v0.9

Overview

Welcome to Solidity Guides!

This section will guide you through writing confidential smart contracts in Solidity using the FHEVM library. With Fully Homomorphic Encryption(FHE), your contracts can operate directly on encrypted data without ever decrypting it onchain.

Where to go next

If you're new to the Zama Protocol, start with the Litepaper ↗ or the Protocol Overview ↗ to understand the foundations.

Otherwise:

🟧 Go to **What is FHEVM** to learn about the core concepts and features.

🟧 Go to **Quick Start Tutorial** to build and test your first confidential smart contract.

🟧 Go to **Smart Contract Guides** for details on encrypted types, supported operations, inputs, ACL, and decryption flows.

🟧 Go to **Development Guides** to set up your local environment with Hardhat or Foundry and deploy FHEVM contracts.

🟧 Go to **Migration Guide** if you're upgrading from a previous version to v0.7.

Help center

Ask technical questions and discuss with the community.

- Community forum ↗
- Discord channel ↗

Getting Started

What is FHEVM Solidity

This document provides an overview of key features of the FHEVM smart contract library.

Configuration and initialization

Smart contracts using FHEVM require proper configuration and initialization:

- **Environment setup**: Import and inherit from environment-specific configuration contracts
- **Relayer configuration**: Configure secure relayer access for cryptographic operations
- **Initialization checks**: Validate encrypted variables are properly initialized before use

For more information see Configuration.

Encrypted data types

FHEVM introduces encrypted data types compatible with Solidity:

- **Booleans**: `ebool`
- **Unsigned Integers**: `euint8`, `euint16`, `euint32`, `euint64`, `euint128`, `euint256`
- **Addresses**: `eaddress`
- **Input**: `externalEbool`, `externalEaddress`, `externalEuintXX` for handling encrypted input data

Encrypted data is represented as ciphertext handles, ensuring secure computation and interaction.

For more information see use of encrypted types.

Casting types

fhevm provides functions to cast between encrypted types:

- **Casting between encrypted types**: `FHE.asEbool` converts encrypted integers to encrypted booleans
- **Casting to encrypted types**: `FHE.asEuintX` converts plaintext values to encrypted types
- **Casting to encrypted addresses**: `FHE.asEaddress` converts plaintext addresses to encrypted addresses

For more information see use of encrypted types.

Confidential computation

fhevm enables symbolic execution of encrypted operations, supporting:

- **Arithmetic:** `FHE.add`, `FHE.sub`, `FHE.mul`, `FHE.min`, `FHE.max`, `FHE.neg`, `FHE.div`, `FHE.rem`
  - Note: `div` and `rem` operations are supported only with plaintext divisors
- **Bitwise:** `FHE.and`, `FHE.or`, `FHE.xor`, `FHE.not`, `FHE.shl`, `FHE.shr`, `FHE.rotl`, `FHE.rotr`
- **Comparison:** `FHE.eq`, `FHE.ne`, `FHE.lt`, `FHE.le`, `FHE.gt`, `FHE.ge`
- **Advanced:** `FHE.select` for branching on encrypted conditions, `FHE.randEuintX` for on-chain randomness.

For more information on operations, see Operations on encrypted types.

For more information on conditional branching, see Conditional logic in FHE.

For more information on random number generation, see Generate Random Encrypted Numbers.

Access control mechanism

fhevm enforces access control with a blockchain-based Access Control List (ACL):

- **Persistent access**: `FHE.allow`, `FHE.allowThis` grants permanent permissions for ciphertexts.

- **Transient access**: `FHE.allowTransient` provides temporary access for specific transactions.
- **Validation**: `FHE.isSenderAllowed` ensures that only authorized entities can interact with ciphertexts.

For more information see ACL.

Set up Hardhat

In this section, you'll learn how to set up a FHEVM Hardhat development environment using the **FHEVM Hardhat template** as a starting point for building and testing fully homomorphic encrypted smart contracts.

Create a local Hardhat Project

1 **Install a Node.js TLS version**

Ensure that Node.js is installed on your machine.

- Download and install the recommended LTS (Long-Term Support) version from the official website ↗.
- Use an **even-numbered** version (e.g., `v18.x`, `v20.x`)

> **Hardhat** does not support odd-numbered Node.js versions. If you're using one (e.g., v21.x, v23.x), Hardhat will display a persistent warning message and may behave unexpectedly.

To verify your installation:

```
node -v
npm -v
```

2 **Create a new GitHub repository from the FHEVM Hardhat template.**

1. On GitHub, navigate to the main page of the FHEVM Hardhat template ↗ repository.
2. Above the file list, click the green **Use this template** button.
3. Follow the instructions to create a new repository from the FHEVM Hardhat template.

