).to.be.revertedWithCustomError(this.timelock, "TransactionTooEarlyFo

  await expect(
    this.governor.connect(this.signers.carol).queueSetTimelockPendingAd
  ).to.be.revertedWithCustomError(this.governor, "OwnableUnauthorizedAc

  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);

  await expect(
    this.governor.connect(this.signers.carol).executeSetTimelockPending
  ).to.be.revertedWithCustomError(this.governor, "OwnableUnauthorizedAc

  const tx3 = await this.governor.executeSetTimelockPendingAdmin(this.s
  await tx3.wait();

  await expect(this.timelock.acceptAdmin()).to.be.revertedWithCustomErr

  const tx4 = await this.timelock.connect(this.signers.bob).acceptAdmin
  await tx4.wait();

  const latestBlockNumber = await ethers.provider.getBlockNumber();
  const block = await ethers.provider.getBlock(latestBlockNumber);

  let expiry2;
  if (block === null) {
    throw "Block is null. Check RPC config.";
  } else {
    expiry2 = BigInt(block.timestamp) + this.TIMELOCK_DELAY + 1n;
  }

  const timeLockAdd = this.timelockAddress;
  const callData = ethers.AbiCoder.defaultAbiCoder().encode(["address"]
  const tx5 = await this.timelock
    .connect(this.signers.bob)
    .queueTransaction(timeLockAdd, 0, "setPendingAdmin(address)", callD
  await tx5.wait();
  await ethers.provider.send("evm_increaseTime", ["0x2a33c"]);
```

```
  const tx6 = await this.timelock
    .connect(this.signers.bob)
    .executeTransaction(timeLockAdd, 0, "setPendingAdmin(address)", cal

  await tx6.wait();

  await expect(this.governor.connect(this.signers.bob).acceptTimelockAd
    this.governor,
    "OwnableUnauthorizedAccount",
  );

  const tx7 = await this.governor.acceptTimelockAdmin();
  await tx7.wait();
  expect(await this.timelock.admin()).to.eq(this.governorAddress);
}
```

});

it("all arrays of a proposal should be of same length, non null and less than max operations", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description";

```
const invalidTargets = [this.signers.bob.address, this.signers.carol.ad
await expect(
  this.governor.connect(this.signers.alice).propose(invalidTargets, val
).to.be.revertedWithCustomError(this.governor, "LengthsDoNotMatch");

const invalidValues = ["0", "0"];
await expect(
  this.governor.connect(this.signers.alice).propose(targets, invalidVal
).to.be.revertedWithCustomError(this.governor, "LengthsDoNotMatch");

const invalidSignatures = ["getBalanceOf(address)", "getBalanceOf(addre
await expect(
  this.governor.connect(this.signers.alice).propose(targets, values, in
).to.be.revertedWithCustomError(this.governor, "LengthsDoNotMatch");

const invalidCalldatas = [
  ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.b
  ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.b
];

await expect(
  this.governor.connect(this.signers.alice).propose(targets, values, si
).to.be.revertedWithCustomError(this.governor, "LengthsDoNotMatch");

await expect(
  this.governor.connect(this.signers.alice).propose([], [], [], [], des
).to.be.revertedWithCustomError(this.governor, "LengthIsNull");
```

```
await expect(
  this.governor
    .connect(this.signers.alice)
    .propose(
      new Array(11).fill(this.signers.alice),
      new Array(11).fill("0"),
      new Array(11).fill("getBalanceOf(address)"),
      new Array(11).fill(calldatas[0]),
      description,
    ),
).to.be.revertedWithCustomError(this.governor, "LengthAboveMaxOperation

});

it("only gateway can call gateway functions", async function () { await
expect(this.governor.connect(this.signers.bob).callbackInitiateProposal(1,
true)).to.be.reverted; await
expect(this.governor.connect(this.signers.bob).callbackVoteDecryption(1, 10,
10)).to.be.reverted; });

it("only owner can call owner functions", async function () { await
expect(this.governor.connect(this.signers.bob).acceptTimelockAdmin()).to.be.revertedWithCustc
this.governor, "OwnableUnauthorizedAccount", );

await expect(
  this.governor.connect(this.signers.bob).executeSetTimelockPendingAdmi
).to.be.revertedWithCustomError(this.governor, "OwnableUnauthorizedAcco

await expect(
  this.governor.connect(this.signers.bob).queueSetTimelockPendingAdmin(
).to.be.revertedWithCustomError(this.governor, "OwnableUnauthorizedAcco

