File: ./modules/hardhat/README.md

Hardhat Template [image] [image] [image] [image]

A Hardhat-based template for developing Solidity smart contracts, with sensible defaults.

- Hardhat: compile, run and test smart contracts
- TypeChain: generate TypeScript bindings for smart contracts
- Ethers: renowned Ethereum library and wallet implementation
- Solhint: code linter
- Solcover: code coverage
- Prettier Plugin Solidity: code formatter

Getting Started

Click the `Use this template` button at the top of the page to create a new repository with this repo as the initial state.

Features

This template builds upon the frameworks and libraries mentioned above, so for details about their specific features, please consult their respective documentations.

For example, for Hardhat, you can refer to the Hardhat Tutorial and the Hardhat Docs. You might be in particular interested in reading the Testing Contracts section.

Sensible Defaults

This template comes with sensible default configurations in the following files:

```
├── .editorconfig
├── .eslintignore
├── .eslintrc.yml
├── .gitignore
├── .prettierignore
├── .prettierrc.yml
├── .solcover.js
├── .solhint.json
└── hardhat.config.ts
```

VSCode Integration

This template is IDE agnostic, but for the best user experience, you may want to use it in VSCode alongside Nomic Foundation's Solidity extension.

GitHub Actions

This template comes with GitHub Actions pre-configured. Your contracts will be linted and tested on every push and pull request made to the `main` branch.

Note though that to make this work, you must use your `INFURA_API_KEY` and your `MNEMONIC` as GitHub secrets.

You can edit the CI script in .github/workflows/ci.yml.

Usage

Pre Requisites

Install pnpm

Before being able to run any command, you need to create a `.env` file and set a BIP-39 compatible mnemonic as the `MNEMONIC` environment variable. You can follow the example in `.env.example` or start with the following command:

```
cp .env.example .env
```

If you don't already have a mnemonic, you can use this website to generate one. An alternative, if you have foundry installed is to use the `cast wallet new-mnemonic` command.

Then, install all needed dependencies - please ***make sure to use Node v20*** or more recent:

```
pnpm install
```

Compile

Compile the smart contracts with Hardhat:

```
pnpm compile
```

TypeChain

Compile the smart contracts and generate TypeChain bindings:

```
pnpm typechain
```

Test

Run the tests with Hardhat - this will run the tests on a local hardhat node in mocked mode (i.e the FHE operations and decryptions will be simulated by default):

```
pnpm test
```

Lint Solidity

Lint the Solidity code:

```
pnpm lint:sol
```

Lint TypeScript

Lint the TypeScript code:

```
pnpm lint:ts
```

Clean

Delete the smart contract artifacts, the coverage reports and the Hardhat cache:

```
pnpm clean
```

Mocked mode

The mocked mode allows faster testing and the ability to analyze coverage of the tests. In this mocked version, encrypted types are not really encrypted, and the tests are run on the original version of the EVM, on a local hardhat network instance. To run the tests in mocked mode, you can use directly the following command:

```
pnpm test
```

You can still use all the usual specific hardhat network methods, such as `evm_snapshot`, `evm_mine`, `evm_increaseTime`, etc, which are very helpful in a testing context. Another useful hardhat feature, is the console.log function which can be used in fhevm smart contracts in mocked mode as well.

To analyze the coverage of the tests (in mocked mode necessarily, as this cannot be done on the real fhEVM node), you can use this command :

```
pnpm coverage
```

Then open the file `coverage/index.html`. You can see there which line or branch for each contract which has been covered or missed by your test suite. This allows increased security by pointing out missing branches not covered yet by the current tests.

Finally, a new fhevm-specific feature is available in mocked mode: the `debug.decrypt[XX]` functions, which can decrypt directly any encrypted value. Please refer to the utils.ts file for the corresponding documentation.

> [!Note] Due to intrinsic limitations of the original EVM, the mocked version differs in rare edge cases from the real fhEVM, the main difference is the gas consumption for the FHE operations (native gas is around 20% underestimated in mocked mode). This means that before deploying to production, developers should still run the tests with the original fhEVM node, as a final check - i.e in non-mocked mode (see next section).

Non-mocked mode - Sepolia

To run your test on a real fhevm node, you can use the coprocessor deployed on the Sepolia test network. To do this, ensure you are using a valid value `SEPOLIA_RPC_URL` in your `.env` file. You can get free Sepolia RPC URLs by creating an account on services such as Infura or Alchemy. Then you can use the following command:

```
npx hardhat test [PATH_TO_YOUR_TEST] --network sepolia
```

The `--network sepolia` flag will make your test run on a real fhevm coprocessor. Obviously, for the same tests to pass on Sepolia, contrarily to mocked mode, you are not allowed to use any hardhat node specific method, and neither use any of the `debug.decrypt[XX]` functions.

> [!Note] For this test to succeed, first ensure you set your own private `MNEMONIC` variable in the `.env` file and then ensure you have funded your test accounts on Sepolia. For example you can use the following command to get the corresponding private keys associated with the first `5` accounts derived from the mnemonic:

```
npx hardhat get-accounts --num-accounts 5
```

This will let you add them to the Metamask app, to easily fund them from your personal wallet.

If you don't own already Sepolia test tokens, you can for example use a free faucet such as https://sepolia-faucet.pk910.de/.

Another faster way to test the coprocessor on Sepolia is to simply run the following command:

```
pnpm deploy-sepolia
```

This would automatically deploy an instance of the `MyConfidentialERC20` example contract on Sepolia. You could then use this other command to mint some amount of confidential tokens:

```
pnpm mint-sepolia
```

Etherscan verification

If you are using a real instance of the fhEVM, you can verify your deployed contracts on the Etherscan explorer. You first need to set the `ETHERSCAN_API_KEY` variable in the `.env` file to a valid value. You can get such an API key for free by creating an account on the Etherscan website.

Then, simply use the `verify-deployed` hardhat task, via this command:

```
npx hardhat verify-deployed --address [ADDRESS_CONTRACT_TO_VERIFY] --co
```

As a concrete example, to verify the deployed `MyConfidentialERC20` from previous section, you can use:

```
npx hardhat verify-deployed --address [CONFIDENTIAL_ERC20_ADDRESS] --co
```

Note that you should replace the address placeholder `[CONFIDENTIAL_ERC20_ADDRESS]` by the concrete address that is logged when you run the `pnpm deploy-sepolia` deployment script.

Syntax Highlighting

If you use VSCode, you can get Solidity syntax highlighting with the hardhat-solidity extension.

License

This project is licensed under MIT.

File: ./modules/hardhat/contracts/TestAsyncDecrypt.sol

// SPDX-License-Identifier: BSD-3-Clause-Clear

pragma solidity ^0.8.24;

import "fhevm/lib/TFHE.sol"; import "fhevm/config/ZamaFHEVMConfig.sol"; import "fhevm/config/ZamaGatewayConfig.sol"; import "fhevm/gateway/GatewayCaller.sol";