> See Github doc: Creating a repository from a template ↗

3 Clone your newly created GitHub repository locally

Now that your GitHub repository has been created, you can clone it to your local machine:

```
cd <your-preferred-location>
git clone <url-to-your-new-repo>

# Navigate to the root of your new FHEVM Hardhat project
cd <your-new-repo-name>
```

Next, let's install your local Hardhat development environment.

4 Install your FHEVM Hardhat project dependencies

From the project root directory, run:

```
npm install
```

This will install all required dependencies defined in your `package.json`, setting up your local FHEVM Hardhat development environment.

5 Set up the Hardhat configuration variables (optional)

If you do plan to deploy to the Sepolia Ethereum Testnet, you'll need to set up the following Hardhat Configuration variables ↗.

`MNEMONIC`

A mnemonic is a 12-word seed phrase used to generate your Ethereum wallet keys.

1. Get one by creating a wallet with MetaMask ↗, or using any trusted mnemonic generator.
2. Set it up in your Hardhat project:

```
npx hardhat vars set MNEMONIC
```

`INFURA_API_KEY`

The INFURA project key allows you to connect to Ethereum testnets like Sepolia.

1. Obtain one by following the Infura + MetaMask ↗ setup guide.
2. Configure it in your project:

```
npx hardhat vars set INFURA_API_KEY
```

Default Values

If you skip this step, Hardhat will fall back to these defaults:

- `MNEMONIC` = "test test test test test test test test test test test junk"
- `INFURA_API_KEY` = "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"

> These defaults are not suitable for real deployments.

> **Missing variable error:**
>
> If any of the requested Hardhat Configuration Variables is missing, you'll get an error message like this one: `Error HH1201: Cannot find a value for the configuration variable 'MNEMONIC'. Use 'npx hardhat vars set MNEMONIC' to set it or 'npx hardhat var setup' to list all the configuration variables used by this project.`

Congratulations! You're all set to start building your confidential dApp.

Optional settings

Install VSCode extensions

If you're using Visual Studio Code, there are some extensions available to improve you your development experience:

- Prettier - Code formatter by prettier.io ↗ — ID: `esbenp.prettier-vscode`,
- ESLint by Microsoft ↗ — ID: `dbaeumer.vscode-eslint`

Solidity support (pick one only):

- Solidity by Juan Blanco ↗ — ID: `juanblanco.solidity`
- Solidity by Nomic Foundation ↗ — ID: `nomicfoundation.hardhat-solidity`

Reset the Hardhat project

If you'd like to start from a clean slate, you can reset your FHEVM Hardhat project by removing all example code and generated files.

```
# Navigate to the root of your new FHEVM Hardhat project
cd <your-new-repo-name>
```

Then run:

```
rm -rf test/* src/* tasks/* deploy ./fhevmTemp ./artifacts ./cache
./coverage ./types ./coverage.json ./dist
```

Quick start tutorial

This tutorial guides you to start quickly with Zama's **Fully Homomorphic Encryption (FHE)** technology for building confidential smart contracts.

What You'll Learn

In **about 30 minutes**, you'll go from a basic Solidity contract to a fully confidential one using **FHEVM**. Here's what you'll do:

1. Set up your development environment
2. Write a simple Solidity smart contract
3. Convert it into an FHEVM-compatible confidential contract
4. Test your FHEVM-compatible confidential contract

Prerequisite

- A basic understanding of **Solidity** library and **Ethereum**.
- Some familiarity with **Hardhat.**

> **About Hardhat**
>
> Hardhat ↗ is a development environment for compiling, deploying, testing, and debugging Ethereum smart contracts. It's widely used in the Ethereum ecosystem.
>
> In this tutorial, we'll introduce the FHEVM hardhat template that provides an easy way to use FHEVM.

2. Write a simple contract

In this tutorial, you'll write and test a simple regular Solidity smart contract within the FHEVM Hardhat template to get familiar with Hardhat workflow.

In the next tutorial, you'll learn how to convert this contract into an FHEVM contract.

Prerequisite

- Set up your Hardhat envrionment.
- Make sure that you Hardhat project is clean and ready to start. See the instructions here.

What you'll learn

By the end of this tutorial, you will learn to:

- Write a minimal Solidity contract using Hardhat.
- Test the contract using TypeScript and Hardhat's testing framework.

Write a simple contract

1 Create `Counter.sol`

Go to your project's `contracts` directory:

```
cd <your-project-root-directory>/contracts
```

From there, create a new file named `Counter.sol` and copy/paste the following Solidity code in it.