});

it("only owner or proposer can cancel proposal", async function () { const targets =
[this.signers.bob.address]; const values = ["0"]; const signatures =
["getBalanceOf(address)"]; const calldatas =
[ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const
description = "description"; const transferAmount = await
this.governor.QUORUM_VOTES();

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

const tx = await this.governor
  .connect(this.signers.bob)
  .propose(targets, values, signatures, calldatas, description);
```

```
await tx.wait();

// @dev ProposalId starts at 1.
await expect(this.governor.connect(this.signers.carol).cancel(1)).to.be
  this.governor,
  "OwnableUnauthorizedAccount",
);

});
```

it("proposer cannot make a new proposal while he still has an already pending or active proposal", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

const tx = await this.governor
  .connect(this.signers.bob)
  .propose(targets, values, signatures, calldatas, description);
await tx.wait();

await expect(
  this.governor.connect(this.signers.bob).propose(targets, values, sign
).to.be.revertedWithCustomError(this.governor, "ProposerHasAnotherPropo

});
```

it("cannot queue twice or execute before queuing", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
// Bob receives 400k tokens and delegates to himself.
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
```

```
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob casts a vote
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();

// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();

// QUEUING
// @dev Cannot execute before queuing.
await expect(this.governor.execute(proposalId)).to.be.revertedWithCusto
  this.governor,
  "ProposalStateInvalid",
);

tx = await this.governor.queue(proposalId);
await tx.wait();

// @dev Cannot queue twice.
await expect(this.governor.queue(proposalId)).to.be.revertedWithCustomE

});
```

it("cannot cancel if state is Rejected/Defeated/Executed/Canceled", async function () { let transferAmount = (await this.governor.PROPOSAL_THRESHOLD()) - 1n; const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description";

```
// CANNOT CANCEL IF REJECTED
```

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();
await awaitAllDecryptionResults();

let proposalId = await this.governor.latestProposalIds(this.signers.bob

await expect(this.governor.connect(this.signers.bob).cancel(proposalId)
  this.governor,
  "ProposalStateInvalid",
);

// CANNOT CANCEL IF DEFEATED
transferAmount = (await this.governor.QUORUM_VOTES()) - 1n;

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "carol",
  "carol",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

tx = await this.governor.connect(this.signers.carol).propose(targets, v
await tx.wait();
await awaitAllDecryptionResults();

proposalId = await this.governor.latestProposalIds(this.signers.carol.a

let input = this.instances.carol.createEncryptedInput(this.governorAddr
input.addBool(true);
let encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.carol)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
```

```
await tx.wait();
await awaitAllDecryptionResults();
await expect(this.governor.connect(this.signers.carol).cancel(proposalI
  this.governor,
  "ProposalStateInvalid",
);

// CANNOT CANCEL IF EXECUTED
transferAmount = await this.governor.QUORUM_VOTES();

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "dave",
  "dave",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

tx = await this.governor.connect(this.signers.dave).propose(targets, va
await tx.wait();
await awaitAllDecryptionResults();

proposalId = await this.governor.latestProposalIds(this.signers.dave.ad

input = this.instances.dave.createEncryptedInput(this.governorAddress,
input.addBool(true);
encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.dave)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();
await awaitAllDecryptionResults();

tx = await this.governor.queue(proposalId);
await tx.wait();

const eta = (await this.governor.getProposalInfo(proposalId)).eta;

// EXECUTE
await ethers.provider.send("evm_setNextBlockTimestamp", [eta.toString()
tx = await this.governor.execute(proposalId);
await tx.wait();

await expect(this.governor.connect(this.signers.dave).cancel(proposalId
```

```
  this.governor,
  "ProposalStateInvalid",
);

// CANNOT CANCEL TWICE
tx = await this.governor.connect(this.signers.carol).propose(targets, v
await tx.wait();

proposalId = await this.governor.latestProposalIds(this.signers.carol.a

tx = await this.governor.connect(this.signers.carol).cancel(proposalId)
await tx.wait();
await expect(this.governor.connect(this.signers.carol).cancel(proposalI
  this.governor,
  "ProposalStateInvalid",
);
```

});

it("cancel function clears the timelock if the proposal is queued", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob votes for
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();
```

```
// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();

// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();

// QUEUING
tx = await this.governor.queue(proposalId);
await tx.wait();

// @dev Alice is the governor's owner.
tx = await this.governor.connect(this.signers.alice).cancel(proposalId)
await expect(tx).to.emit(this.governor, "ProposalCanceled").withArgs(
  1n, // @dev proposalId
);

// 5 ==> Canceled
expect((await this.governor.getProposalInfo(proposalId)).state).to.equa