/// @notice Contract for testing asynchronous decryption using the Gateway /* solhint-disable max-states-count*/ /* solhint-disable var-name-mixedcase*/ contract TestAsyncDecrypt is SepoliaZamaFHEVMConfig, SepoliaZamaGatewayConfig, GatewayCaller { /// @dev Encrypted state variables ebool xBool; euint4 xUint4; euint8 xUint8; euint16 xUint16; euint32 xUint32; euint64 xUint64; euint64 xUint64_2; euint64 xUint64_3; euint128 xUint128; eaddress xAddress; eaddress xAddress2; euint256 xUint256;

```
/// @dev Decrypted state variables
bool public yBool;
uint8 public yUint4;
uint8 public yUint8;
uint16 public yUint16;
uint32 public yUint32;
uint64 public yUint64;
uint64 public yUint64_2;
uint64 public yUint64_3;
uint128 public yUint128;
address public yAddress;
address public yAddress2;
uint256 public yUint256;
bytes public yBytes64;
bytes public yBytes128;
bytes public yBytes256;

/// @dev Tracks the latest decryption request ID
uint256 public latestRequestID;

/// @notice Constructor to initialize the contract and set up encrypted
constructor() {
    /// @dev Initialize encrypted variables with sample values
    xBool = TFHE.asEbool(true);
    TFHE.allowThis(xBool);
    xUint4 = TFHE.asEuint4(4);
    TFHE.allowThis(xUint4);
    xUint8 = TFHE.asEuint8(42);
    TFHE.allowThis(xUint8);
    xUint16 = TFHE.asEuint16(16);
```

```
    TFHE.allowThis(xUint16);
    xUint32 = TFHE.asEuint32(32);
    TFHE.allowThis(xUint32);
    xUint64 = TFHE.asEuint64(18446744073709551600);
    TFHE.allowThis(xUint64);
    xUint64_2 = TFHE.asEuint64(76575465786);
    TFHE.allowThis(xUint64_2);
    xUint64_3 = TFHE.asEuint64(6400);
    TFHE.allowThis(xUint64_3);
    xUint128 = TFHE.asEuint128(126765060022822940149670320544 3);
    TFHE.allowThis(xUint128);
    xUint256 = TFHE.asEuint256(27606985387162255149739023449108101809 80
    TFHE.allowThis(xUint256);
    xAddress = TFHE.asEaddress(0x8ba1f109551bD432803012645Ac136ddd64DBA
    TFHE.allowThis(xAddress);
    xAddress2 = TFHE.asEaddress(0xf48b8840387ba3809DAE990c930F3b4766A86
    TFHE.allowThis(xAddress2);
}

/// @notice Function to request decryption with an excessive delay (sho
function requestBoolAboveDelay() public {
    /// @dev This should revert due to the excessive delay
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xBool);
    Gateway.requestDecryption(cts, this.callbackBool.selector, 0, block
}

/// @notice Request decryption of a boolean value
function requestBool() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xBool);
    /// @dev Request decryption with a 100-second deadline and non-trus
    Gateway.requestDecryption(cts, this.callbackBool.selector, 0, block
}

/// @notice Request decryption of a boolean value in trustless mode
function requestBoolTrustless() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xBool);
    /// @dev Request decryption with a 100-second deadline and trustles
    uint256 requestID = Gateway.requestDecryption(
        cts,
        this.callbackBoolTrustless.selector,
        0,
        block.timestamp + 100,
        true
    );
    latestRequestID = requestID;
    /// @dev Save the requested handles for later verification
    saveRequestedHandles(requestID, cts);
}

/// @notice Callback function for non-trustless boolean decryption
```

```
function callbackBool(uint256, bool decryptedInput) public onlyGateway
    yBool = decryptedInput;
    return yBool;
}

/// @notice Callback function for trustless boolean decryption
function callbackBoolTrustless(
    uint256 requestID,
    bool decryptedInput,
    bytes[] memory signatures
) public onlyGateway returns (bool) {
    /// @dev Verify that the requestID matches the latest request
    require(latestRequestID == requestID, "wrong requestID passed by Ga
    /// @dev Load the previously saved handles for verification
    uint256[] memory requestedHandles = loadRequestedHandles(latestRequ
    /// @dev Verify the signatures provided by the KMS (Key Management
    bool isKMSVerified = Gateway.verifySignatures(requestedHandles, sig
    require(isKMSVerified, "KMS did not verify this decryption result")
    /// @dev If verification passes, store the decrypted value
    yBool = decryptedInput;
    return yBool;
}

/// @notice Request decryption of a 4-bit unsigned integer
function requestUint4() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint4);
    Gateway.requestDecryption(cts, this.callbackUint4.selector, 0, bloc
}

/// @notice Callback function for 4-bit unsigned integer decryption
/// @param decryptedInput The decrypted 4-bit unsigned integer
/// @return The decrypted value
function callbackUint4(uint256, uint8 decryptedInput) public onlyGatewa
    yUint4 = decryptedInput;
    return decryptedInput;
}

/// @notice Request decryption of an 8-bit unsigned integer
function requestUint8() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint8);
    Gateway.requestDecryption(cts, this.callbackUint8.selector, 0, bloc
}

/// @notice Callback function for 8-bit unsigned integer decryption
/// @param decryptedInput The decrypted 8-bit unsigned integer
/// @return The decrypted value
function callbackUint8(uint256, uint8 decryptedInput) public onlyGatewa
    yUint8 = decryptedInput;
    return decryptedInput;
}
```

```
/// @notice Request decryption of a 16-bit unsigned integer
function requestUint16() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint16);
    Gateway.requestDecryption(cts, this.callbackUint16.selector, 0, blo
}

/// @notice Callback function for 16-bit unsigned integer decryption
/// @param decryptedInput The decrypted 16-bit unsigned integer
/// @return The decrypted value
function callbackUint16(uint256, uint16 decryptedInput) public onlyGate
    yUint16 = decryptedInput;
    return decryptedInput;
}

/// @notice Request decryption of a 32-bit unsigned integer with additi
/// @param input1 First additional input
/// @param input2 Second additional input
function requestUint32(uint32 input1, uint32 input2) public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint32);
    uint256 requestID = Gateway.requestDecryption(
        cts,
        this.callbackUint32.selector,
        0,
        block.timestamp + 100,
        false
    );
    addParamsUint256(requestID, input1);
    addParamsUint256(requestID, input2);
}

/// @notice Callback function for 32-bit unsigned integer decryption
/// @param requestID The ID of the decryption request
/// @param decryptedInput The decrypted 32-bit unsigned integer
/// @return The result of the computation
function callbackUint32(uint256 requestID, uint32 decryptedInput) publi
    uint256[] memory params = getParamsUint256(requestID);
    unchecked {
        uint32 result = uint32(params[0]) + uint32(params[1]) + decrypt
        yUint32 = result;
        return result;
    }
}

/// @notice Request decryption of a 64-bit unsigned integer
function requestUint64() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint64);
    Gateway.requestDecryption(cts, this.callbackUint64.selector, 0, blo
}

/// @notice Request decryption of a non-trivial 64-bit unsigned integer
```

```
/// @param inputHandle The input handle for the encrypted value
/// @param inputProof The input proof for the encrypted value
function requestUint64NonTrivial(einput inputHandle, bytes calldata inp
    euint64 inputNonTrivial = TFHE.asEuint64(inputHandle, inputProof);
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackUint64.selector, 0, blo
}

/// @notice Callback function for 64-bit unsigned integer decryption
/// @param decryptedInput The decrypted 64-bit unsigned integer
/// @return The decrypted value
function callbackUint64(uint256, uint64 decryptedInput) public onlyGate
    yUint64 = decryptedInput;
    return decryptedInput;
}

function requestUint128() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint128);
    Gateway.requestDecryption(cts, this.callbackUint128.selector, 0, bl
}

function requestUint128NonTrivial(einput inputHandle, bytes calldata in
    euint128 inputNonTrivial = TFHE.asEuint128(inputHandle, inputProof)
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackUint128.selector, 0, bl
}

function callbackUint128(uint256, uint128 decryptedInput) public onlyGa
    yUint128 = decryptedInput;
    return decryptedInput;
}

function requestUint256() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xUint256);
    Gateway.requestDecryption(cts, this.callbackUint256.selector, 0, bl
}

function requestUint256NonTrivial(einput inputHandle, bytes calldata in
    euint256 inputNonTrivial = TFHE.asEuint256(inputHandle, inputProof)
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackUint256.selector, 0, bl
}

function callbackUint256(uint256, uint256 decryptedInput) public onlyGa
    yUint256 = decryptedInput;
    return decryptedInput;
}
```

```
function requestEbytes64NonTrivial(einput inputHandle, bytes calldata i
    ebytes64 inputNonTrivial = TFHE.asEbytes64(inputHandle, inputProof)
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes64.selector, 0, bl
}

function requestEbytes64Trivial(bytes calldata value) public {
    ebytes64 inputTrivial = TFHE.asEbytes64(TFHE.padToBytes64(value));
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes64.selector, 0, bl
}

function callbackBytes64(uint256, bytes calldata decryptedInput) public
    yBytes64 = decryptedInput;
    return decryptedInput;
}

function requestEbytes128NonTrivial(einput inputHandle, bytes calldata
    ebytes128 inputNonTrivial = TFHE.asEbytes128(inputHandle, inputProo
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes128.selector, 0, b
}

function requestEbytes128Trivial(bytes calldata value) public {
    ebytes128 inputTrivial = TFHE.asEbytes128(TFHE.padToBytes128(value)
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes128.selector, 0, b
}

function callbackBytes128(uint256, bytes calldata decryptedInput) publi
    yBytes128 = decryptedInput;
    return decryptedInput;
}

function requestEbytes256Trivial(bytes calldata value) public {
    ebytes256 inputTrivial = TFHE.asEbytes256(TFHE.padToBytes256(value)
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes256.selector, 0, b
}

function requestEbytes256NonTrivial(einput inputHandle, bytes calldata
    ebytes256 inputNonTrivial = TFHE.asEbytes256(inputHandle, inputProo
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    Gateway.requestDecryption(cts, this.callbackBytes256.selector, 0, b
}

/// @notice Callback function for 256-bit encrypted bytes decryption
```

```
/// @param decryptedInput The decrypted 256-bit bytes
/// @return The decrypted value
function callbackBytes256(uint256, bytes calldata decryptedInput) publi
    yBytes256 = decryptedInput;
    return decryptedInput;
}

/// @notice Request decryption of an encrypted address
function requestAddress() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xAddress);
    Gateway.requestDecryption(cts, this.callbackAddress.selector, 0, bl
}

/// @notice Request decryption of multiple encrypted addresses
function requestSeveralAddresses() public {
    uint256[] memory cts = new uint256[](2);
    cts[0] = Gateway.toUint256(xAddress);
    cts[1] = Gateway.toUint256(xAddress2);
    Gateway.requestDecryption(cts, this.callbackAddresses.selector, 0,
}

/// @notice Callback function for multiple address decryption
/// @param decryptedInput1 The first decrypted address
/// @param decryptedInput2 The second decrypted address
/// @return The first decrypted address
function callbackAddresses(
    uint256 /*requestID*/,
    address decryptedInput1,
    address decryptedInput2
) public onlyGateway returns (address) {
    yAddress = decryptedInput1;
    yAddress2 = decryptedInput2;
    return decryptedInput1;
}

/// @notice Callback function for address decryption
/// @param decryptedInput The decrypted address
/// @return The decrypted address
function callbackAddress(uint256, address decryptedInput) public onlyGa
    yAddress = decryptedInput;
    return decryptedInput;
}

/// @notice Request decryption of multiple encrypted data types
/// @dev This function demonstrates how to request decryption for vario
/// @param input1 First additional input parameter for the callback fun
/// @param input2 Second additional input parameter for the callback fu
function requestMixed(uint32 input1, uint32 input2) public {
    uint256[] memory cts = new uint256[](10);
    cts[0] = Gateway.toUint256(xBool);
    cts[1] = Gateway.toUint256(xBool);
    cts[2] = Gateway.toUint256(xUint4);
```

```
        cts[3] = Gateway.toUint256(xUint8);
        cts[4] = Gateway.toUint256(xUint16);
        cts[5] = Gateway.toUint256(xUint32);
        cts[6] = Gateway.toUint256(xUint64);
        cts[7] = Gateway.toUint256(xUint64);
        cts[8] = Gateway.toUint256(xUint64);
        cts[9] = Gateway.toUint256(xAddress);
        uint256 requestID = Gateway.requestDecryption(
            cts,
            this.callbackMixed.selector,
            0,
            block.timestamp + 100,
            false
        );
        addParamsUint256(requestID, input1);
        addParamsUint256(requestID, input2);
    }

    /// @notice Callback function for mixed data type decryption
    /// @dev Processes the decrypted values and performs some basic checks
    /// @param requestID The ID of the decryption request
    /// @param decBool_1 First decrypted boolean
    /// @param decBool_2 Second decrypted boolean
    /// @param decUint4 Decrypted 4-bit unsigned integer
    /// @param decUint8 Decrypted 8-bit unsigned integer
    /// @param decUint16 Decrypted 16-bit unsigned integer
    /// @param decUint32 Decrypted 32-bit unsigned integer
    /// @param decUint64_1 First decrypted 64-bit unsigned integer
    /// @param decUint64_2 Second decrypted 64-bit unsigned integer
    /// @param decUint64_3 Third decrypted 64-bit unsigned integer
    /// @param decAddress Decrypted address
    /// @return The decrypted 4-bit unsigned integer
    function callbackMixed(
        uint256 requestID,
        bool decBool_1,
        bool decBool_2,
        uint8 decUint4,
        uint8 decUint8,
        uint16 decUint16,
        uint32 decUint32,
        uint64 decUint64_1,
        uint64 decUint64_2,
        uint64 decUint64_3,
        address decAddress
    ) public onlyGateway returns (uint8) {
        yBool = decBool_1;
        require(decBool_1 == decBool_2, "Wrong decryption");
        yUint4 = decUint4;
        yUint8 = decUint8;
        yUint16 = decUint16;
        uint256[] memory params = getParamsUint256(requestID);
        unchecked {
            uint32 result = uint32(params[0]) + uint32(params[1]) + decUint
```

```
        yUint32 = result;
    }
    yUint64 = decUint64_1;
    require(decUint64_1 == decUint64_2 && decUint64_2 == decUint64_3, "
    yAddress = decAddress;
    return yUint4;
}

/// @notice Request decryption of mixed data types including 256-bit en
/// @dev Demonstrates how to include encrypted bytes256 in a mixed decr
/// @param inputHandle The encrypted input handle for the bytes256
/// @param inputProof The proof for the encrypted bytes256
function requestMixedBytes256(einput inputHandle, bytes calldata inputP
    ebytes256 xBytes256 = TFHE.asEbytes256(inputHandle, inputProof);
    uint256[] memory cts = new uint256[](4);
    cts[0] = Gateway.toUint256(xBool);
    cts[1] = Gateway.toUint256(xAddress);
    cts[2] = Gateway.toUint256(xBytes256);
    ebytes64 input64Bytes = TFHE.asEbytes64(TFHE.padToBytes64(hex"aaff4
    cts[3] = Gateway.toUint256(input64Bytes);
    Gateway.requestDecryption(cts, this.callbackMixedBytes256.selector,
}

/// @notice Callback function for mixed data type decryption including
/// @dev Processes and stores the decrypted values
/// @param decBool Decrypted boolean
/// @param decAddress Decrypted address
/// @param bytesRes Decrypted 256-bit bytes
function callbackMixedBytes256(
    uint256,
    bool decBool,
    address decAddress,
    bytes memory bytesRes,
    bytes memory bytesRes2
) public onlyGateway {
    yBool = decBool;
    yAddress = decAddress;
    yBytes256 = bytesRes;
    yBytes64 = bytesRes2;
}

/// @notice Request trustless decryption of non-trivial 256-bit encrypt
/// @dev Demonstrates how to request trustless decryption for complex e
/// @param inputHandle The encrypted input handle for the bytes256
/// @param inputProof The proof for the encrypted bytes256
function requestEbytes256NonTrivialTrustless(einput inputHandle, bytes
    ebytes256 inputNonTrivial = TFHE.asEbytes256(inputHandle, inputProo
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(inputNonTrivial);
    uint256 requestID = Gateway.requestDecryption(
        cts,
        this.callbackBytes256Trustless.selector,
        0,
```

```
        block.timestamp + 100,
        true
    );
    latestRequestID = requestID;
    saveRequestedHandles(requestID, cts);
}

/// @notice Callback function for trustless decryption of 256-bit encry
/// @dev Verifies the decryption result using KMS signatures
/// @param requestID The ID of the decryption request
/// @param decryptedInput The decrypted bytes256 value
/// @param signatures The signatures from the KMS for verification
/// @return The decrypted bytes256 value
function callbackBytes256Trustless(
    uint256 requestID,
    bytes calldata decryptedInput,
    bytes[] memory signatures
) public onlyGateway returns (bytes memory) {
    require(latestRequestID == requestID, "wrong requestID passed by Ga
    uint256[] memory requestedHandles = loadRequestedHandles(latestRequ
    bool isKMSVerified = Gateway.verifySignatures(requestedHandles, sig
    require(isKMSVerified, "KMS did not verify this decryption result")
    yBytes256 = decryptedInput;
    return decryptedInput;
}

/// @notice Request trustless decryption of mixed data types including
/// @dev Demonstrates how to request trustless decryption for multiple
/// @param inputHandle The encrypted input handle for the bytes256
/// @param inputProof The proof for the encrypted bytes256
function requestMixedBytes256Trustless(einput inputHandle, bytes callda
    ebytes256 xBytes256 = TFHE.asEbytes256(inputHandle, inputProof);
    uint256[] memory cts = new uint256[](3);
    cts[0] = Gateway.toUint256(xBool);
    cts[1] = Gateway.toUint256(xBytes256);
    cts[2] = Gateway.toUint256(xAddress);
    uint256 requestID = Gateway.requestDecryption(
        cts,
        this.callbackMixedBytes256Trustless.selector,
        0,
        block.timestamp + 100,
        true
    );
    latestRequestID = requestID;
    saveRequestedHandles(requestID, cts);
}

/// @notice Callback function for trustless decryption of mixed data ty
/// @dev Verifies and processes the decrypted values
/// @param requestID The ID of the decryption request
/// @param decBool Decrypted boolean
/// @param bytesRes Decrypted 256-bit bytes
/// @param decAddress Decrypted address
```

```
/// @param signatures The signatures from the KMS for verification
function callbackMixedBytes256Trustless(
    uint256 requestID,
    bool decBool,
    bytes memory bytesRes,
    address decAddress,
    bytes[] memory signatures
) public onlyGateway {
    require(latestRequestID == requestID, "wrong requestID passed by Ga
    uint256[] memory requestedHandles = loadRequestedHandles(latestRequ
    bool isKMSVerified = Gateway.verifySignatures(requestedHandles, sig
    require(isKMSVerified, "KMS did not verify this decryption result")
    yBool = decBool;
    yAddress = decAddress;
    yBytes256 = bytesRes;
}

}
```

File: ./modules/hardhat/contracts/MyConfidentialERC20.sol

// SPDX-License-Identifier: BSD-3-Clause-Clear

pragma solidity ^0.8.24;

import "fhevm/lib/TFHE.sol"; import "fhevm/config/ZamaFHEVMConfig.sol"; import "fhevm-contracts/contracts/token/ERC20/extensions/ConfidentialERC20Mintable.sol";

/// @notice This contract implements an encrypted ERC20-like token with confidential balances using Zama's FHE library. /// @dev It supports typical ERC20 functionality such as transferring tokens, minting, and setting allowances, /// @dev but uses encrypted data types. contract MyConfidentialERC20 is SepoliaZamaFHEVMConfig, ConfidentialERC20Mintable { /// @notice Constructor to initialize the token's name and symbol, and set up the owner /// @param name_ The name of the token /// @param symbol_ The symbol of the token constructor(string memory name_, string memory symbol_) ConfidentialERC20Mintable(name_, symbol_, msg.sender) {} }