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

/// @title A simple counter contract
contract Counter {
  uint32 private _count;

  /// @notice Returns the current count
  function getCount() external view returns (uint32) {
    return _count;
  }

  /// @notice Increments the counter by a specific value
  function increment(uint32 value) external {
    _count += value;
  }

  /// @notice Decrements the counter by a specific value
  function decrement(uint32 value) external {
    require(_count >= value, "Counter: cannot decrement below
zero");
    _count -= value;
  }
}
```

2 Compile `Counter.sol`

From your project's root directory, run:

```
npx hardhat compile
```

Great! Your Smart Contract is now compiled.

Set up the testing environment

1 Create a test script `test/Counter.ts`

Go to your project's `test` directory

```
cd <your-project-root-directory>/test
```

From there, create a new file named `Counter.ts` and copy/paste the following Typescript skeleton code in it.

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

describe("Counter", function () {
  it("empty test", async function () {
    console.log("Cool! The test basic skeleton is running!");
  });
});
```

The file contains the following:

- all the required `import` statements we will need during the various tests
- The `chai` basic statements to run a first empty test named `empty test`

2 Run the test `test/Counter.ts`

From your project's root directory, run:

```
npx hardhat test
```

Output:

```
  Counter
Cool! The test basic skeleton is running!
    ✓ empty test


  1 passing (1ms)
```

Great! Your Hardhat test environment is properly setup.

3 Set up the test signers

Before interacting with smart contracts in Hardhat tests, we need to initialize signers.

> In the context of Ethereum development, a signer represents an entity (usually a wallet) that can send transactions and sign messages. In Hardhat, `ethers.getSigners()` returns a list of pre-funded test accounts.

We'll define three named signers for convenience:

- `owner` — the deployer of the contract
- `alice` and `bob` — additional simulated users

Replace the contents of `test/Counter.ts` with the following:

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

type Signers = {
  owner: HardhatEthersSigner;
  alice: HardhatEthersSigner;
  bob: HardhatEthersSigner;
};

describe("Counter", function () {
  let signers: Signers;

  before(async function () {
    const ethSigners: HardhatEthersSigner[] = await
ethers.getSigners();
    signers = { owner: ethSigners[0], alice: ethSigners[1], bob:
ethSigners[2] };
  });

  it("should work", async function () {
    console.log(`address of user owner is
${signers.owner.address}`);
    console.log(`address of user alice is
${signers.alice.address}`);
    console.log(`address of user bob is ${signers.bob.address}`);
  });
});
```

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  Counter
address of user owner is
0x37AC010c1c566696326813b840319B58Bb5840E4
address of user alice is
0xD9F9298BbcD72843586e7E08DAe577E3a0aC8866
address of user bob is 0x3f0CdAe6ebd93F9F776BCBB7da1D42180cC8fcC1
    ✓ should work


  1 passing (2ms)
```

4 Set up testing instance

Now that we have our signers set up, we can deploy the smart contract.

To ensure isolated and deterministic tests, we should deploy a fresh instance of `Counter.sol` before each test. This avoids any side effects from previous tests.

The standard approach is to define a `deployFixture()` function that handles contract deployment.

```
async function deployFixture() {
  const factory = (await ethers.getContractFactory("Counter")) as
Counter__factory;
  const counterContract = (await factory.deploy()) as Counter;
  const counterContractAddress = await
counterContract.getAddress();

  return { counterContract, counterContractAddress };
}
```

To run this setup before each test case, call `deployFixture()` inside a `beforeEach` block:

```
beforeEach(async () => {
  ({ counterContract, counterContractAddress } = await
deployFixture());
});
```

This ensures each test runs with a clean, independent contract instance.

Let's put it together. Now your `test/Counter.ts` should look like the following:

```
import { Counter, Counter__factory } from "../types";
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-
ethers/signers";
import { expect } from "chai";
import { ethers } from "hardhat";

type Signers = {
  deployer: HardhatEthersSigner;
  alice: HardhatEthersSigner;
  bob: HardhatEthersSigner;
};

async function deployFixture() {
  const factory = (await ethers.getContractFactory("Counter")) as
Counter__factory;
  const counterContract = (await factory.deploy()) as Counter;
  const counterContractAddress = await
counterContract.getAddress();

  return { counterContract, counterContractAddress };
}

describe("Counter", function () {
  let signers: Signers;
  let counterContract: Counter;
  let counterContractAddress: Counter;

  before(async function () {
    const ethSigners: HardhatEthersSigner[] = await
ethers.getSigners();
    signers = { deployer: ethSigners[0], alice: ethSigners[1],
bob: ethSigners[2] };
  });

  beforeEach(async () => {
    // Deploy a new instance of the contract before each test
    ({ counterContract, counterContractAddress } = await
deployFixture());
  });

  it("should be deployed", async function () {
    console.log(`Counter has been deployed at address
${counterContractAddress}`);
    // Test the deployed address is valid
    expect(ethers.isAddress(counterContractAddress)).to.eq(true);
  });
});
```

Run the test:

From your project's root directory, run:

```
npx hardhat test
```

Expected Output:

```
  Counter
Counter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed


  1 passing (7ms)
```

Test functions

Now everything is up and running, you can start testing your contract functions.