});

it("cannot request vote decryption if state is not Active or if endBlock >= block.number",
async function () { await
expect(this.governor.connect(this.signers.dave).requestVoteDecryption(0)).to.be.revertedWithCu
this.governor, "ProposalStateInvalid", );

const targets = [this.signers.bob.address];
const values = ["0"];
const signatures = ["getBalanceOf(address)"];
const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"]
const description = "description";
const transferAmount = await this.governor.QUORUM_VOTES();

await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
```

```
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob votes for
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Mine blocks but not enough
await mineNBlocks(3);

await expect(
  this.governor.connect(this.signers.dave).requestVoteDecryption(propos
).to.be.revertedWithCustomError(this.governor, "ProposalStateStillActiv

});
```

it("cannot cast a vote if state is not Active or if endBlock > block.number", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

let tx = await this.governor.connect(this.signers.bob).propose(targets,
const proposalId = await this.governor.latestProposalIds(this.signers.b

const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();

await expect(
  this.governor
    .connect(this.signers.bob)
    ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handl
).to.be.revertedWithCustomError(this.governor, "ProposalStateInvalid");

tx = await this.governor.connect(this.signers.bob).cancel(proposalId);
```

```
await tx.wait();

tx = await this.governor.connect(this.signers.bob).propose(targets, val
await tx.wait();
await awaitAllDecryptionResults();

const newProposalId = await this.governor.latestProposalIds(this.signer
// 3 --> Active
expect((await this.governor.getProposalInfo(newProposalId)).state).to.e

// Mine too many blocks so that it becomes too late to cast vote
await mineNBlocks(5);

await expect(
  this.governor
    .connect(this.signers.bob)
    ["castVote(uint256,bytes32,bytes)"](newProposalId, encryptedVote.ha
).to.be.revertedWithCustomError(this.governor, "ProposalStateNotActive"

});
```

it("cannot cast a vote twice", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
// Bob receives 400k tokens and delegates to himself.
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob casts a vote
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
```

```
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

await expect(
  this.governor
    .connect(this.signers.bob)
    ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handl
).to.be.revertedWithCustomError(this.governor, "VoterHasAlreadyVoted");

});
```

it("proposal expires after grace period", async function () { const targets = [this.signers.bob.address]; const values = ["0"]; const signatures = ["getBalanceOf(address)"]; const calldatas = [ethers.AbiCoder.defaultAbiCoder().encode(["address"], [this.signers.bob.address])]; const description = "description"; const transferAmount = await this.governor.QUORUM_VOTES();

```
// Bob receives 400k tokens and delegates to himself.
await transferTokensAndDelegate(
  this.signers,
  this.instances,
  transferAmount,
  "bob",
  "bob",
  this.confidentialERC20Votes,
  this.confidentialERC20VotesAddress,
);

// INITIATE A PROPOSAL
let tx = await this.governor.connect(this.signers.bob).propose(targets,
await tx.wait();

// DECRYPTION FOR THE TOKEN THRESHOLD
await awaitAllDecryptionResults();
const proposalId = await this.governor.latestProposalIds(this.signers.b

// VOTE
// Bob casts a vote
const input = this.instances.bob.createEncryptedInput(this.governorAddr
input.addBool(true);
const encryptedVote = await input.encrypt();
tx = await this.governor
  .connect(this.signers.bob)
  ["castVote(uint256,bytes32,bytes)"](proposalId, encryptedVote.handles
await tx.wait();

// Mine blocks
await mineNBlocks(4);

// REQUEST DECRYPTION
tx = await this.governor.requestVoteDecryption(proposalId);
await tx.wait();
```

```
// POST-DECRYPTION RESULTS
await awaitAllDecryptionResults();

// Proposal is queued
tx = await this.governor.queue(proposalId);
await tx.wait();

let proposalInfo = await this.governor.getProposalInfo(proposalId);
const eta = proposalInfo.eta;
const deadlineExecutionTransaction = eta + (await this.timelock.GRACE_P

await ethers.provider.send("evm_setNextBlockTimestamp", [deadlineExecut
await mineNBlocks(1);

await expect(this.governor.execute(proposalId)).to.be.revertedWithCusto
  this.timelock,
  "TransactionTooLateForExecution",
);

proposalInfo = await this.governor.getProposalInfo(proposalId);
// 9 ==> Expired
expect(proposalInfo.state).to.equal(9);
```

});

it("cannot deploy if maxDecryptionDelay is higher than 1 day (86_400 seconds)", async function () { const maxDecryptionDelay = 86_401; const votingPeriod = 5;