File: ./modules/hardhat/deploy/deploy.ts

import { DeployFunction } from "hardhat-deploy/types"; import { HardhatRuntimeEnvironment } from "hardhat/types";

const func: DeployFunction = async function (hre: HardhatRuntimeEnvironment) { const { deployer } = await hre.getNamedAccounts(); const { deploy } = hre.deployments;

const deployed = await deploy("MyConfidentialERC20", { from: deployer, args: ["Naraggara", "NARA"], log: true, });

console.log(`MyConfidentialERC20 contract: `, deployed.address); }; export default func; func.id = "deploy_confidentialERC20"; // id required to prevent reexecution func.tags

= ["MyConfidentialERC20"];

File: ./modules/hardhat/tasks/mintMyConfidentialERC20.ts

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

import { MyConfidentialERC20 } from "../types";

task("mint") .addParam("amount", "Tokens to mint") .setAction(async function (taskArguments: TaskArguments, hre: HardhatRuntimeEnvironment) { const { ethers, deployments } = hre; const ERC20 = await deployments.get("MyConfidentialERC20"); const signers = await ethers.getSigners(); const erc20 = (await ethers.getContractAt("MyConfidentialERC20", ERC20.address)) as MyConfidentialERC20; const tx = await erc20.connect(signers[0]).mint(+taskArguments.amount); const rcpt = await tx.wait(); console.info("Mint tx hash: ", rcpt!.hash); console.info("Mint done: ", taskArguments.amount, "tokens were minted succesfully"); });

File: ./modules/hardhat/tasks/etherscanVerify.ts

import { task } from "hardhat/config";

task("verify-deployed", "Verifies an already deployed contract on Etherscan") .addParam("address", "The contract's address") .addParam("contract", "Full contract path (e.g., 'contracts/MyConfidentialERC20.sol:MyConfidentialERC20.sol')") .addParam("args", "Constructor arguments as comma-separated values", "") .setAction(async (taskArgs, hre) => { if (hre.network.name === "hardhat") { throw Error("Etherscan verification is not possilbe in mocked mode, choose another network"); } const { address, contract, args } = taskArgs;

```
console.info("\nStarting verification for deployed contract...");
console.info("Contract:", contract);
console.info("Address:", address);

try {
  // Parse constructor arguments
  const constructorArgs = args
    ? args.split(",").map((arg) => {
        const trimmed = arg.trim();
        // Try to parse as JSON
        try {
          return JSON.parse(trimmed);
        } catch {
          // If it's a number
          if (!isNaN(trimmed)) {
            return Number(trimmed);
          }
          // If it's a boolean
```

```
          if (trimmed.toLowerCase() === "true") return true;
          if (trimmed.toLowerCase() === "false") return false;
          // Otherwise return as string
          return trimmed;
        }
      })
    : [];

  console.info("Constructor Arguments:", constructorArgs);

  // Prepare verification arguments
  const verificationArgs = {
    address: address,
    contract: contract,
    constructorArguments: constructorArgs,
  };

  console.info("\nSubmitting verification request...");
  await hre.run("verify:verify", verificationArgs);

  console.info("\n✓ Contract verification completed successfully!");
} catch (error) {
  if (error.message.includes("Already Verified")) {
    console.info("\n✓ Contract is already verified!");
  } else {
    console.error("\n✗ Verification failed:", error.message);
    console.info("\nTo verify your contract, use the following format:"
    console.info("\nnpx hardhat verify-deployed \\");
    console.info("  --address", address, "\\");
    console.info("  --contract", contract, "\\");
    console.info('  --args "arg1,arg2,arg3" \\');
    console.info("  --network <network>");
  }
}

});
```

File: ./modules/hardhat/tasks/accounts.ts

import { task, types } from "hardhat/config";

import { ACCOUNT_NAMES } from "../test/constants";

task("get-accounts", "Prints the list of accounts") .addParam("numAccounts", "Number of accounts to return (1-10)", 3, types.int) .setAction(async ({ numAccounts }, hre) = > { // Validate input if (numAccounts < 1 || numAccounts > 10) { throw new Error("Number of accounts must be between 1 and 10"); }

```
// Get signers from hardhat
const signers = await hre.ethers.getSigners();
const accounts = [];
const { mnemonic } = hre.network.config.accounts;
```

```
// Get details for specified number of accounts
for (let i = 0; i < numAccounts && i < signers.length; i++) {
  const signer = signers[i];
  const address = await signer.getAddress();
  const phrase = hre.ethers.Mnemonic.fromPhrase(mnemonic);
  const pathDeployer = "m/44'/60'/0'/0/" + i;
  const privateKey = hre.ethers.HDNodeWallet.fromMnemonic(phrase, pathD

  accounts.push({
    index: i,
    privateKey: privateKey,
    address: address,
  });
}
console.info("\nAccount Details:");
console.info("================");
accounts.forEach(({ index, privateKey, address }) => {
  console.info(`\nAccount ${index}: (${ACCOUNT_NAMES[index]})`);
  console.info(`Address:     ${address}`);
  console.info(`Private Key: ${privateKey}`);
});

});
```

File: ./modules/hardhat/test/utils.ts

import { toBufferBE } from "bigint-buffer"; import { ethers, network } from "hardhat";

import { awaitCoprocessor, getClearText } from "./coprocessorUtils";

export const mineNBlocks = async (n: number) => { for (let index = 0; index < n; index ++) { await ethers.provider.send("evm_mine"); } };

export const bigIntToBytes64 = (value: bigint) => { return new Uint8Array(toBufferBE(value, 64)); };

export const bigIntToBytes128 = (value: bigint) => { return new Uint8Array(toBufferBE(value, 128)); };

export const bigIntToBytes256 = (value: bigint) => { return new Uint8Array(toBufferBE(value, 256)); };

export const waitNBlocks = async (Nblocks: number) => { const currentBlock = await ethers.provider.getBlockNumber(); if (network.name === "hardhat") { await produceDummyTransactions(Nblocks); } await waitForBlock(currentBlock + Nblocks); };

export const produceDummyTransactions = async (blockCount: number) => { let counter = blockCount; while (counter >= 0) { counter--; const [signer] = await ethers.getSigners(); const nullAddress = "0x0000000000000000000000000000000000000000"; const tx = { to: nullAddress, value: 0n, }; const receipt = await signer.sendTransaction(tx); await receipt.wait(); } };

const waitForBlock = (blockNumber: bigint | number) => { return new Promise((resolve, reject) => { const waitBlock = async (currentBlock: number) => { if (blockNumber <= BigInt(currentBlock)) { await ethers.provider.off("block", waitBlock); resolve(blockNumber); } }; ethers.provider.on("block", waitBlock).catch((err) => { reject(err); }); }); };

const EBOOL_T = 0; const EUINT4_T = 1; const EUINT8_T = 2; const EUINT16_T = 3; const EUINT32_T = 4; const EUINT64_T = 5; const EUINT128_T = 6; const EUINT160_T = 7; // @dev It is the one for eaddresses. const EUINT256_T = 8; const EBYTES64_T = 9; const EBYTES128_T = 10; const EBYTES256_T = 11;

function verifyType(handle: bigint, expectedType: number) { if (handle === 0n) { throw "Handle is not initialized"; } if (handle.toString(2).length > 256) { throw "Handle is not a bytes32"; } const typeCt = handle >> 8n; if (Number(typeCt % 256n) !== expectedType) { throw "Wrong encrypted type for the handle"; } }

export const debug = { /**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} a handle to decrypt
- @returns {bool} */ decryptBool: async (handle: bigint): Promise => { verifyType(handle, EBOOL_T); if (network.name === "hardhat") { await awaitCoprocessor(); return (await getClearText(handle)) === "1"; } else { throw Error("The debug.decryptBool function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt4: async (handle: bigint): Promise => { verifyType(handle, EUINT4_T); if (network.name === "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt4 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-

- @param {bigint} a handle to decrypt
- @returns {bigint} */ decrypt8: async (handle: bigint): Promise = > { verifyType(handle, EUINT8_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt8 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt16: async (handle: bigint): Promise = > { verifyType(handle, EUINT16_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt16 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt32: async (handle: bigint): Promise = > { verifyType(handle, EUINT32_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt32 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt64: async (handle: bigint): Promise = > { verifyType(handle, EUINT64_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt64 function can only be called in mocked mode"); } },

/**

- @debug

- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt128: async (handle: bigint): Promise = > { verifyType(handle, EUINT128_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt128 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decrypt256: async (handle: bigint): Promise = > { verifyType(handle, EUINT256_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return BigInt(await getClearText(handle)); } else { throw Error("The debug.decrypt256 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {string} */ decryptAddress: async (handle: bigint): Promise = > { verifyType(handle, EUINT160_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); const bigintAdd = BigInt(await getClearText(handle)); const handleStr = "0x" + bigintAdd.toString(16).padStart(40, "0"); return handleStr; } else { throw Error("The debug.decryptAddress function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} a handle to decrypt

- @returns {bigint} */ decryptEbytes64: async (handle: bigint): Promise = > { verifyType(handle, EBYTES64_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return ethers.toBeHex(await getClearText(handle), 64); } else { throw Error("The debug.decryptEbytes64 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decryptEbytes128: async (handle: bigint): Promise = > { verifyType(handle, EBYTES128_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return ethers.toBeHex(await getClearText(handle), 128); } else { throw Error("The debug.decryptEbytes128 function can only be called in mocked mode"); } },

/**

- @debug
- This function is intended for debugging purposes only.
- It cannot be used in production code, since it requires the FHE private key for decryption.
- In production, decryption is only possible via an asyncronous on-chain call to the Gateway.
-
- @param {bigint} handle to decrypt
- @returns {bigint} */ decryptEbytes256: async (handle: bigint): Promise = > { verifyType(handle, EBYTES256_T); if (network.name = = = "hardhat") { await awaitCoprocessor(); return ethers.toBeHex(await getClearText(handle), 256); } else { throw Error("The debug.decryptEbytes256 function can only be called in mocked mode"); } }, };

File: ./modules/hardhat/test/mockedSetup.ts

import { ZeroAddress } from "ethers"; import { HardhatRuntimeEnvironment } from "hardhat/types";

import { ACL_ADDRESS, FHEPAYMENT_ADDRESS, GATEWAYCONTRACT_ADDRESS, INPUTVERIFIER_ADDRESS, KMSVERIFIER_ADDRESS, PRIVATE_KEY_KMS_SIGNER, TFHEEXECUTOR_ADDRESS, } from "./constants";

const OneAddress = "0x0000000000000000000000000000000000000001";

export async function setCodeMocked(hre: HardhatRuntimeEnvironment) { const aclArtifact = require("fhevm-core-contracts/artifacts/contracts/ACL.sol/ACL.json"); const aclBytecode

= aclArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [ACL_ADDRESS, aclBytecode]); const execArtifact = require("fhevm-core-contracts/artifacts/contracts/TFHEExecutorWithEvents.sol/TFHEExecutorWithEvents.json"); const execBytecode = execArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [TFHEEXECUTOR_ADDRESS, execBytecode]); const kmsArtifact = require("fhevm-core-contracts/artifacts/contracts/KMSVerifier.sol/KMSVerifier.json"); const kmsBytecode = kmsArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [KMSVERIFIER_ADDRESS, kmsBytecode]); const inputArtifact = require("fhevm-core-contracts/artifacts/contracts/InputVerifier.coprocessor.sol/InputVerifier.json"); const inputBytecode = inputArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [INPUTVERIFIER_ADDRESS, inputBytecode]); const fhepaymentArtifact = require("fhevm-core-contracts/artifacts/contracts/FHEPayment.sol/FHEPayment.json"); const fhepaymentBytecode = fhepaymentArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [FHEPAYMENT_ADDRESS, fhepaymentBytecode]); const gatewayArtifact = require("fhevm-core-contracts/artifacts/gateway/GatewayContract.sol/GatewayContract.json"); const gatewayBytecode = gatewayArtifact.deployedBytecode; await hre.network.provider.send("hardhat_setCode", [GATEWAYCONTRACT_ADDRESS, gatewayBytecode]); const zero = await impersonateAddress(hre, ZeroAddress, hre.ethers.parseEther("100")); const one = await impersonateAddress(hre, OneAddress, hre.ethers.parseEther("100")); const kmsSigner = new hre.ethers.Wallet(PRIVATE_KEY_KMS_SIGNER); const kms = await hre.ethers.getContractAt(kmsArtifact.abi, KMSVERIFIER_ADDRESS); await kms.connect(zero).initialize(OneAddress); await kms.connect(one).addSigner(kmsSigner); const input = await hre.ethers.getContractAt(inputArtifact.abi, INPUTVERIFIER_ADDRESS); await input.connect(zero).initialize(OneAddress); const gateway = await hre.ethers.getContractAt(gatewayArtifact.abi, GATEWAYCONTRACT_ADDRESS); await gateway.connect(zero).addRelayer(ZeroAddress); }

export async function impersonateAddress(hre: HardhatRuntimeEnvironment, address: string, amount: bigint) { // for mocked mode await hre.network.provider.request({ method: "hardhat_impersonateAccount", params: [address], }); await hre.network.provider.send("hardhat_setBalance", [address, hre.ethers.toBeHex(amount)]); const impersonatedSigner = await hre.ethers.getSigner(address); return impersonatedSigner; }