1 Call the contract `getCount()` view function

Everything is up and running, we can now call the `Counter.sol` view function `getCount()` !

Just below the test block `it("should be deployed", async function () {...}`,

add the following unit test:

```
it("count should be zero after deployment", async function () {
  const count = await counterContract.getCount();
  console.log(`Counter.getCount() === ${count}`);
  // Expect initial count to be 0 after deployment
  expect(count).to.eq(0);
});
```

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  Counter
Counter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
Counter.getCount() === 0
    ✓ count should be zero after deployment


  1 passing (7ms)
```

2 Call the contract `increment()` transaction function

Just below the test block `it("count should be zero after deployment", async function () {...}`, add the following test block:

```
it("increment the counter by 1", async function () {
  const countBeforeInc = await counterContract.getCount();
  const tx = await
counterContract.connect(signers.alice).increment(1);
  await tx.wait();
  const countAfterInc = await counterContract.getCount();
  expect(countAfterInc).to.eq(countBeforeInc + 1n);
});
```

Remarks:

- `increment()` is a transactional function that modifies the blockchain state.
- It must be signed by a user — here we use `alice`.
- `await wait()` to wait for the transaction to mined.
- The test compares the counter before and after the transaction to ensure it incremented as expected.

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  Counter
Counter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
Counter.getCount() === 0
    ✓ count should be zero after deployment
    ✓ increment the counter by 1


  2 passing (12ms)
```

3 Call the contract `decrement()` transaction function

Just below the test block `it("increment the counter by 1", async function () {...}`,

add the following test block:

```
it("decrement the counter by 1", async function () {
  // First increment, count becomes 1
  let tx = await
counterContract.connect(signers.alice).increment(1);
  await tx.wait();
  // Then decrement, count goes back to 0
  tx = await counterContract.connect(signers.alice).decrement(1);
  await tx.wait();
  const count = await counterContract.getCount();
  expect(count).to.eq(0);
});
```

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  Counter
Counter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
Counter.getCount() === 0
    ✓ count should be zero after deployment
    ✓ increment the counter by 1
    ✓ decrement the counter by 1


  2 passing (12ms)
```

Now you have successfully written and tested your counter contract. You should have the following files in your project:

- `contracts/Counter.sol` ↗ — your Solidity smart contract
- `test/Counter.ts` ↗ — your Hardhat test suite written in TypeScript

These files form the foundation of a basic Hardhat-based smart contract project.

Next step

Now that you've written and tested a basic Solidity smart contract, you're ready to take the next step.

In the next tutorial, we'll transform this standard `Counter.sol` contract into `FHECounter.sol`, a trivial FHEVM-compatible version — allowing the counter value to be stored and updated using trivial fully homomorphic encryption.

3. Turn it into FHEVM

In this tutorial, you'll learn how to take a basic Solidity smart contract and progressively upgrade it to support Fully Homomorphic Encryption using the FHEVM library by Zama.

Starting with the plain `Counter.sol` contract that you built from the "Write a simple contract" tutorial, and step-by-step, you'll learn how to:

- Replace standard types with encrypted equivalents
- Integrate zero-knowledge proof validation
- Enable encrypted on-chain computation
- Grant permissions for secure off-chain decryption

By the end, you'll have a fully functional smart contract that supports FHE computation.

Initiate the contract

1 Create the `FHECounter.sol` file

Navigate to your project's `contracts` directory:

```
cd <your-project-root-directory>/contracts
```

From there, create a new file named `FHECounter.sol`, and copy the following Solidity code into it:

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

/// @title A simple counter contract
contract Counter {
  uint32 private _count;

  /// @notice Returns the current count
  function getCount() external view returns (uint32) {
    return _count;
  }

  /// @notice Increments the counter by a specific value
  function increment(uint32 value) external {
    _count += value;
  }

  /// @notice Decrements the counter by a specific value
  function decrement(uint32 value) external {
    require(_count >= value, "Counter: cannot decrement below
zero");
    _count -= value;
  }
}
```

This is a plain `Counter` contract that we'll use as the starting point for adding FHEVM functionality. We will modify this contract step-by-step to progressively integrate FHEVM capabilities.

2 Turn `Counter` into `FHECounter`

To begin integrating FHEVM features into your contract, we first need to import the required FHEVM libraries.