```
const contractFactory = await ethers.getContractFactory("TestConfidenti
await expect(
  contractFactory
    .connect(this.signers.alice)
    .deploy(
      this.signers.alice.address,
      this.timelockAddress,
      this.confidentialERC20VotesAddress,
      votingPeriod,
      maxDecryptionDelay,
    ),
).to.be.revertedWithCustomError(this.governor, "MaxDecryptionDelayTooHi
```

}); });

# File: ./modules/contracts/test/governance/ConfidentialERC20Votes.test.ts

import { expect } from "chai"; import { parseUnits } from "ethers"; import { ethers, network } from "hardhat";

import { reencryptBalance } from "../confidentialERC20/ConfidentialERC20.fixture"; import { createInstances } from "../instance"; import { reencryptEuint64 } from "../reencrypt"; import { getSigners, initSigners } from "../signers"; import { waitNBlocks } from "../utils"; import { deployConfidentialERC20Votes, reencryptCurrentVotes, reencryptPriorVotes, } from "./ConfidentialERC20Votes.fixture"; import { delegateBySig } from "./DelegateBySig";

describe("ConfidentialERC20Votes", function () { // @dev The placeholder is type(uint256).max --> 2**256 - 1. const PLACEHOLDER = 2n ** 256n - 1n; const NULL_ADDRESS = "0x0000000000000000000000000000000000000000";

before(async function () { await initSigners(3); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20Votes(this.signers); this.confidentialERC20VotesAddress = await contract.getAddress(); this.confidentialERC20Votes = contract; this.instances = await createInstances(this.signers); });

it("should transfer tokens", async function () { const transferAmount = parseUnits(String(2_000_000), 6);

```
const input = this.instances.alice.createEncryptedInput(
  this.confidentialERC20VotesAddress,
  this.signers.alice.address,
);
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

const tx = await this.confidentialERC20Votes["transfer(address,bytes32,
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

await expect(tx)
  .to.emit(this.confidentialERC20Votes, "Transfer")
  .withArgs(this.signers.alice, this.signers.bob, PLACEHOLDER);

// Decrypt Alice's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "alice",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(8_000_000), 6));

// Decrypt Bob's balance
expect(
  await reencryptBalance(
    this.signers,
    this.instances,
    "bob",
```

```
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(2_000_000), 6));

});
```

it("can delegate tokens on-chain", async function () { const tx = await this.confidentialERC20Votes.connect(this.signers.alice).delegate(this.signers.bob.address); await expect(tx) .to.emit(this.confidentialERC20Votes, "DelegateChanged") .withArgs(this.signers.alice, NULL_ADDRESS, this.signers.bob);

```
const latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "bob",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(10_000_000), 6));

// Verify the two functions return the same.
expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "bob",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
);

});
```

it("can delegate votes via delegateBySig if signature is valid", async function () { const delegator = this.signers.alice; const delegatee = this.signers.bob; const nonce = 0; let latestBlockNumber = await ethers.provider.getBlockNumber(); const block = await ethers.provider.getBlock(latestBlockNumber); const expiry = block!.timestamp + 100; const signature = await delegateBySig(delegator, delegatee.address, this.confidentialERC20Votes, nonce, expiry);

```
const tx = await this.confidentialERC20Votes
  .connect(this.signers.alice)
  .delegateBySig(delegator, delegatee, nonce, expiry, signature);

await expect(tx)
  .to.emit(this.confidentialERC20Votes, "DelegateChanged")
  .withArgs(this.signers.alice, NULL_ADDRESS, this.signers.bob);

latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "bob",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(10_000_000), 6));

// Verify the two functions return the same.
expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "bob",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
);
```

});

it("cannot delegate votes to self but it gets removed once the tokens are transferred", async function () { let tx = await this.confidentialERC20Votes.connect(this.signers.alice).delegate(this.signers.alice.address); await tx.wait();

```
let latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

expect(
```

```
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "alice",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(10_000_000), 6));

const transferAmount = parseUnits(String(10_000_000), 6);
const input = this.instances.alice.createEncryptedInput(
  this.confidentialERC20VotesAddress,
  this.signers.alice.address,
);
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

tx = await this.confidentialERC20Votes
  .connect(this.signers.alice)
  [
    "transfer(address,bytes32,bytes)"
  ](this.signers.bob.address, encryptedTransferAmount.handles[0], encry

await tx.wait();

latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "alice",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(0);