File: ./modules/hardhat/test/asyncDecrypt.ts

import { Wallet, ZeroAddress } from "ethers"; import gatewayArtifact from "fhevm-core-contracts/artifacts/gateway/GatewayContract.sol/GatewayContract.json"; import { ethers, network } from "hardhat";

import { ACL_ADDRESS, GATEWAYCONTRACT_ADDRESS, KMSVERIFIER_ADDRESS, PRIVATE_KEY_KMS_SIGNER } from "./constants"; import { awaitCoprocessor, getClearText } from "./coprocessorUtils"; import { impersonateAddress } from "./mockedSetup"; import { waitNBlocks } from "./utils";

const networkName = network.name; const aclAdd = ACL_ADDRESS;

const CiphertextType = { 0: "bool", 1: "uint8", // corresponding to euint4 2: "uint8", //

corresponding to euint8 3: "uint16", 4: "uint32", 5: "uint64", 6: "uint128", 7: "address", 8: "uint256", 9: "bytes", 10: "bytes", 11: "bytes", };

const currentTime = (): string => { const now = new Date(); return now.toLocaleTimeString("en-US", { hour12: true, hour: "numeric", minute: "numeric", second: "numeric" }); };

const argEvents = "(uint256 indexed requestID, uint256[] cts, address contractCaller, bytes4 callbackSelector, uint256 msgValue, uint256 maxTimestamp, bool passSignaturesToCaller)"; const ifaceEventDecryption = new ethers.Interface(["event EventDecryption" + argEvents]);

const argEvents2 = "(uint256 indexed requestID, bool success, bytes result)"; const ifaceResultCallback = new ethers.Interface(["event ResultCallback" + argEvents2]);

let gateway; let firstBlockListening: number; let lastBlockSnapshotForDecrypt: number;

export const initGateway = async (): Promise => { firstBlockListening = await ethers.provider.getBlockNumber(); if (networkName === "hardhat" && hre._SOLIDITY_COVERAGE_RUNNING !== true) { // evm_snapshot is not supported in coverage mode await ethers.provider.send("set_lastBlockSnapshotForDecrypt", [firstBlockListening]); } // this function will emit logs for every request and fulfilment of a decryption gateway = await ethers.getContractAt(gatewayArtifact.abi, GATEWAYCONTRACT_ADDRESS); gateway.on("EventDecryption", async (requestID, cts, contractCaller, callbackSelector, msgValue, maxTimestamp, eventData) => { const blockNumber = eventData.log.blockNumber; console.log(`${await currentTime()} - Requested decrypt on block ${blockNumber} (requestID ${requestID})`); },); gateway.on("ResultCallback", async (requestID, success, result, eventData) => { const blockNumber = eventData.log.blockNumber; console.log(`${await currentTime()} - Fulfilled decrypt on block ${blockNumber} (requestID ${requestID})`); }); };

export const awaitAllDecryptionResults = async (): Promise => { gateway = await ethers.getContractAt(gatewayArtifact.abi, GATEWAYCONTRACT_ADDRESS); const provider = ethers.provider; if (networkName === "hardhat" && hre._SOLIDITY_COVERAGE_RUNNING !== true) { // evm_snapshot is not supported in coverage mode lastBlockSnapshotForDecrypt = await provider.send("get_lastBlockSnapshotForDecrypt"); if (lastBlockSnapshotForDecrypt < firstBlockListening) { firstBlockListening = lastBlockSnapshotForDecrypt + 1; } } await fulfillAllPastRequestsIds(networkName === "hardhat"); firstBlockListening = (await ethers.provider.getBlockNumber()) + 1; if (networkName === "hardhat" && hre._SOLIDITY_COVERAGE_RUNNING !== true) { // evm_snapshot is not supported in coverage mode await provider.send("set_lastBlockSnapshotForDecrypt", [firstBlockListening]); } };

const getAlreadyFulfilledDecryptions = async (): Promise<[bigint]> => { let results = []; const eventDecryptionResult = await gateway.filters.ResultCallback().getTopicFilter(); const filterDecryptionResult = { address: GATEWAYCONTRACT_ADDRESS, fromBlock: firstBlockListening, toBlock: "latest", topics: eventDecryptionResult, }; const pastResults = await ethers.provider.getLogs(filterDecryptionResult); results = results.concat(pastResults.map((result) => ifaceResultCallback.parseLog(result).args[0])); return results; };

const allTrue = (arr: boolean[], fn = Boolean) => arr.every(fn);

const fulfillAllPastRequestsIds = async (mocked: boolean) => { const eventDecryption = await gateway.filters.EventDecryption().getTopicFilter(); const results = await getAlreadyFulfilledDecryptions(); const filterDecryption = { address: GATEWAYCONTRACT_ADDRESS, fromBlock: firstBlockListening, toBlock: "latest", topics: eventDecryption, }; const pastRequests = await ethers.provider.getLogs(filterDecryption); for (const request of pastRequests) { const event = ifaceEventDecryption.parseLog(request); const requestID = event.args[0]; const handles = event.args[1]; const typesList = handles.map((handle) => parseInt(handle.toString(16).slice(-4, -2), 16)); const msgValue = event.args[4]; const passSignaturesToCaller = event.args[6]; if (!results.includes(requestID)) { // if request is not already fulfilled if (mocked) { // in mocked mode, we trigger the decryption fulfillment manually await awaitCoprocessor();

```
    // first check tat all handles are allowed for decryption
    const aclArtifact = require("fhevm-core-contracts/artifacts/contrac
    const acl = await ethers.getContractAt(aclArtifact.abi, ACL_ADDRESS
    const isAllowedForDec = await Promise.all(handles.map(async (handle
    if (!allTrue(isAllowedForDec)) {
      throw new Error("Some handle is not authorized for decryption");
    }
    const types = typesList.map((num) => CiphertextType[num]);
    const values = await Promise.all(handles.map(async (handle) => BigI
    const valuesFormatted = values.map((value, index) =>
      types[index] === "address" ? "0x" + value.toString(16).padStart(4
    );
    const valuesFormatted2 = valuesFormatted.map((value, index) =>
      typesList[index] === 9 ? "0x" + value.toString(16).padStart(128,
    );
    const valuesFormatted3 = valuesFormatted2.map((value, index) =>
      typesList[index] === 10 ? "0x" + value.toString(16).padStart(256,
    );
    const valuesFormatted4 = valuesFormatted3.map((value, index) =>
      typesList[index] === 11 ? "0x" + value.toString(16).padStart(512,
    );

    const abiCoder = new ethers.AbiCoder();
    let encodedData;
    let calldata;
    if (!passSignaturesToCaller) {
      encodedData = abiCoder.encode(["uint256", ...types], [31, ...valu
      calldata = "0x" + encodedData.slice(66); // we just pop the dummy
    } else {
      encodedData = abiCoder.encode(["uint256", ...types, "bytes[]"], [
      calldata = "0x" + encodedData.slice(66).slice(0, -64); // we also
    }

    const numSigners = 1; // for the moment mocked mode only uses 1 sig
    const decryptResultsEIP712signatures = await computeDecryptSignatur
    const relayer = await impersonateAddress(hre, ZeroAddress, ethers.p
    await gateway
      .connect(relayer)
      .fulfillRequest(requestID, calldata, decryptResultsEIP712signatur
  } else {
    // in non-mocked mode we must wait until the gateway service relaye
```

```
    await waitNBlocks(1);
    await fulfillAllPastRequestsIds(mocked);
  }
}

} };
```

async function computeDecryptSignatures(handlesList: bigint[], decryptedResult: string, numSigners: number,): Promise<string[]> { const signatures: string[] = [];

for (let idx = 0; idx < numSigners; idx++) { const privKeySigner = PRIVATE_KEY_KMS_SIGNER; if (privKeySigner) { const kmsSigner = new ethers.Wallet(privKeySigner).connect(ethers.provider); const signature = await kmsSign(handlesList, decryptedResult, kmsSigner); signatures.push(signature); } else { throw new Error(`Private key for signer ${idx} not found in environment variables`); } } return signatures; }

async function kmsSign(handlesList: bigint[], decryptedResult: string, kmsSigner: Wallet) { const kmsAdd = KMSVERIFIER_ADDRESS; const chainId = (await ethers.provider.getNetwork()).chainId;

const domain = { name: "KMSVerifier", version: "1", chainId: chainId, verifyingContract: kmsAdd, };

const types = { DecryptionResult: [{ name: "aclAddress", type: "address", }, { name: "handlesList", type: "uint256[]", }, { name: "decryptedResult", type: "bytes", },], }; const message = { aclAddress: aclAdd, handlesList: handlesList, decryptedResult: decryptedResult, };

const signature = await kmsSigner.signTypedData(domain, types, message); const sigRSV = ethers.Signature.from(signature); const v = 27 + sigRSV.yParity; const r = sigRSV.r; const s = sigRSV.s;

const result = r + s.substring(2) + v.toString(16); return result; }

File: ./modules/hardhat/test/fhevmjsMocked.ts

import { toBigIntBE, toBufferBE } from "bigint-buffer"; import crypto from "crypto"; import { Wallet, ethers } from "ethers"; import hre from "hardhat"; import { Keccak } from "sha3"; import { isAddress } from "web3-validator";

import { ACL_ADDRESS, INPUTVERIFIER_ADDRESS, KMSVERIFIER_ADDRESS, PRIVATE_KEY_COPROCESSOR_ACCOUNT, PRIVATE_KEY_KMS_SIGNER, } from "./constants"; import { insertSQL } from "./coprocessorUtils"; import { awaitCoprocessor, getClearText } from "./coprocessorUtils";

enum Types { ebool = 0, euint4, euint8, euint16, euint32, euint64, euint128, eaddress, euint256, ebytes64, ebytes128, ebytes256, }

const sum = (arr: number[]) => arr.reduce((acc, val) => acc + val, 0);

function bytesToBigInt(byteArray: Uint8Array): bigint { if (!byteArray || byteArray?.length

=== 0) { return BigInt(0); } const buffer = Buffer.from(byteArray); const result = toBigIntBE(buffer); return result; }

function createUintToUint8ArrayFunction(numBits: number) { const numBytes = Math.ceil(numBits / 8); return function (uint: number | bigint | boolean) { const buffer = toBufferBE(BigInt(uint), numBytes);

```
// concatenate 32 random bytes at the end of buffer to simulate encrypt
const randomBytes = crypto.randomBytes(32);
const combinedBuffer = Buffer.concat([buffer, randomBytes]);

let byteBuffer;
let totalBuffer;

switch (numBits) {
  case 2: // ebool takes 2 bits
    byteBuffer = Buffer.from([Types.ebool]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 4:
    byteBuffer = Buffer.from([Types.euint4]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 8:
    byteBuffer = Buffer.from([Types.euint8]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 16:
    byteBuffer = Buffer.from([Types.euint16]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 32:
    byteBuffer = Buffer.from([Types.euint32]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 64:
    byteBuffer = Buffer.from([Types.euint64]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 128:
    byteBuffer = Buffer.from([Types.euint128]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 160:
    byteBuffer = Buffer.from([Types.eaddress]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 256:
    byteBuffer = Buffer.from([Types.euint256]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 512:
    byteBuffer = Buffer.from([Types.ebytes64]);
```

```
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 1024:
    byteBuffer = Buffer.from([Types.ebytes128]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  case 2048:
    byteBuffer = Buffer.from([Types.ebytes256]);
    totalBuffer = Buffer.concat([byteBuffer, combinedBuffer]);
    break;
  default:
    throw Error("Non-supported numBits");
}

return totalBuffer;

}; }

export const reencryptRequestMocked = async ( handle: bigint, privateKey: string, publicKey: string, signature: string, contractAddress: string, userAddress: string, ) => { // Signature checking: const domain = { name: "Authorization token", version: "1", chainId: hre.network.config.chainId, verifyingContract: contractAddress, }; const types = { Reencrypt: [{ name: "publicKey", type: "bytes" }], }; const value = { publicKey: 0x ${publicKey}, }; const signerAddress = ethers.verifyTypedData(domain, types, value, 0x ${signature}); const normalizedSignerAddress = ethers.getAddress(signerAddress); const normalizedUserAddress = ethers.getAddress(userAddress); if (normalizedSignerAddress !== normalizedUserAddress) { throw new Error("Invalid EIP-712 signature!"); }

// ACL checking const aclArtifact = require("fhevm-core-contracts/artifacts/contracts/ACL.sol/ACL.json"); const acl = await hre.ethers.getContractAt(aclArtifact.abi, ACL_ADDRESS); const userAllowed = await acl.persistAllowed(handle, userAddress); const contractAllowed = await acl.persistAllowed(handle, contractAddress); const isAllowed = userAllowed && contractAllowed; if (!isAllowed) { throw new Error("User is not authorized to reencrypt this handle!"); } if (userAddress === contractAddress) { throw new Error("userAddress should not be equal to contractAddress when requesting reencryption!"); } await awaitCoprocessor(); return BigInt(await getClearText(handle)); };

export const createEncryptedInputMocked = (contractAddress: string, userAddress: string) => { if (!isAddress(contractAddress)) { throw new Error("Contract address is not a valid address."); }

if (!isAddress(userAddress)) { throw new Error("User address is not a valid address."); }

const values: bigint[] = []; const bits: (keyof typeof ENCRYPTION_TYPES)[] = []; return { addBool(value: boolean | number | bigint) { if (value == null) throw new Error("Missing value"); if (typeof value !== "boolean" && typeof value !== "number" && typeof value !== "bigint") throw new Error("The value must be a boolean, a number or a bigint."); if ((typeof value !== "bigint" || typeof value !== "number") && Number(value) > 1) throw new Error("The value must be 1 or 0."); values.push(BigInt(value)); bits.push(2); // ebool takes 2 bits instead of one: only exception in TFHE-rs if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add4(value: number | bigint) { checkEncryptedValue(value, 4); values.push(BigInt(value)); bits.push(4); if (sum(bits) >
```