Replace the current header

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

With this updated header:

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

import { FHE, euint32, externalEuint32 } from
"@fhevm/solidity/lib/FHE.sol";
import { EthereumConfig } from
"@fhevm/solidity/config/ZamaConfig.sol";
```

These imports:

- **FHE** — the core library to work with FHEVM encrypted types
- **euint32** and **externalEuint32** — encrypted uint32 types used in FHEVM
- **EthereumConfig** — provides the FHEVM configuration for the Ethereum mainnet or Ethereum Sepolia testnet networks.
  Inheriting from it enables your contract to use the FHE library

Replace the current contract declaration:

```
/// @title A simple counter contract
contract Counter {
```

With the updated declaration :

```
/// @title A simple FHE counter contract
contract FHECounter is EthereumConfig {
```

This change:

- Renames the contract to `FHECounter`
- Inherits from `EthereumConfig` to enable FHEVM support

> This contract must inherit from the `EthereumConfig` abstract contract; otherwise, it will not be able to execute any FHEVM-related functionality on Sepolia or Hardhat.

From your project's root directory, run:

```
npx hardhat compile
```

Great! Your smart contract is now compiled and ready to use **FHEVM features.**

Apply FHE functions and types

1 Comment out the `increment()` and `decrement()` Functions

Before we move forward, let's comment out the `increment()` and `decrement()` functions in `FHECounter`. We'll replace them later with updated versions that support FHE-encrypted operations.

```
 /// @notice Increments the counter by a specific value
// function increment(uint32 value) external {
//     _count += value;
// }

/// @notice Decrements the counter by a specific value
// function decrement(uint32 value) external {
//     require(_count >= value, "Counter: cannot decrement below
zero");
//     _count -= value;
// }
```

2 Replace `uint32` with the FHEVM `euint32` Type

We'll now switch from the standard Solidity `uint32` type to the encrypted FHEVM type `euint32`.

This enables private, homomorphic computation on encrypted integers.

Replace

```
uint32 _count;
```

and

```
function getCount() external view returns (uint32) {
```

With :

```
euint32 _count;
```

and

```
function getCount() external view returns (euint32) {
```

3 Replace `increment(uint32 value)` with the FHEVM version `increment(externalEuint32 value)`

To support encrypted input, we will update the increment function to accept a value encrypted off-chain.

Instead of using a `uint32`, the new version will accept an `externalEuint32`, which is an encrypted integer produced off-chain and sent to the smart contract.

To ensure the validity of this encrypted value, we also include a second argument: `inputProof`, a bytes array containing a Zero-Knowledge Proof of Knowledge (ZKPoK) that proves two things:

1. The `externalEuint32` was encrypted off-chain by the function caller (`msg.sender`)
2. The `externalEuint32` is bound to the contract (`address(this)`) and can only be processed by it.

Replace

```
 /// @notice Increments the counter by a specific value
// function increment(uint32 value) external {
//     _count += value;
// }
```

With :

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  //     _count += value;
}
```

4 Convert `externalEuint32` to `euint32`

You cannot directly use `externalEuint32` in FHE operations. To manipulate it with the FHEVM library, you first need to convert it into the native FHE type `euint32`.

This conversion is done using:

```
FHE.fromExternal(inputEuint32, inputProof);
```

This method verifies the zero-knowledge proof and returns a usable encrypted value within the contract.

Replace

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  //     _count += value;
}
```

With :

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 evalue = FHE.fromExternal(inputEuint32, inputProof);
  //     _count += value;
}
```

5 Convert `_count += value` into its FHEVM equivalent

To perform the update `_count += value` in a Fully Homomorphic way, we use the `FHE.add()` operator. This function allows us to compute the FHE sum of 2 encrypted integers.

Replace

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 evalue = FHE.fromExternal(inputEuint32, inputProof);
  //     _count += value;
}
```

With :

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 evalue = FHE.fromExternal(inputEuint32, inputProof);
  _count = FHE.add(_count, evalue);
}
```

ⓘ This FHE operation allows the smart contract to process encrypted values without ever decrypting them — a core feature of FHEVM that enables on-chain privacy.

Grant FHE Permissions

⚠ This step is critical! You must grant FHE permissions to both the contract and the caller to ensure the encrypted `_count` value can be decrypted off-chain by the caller. Without these 2 permissions, the caller will not be able to compute the clear result.

To grant FHE permission we will call the `FHE.allow()` function.