});
```

it("cannot delegate votes if nonce is invalid", async function () { const delegator = this.signers.alice; const delegatee = this.signers.bob; const nonce = 0; const block = await ethers.provider.getBlock(await ethers.provider.getBlockNumber()); const expiry = block!.timestamp + 100; const signature = await delegateBySig(delegator, delegatee.address, this.confidentialERC20Votes, nonce, expiry);

```
const tx = await this.confidentialERC20Votes
  .connect(this.signers.alice)
  .delegateBySig(delegator, delegatee, nonce, expiry, signature);
await tx.wait();
```

```
// Cannot reuse same nonce when delegating by sig
await expect(
  this.confidentialERC20Votes.delegateBySig(delegator, delegatee, nonce
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "Signature

});
```

it("cannot delegate votes if nonce is invalid due to the delegator incrementing her nonce", async function () { const delegator = this.signers.alice; const delegatee = this.signers.bob; const nonce = 0; const block = await ethers.provider.getBlock(await ethers.provider.getBlockNumber()); const expiry = block!.timestamp + 100; const signature = await delegateBySig(delegator, delegatee.address, this.confidentialERC20Votes, nonce, expiry);

```
const tx = await this.confidentialERC20Votes.connect(delegator).increme
// @dev the newNonce is 1
await expect(tx).to.emit(this.confidentialERC20Votes, "NonceIncremented

// Cannot reuse same nonce when delegating by sig
await expect(
  this.confidentialERC20Votes.delegateBySig(delegator, delegatee, nonce
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "Signature

});
```

it("cannot delegate votes if signer is invalid", async function () { const delegator = this.signers.alice; const delegatee = this.signers.bob; const nonce = 0; const block = await ethers.provider.getBlock(await ethers.provider.getBlockNumber()); const expiry = block!.timestamp + 100;

```
// Signer is not the delegator
const signature = await delegateBySig(
  this.signers.carol,
  delegatee.address,
  this.confidentialERC20Votes,
  nonce,
  expiry,
);
await expect(
  this.confidentialERC20Votes.delegateBySig(delegator, delegatee, nonce
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "Signature

});
```

it("cannot delegate votes if signature has expired", async function () { const delegator = this.signers.alice; const delegatee = this.signers.bob; const nonce = 0; const block = await ethers.provider.getBlock(await ethers.provider.getBlockNumber()); const expiry = block!.timestamp + 100; const signature = await delegateBySig(delegator, delegatee.address, this.confidentialERC20Votes, nonce, expiry);

```
await ethers.provider.send("evm_increaseTime", ["0xffff"]);

await expect(
  this.confidentialERC20Votes.connect(delegatee).delegateBySig(delegato
```

```
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "Signature
```

});

it("cannot request votes if blocktime is equal to current blocktime", async function () { let blockNumber = await ethers.provider.getBlockNumber();

```
await expect(
  this.confidentialERC20Votes.getPriorVotes(this.signers.alice, blockNu
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "BlockNumb

const tx = await this.confidentialERC20Votes.connect(this.signers.alice
await expect(tx).to.emit(this.confidentialERC20Votes, "NewGovernor").wi

blockNumber = await ethers.provider.getBlockNumber();

await expect(
  this.confidentialERC20Votes
    .connect(this.signers.bob)
    .getPriorVotesForGovernor(this.signers.alice, blockNumber + 1),
).to.be.revertedWithCustomError(this.confidentialERC20Votes, "BlockNumb
```

});

it("users can request past votes getPriorVotes", async function () { // Alice transfers 1M tokens to Bob, 1M tokens to Carol, 1M tokens to Dave const transferAmount = parseUnits(String(1_000_000), 6);

```
const input = this.instances.alice.createEncryptedInput(
  this.confidentialERC20VotesAddress,
  this.signers.alice.address,
);
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

let tx = await this.confidentialERC20Votes["transfer(address,bytes32,by
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

await tx.wait();

tx = await this.confidentialERC20Votes["transfer(address,bytes32,bytes)
  this.signers.carol.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

await tx.wait();

tx = await this.confidentialERC20Votes["transfer(address,bytes32,bytes)
  this.signers.dave.address,
  encryptedTransferAmount.handles[0],
```

```
  encryptedTransferAmount.inputProof,
);

await tx.wait();

tx = await this.confidentialERC20Votes.connect(this.signers.bob).delega
await tx.wait();

const firstCheckPointBlockNumber = await ethers.provider.getBlockNumber
await waitNBlocks(1);

tx = await this.confidentialERC20Votes.connect(this.signers.carol).dele
await tx.wait();

const secondCheckPointBlockNumber = await ethers.provider.getBlockNumbe
await waitNBlocks(1);

expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "dave",
    firstCheckPointBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.be.equal(parseUnits(String(1_000_000), 6));

expect(
  await reencryptPriorVotes(
    this.signers,
    this.instances,
    "dave",
    secondCheckPointBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.be.equal(parseUnits(String(2_000_000), 6));