2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add8(value: number | bigint) { checkEncryptedValue(value, 8); values.push(BigInt(value)); bits.push(8); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add16(value: number | bigint) { checkEncryptedValue(value, 16); values.push(BigInt(value)); bits.push(16); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add32(value: number | bigint) { checkEncryptedValue(value, 32); values.push(BigInt(value)); bits.push(32); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add64(value: number | bigint) { checkEncryptedValue(value, 64); values.push(BigInt(value)); bits.push(64); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add128(value: number | bigint) { checkEncryptedValue(value, 128); values.push(BigInt(value)); bits.push(128); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, addAddress(value: string) { if (! isAddress(value)) { throw new Error("The value must be a valid address."); } values.push(BigInt(value)); bits.push(160); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, add256(value: number | bigint) { checkEncryptedValue(value, 256); values.push(BigInt(value)); bits.push(256); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, addBytes64(value: Uint8Array) { if (value.length !== 64) throw Error("Uncorrect length of input Uint8Array, should be 64 for an ebytes64"); const bigIntValue = bytesToBigInt(value); checkEncryptedValue(bigIntValue, 512); values.push(bigIntValue); bits.push(512); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, addBytes128(value: Uint8Array) { if (value.length !== 128) throw Error("Uncorrect length of input Uint8Array, should be 128 for an ebytes128"); const bigIntValue = bytesToBigInt(value); checkEncryptedValue(bigIntValue, 1024); values.push(bigIntValue); bits.push(1024); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, addBytes256(value: Uint8Array) { if (value.length !== 256) throw Error("Uncorrect length of input Uint8Array, should be 256 for an ebytes256"); const bigIntValue = bytesToBigInt(value); checkEncryptedValue(bigIntValue, 2048); values.push(bigIntValue); bits.push(2048); if (sum(bits) > 2048) throw Error("Packing more than 2048 bits in a single input ciphertext is unsupported"); if (bits.length > 256) throw Error("Packing more than 256 variables in a single input ciphertext is unsupported"); return this; }, getValues() { return values; }, getBits() { return bits; }, resetValues() { values.length = 0; bits.length = 0; return this; }, async encrypt() { let encrypted = Buffer.alloc(0);

```
  bits.map((v, i) => {
```

```
  encrypted = Buffer.concat([encrypted, createUintToUint8ArrayFunctio
});

const encryptedArray = new Uint8Array(encrypted);
const hash = new Keccak(256).update(Buffer.from(encryptedArray)).dige

const handles = bits.map((v, i) => {
  const dataWithIndex = new Uint8Array(hash.length + 1);
  dataWithIndex.set(hash, 0);
  dataWithIndex.set([i], hash.length);
  const finalHash = new Keccak(256).update(Buffer.from(dataWithIndex)
  const dataInput = new Uint8Array(32);
  dataInput.set(finalHash, 0);
  dataInput.set([i, ENCRYPTION_TYPES[v], 0], 29);
  return dataInput;
});
let inputProof = "0x" + numberToHex(handles.length); // for coprocess
// for native : numHandles + numSignersKMS + list_handles + signature
const numSigners = 1; // @note: only 1 signer in mocked mode for the
inputProof += numberToHex(numSigners);
//if (process.env.IS_COPROCESSOR === "true") { // @note: for now we s
// coprocessor
inputProof += hash.toString("hex");

const listHandlesStr = handles.map((i) => uint8ArrayToHexString(i));
listHandlesStr.map((handle) => (inputProof += handle));
const listHandles = listHandlesStr.map((i) => BigInt("0x" + i));
const sigCoproc = await computeInputSignatureCopro(
  "0x" + hash.toString("hex"),
  listHandles,
  userAddress,
  contractAddress,
);
inputProof += sigCoproc.slice(2);

const signaturesKMS = await computeInputSignaturesKMS("0x" + hash.toS
signaturesKMS.map((sigKMS) => (inputProof += sigKMS.slice(2)));
listHandlesStr.map((handle, i) => insertSQL("0x" + handle, values[i])
/*} else {
  // native
  const listHandlesStr = handles.map((i) => uint8ArrayToHexString(i))
  listHandlesStr.map((handle) => (inputProof += handle));
  const signaturesKMS = await computeInputSignaturesKMS(
    "0x" + hash.toString("hex"),
    userAddress,
    contractAddress,
  );
  signaturesKMS.map((sigKMS) => (inputProof += sigKMS.slice(2)));
  listHandlesStr.map((handle, i) => insertSQL("0x" + handle, values[i

  inputProof += encrypted.toString("hex");
}*/
```

```
  return {
    handles,
    inputProof,
  };
},

}; };

function uint8ArrayToHexString(uint8Array: Uint8Array) { return Array.from(uint8Array)
.map((byte) => byte.toString(16).padStart(2, "0")) .join(""); }

function numberToHex(num: number) { const hex = num.toString(16); return hex.length %
2 ? "0" + hex : hex; }

const checkEncryptedValue = (value: number | bigint, bits: number) => { if (value ==
null) throw new Error("Missing value"); let limit; if (bits >= 8) { limit = BigInt(0x${new
Array(bits / 8).fill(null).reduce((v) => ${v}ff, "")}); } else { limit =
BigInt(2 ** bits - 1); } if (typeof value !== "number" && typeof value !== "bigint") throw
new Error("Value must be a number or a bigint."); if (value > limit) { throw new Error(The
value exceeds the limit for ${bits}bits integer
(${limit.toString()}).); } };

export const ENCRYPTION_TYPES = { 2: 0, // ebool takes 2 bits 4: 1, 8: 2, 16: 3, 32: 4, 64:
5, 128: 6, 160: 7, 256: 8, 512: 9, 1024: 10, 2048: 11, };

async function computeInputSignatureCopro( hash: string, handlesList: bigint[], userAddress:
string, contractAddress: string, ): Promise { const privKeySigner =
PRIVATE_KEY_COPROCESSOR_ACCOUNT; const coprocSigner = new
Wallet(privKeySigner).connect(ethers.provider); const signature = await coprocSign(hash,
handlesList, userAddress, contractAddress, coprocSigner); return signature; }

async function computeInputSignaturesKMS( hash: string, userAddress: string,
contractAddress: string, ): Promise<string[]> { const signatures: string[] = []; const
numSigners = 1; // @note: only 1 KMS signer in mocked mode for now for (let idx = 0; idx
< numSigners; idx++) { const privKeySigner = PRIVATE_KEY_KMS_SIGNER; const
kmsSigner = new ethers.Wallet(privKeySigner).connect(ethers.provider); const signature =
await kmsSign(hash, userAddress, contractAddress, kmsSigner); signatures.push(signature); }
return signatures; }

async function coprocSign( hashOfCiphertext: string, handlesList: bigint[], userAddress:
string, contractAddress: string, signer: Wallet, ): Promise { const inputAdd =
INPUTVERIFIER_ADDRESS; const chainId = hre.__SOLIDITY_COVERAGE_RUNNING ? 31337
: network.config.chainId; const aclAdd = ACL_ADDRESS;

const domain = { name: "InputVerifier", version: "1", chainId: chainId, verifyingContract:
inputAdd, };

const types = { CiphertextVerificationForCopro: [ { name: "aclAddress", type: "address", }, {
name: "hashOfCiphertext", type: "bytes32", }, { name: "handlesList", type: "uint256[]", }, {
name: "userAddress", type: "address", }, { name: "contractAddress", type: "address", }, ], };
const message = { aclAddress: aclAdd, hashOfCiphertext: hashOfCiphertext, handlesList:
handlesList, userAddress: userAddress, contractAddress: contractAddress, };

const signature = await signer.signTypedData(domain, types, message); const sigRSV =
ethers.Signature.from(signature); const v = 27 + sigRSV.yParity; const r = sigRSV.r; const s
```

= sigRSV.s;

const result = r + s.substring(2) + v.toString(16); return result; }

async function kmsSign(hashOfCiphertext: string, userAddress: string, contractAddress: string, signer: Wallet,): Promise { const kmsVerifierAdd = KMSVERIFIER_ADDRESS; const chainId = hre.__SOLIDITY_COVERAGE_RUNNING ? 31337 : network.config.chainId; const aclAdd = ACL_ADDRESS;

const domain = { name: "KMSVerifier", version: "1", chainId: chainId, verifyingContract: kmsVerifierAdd, };

const types = { CiphertextVerificationForKMS: [{ name: "aclAddress", type: "address", }, { name: "hashOfCiphertext", type: "bytes32", }, { name: "userAddress", type: "address", }, { name: "contractAddress", type: "address", },], }; const message = { aclAddress: aclAdd, hashOfCiphertext: hashOfCiphertext, userAddress: userAddress, contractAddress: contractAddress, };

const signature = await signer.signTypedData(domain, types, message); const sigRSV = ethers.Signature.from(signature); const v = 27 + sigRSV.yParity; const r = sigRSV.r; const s = sigRSV.s;

const result = r + s.substring(2) + v.toString(16); return result; }

File: ./modules/hardhat/test/coprocessorUtils.ts

import { log2 } from "extra-bigint"; import { ethers } from "hardhat"; import hre from "hardhat"; import { Database } from "sqlite3";

import { TFHEEXECUTOR_ADDRESS } from "./constants"; import operatorPrices from "./operatorPrices.json";

const executorAddress = TFHEEXECUTOR_ADDRESS;

let firstBlockListening = 0; let lastBlockSnapshot = 0; let lastCounterRand = 0; let counterRand = 0;

//const db = new Database('./sql.db'); // on-disk db for debugging const db = new Database(":memory:");

export function insertSQL(handle: string, clearText: bigint, replace: boolean = false) { if (replace) { // this is useful if using snapshots while sampling different random numbers on each revert db.run("INSERT OR REPLACE INTO ciphertexts (handle, clearText) VALUES (?, ?)", [handle, clearText.toString()]); } else { db.run("INSERT OR IGNORE INTO ciphertexts (handle, clearText) VALUES (?, ?)", [handle, clearText.toString()]); } }

// Decrypt any handle, bypassing ACL // WARNING : only for testing or internal use export const getClearText = async (handle: bigint): Promise = > { const handleStr = "0x" + handle.toString(16).padStart(64, "0");

return new Promise((resolve, reject) = > { let attempts = 0; const maxRetries = 100;

```
function executeQuery() {
  db.get("SELECT clearText FROM ciphertexts WHERE handle = ?", [handleS
    if (err) {
      reject(new Error(`Error querying database: ${err.message}`));
    } else if (row) {
      resolve(row.clearText);
    } else if (attempts < maxRetries) {
      attempts++;
      executeQuery();
    } else {
      reject(new Error("No record found after maximum retries"));
    }
  });
}

executeQuery();

}); };
```

db.serialize(() => db.run("CREATE TABLE IF NOT EXISTS ciphertexts (handle BINARY PRIMARY KEY,clearText TEXT)"));

interface FHEVMEvent { eventName: string; args: object; }

const NumBits = { 0: 1n, //ebool 1: 4n, //euint4 2: 8n, //euint8 3: 16n, //euint16 4: 32n, //euint32 5: 64n, //euint64 6: 128n, //euint128 7: 160n, //eaddress 8: 256n, //euint256 9: 512n, //ebytes64 10: 1024n, //ebytes128 11: 2048n, //ebytes256 };

export function numberToEvenHexString(num: number) { if (typeof num !== "number" || num < 0) { throw new Error("Input should be a non-negative number."); } let hexString = num.toString(16); if (hexString.length % 2 !== 0) { hexString = "0" + hexString; } return hexString; }

function getRandomBigInt(numBits: number): bigint { if (numBits <= 0) { throw new Error("Number of bits must be greater than 0"); } const numBytes = Math.ceil(numBits / 8); const randomBytes = new Uint8Array(numBytes); crypto.getRandomValues(randomBytes); let randomBigInt = BigInt(0); for (let i = 0; i < numBytes; i++) { randomBigInt = (randomBigInt << BigInt(8)) | BigInt(randomBytes[i]); } const mask = (BigInt(1) << BigInt(numBits)) - BigInt(1); randomBigInt = randomBigInt & mask; return randomBigInt; }

function bitwiseNotUintBits(value: bigint, numBits: number) { if (typeof value !== "bigint") { throw new TypeError("The input value must be a BigInt."); } if (typeof numBits !== "number" || numBits <= 0) { throw new TypeError("The numBits parameter must be a positive integer."); } // Create the mask with numBits bits set to 1 const BIT_MASK = (BigInt(1) << BigInt(numBits)) - BigInt(1); return ~value & BIT_MASK; }

export const awaitCoprocessor = async (): Promise => { await processAllPastTFHEExecutorEvents(); };

const abi = ["event FheAdd(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheSub(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheMul(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheDiv(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheRem(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheBitAnd(uint256 lhs, uint256 rhs, bytes1

scalarByte, uint256 result)", "event FheBitOr(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheBitXor(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheShl(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheShr(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheRotl(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheRotr(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheEq(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheEqBytes(uint256 lhs, bytes rhs, bytes1 scalarByte, uint256 result)", "event FheNe(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheNeBytes(uint256 lhs, bytes rhs, bytes1 scalarByte, uint256 result)", "event FheGe(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheGt(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheLe(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheLt(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheMin(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheMax(uint256 lhs, uint256 rhs, bytes1 scalarByte, uint256 result)", "event FheNeg(uint256 ct, uint256 result)", "event FheNot(uint256 ct, uint256 result)", "event VerifyCiphertext(bytes32 inputHandle,address userAddress,bytes inputProof,bytes1 inputType,uint256 result)", "event Cast(uint256 ct, bytes1 toType, uint256 result)", "event TrivialEncrypt(uint256 pt, bytes1 toType, uint256 result)", "event TrivialEncryptBytes(bytes pt, bytes1 toType, uint256 result)", "event FheIfThenElse(uint256 control, uint256 ifTrue, uint256 ifFalse, uint256 result)", "event FheRand(bytes1 randType, uint256 result)", "event FheRandBounded(uint256 upperBound, bytes1 randType, uint256 result)",];

async function processAllPastTFHEExecutorEvents() { const provider = ethers.provider; const latestBlockNumber = await provider.getBlockNumber();

if (hre._SOLIDITY_COVERAGE_RUNNING !== true) { // evm_snapshot is not supported in coverage mode [lastBlockSnapshot, lastCounterRand] = await provider.send("get_lastBlockSnapshot"); if (lastBlockSnapshot < firstBlockListening) { firstBlockListening = lastBlockSnapshot + 1; counterRand = Number(lastCounterRand); } }

const contract = new ethers.Contract(executorAddress, abi, provider);