Replace

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 evalue = FHE.fromExternal(inputEuint32, inputProof);
  _count = FHE.add(_count, evalue);
}
```

With :

```
/// @notice Increments the counter by a specific value
function increment(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 evalue = FHE.fromExternal(inputEuint32, inputProof);
  _count = FHE.add(_count, evalue);

  FHE.allowThis(_count);
  FHE.allow(_count, msg.sender);
}
```

> We grant **two** FHE permissions here — not just one. In the next part of the tutorial, you'll learn why **both** are necessary.

Convert `decrement()` to its FHEVM equivalent

Just like with the `increment()` migration, we'll now convert the `decrement()` function to its FHEVM-compatible version.

Replace :

```
/// @notice Decrements the counter by a specific value
function decrement(uint32 value) external {
  require(_count >= value, "Counter: cannot decrement below zero");
  _count -= value;
}
```

with the following :

```
/// @notice Decrements the counter by a specific value
/// @dev This example omits overflow/underflow checks for simplicity
and readability.
/// In a production contract, proper range checks should be
implemented.
function decrement(externalEuint32 inputEuint32, bytes calldata
inputProof) external {
  euint32 encryptedEuint32 = FHE.fromExternal(inputEuint32,
inputProof);

  _count = FHE.sub(_count, encryptedEuint32);

  FHE.allowThis(_count);
  FHE.allow(_count, msg.sender);
}
```

> The `increment()` and `decrement()` functions do not perform any overflow or underflow checks.

Compile `FHECounter.sol`

From your project's root directory, run:

```
npx hardhat compile
```

Congratulations! Your smart contract is now fully **FHEVM-compatible**.

Now you should have the following files in your project:

- `contracts/FHECounter.sol` ↗ — your Solidity smart FHEVM contract
- `test/FHECounter.ts` ↗ — your FHEVM Hardhat test suite written in TypeScript

In the next tutorial ↗, we'll move on to the **TypeScript integration**, where you'll learn how to interact with your newly upgraded FHEVM contract in a test suite.

4. Test the FHEVM contract

In this tutorial, you'll learn how to migrate a standard Hardhat test suite - from `Counter.ts` to its FHEVM-compatible version `FHECounter.ts` — and progressively enhance it to support Fully Homomorphic Encryption using Zama's FHEVM library.

Set up the FHEVM testing environment

1 Create a test script `test/FHECounter.ts`

Go to your project's `test` directory

```
cd <your-project-root-directory>/test
```

From there, create a new file named `FHECounter.ts` and copy/paste the following Typescript skeleton code in it.

```
import { FHECounter, FHECounter__factory } from "../types";
import { FhevmType } from "@fhevm/hardhat-plugin";
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-
ethers/signers";
import { expect } from "chai";
import { ethers, fhevm } from "hardhat";

type Signers = {
  deployer: HardhatEthersSigner;
  alice: HardhatEthersSigner;
  bob: HardhatEthersSigner;
};

async function deployFixture() {
  const factory = (await ethers.getContractFactory("FHECounter"))
as FHECounter__factory;
  const fheCounterContract = (await factory.deploy()) as
FHECounter;
  const fheCounterContractAddress = await
fheCounterContract.getAddress();

  return { fheCounterContract, fheCounterContractAddress };
}

describe("FHECounter", function () {
  let signers: Signers;
  let fheCounterContract: FHECounter;
  let fheCounterContractAddress: string;

  before(async function () {
    const ethSigners: HardhatEthersSigner[] = await
ethers.getSigners();
    signers = { deployer: ethSigners[0], alice: ethSigners[1],
bob: ethSigners[2] };
  });

  beforeEach(async () => {
    ({ fheCounterContract, fheCounterContractAddress } = await
deployFixture());
  });

  it("should be deployed", async function () {
    console.log(`FHECounter has been deployed at address
${fheCounterContractAddress}`);
    // Test the deployed address is valid

expect(ethers.isAddress(fheCounterContractAddress)).to.eq(true);
  });
```

```
  //   it("count should be zero after deployment", async function
() {
  //     const count = await counterContract.getCount();
  //     console.log(`Counter.getCount() === ${count}`);
  //     // Expect initial count to be 0 after deployment
  //     expect(count).to.eq(0);
  //   });

  //   it("increment the counter by 1", async function () {
  //     const countBeforeInc = await counterContract.getCount();
  //     const tx = await
counterContract.connect(signers.alice).increment(1);
  //     await tx.wait();
  //     const countAfterInc = await counterContract.getCount();
  //     expect(countAfterInc).to.eq(countBeforeInc + 1n);
  //   });

  //   it("decrement the counter by 1", async function () {
  //     // First increment, count becomes 1
  //     let tx = await
counterContract.connect(signers.alice).increment();
  //     await tx.wait();
  //     // Then decrement, count goes back to 0
  //     tx = await
counterContract.connect(signers.alice).decrement(1);
  //     await tx.wait();
  //     const count = await counterContract.getCount();
  //     expect(count).to.eq(0);
  //   });
});
```

What's Different from `Counter.ts`?

- This test file is structurally similar to the original `Counter.ts`, but it uses the FHEVM-compatible smart contract `FHECounter` instead of the regular `Counter`.

– For clarity, the `Counter` unit tests are included as comments, allowing you to better understand how each part is adapted during the migration to FHEVM.

- While the test logic remains the same, this version is now set up to support encrypted computations via the FHEVM library — enabling tests that manipulate confidential values directly on-chain.

2 Run the test `test/FHECounter.ts`

From your project's root directory, run:

```
npx hardhat test
```

Output:

```
  FHECounter
FHECounter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed


  1 passing (1ms)
```

Great! Your Hardhat FHEVM test environment is properly setup.

Test functions

Now everything is up and running, you can start testing your contract functions.

1 Call the contract `getCount()` view function

Replace the commented-out test for the legacy `Counter` contract:

```
//   it("count should be zero after deployment", async function ()
{
//     const count = await counterContract.getCount();
//     console.log(`Counter.getCount() === ${count}`);
//     // Expect initial count to be 0 after deployment
//     expect(count).to.eq(0);
//   });
```

with its FHEVM equivalent:

```
it("encrypted count should be uninitialized after deployment",
async function () {
  const encryptedCount = await fheCounterContract.getCount();
  // Expect initial count to be bytes32(0) after deployment,
  // (meaning the encrypted count value is uninitialized)
  expect(encryptedCount).to.eq(ethers.ZeroHash);
});
```

What's different?

– `encryptedCount` is no longer a plain TypeScript number. It is now a hexadecimal string representing a Solidity `bytes32` value, known as an **FHEVM handle**. This handle points to an encrypted FHEVM primitive of type `euint32`, which internally represents an encrypted Solidity `uint32` primitive type.

- `encryptedCount` is equal to `0x0000000000000000000000000000000000000000000000000000000000000000` which means that `encryptedCount` is uninitialized, and does not reference to any encrypted value at this point.

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  Counter
Counter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
    ✓ encrypted count should be uninitialized after deployment


  2 passing (7ms)
```

2 Setup the `increment()` function unit test

We'll migrate the `increment()` unit test to FHEVM step by step. To start, let's handle the value of the counter before the first increment. As explained above, the counter is initially a `bytes32` value equal to zero, meaning the FHEVM `euint32` variable is uninitialized.

We'll interpret this as if the underlying clear value is 0.

Replace the commented-out test for the legacy `Counter` contract:

```
//   it("increment the counter by 1", async function () {
//     const countBeforeInc = await counterContract.getCount();
//     const tx = await
counterContract.connect(signers.alice).increment(1);
//     await tx.wait();
//     const countAfterInc = await counterContract.getCount();
//     expect(countAfterInc).to.eq(countBeforeInc + 1n);
//   });
```

with the following:

```
it("increment the counter by 1", async function () {
  const encryptedCountBeforeInc = await
fheCounterContract.getCount();
  expect(encryptedCountBeforeInc).to.eq(ethers.ZeroHash);
  const clearCountBeforeInc = 0;

  // const tx = await
counterContract.connect(signers.alice).increment(1);
  // await tx.wait();
  // const countAfterInc = await counterContract.getCount();
  // expect(countAfterInc).to.eq(countBeforeInc + 1n);
});
```

3 Encrypt the `increment()` function argument

The `increment()` function takes a single argument: the value by which the counter should be incremented. In the initial version of `Counter.sol`, this value is a clear `uint32`.

We'll switch to passing an encrypted value instead, using FHEVM `externalEuint32` primitive type. This allows us to securely increment the counter without revealing the input value on-chain.

> We are using an `externalEuint32` instead of a regular `euint32`. This tells the FHEVM that the encrypted `uint32` was provided externally (e.g., by a user) and must be verified for integrity and authenticity before it can be used within the contract.

Replace :

```
it("increment the counter by 1", async function () {
  const encryptedCountBeforeInc = await
fheCounterContract.getCount();
  expect(encryptedCountBeforeInc).to.eq(ethers.ZeroHash);
  const clearCountBeforeInc = 0;

  // const tx = await
counterContract.connect(signers.alice).increment(1);
  // await tx.wait();
  // const countAfterInc = await counterContract.getCount();
  // expect(countAfterInc).to.eq(countBeforeInc + 1n);
});
```

with the following:

```
it("increment the counter by 1", async function () {
  const encryptedCountBeforeInc = await
fheCounterContract.getCount();
  expect(encryptedCountBeforeInc).to.eq(ethers.ZeroHash);
  const clearCountBeforeInc = 0;

  // Encrypt constant 1 as a euint32
  const clearOne = 1;
  const encryptedOne = await fhevm
    .createEncryptedInput(fheCounterContractAddress,
signers.alice.address)
    .add32(clearOne)
    .encrypt();

  // const tx = await
counterContract.connect(signers.alice).increment(1);
  // await tx.wait();
  // const countAfterInc = await counterContract.getCount();
  // expect(countAfterInc).to.eq(countBeforeInc + 1n);
});
```

`fhevm.createEncryptedInput(fheCounterContractAddress, signers.alice.address)` creates an encrypted value that is bound to both the contract (`fheCounterContractAddress`) and the user (`signers.alice.address`). This means only Alice can use this encrypted value, and only within the `FHECounter.sol` contract at that specific address. **It cannot be reused by another user or in a different contract, ensuring data confidentiality and binding context-specific encryption.**

4 Call the `increment()` function with the encrypted argument

Now that we have an encrypted argument, we can call the `increment()` function with it.