});
```

it("only governor contract can call getPriorVotes", async function () { await expect( this.confidentialERC20Votes.getPriorVotesForGovernor("0xE359a77c3bFE58792FB167D05720e 0), ).to.be.revertedWithCustomError(this.confidentialERC20Votes, "GovernorInvalid"); });

it("only owner can set governor contract", async function () { const newAllowedContract = "0x9d3e06a2952dc49EDCc73e41C76645797fC53967"; await expect(this.confidentialERC20Votes.connect(this.signers.bob).setGovernor(newAllowedContract) .to.be.revertedWithCustomError(this.confidentialERC20Votes, "OwnableUnauthorizedAccount") .withArgs(this.signers.bob.address); });

it("getCurrentVote/getPriorVotes without any vote cannot be decrypted", async function () { // 1. If no checkpoint exists using getCurrentVotes let currentVoteHandle = await this.confidentialERC20Votes .connect(this.signers.bob)

.getCurrentVotes(this.signers.bob.address); expect(currentVoteHandle).to.be.eq(0n);

```
await expect(
  reencryptEuint64(this.signers, this.instances, "bob", currentVoteHand
).to.be.rejectedWith("Handle is not initialized");

// 2. If no checkpoint exists using getPriorVotes
let latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

currentVoteHandle = await this.confidentialERC20Votes
  .connect(this.signers.bob)
  .getPriorVotes(this.signers.bob.address, latestBlockNumber);

// It is an encrypted constant that is not reencryptable by Bob.
expect(currentVoteHandle).not.to.be.eq(0n);

await expect(
  reencryptEuint64(this.signers, this.instances, "bob", currentVoteHand
).to.be.rejectedWith("Invalid contract address.");

// 3. If a checkpoint exists using getPriorVotes but block.number < blo
latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

const tx = await this.confidentialERC20Votes.connect(this.signers.alice
await tx.wait();

currentVoteHandle = await this.confidentialERC20Votes
  .connect(this.signers.bob)
  .getPriorVotes(this.signers.bob.address, latestBlockNumber);

// It is an encrypted constant that is not reencryptable by Bob.
expect(currentVoteHandle).not.to.be.eq(0n);

await expect(
  reencryptEuint64(this.signers, this.instances, "bob", currentVoteHand
).to.be.rejectedWith("Invalid contract address.");
```

});

it("can do multiple checkpoints and access the values when needed", async function () { let i = 0;

```
const blockNumbers = [];

const thisBlockNumber = await ethers.provider.getBlockNumber();

while (i < 20) {
  let tx = await this.confidentialERC20Votes.connect(this.signers.alice
  await tx.wait();
  blockNumbers.push(await ethers.provider.getBlockNumber());

  tx = await this.confidentialERC20Votes.connect(this.signers.alice).de
```

```
  await tx.wait();
  blockNumbers.push(await ethers.provider.getBlockNumber());
  i++;
}

await waitNBlocks(1);

// There are 40 checkpoints for Alice and 39 checkpoints for Carol
expect(await this.confidentialERC20Votes.numCheckpoints(this.signers.al
expect(await this.confidentialERC20Votes.numCheckpoints(this.signers.ca

i = 0;

const startWithAlice = thisBlockNumber % 2 === 1;

while (i < 40) {
  if (blockNumbers[i] % 2 === 0) {
    expect(
      await reencryptPriorVotes(
        this.signers,
        this.instances,
        startWithAlice ? "alice" : "carol",
        blockNumbers[i],
        this.confidentialERC20Votes,
        this.confidentialERC20VotesAddress,
      ),
    ).to.be.eq(parseUnits(String(10_000_000), 6));
  } else {
    expect(
      await reencryptPriorVotes(
        this.signers,
        this.instances,
        startWithAlice ? "carol" : "alice",
        blockNumbers[i],
        this.confidentialERC20Votes,
        this.confidentialERC20VotesAddress,
      ),
    ).to.be.eq(parseUnits(String(10_000_000), 6));
  }
  i++;
}

});
```

it("governor address can access votes for any account", async function () { // Bob becomes the governor address. let tx = await this.confidentialERC20Votes.connect(this.signers.alice).setGovernor(this.signers.bob.address); await expect(tx).to.emit(this.confidentialERC20Votes, "NewGovernor").withArgs(this.signers.bob);