// Fetch all events emitted by the contract const filter = { address: executorAddress, fromBlock: firstBlockListening, toBlock: latestBlockNumber, };

const logs = await provider.getLogs(filter);

const events = logs .map((log) => { try { const parsedLog = contract.interface.parseLog(log); return { eventName: parsedLog.name, args: parsedLog.args, }; } catch (e) { // If the log cannot be parsed, skip it return null; } }) .filter((event) => event !== null);

firstBlockListening = latestBlockNumber + 1; if (hre._SOLIDITY_COVERAGE_RUNNING !== true) { // evm_snapshot is not supported in coverage mode await provider.send("set_lastBlockSnapshot", [firstBlockListening]); } events.map(async (event) => await insertHandleFromEvent(event)); }

async function insertHandleFromEvent(event: FHEVMEvent) { let handle; let clearText; let clearLHS; let clearRHS; let resultType; let shift;

switch (event.eventName) { case "TrivialEncrypt": clearText = event.args[0]; handle = ethers.toBeHex(event.args[2], 32); insertSQL(handle, clearText); break;

```
case "TrivialEncryptBytes":
  clearText = event.args[0];
```

```
  handle = ethers.toBeHex(event.args[2], 32);
  insertSQL(handle, clearText);
  break;

case "FheAdd":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    clearText = BigInt(clearLHS) + event.args[1];
    clearText = clearText % 2n ** NumBits[resultType];
  } else {
    clearRHS = await getClearText(event.args[1]);
    clearText = BigInt(clearLHS) + BigInt(clearRHS);
    clearText = clearText % 2n ** NumBits[resultType];
  }
  insertSQL(ethers.toBeHex(handle, 32), clearText);
  break;

case "FheSub":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    clearText = BigInt(clearLHS) - event.args[1];
    if (clearText < 0n) clearText = clearText + 2n ** NumBits[resultTyp
    clearText = clearText % 2n ** NumBits[resultType];
  } else {
    clearRHS = await getClearText(event.args[1]);
    clearText = BigInt(clearLHS) - BigInt(clearRHS);
    if (clearText < 0n) clearText = clearText + 2n ** NumBits[resultTyp
    clearText = clearText % 2n ** NumBits[resultType];
  }
  insertSQL(handle, clearText);
  break;

case "FheMul":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    clearText = BigInt(clearLHS) * event.args[1];
    clearText = clearText % 2n ** NumBits[resultType];
  } else {
    clearRHS = await getClearText(event.args[1]);
    clearText = BigInt(clearLHS) * BigInt(clearRHS);
    clearText = clearText % 2n ** NumBits[resultType];
  }
  insertSQL(handle, clearText);
  break;

case "FheDiv":
  handle = ethers.toBeHex(event.args[3], 32);
```

```
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) / event.args[1];
    } else {
      throw new Error("Non-scalar div not implemented yet");
    }
    insertSQL(handle, clearText);
    break;

  case "FheRem":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) % event.args[1];
    } else {
      throw new Error("Non-scalar rem not implemented yet");
    }
    insertSQL(handle, clearText);
    break;

  case "FheBitAnd":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) & event.args[1];
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) & BigInt(clearRHS);
      clearText = clearText % 2n ** NumBits[resultType];
    }
    insertSQL(handle, clearText);
    break;

  case "FheBitOr":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) | event.args[1];
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) | BigInt(clearRHS);
      clearText = clearText % 2n ** NumBits[resultType];
    }
    insertSQL(handle, clearText);
    break;

  case "FheBitXor":
```

```
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) ^ event.args[1];
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) ^ BigInt(clearRHS);
      clearText = clearText % 2n ** NumBits[resultType];
    }
    insertSQL(handle, clearText);
    break;

  case "FheShl":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) << event.args[1] % NumBits[resultType]
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) << BigInt(clearRHS) % NumBits[resultTy
      clearText = clearText % 2n ** NumBits[resultType];
    }
    insertSQL(handle, clearText);
    break;

  case "FheShr":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) >> event.args[1] % NumBits[resultType]
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) >> BigInt(clearRHS) % NumBits[resultTy
      clearText = clearText % 2n ** NumBits[resultType];
    }
    insertSQL(handle, clearText);
    break;

  case "FheRotl":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      shift = event.args[1] % NumBits[resultType];
      clearText = (BigInt(clearLHS) << shift) | (BigInt(clearLHS) >> (Num
      clearText = clearText % 2n ** NumBits[resultType];
    } else {
```

```
    clearRHS = await getClearText(event.args[1]);
    shift = BigInt(clearRHS) % NumBits[resultType];
    clearText = (BigInt(clearLHS) << shift) | (BigInt(clearLHS) >> (Num
    clearText = clearText % 2n ** NumBits[resultType];
  }
  insertSQL(handle, clearText);
  break;

case "FheRotr":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    shift = event.args[1] % NumBits[resultType];
    clearText = (BigInt(clearLHS) >> shift) | (BigInt(clearLHS) << (Num
    clearText = clearText % 2n ** NumBits[resultType];
  } else {
    clearRHS = await getClearText(event.args[1]);
    shift = BigInt(clearRHS) % NumBits[resultType];
    clearText = (BigInt(clearLHS) >> shift) | (BigInt(clearLHS) << (Num
    clearText = clearText % 2n ** NumBits[resultType];
  }
  insertSQL(handle, clearText);
  break;

case "FheEq":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    clearText = BigInt(clearLHS) === event.args[1] ? 1n : 0n;
  } else {
    clearRHS = await getClearText(event.args[1]);
    clearText = BigInt(clearLHS) === BigInt(clearRHS) ? 1n : 0n;
  }
  insertSQL(handle, clearText);
  break;

case "FheEqBytes":
  handle = ethers.toBeHex(event.args[3], 32);
  resultType = parseInt(handle.slice(-4, -2), 16);
  clearLHS = await getClearText(event.args[0]);
  if (event.args[2] === "0x01") {
    clearText = BigInt(clearLHS) === BigInt(event.args[1]) ? 1n : 0n;
  } else {
    clearRHS = await getClearText(event.args[1]);
    clearText = BigInt(clearLHS) === BigInt(clearRHS) ? 1n : 0n;
  }
  insertSQL(handle, clearText);
  break;

case "FheNe":
  handle = ethers.toBeHex(event.args[3], 32);
```

```
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) !== event.args[1] ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) !== BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheNeBytes":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) !== BigInt(event.args[1]) ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) !== BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheGe":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) >= event.args[1] ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) >= BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheGt":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) > event.args[1] ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) > BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheLe":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
```

```
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) <= event.args[1] ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) <= BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheLt":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) < event.args[1] ? 1n : 0n;
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) < BigInt(clearRHS) ? 1n : 0n;
    }
    insertSQL(handle, clearText);
    break;

  case "FheMax":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) > event.args[1] ? clearLHS : event.arg
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) > BigInt(clearRHS) ? clearLHS : clearR
    }
    insertSQL(handle, clearText);
    break;

  case "FheMin":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearLHS = await getClearText(event.args[0]);
    if (event.args[2] === "0x01") {
      clearText = BigInt(clearLHS) < event.args[1] ? clearLHS : event.arg
    } else {
      clearRHS = await getClearText(event.args[1]);
      clearText = BigInt(clearLHS) < BigInt(clearRHS) ? clearLHS : clearR
    }
    insertSQL(handle, clearText);
    break;

  case "Cast":
    resultType = parseInt(event.args[1]);
    handle = ethers.toBeHex(event.args[2], 32);
    clearText = BigInt(await getClearText(event.args[0])) % 2n ** NumBits
```

```
    insertSQL(handle, clearText);
    break;

  case "FheNot":
    handle = ethers.toBeHex(event.args[1], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearText = BigInt(await getClearText(event.args[0]));
    clearText = bitwiseNotUintBits(clearText, Number(NumBits[resultType])
    insertSQL(handle, clearText);
    break;

  case "FheNeg":
    handle = ethers.toBeHex(event.args[1], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    clearText = BigInt(await getClearText(event.args[0]));
    clearText = bitwiseNotUintBits(clearText, Number(NumBits[resultType])
    clearText = (clearText + 1n) % 2n ** NumBits[resultType];
    insertSQL(handle, clearText);
    break;

  case "VerifyCiphertext":
    handle = event.args[0];
    try {
      await getClearText(BigInt(handle));
    } catch {
      throw Error("User input was not found in DB");
    }
    break;

  case "FheIfThenElse":
    handle = ethers.toBeHex(event.args[3], 32);
    resultType = parseInt(handle.slice(-4, -2), 16);
    handle = ethers.toBeHex(event.args[3], 32);
    const clearControl = BigInt(await getClearText(event.args[0]));
    const clearIfTrue = BigInt(await getClearText(event.args[1]));
    const clearIfFalse = BigInt(await getClearText(event.args[2]));
    if (clearControl === 1n) {
      clearText = clearIfTrue;
    } else {
      clearText = clearIfFalse;
    }
    insertSQL(handle, clearText);
    break;

  case "FheRand":
    resultType = parseInt(event.args[0], 16);
    handle = ethers.toBeHex(event.args[1], 32);
    clearText = getRandomBigInt(Number(NumBits[resultType]));
    insertSQL(handle, clearText, true);
    counterRand++;
    break;

  case "FheRandBounded":
```

```
  resultType = parseInt(event.args[1], 16);
  handle = ethers.toBeHex(event.args[2], 32);
  clearText = getRandomBigInt(Number(log2(BigInt(event.args[0]))));
  insertSQL(handle, clearText, true);
  counterRand++;
  break;

}}
```

export function getFHEGasFromTxReceipt(receipt: ethers.TransactionReceipt): number { if (hre.network.name !== "hardhat") { throw Error("FHEGas tracking is currently implemented only in mocked mode"); } if (receipt.status === 0) { throw new Error("Transaction reverted"); } const contract = new ethers.Contract(executorAddress, abi, ethers.provider); const relevantLogs = receipt.logs.filter((log: ethers.Log) => { if (log.address.toLowerCase() !== executorAddress.toLowerCase()) { return false; } try { const parsedLog = contract.interface.parseLog({ topics: log.topics, data: log.data, }); return abi.some((item) => item.startsWith(`event ${parsedLog.name}`) && parsedLog.name !== "VerifyCiphertext"); } catch { return false; } }); const FHELogs = relevantLogs.map((log: ethers.Log) => { const parsedLog = contract.interface.parseLog({ topics: log.topics, data: log.data, }); return { name: parsedLog.name, args: parsedLog.args, }; }); let FHEGasConsumed = 0; for (const event of FHELogs) { let type; let handle; switch (event.name) { case "TrivialEncrypt": type = parseInt(event.args[1], 16); FHEGasConsumed += operatorPrices["trivialEncrypt"].types[type]; break;

```
  case "TrivialEncryptBytes":
    type = parseInt(event.args[1], 16);
    FHEGasConsumed += operatorPrices["trivialEncrypt"].types[type];
    break;

  case "FheAdd":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheAdd"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheAdd"].nonScalar[type];
    }
    break;

  case "FheSub":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheSub"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheSub"].nonScalar[type];
    }
    break;

  case "FheMul":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
```

```
    FHEGasConsumed += operatorPrices["fheMul"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheMul"].nonScalar[type];
  }
  break;

case "FheDiv":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheDiv"].scalar[type];
  } else {
    throw new Error("Non-scalar div not implemented yet");
  }
  break;

case "FheRem":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheRem"].scalar[type];
  } else {
    throw new Error("Non-scalar rem not implemented yet");
  }
  break;

case "FheBitAnd":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheBitAnd"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheBitAnd"].nonScalar[type];
  }
  break;

case "FheBitOr":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheBitOr"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheBitOr"].nonScalar[type];
  }
  break;

case "FheBitXor":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheBitXor"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheBitXor"].nonScalar[type];
```

```
    }
    break;

  case "FheShl":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheBitShl"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheBitShl"].nonScalar[type];
    }
    break;

  case "FheShr":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheBitShr"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheBitShr"].nonScalar[type];
    }
    break;

  case "FheRotl":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheRotl"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheRotl"].nonScalar[type];
    }
    break;

  case "FheRotr":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheRotr"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheRotr"].nonScalar[type];
    }
    break;

  case "FheEq":
    handle = ethers.toBeHex(event.args[0], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    if (event.args[2] === "0x01") {
      FHEGasConsumed += operatorPrices["fheEq"].scalar[type];
    } else {
      FHEGasConsumed += operatorPrices["fheEq"].nonScalar[type];
    }
    break;
```

```
case "FheEqBytes":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheEq"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheEq"].nonScalar[type];
  }

case "FheNe":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheNe"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheNe"].nonScalar[type];
  }
  break;

case "FheNeBytes":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheNe"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheNe"].nonScalar[type];
  }
  break;

case "FheGe":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheGe"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheGe"].nonScalar[type];
  }
  break;

case "FheGt":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheGt"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheGt"].nonScalar[type];
  }
  break;

case "FheLe":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
```

```
    FHEGasConsumed += operatorPrices["fheLe"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheLe"].nonScalar[type];
  }
  break;

case "FheLt":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheLt"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheLt"].nonScalar[type];
  }
  break;

case "FheMax":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheMax"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheMax"].nonScalar[type];
  }
  break;

case "FheMin":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  if (event.args[2] === "0x01") {
    FHEGasConsumed += operatorPrices["fheMin"].scalar[type];
  } else {
    FHEGasConsumed += operatorPrices["fheMin"].nonScalar[type];
  }
  break;

case "Cast":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  FHEGasConsumed += operatorPrices["cast"].types[type];
  break;

case "FheNot":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  FHEGasConsumed += operatorPrices["fheNot"].types[type];
  break;

case "FheNeg":
  handle = ethers.toBeHex(event.args[0], 32);
  type = parseInt(handle.slice(-4, -2), 16);
  FHEGasConsumed += operatorPrices["fheNeg"].types[type];
  break;
```

```
  case "FheIfThenElse":
    handle = ethers.toBeHex(event.args[3], 32);
    type = parseInt(handle.slice(-4, -2), 16);
    FHEGasConsumed += operatorPrices["ifThenElse"].types[type];
    break;

  case "FheRand":
    type = parseInt(event.args[0], 16);
    FHEGasConsumed += operatorPrices["fheRand"].types[type];
    break;

  case "FheRandBounded":
    type = parseInt(event.args[1], 16);
    FHEGasConsumed += operatorPrices["fheRandBounded"].types[type];
    break;
}
```

} return FHEGasConsumed; }

File: ./modules/hardhat/test/constants.ts

export const TFHEEXECUTOR_ADDRESS = "0x687408ab54661ba0b4aef3a44156c616c6955e07"; export const ACL_ADDRESS = "0xfee8407e2f5e3ee68ad77cae98c434e637f516e5"; export const FHEPAYMENT_ADDRESS = "0xfb03be574d14c256d56f09a198b586bdfc0a9de2"; export const KMSVERIFIER_ADDRESS = "0x9d6891a6240d6130c54ae243d8005063d05fe14b"; export const INPUTVERIFIER_ADDRESS = "0x3a2DA6f1daE9eF988B48d9CF27523FA31a8eBE50"; export const GATEWAYCONTRACT_ADDRESS = "0x33347831500f1e73f0cccbb95c9f86b94d7b1123"; export const PRIVATE_KEY_KMS_SIGNER = "388b7680e4e1afa06efbfd45cdd1fe39f3c6af381df6555a19661f283b97de91"; export const PRIVATE_KEY_COPROCESSOR_ACCOUNT = "7ec8ada6642fc4ccfb7729bc29c17cf8d21b61abd5642d1db992c0b8672ab901"; export const GATEWAY_URL = "https://gateway.sepolia.zama.ai/"; export const ACCOUNT_NAMES = ["alice", "bob", "carol", "dave", "eve", "fred", "greg", "hugo", "ian", "jane"];

File: ./modules/hardhat/test/signers.ts

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

import { ACCOUNT_NAMES } from "./constants";

type AccountNames = (typeof ACCOUNT_NAMES)[number];

export interface Signers { [K in AccountNames]: HardhatEthersSigner; }

const signers: Signers = {} as Signers;

export const initSigners = async (): Promise = > { if (Object.entries(signers).length = = = 0) { const eSigners = await ethers.getSigners(); for (let index = 0; index <

ACCOUNT_NAMES.length; index++) { const name = ACCOUNT_NAMES[index]; signers[name] = eSigners[index]; } } };

export const getSigners = async (): Promise => { return signers; };

File: ./modules/hardhat/test/reencrypt.ts

import { Signer } from "ethers"; import { FhevmInstance } from "fhevmjs/node";

const EBOOL_T = 0; const EUINT4_T = 1; const EUINT8_T = 2; const EUINT16_T = 3; const EUINT32_T = 4; const EUINT64_T = 5; const EUINT128_T = 6; const EUINT160_T = 7; // @dev It is the one for eaddresses. const EUINT256_T = 8; const EBYTES64_T = 9; const EBYTES128_T = 10; const EBYTES256_T = 11;

export function verifyType(handle: bigint, expectedType: number) { if (handle === 0n) { throw "Handle is not initialized"; }

if (handle.toString(2).length > 256) { throw "Handle is not a bytes32"; }

const typeCt = handle >> 8n;

if (Number(typeCt % 256n) !== expectedType) { throw "Wrong encrypted type for the handle"; } }

export async function reencryptEbool(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EBOOL_T); return (await reencryptHandle(signer, instance, handle, contractAddress)) === 1n; }

export async function reencryptEuint4(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT4_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEuint8(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT8_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEuint16(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT16_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEuint32(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT32_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEuint64(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT64_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEuint128(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT128_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEaddress(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT160_T); const

addressAsUint160: bigint = await reencryptHandle(signer, instance, handle, contractAddress); const handleStr = "0x" + addressAsUint160.toString(16).padStart(40, "0"); return handleStr; }

export async function reencryptEuint256(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EUINT256_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEbytes64(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EBYTES64_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEbytes128(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EBYTES128_T); return reencryptHandle(signer, instance, handle, contractAddress); }

export async function reencryptEbytes256(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { verifyType(handle, EBYTES256_T); return reencryptHandle(signer, instance, handle, contractAddress); }

/**

- @dev This function is to reencrypt handles.
- `It does not verify types.`

*/ async function reencryptHandle(signer: Signer, instance: FhevmInstance, handle: bigint, contractAddress: string,): Promise { const { publicKey: publicKey, privateKey: privateKey } = instance.generateKeypair(); const eip712 = instance.createEIP712(publicKey, contractAddress); const signature = await signer.signTypedData(eip712.domain, { Reencrypt: eip712.types.Reencrypt }, eip712.message);

const reencryptedHandle = await instance.reencrypt(handle, privateKey, publicKey, signature.replace("0x", ""), contractAddress, await signer.getAddress(),);

return reencryptedHandle; }

File: ./modules/hardhat/test/instance.ts

import { createEIP712, createInstance as createFhevmInstance, generateKeypair } from "fhevmjs"; import { FhevmInstance } from "fhevmjs/node"; import { network } from "hardhat";

import { ACL_ADDRESS, GATEWAY_URL, KMSVERIFIER_ADDRESS } from "./constants"; import { createEncryptedInputMocked, reencryptRequestMocked } from "./fhevmjsMocked";

const kmsAdd = KMSVERIFIER_ADDRESS; const aclAdd = ACL_ADDRESS;

export const createInstance = async (): Promise = > { if (network.name = = = "hardhat") { const instance = { reencrypt: reencryptRequestMocked, createEncryptedInput: createEncryptedInputMocked, getPublicKey: () = > "0xFFAA44433", generateKeypair: generateKeypair, createEIP712: createEIP712(network.config.chainId), }; return instance; } else { const instance = await createFhevmInstance({ kmsContractAddress: kmsAdd, aclContractAddress: aclAdd, networkUrl: network.config.url, gatewayUrl: GATEWAY_URL, });

return instance; } };

File: ./modules/hardhat/test/confidentialERC20/ConfidentialERC20.fixture.ts

import { ethers } from "hardhat";

import type { MyConfidentialERC20 } from "../../types"; import { getSigners } from "../signers";

export async function deployConfidentialERC20Fixture(): Promise { const signers = await getSigners();

const contractFactory = await ethers.getContractFactory("MyConfidentialERC20"); const contract = await contractFactory.connect(signers.alice).deploy("Naraggara", "NARA"); // City of Zama's battle await contract.waitForDeployment();

return contract; }

File: ./modules/hardhat/test/confidentialERC20/ConfidentialERC20.FHEGas.ts

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

import { getFHEGasFromTxReceipt } from "../coprocessorUtils"; import { createInstance } from "../instance"; import { getSigners, initSigners } from "../signers"; import { deployConfidentialERC20Fixture } from "./ConfidentialERC20.fixture";

describe("ConfidentialERC20:FHEGas", function () { before(async function () { await initSigners(); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20Fixture(); this.contractAddress = await contract.getAddress(); this.erc20 = contract; this.fhevm = await createInstance(); });

it("gas consumed during transfer", async function () { const transaction = await this.erc20.mint(10000); const t1 = await transaction.wait(); expect(t1?.status).to.eq(1);

```
const input = this.fhevm.createEncryptedInput(this.contractAddress, thi
input.add64(1337);
const encryptedTransferAmount = await input.encrypt();
const tx = await this.erc20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);
```

```
const t2 = await tx.wait();
expect(t2?.status).to.eq(1);
if (network.name === "hardhat") {
  // `getFHEGasFromTxReceipt` function only works in mocked mode but gi
  const FHEGasConsumedTransfer = getFHEGasFromTxReceipt(t2);
  console.log("FHEGas Consumed during transfer", FHEGasConsumedTransfer
}
// contrarily to FHEGas, native gas in mocked mode slightly differs fro
console.log("Native Gas Consumed during transfer", t2.gasUsed);

});
```

it("gas consumed during transferFrom", async function () { const transaction = await this.erc20.mint(10000); await transaction.wait();

```
const inputAlice = this.fhevm.createEncryptedInput(this.contractAddress
inputAlice.add64(1337);
const encryptedAllowanceAmount = await inputAlice.encrypt();
const tx = await this.erc20["approve(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedAllowanceAmount.handles[0],
  encryptedAllowanceAmount.inputProof,
);
await tx.wait();

const bobErc20 = this.erc20.connect(this.signers.bob);
const inputBob2 = this.fhevm.createEncryptedInput(this.contractAddress,
inputBob2.add64(1337); // below allowance so next tx should send token
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,
);
const t3 = await tx3.wait();
if (network.name === "hardhat") {
  // `getFHEGasFromTxReceipt` function only works in mocked mode but gi
  const FHEGasConsumedTransferFrom = getFHEGasFromTxReceipt(t3);
  console.log("FHEGas Consumed during transfer", FHEGasConsumedTransfer
}
// contrarily to FHEGas, native gas in mocked mode slightly differs fro
console.log("Native Gas Consumed during transfer", t3.gasUsed);

}); });
```

File: ./modules/hardhat/test/confidentialERC20/ConfidentialERC20.ts

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

import { createInstance } from "../instance"; import { reencryptEuint64 } from "../reencrypt";

import { getSigners, initSigners } from "../signers"; import { debug } from "../utils"; import { deployConfidentialERC20Fixture } from "./ConfidentialERC20.fixture";

describe("ConfidentialERC20", function () { before(async function () { await initSigners(); this.signers = await getSigners(); });

beforeEach(async function () { const contract = await deployConfidentialERC20Fixture(); this.contractAddress = await contract.getAddress(); this.erc20 = contract; this.fhevm = await createInstance(); });

it("should mint the contract", async function () { const transaction = await this.erc20.mint(1000); await transaction.wait();

```
// Reencrypt Alice's balance
const balanceHandleAlice = await this.erc20.balanceOf(this.signers.alic
const balanceAlice = await reencryptEuint64(
  this.signers.alice,
  this.fhevm,
  balanceHandleAlice,
  this.contractAddress,
);

expect(balanceAlice).to.equal(1000);

const totalSupply = await this.erc20.totalSupply();
expect(totalSupply).to.equal(1000);
```