```
// Alice delegates her votes to Carol.
tx = await this.confidentialERC20Votes.connect(this.signers.alice).dele
await tx.wait();
```

```
const latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);
await waitNBlocks(1);

// Bob, the governor address, gets the prior votes of Carol.
// @dev It is not possible to catch the return value since it is not a
// ConfidentialGovernorAlpha.test.ts contains tests that use this funct
await this.confidentialERC20Votes
  .connect(this.signers.bob)
  .getPriorVotesForGovernor(this.signers.carol.address, latestBlockNumb

});
```

it("different voters can delegate to same delegatee", async function () { const transferAmount = parseUnits(String(2_000_000), 6);

```
const input = this.instances.alice.createEncryptedInput(
  this.confidentialERC20VotesAddress,
  this.signers.alice.address,
);
input.add64(transferAmount);
const encryptedTransferAmount = await input.encrypt();

let tx = await this.confidentialERC20Votes["transfer(address,bytes32,by
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);

await tx.wait();

tx = await this.confidentialERC20Votes.connect(this.signers.alice).dele
await tx.wait();

tx = await this.confidentialERC20Votes.connect(this.signers.bob).delega
await tx.wait();

const latestBlockNumber = await ethers.provider.getBlockNumber();
await waitNBlocks(1);

expect(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "carol",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(parseUnits(String(10_000_000), 6));

expect(
  await reencryptPriorVotes(
```

```
    this.signers,
    this.instances,
    "carol",
    latestBlockNumber,
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "carol",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
);

});
```

// TODO: fix issue with mining it.skip("number of checkpoints is incremented once per block, even when written multiple times in same block", async function () { await network.provider.send("evm_setAutomine", [false]); await network.provider.send("evm_setIntervalMining", [0]);

```
// do two checkpoints in same block
const tx1 = this.confidentialERC20Votes.connect(this.signers.alice).del
const tx2 = this.confidentialERC20Votes.connect(this.signers.alice).del

await network.provider.send("evm_mine");
await network.provider.send("evm_setAutomine", [true]);
await Promise.all([tx1, tx2]);

expect(await this.confidentialERC20Votes.numCheckpoints(this.signers.al
expect(await this.confidentialERC20Votes.numCheckpoints(this.signers.bo
expect(await this.confidentialERC20Votes.numCheckpoints(this.signers.ca

expect(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "bob",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
  ),
).to.equal(0);

expect(
  await reencryptCurrentVotes(
    this.signers,
    this.instances,
    "carol",
    this.confidentialERC20Votes,
    this.confidentialERC20VotesAddress,
```

```
  ),
).to.equal(parseUnits(String(10_000_000), 6));

}); });
```

# File: ./modules/contracts/test/utils/EncryptedErrors.fixture.ts

import { ethers } from "hardhat";

import type { TestEncryptedErrors } from "../../types"; import { Signers } from "../signers";

export async function deployEncryptedErrors(signers: Signers, numberErrors: number): Promise { const contractFactory = await ethers.getContractFactory("TestEncryptedErrors"); const contract = await contractFactory.connect(signers.alice).deploy(numberErrors); await contract.waitForDeployment(); return contract; }

# File: ./modules/contracts/test/utils/EncryptedErrors.test.ts

import { expect } from "chai"; import { ethers } from "hardhat";

import { createInstances } from "../instance"; import { reencryptEuint8 } from "../reencrypt"; import { getSigners, initSigners } from "../signers"; import { deployEncryptedErrors } from "./EncryptedErrors.fixture";

describe("EncryptedErrors", function () { const NO_ERROR_CODE = 0n;

before(async function () { await initSigners(3); this.signers = await getSigners(); this.instances = await createInstances(this.signers); });

beforeEach(async function () { this.numberErrors = 3; const contract = await deployEncryptedErrors(this.signers, this.numberErrors); this.encryptedErrorsAddress = await contract.getAddress(); this.encryptedErrors = contract; });

it("post-deployment", async function () { expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("0")); expect(await this.encryptedErrors.errorGetNumCodesDefined()).to.be.eq(BigInt("3"));