});

it("should transfer tokens between two users", async function () { const transaction = await this.erc20.mint(10000); const t1 = await transaction.wait(); expect(t1?.status).to.eq(1);

```
const input = this.fhevm.createEncryptedInput(this.contractAddress, thi
input.add64(1337);
const encryptedTransferAmount = await input.encrypt();
const tx = await this.erc20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);
const t2 = await tx.wait();
expect(t2?.status).to.eq(1);

// Reencrypt Alice's balance
const balanceHandleAlice = await this.erc20.balanceOf(this.signers.alic
const balanceAlice = await reencryptEuint64(
  this.signers.alice,
  this.fhevm,
  balanceHandleAlice,
  this.contractAddress,
);
expect(balanceAlice).to.equal(10000 - 1337);

// Reencrypt Bob's balance
```

```
const balanceHandleBob = await this.erc20.balanceOf(this.signers.bob);
const balanceBob = await reencryptEuint64(this.signers.bob, this.fhevm,
expect(balanceBob).to.equal(1337);

// on the other hand, Bob should be unable to read Alice's balance
await expect(
  reencryptEuint64(this.signers.bob, this.fhevm, balanceHandleAlice, th
).to.be.rejectedWith("User is not authorized to reencrypt this handle!"

// and should be impossible to call reencrypt if contractAddress === us
await expect(
  reencryptEuint64(this.signers.alice, this.fhevm, balanceHandleAlice,
).to.be.rejectedWith("userAddress should not be equal to contractAddres

});
```

it("should not transfer tokens between two users", async function () { const transaction = await this.erc20.mint(1000); await transaction.wait();

```
const input = this.fhevm.createEncryptedInput(this.contractAddress, thi
input.add64(1337);
const encryptedTransferAmount = await input.encrypt();
const tx = await this.erc20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);
await tx.wait();

const balanceHandleAlice = await this.erc20.balanceOf(this.signers.alic
const balanceAlice = await reencryptEuint64(
  this.signers.alice,
  this.fhevm,
  balanceHandleAlice,
  this.contractAddress,
);
expect(balanceAlice).to.equal(1000);

// Reencrypt Bob's balance
const balanceHandleBob = await this.erc20.balanceOf(this.signers.bob);
const balanceBob = await reencryptEuint64(this.signers.bob, this.fhevm,
expect(balanceBob).to.equal(0);

});
```

it("should be able to transferFrom only if allowance is sufficient", async function () { const transaction = await this.erc20.mint(10000); await transaction.wait();

```
const inputAlice = this.fhevm.createEncryptedInput(this.contractAddress
inputAlice.add64(1337);
const encryptedAllowanceAmount = await inputAlice.encrypt();
const tx = await this.erc20["approve(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedAllowanceAmount.handles[0],
```

```
  encryptedAllowanceAmount.inputProof,
);
await tx.wait();

const bobErc20 = this.erc20.connect(this.signers.bob);
const inputBob1 = this.fhevm.createEncryptedInput(this.contractAddress,
inputBob1.add64(1338); // above allowance so next tx should actually no
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 tx2.wait();

// Decrypt Alice's balance
const balanceHandleAlice = await this.erc20.balanceOf(this.signers.alic
const balanceAlice = await reencryptEuint64(
  this.signers.alice,
  this.fhevm,
  balanceHandleAlice,
  this.contractAddress,
);
expect(balanceAlice).to.equal(10000); // check that transfer did not ha

// Decrypt Bob's balance
const balanceHandleBob = await this.erc20.balanceOf(this.signers.bob);
const balanceBob = await reencryptEuint64(this.signers.bob, this.fhevm,
expect(balanceBob).to.equal(0); // check that transfer did not happen,

const inputBob2 = this.fhevm.createEncryptedInput(this.contractAddress,
inputBob2.add64(1337); // below allowance so next tx should send token
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
const balanceHandleAlice2 = await this.erc20.balanceOf(this.signers.ali
const balanceAlice2 = await reencryptEuint64(
  this.signers.alice,
  this.fhevm,
  balanceHandleAlice2,
  this.contractAddress,
);
expect(balanceAlice2).to.equal(10000 - 1337); // check that transfer di

// Decrypt Bob's balance
```

```
const balanceHandleBob2 = await this.erc20.balanceOf(this.signers.bob);
const balanceBob2 = await reencryptEuint64(this.signers.bob, this.fhevm
expect(balanceBob2).to.equal(1337); // check that transfer did happen t

});

it("DEBUG - using debug.decrypt64 for debugging transfer", async function () { if (network.name === "hardhat") { // using the debug.decryptXX functions is possible only in mocked mode

  const transaction = await this.erc20.mint(1000);
  await transaction.wait();
  const input = this.fhevm.createEncryptedInput(this.contractAddress, t
  input.add64(1337);
  const encryptedTransferAmount = await input.encrypt();
  const tx = await this.erc20["transfer(address,bytes32,bytes)"](
    this.signers.bob.address,
    encryptedTransferAmount.handles[0],
    encryptedTransferAmount.inputProof,
  );
  await tx.wait();

  const balanceHandleAlice = await this.erc20.balanceOf(this.signers.al
  const balanceAlice = await debug.decrypt64(balanceHandleAlice);
  expect(balanceAlice).to.equal(1000);

  // Reencrypt Bob's balance
  const balanceHandleBob = await this.erc20.balanceOf(this.signers.bob)
  const balanceBob = await debug.decrypt64(balanceHandleBob);
  expect(balanceBob).to.equal(0);
}

}); });
```

File: ./modules/hardhat/test/gatewayDecrypt/testAsyncDecrypt.ts

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

import { awaitAllDecryptionResults, initGateway } from "../asyncDecrypt"; import { createInstance } from "../instance"; import { getSigners, initSigners } from "../signers"; import { bigIntToBytes64, bigIntToBytes128, bigIntToBytes256 } from "../utils";

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

beforeEach(async function () { const contractFactory = await ethers.getContractFactory("TestAsyncDecrypt"); this.contract = await contractFactory.connect(this.signers.alice).deploy(); await this.contract.waitForDeployment(); this.contractAddress = await this.contract.getAddress(); this.fhevm = await createInstance(); });

```
it("test async decrypt bool", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestBool(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yBool(); expect(y).to.equal(true);
});

it("test async decrypt bool trustless", async function () { const tx2 = await
this.contract.requestBoolTrustless(); await tx2.wait(); await awaitAllDecryptionResults();
const y = await this.contract.yBool(); expect(y).to.equal(true); });

it("test async decrypt bool would fail if maxTimestamp is above 1 day", async function () {
await
expect(this.contract.connect(this.signers.carol).requestBoolAboveDelay()).to.be.revertedWith(
"maxTimestamp exceeded MAX_DELAY", ); });

it("test async decrypt uint4", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint4(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint4(); expect(y).to.equal(4);
});

it("test async decrypt uint8", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint8(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint8(); expect(y).to.equal(42);
});

it("test async decrypt uint16", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint16(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint16(); expect(y).to.equal(16);
});

it("test async decrypt uint32", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint32(5, 15); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint32(); expect(y).to.equal(52);
// 5+15+32 });

it("test async decrypt uint64", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint64(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint64();
expect(y).to.equal(18446744073709551600n); });

it("test async decrypt uint128", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint128(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint128();
expect(y).to.equal(126765060022822940149670320544 3n); });

it("test async decrypt uint128 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.add128(184467440737095500429401496n); const encryptedAmount = await
inputAlice.encrypt(); const tx = await
this.contract.requestUint128NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yUint128(); expect(y).to.equal(184467440737095500429401496n); });

it("test async decrypt uint256", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestUint256(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yUint256();
```

```
expect(y).to.equal(276069853871622551497390234491081018098044358886815462206500
});

it("test async decrypt uint256 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.add256(69853871622551497390234491081018098044358886815462n); const
encryptedAmount = await inputAlice.encrypt(); const tx = await
this.contract.requestUint256NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yUint256();
expect(y).to.equal(69853871622551497390234491081018098044358886815462n); });

it("test async decrypt address", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestAddress(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yAddress();
expect(y).to.equal("0x8ba1f109551bD432803012645Ac136ddd64DBA72"); });

it("test async decrypt several addresses", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestSeveralAddresses(); await tx2.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yAddress(); const y2 = await
this.contract.yAddress2();
expect(y).to.equal("0x8ba1f109551bD432803012645Ac136ddd64DBA72");
expect(y2).to.equal("0xf48b8840387ba3809DAE990c930F3b4766A86ca3"); });

it("test async decrypt mixed", async function () { const tx2 = await
this.contract.connect(this.signers.carol).requestMixed(5, 15); await tx2.wait(); await
awaitAllDecryptionResults(); const yB = await this.contract.yBool();
expect(yB).to.equal(true); let y = await this.contract.yUint4(); expect(y).to.equal(4); y =
await this.contract.yUint8(); expect(y).to.equal(42); y = await this.contract.yUint16();
expect(y).to.equal(16); const yAdd = await this.contract.yAddress();
expect(yAdd).to.equal("0x8ba1f109551bD432803012645Ac136ddd64DBA72"); y = await
this.contract.yUint32(); expect(y).to.equal(52); // 5+15+32 y = await
this.contract.yUint64(); expect(y).to.equal(18446744073709551600n); });

it("test async decrypt uint64 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.add64(18446744073709550042n); const encryptedAmount = await
inputAlice.encrypt(); const tx = await
this.contract.requestUint64NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yUint64(); expect(y).to.equal(18446744073709550042n); });

it("test async decrypt ebytes64 trivial", async function () { const tx = await
this.contract.requestEbytes64Trivial("0x78685689"); await tx.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yBytes64();
expect(y).to.equal(ethers.toBeHex(BigInt("0x78685689"), 64)); });

it("test async decrypt ebytes64 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.addBytes64(
bigIntToBytes64(98870780878070870878787887072921111299111111000000292928818818
); const encryptedAmount = await inputAlice.encrypt(); const tx = await
this.contract.requestEbytes64NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
```

```
await this.contract.yBytes64(); expect(y).to.equal(
ethers.toBeHex(9887078087807087087878788707292111129911111100000029292881881881
64), ); });

it("test async decrypt ebytes128 trivial", async function () { const tx = await
this.contract.requestEbytes128Trivial(
"0x8701d11594415047dfac2d9cb87e6631df5a735a2f364fba1511fa7b812dfad2972b809b80ff25
); await tx.wait(); await awaitAllDecryptionResults(); const y = await
this.contract.yBytes128(); expect(y).to.equal( ethers.toBeHex( BigInt(
"0x8701d11594415047dfac2d9cb87e6631df5a735a2f364fba1511fa7b812dfad2972b809b80ff25
), 128, ), ); });

it("test async decrypt ebytes128 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.addBytes128( bigIntToBytes128(
9887078087807087087878788707292111129911111100000029292881881881881822111211
), ); const encryptedAmount = await inputAlice.encrypt(); const tx = await
this.contract.requestEbytes128NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yBytes128(); expect(y).to.equal( ethers.toBeHex(
9887078087807087087878788707292111129911111100000029292881881881881822111211
128, ), ); });

it("test async decrypt ebytes256 trivial", async function () { const tx = await
this.contract.requestEbytes256Trivial("0x78685689"); await tx.wait(); await
awaitAllDecryptionResults(); const y = await this.contract.yBytes256();
expect(y).to.equal(ethers.toBeHex(BigInt("0x78685689"), 256)); });

it("test async decrypt ebytes256 non-trivial", async function () { const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.addBytes256(bigIntToBytes256(18446744073709550022n)); const
encryptedAmount = await inputAlice.encrypt(); const tx = await
this.contract.requestEbytes256NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yBytes256();
expect(y).to.equal(ethers.toBeHex(18446744073709550022n, 256)); });

it("test async decrypt ebytes256 non-trivial with snapshot [skip-on-coverage]", async function
() { if (network.name === "hardhat") { // calling evm_snapshot is possible only in mocked
mode this.snapshotId = await ethers.provider.send("evm_snapshot"); const inputAlice =
this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address);
inputAlice.addBytes256(bigIntToBytes256(18446744073709550022n)); const
encryptedAmount = await inputAlice.encrypt(); const tx = await
this.contract.requestEbytes256NonTrivial(encryptedAmount.handles[0],
encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y =
await this.contract.yBytes256();
expect(y).to.equal(ethers.toBeHex(18446744073709550022n, 256));

  await ethers.provider.send("evm_revert", [this.snapshotId]);
  const inputAlice2 = this.fhevm.createEncryptedInput(this.contractAddr
  inputAlice2.addBytes256(bigIntToBytes256(424242n));
  const encryptedAmount2 = await inputAlice2.encrypt();
  const tx2 = await this.contract.requestEbytes256NonTrivial(
    encryptedAmount2.handles[0],
```

```
    encryptedAmount2.inputProof,
  );
  await tx2.wait();
  await awaitAllDecryptionResults();
  const y2 = await this.contract.yBytes256();
  expect(y2).to.equal(ethers.toBeHex(424242n, 256));
}

});
```

it("test async decrypt mixed with ebytes256", async function () { const inputAlice = this.fhevm.createEncryptedInput(this.contractAddress, this.signers.alice.address); inputAlice.addBytes256(bigIntToBytes256(18446744073709550032n)); const encryptedAmount = await inputAlice.encrypt(); const tx = await this.contract.requestMixedBytes256(encryptedAmount.handles[0], encryptedAmount.inputProof); await tx.wait(); await awaitAllDecryptionResults(); const y = await this.contract.yBytes256(); expect(y).to.equal(ethers.toBeHex(18446744073709550032n, 256)); const y2 = await this.contract.yBytes64(); expect(y2).to.equal(ethers.toBeHex(BigInt("0xaaff42"), 64)); const yb = await this.contract.yBool(); expect(yb).to.equal(true); const yAdd = await this.contract.yAddress(); expect(yAdd).to.equal("0x8ba1f109551bD432803012645Ac136ddd64DBA72"); });

it("test async decrypt ebytes256 non-trivial trustless", async function () { const inputAlice = this.fhevm.createEncryptedInput(await this.contract.getAddress(), this.signers.alice.address); inputAlice.addBytes256(bigIntToBytes256(18446744073709550022n)); const encryptedAmount = await inputAlice.encrypt(); const tx = await this.contract.requestEbytes256NonTrivialTrustless(encryptedAmount.handles[0], encryptedAmount.inputProof,); await tx.wait(); await awaitAllDecryptionResults(); const y = await this.contract.yBytes256(); expect(y).to.equal(ethers.toBeHex(18446744073709550022n, 256)); });

it("test async decrypt mixed with ebytes256 trustless", async function () { const inputAlice = this.fhevm.createEncryptedInput(await this.contract.getAddress(), this.signers.alice.address); inputAlice.addBytes256(bigIntToBytes256(18446744073709550032n)); const encryptedAmount = await inputAlice.encrypt(); const tx = await this.contract.requestMixedBytes256Trustless(encryptedAmount.handles[0], encryptedAmount.inputProof,); await tx.wait(); await awaitAllDecryptionResults(); const y = await this.contract.yBytes256(); expect(y).to.equal(ethers.toBeHex(18446744073709550032n, 256)); const yb = await this.contract.yBool(); expect(yb).to.equal(true); const yAdd = await this.contract.yAddress(); expect(yAdd).to.equal("0x8ba1f109551bD432803012645Ac136ddd64DBA72"); }); });