```
for (let i = 0; i < 3; i++) {
  const handle = await this.encryptedErrors.connect(this.signers.alice)
  expect(
    await reencryptEuint8(this.signers, this.instances, "alice", handle
  ).to.be.eq(i);
}
```

});

it("errorDefineIf --> true", async function () { // True --> errorId=0 has errorCode=2 const condition = true; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorDefineIf(encryptedData.handles[0], encryptedData.inputProof, ta

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  targetErrorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorDefineIf --> false", async function () { // False --> errorId = 1 has errorCode = 0
const condition = false; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorDefineIf(encryptedData.handles[0], encryptedData.inputProof, ta

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  NO_ERROR_CODE,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorDefineIfNot --> true", async function () { // True --> errorId = 0 has errorCode = 0
const condition = true; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorDefineIfNot(encryptedData.handles[0], encryptedData.inputProof,

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  NO_ERROR_CODE,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorDefineIf --> false", async function () { // False --> errorId = 1 has errorCode = 2
const condition = false; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorDefineIfNot(encryptedData.handles[0], encryptedData.inputProof,

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  targetErrorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorChangeIf --> true --> change error code", async function () { // True --> change errorCode const condition = true; const errorCode = 1; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).add8(errorCode).en

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorChangeIf(encryptedData.handles[0], encryptedData.handles[1], en

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  targetErrorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorChangeIf --> false --> no change for error code", async function () { // False --> no change in errorCode const condition = false; const errorCode = 1; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).add8(errorCode).en

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorChangeIf(encryptedData.handles[0], encryptedData.handles[1], en

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  errorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorChangeIfNot --> true --> no change for error code", async function () { // True --> no change errorCode const condition = true; const errorCode = 1; const targetErrorCode =

2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).add8(errorCode).en

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorChangeIfNot(encryptedData.handles[0], encryptedData.handles[1],

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  errorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("errorChangeIfNot --> false --> change error code", async function () { // False --> change in errorCode const condition = false; const errorCode = 1; const targetErrorCode = 2;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).add8(errorCode).en

await this.encryptedErrors
  .connect(this.signers.alice)
  .errorChangeIfNot(encryptedData.handles[0], encryptedData.handles[1],

const handle = await this.encryptedErrors.connect(this.signers.alice).e
expect(await reencryptEuint8(this.signers, this.instances, "alice", han
  targetErrorCode,
);
expect(await this.encryptedErrors.errorGetCounter()).to.be.eq(BigInt("1

});
```

it("cannot deploy if totalNumberErrorCodes_ == 0", async function () { const numberErrors = 0; const contractFactory = await ethers.getContractFactory("TestEncryptedErrors"); await expect(contractFactory.connect(this.signers.alice).deploy(numberErrors)).to.be.revertedWithCus this.encryptedErrors, "TotalNumberErrorCodesEqualToZero", ); });

it("cannot define errors if indexCode is greater or equal than totalNumberErrorCodes", async function () { const condition = true; const targetErrorCode = (await this.encryptedErrors.errorGetNumCodesDefined()) + 1n;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorDefineIf(encryptedData.handles[0], encryptedData.inputProof,
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali
```

```
await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorDefineIfNot(encryptedData.handles[0], encryptedData.inputProo
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali

});
```

it("cannot define errors if indexCode is 0 or equal", async function () { const condition = true; const targetErrorCode = 0;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).encrypt();

await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorDefineIf(encryptedData.handles[0], encryptedData.inputProof,
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexIsNull

await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorDefineIfNot(encryptedData.handles[0], encryptedData.inputProo
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexIsNull

});
```

it("cannot change errors if indexCode is greater or equal than totalNumberErrorCodes", async function () { const condition = true; const errorCode = 1; const targetErrorCode = (await this.encryptedErrors.errorGetNumCodesDefined()) + 1n;

```
const input = this.instances.alice.createEncryptedInput(this.encryptedE
const encryptedData = await input.addBool(condition).add8(errorCode).en

await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorChangeIf(encryptedData.handles[0], encryptedData.handles[1],
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali

await expect(
  this.encryptedErrors
    .connect(this.signers.alice)
    .errorChangeIfNot(
      encryptedData.handles[0],
      encryptedData.handles[1],
      encryptedData.inputProof,
      targetErrorCode,
    ),
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali

});
```

it("cannot call _errorGetCodeDefinition if indexCode is greater or equal than totalNumberErrorCodes", async function () { const indexCodeDefinition = await this.encryptedErrors.errorGetNumCodesDefined();

```
await expect(
  this.encryptedErrors.connect(this.signers.alice).errorGetCodeDefiniti
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali
```

});

it("cannot call _errorGetCodeEmitted if errorId is greater than errorCounter", async function () { const errorCounter = await this.encryptedErrors.errorGetCounter();

```
await expect(
  this.encryptedErrors.connect(this.signers.alice).errorGetCodeEmitted(
).to.be.revertedWithCustomError(this.encryptedErrors, "ErrorIndexInvali
```

}); });