Below, you'll notice that the updated `increment()` function now takes **two arguments instead of one.**

This is because the FHEVM requires both:

1. The `externalEuint32` — the encrypted value itself
2. An accompanying **Zero-Knowledge Proof of Knowledge** (`inputProof`) — which verifies that the encrypted input is securely bound to:
   - the caller (Alice, the transaction signer), and
   - the target smart contract (where `increment()` is being executed)

This ensures that the encrypted value cannot be reused in a different context or by a different user, preserving **confidentiality and integrity.**

Replace :

```
// const tx = await
counterContract.connect(signers.alice).increment(1);
// await tx.wait();
```

with the following:

```
const tx = await
fheCounterContract.connect(signers.alice).increment(encryptedOne.h
andles[0], encryptedOne.inputProof);
await tx.wait();
```

At this point the counter has been successfully incremented by 1 using a **Fully Homomorphic Encryption (FHE)**. In the next step, we will retrieve the updated encrypted counter value and decrypt it locally. But before we move on, let's quickly run the tests to make sure everything is working correctly.

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  FHECounter
FHECounter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
    ✓ encrypted count should be uninitialized after deployment
    ✓ increment the counter by 1


  3 passing (7ms)
```

5 Call the `getCount()` function and Decrypt the value

Now that the counter has been incremented using an encrypted input, it's time to **read the updated encrypted value** from the smart contract and **decrypt it** using the `userDecryptEuint` function provided by the FHEVM Hardhat Plugin.

The `userDecryptEuint` function takes four parameters:

1. **FhevmType**: The integer type of the FHE-encrypted value. In this case, we're using `FhevmType.euint32` because the counter is a `uint32`.
2. **Encrypted handle**: A 32-byte FHEVM handle representing the encrypted value you want to decrypt.
3. **Smart contract address**: The address of the contract that has permission to access the encrypted handle.
4. **User signer**: The signer (e.g., signers.alice) who has permission to access the handle.

> Note: Permissions to access the FHEVM handle are set on-chain using the `FHE.allow()` Solidity function (see FHECounter.sol).

Replace :

```
// const countAfterInc = await counterContract.getCount();
// expect(countAfterInc).to.eq(countBeforeInc + 1n);
```

with the following:

```
const encryptedCountAfterInc = await
fheCounterContract.getCount();
const clearCountAfterInc = await fhevm.userDecryptEuint(
  FhevmType.euint32,
  encryptedCountAfterInc,
  fheCounterContractAddress,
  signers.alice,
);
expect(clearCountAfterInc).to.eq(clearCountBeforeInc + clearOne);
```

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  FHECounter
FHECounter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
    ✓ encrypted count should be uninitialized after deployment
    ✓ increment the counter by 1


  3 passing (7ms)
```

6 Call the contract `decrement()` function

Similarly to the previous test, we'll now call the `decrement()` function using an encrypted input.

Replace :

```
//   it("decrement the counter by 1", async function () {
//     // First increment, count becomes 1
//     let tx = await
counterContract.connect(signers.alice).increment();
//     await tx.wait();
//     // Then decrement, count goes back to 0
//     tx = await
counterContract.connect(signers.alice).decrement(1);
//     await tx.wait();
//     const count = await counterContract.getCount();
//     expect(count).to.eq(0);
//   });
```

with the following:

```
it("decrement the counter by 1", async function () {
  // Encrypt constant 1 as a euint32
  const clearOne = 1;
  const encryptedOne = await fhevm
    .createEncryptedInput(fheCounterContractAddress,
signers.alice.address)
    .add32(clearOne)
    .encrypt();

  // First increment by 1, count becomes 1
  let tx = await
fheCounterContract.connect(signers.alice).increment(encryptedOne.h
andles[0], encryptedOne.inputProof);
  await tx.wait();

  // Then decrement by 1, count goes back to 0
  tx = await
fheCounterContract.connect(signers.alice).decrement(encryptedOne.h
andles[0], encryptedOne.inputProof);
  await tx.wait();

  const encryptedCountAfterDec = await
fheCounterContract.getCount();
  const clearCountAfterDec = await fhevm.userDecryptEuint(
    FhevmType.euint32,
    encryptedCountAfterDec,
    fheCounterContractAddress,
    signers.alice,
  );

  expect(clearCountAfterDec).to.eq(0);
});
```

Run the test

From your project's root directory, run:

```
npx hardhat test
```

Expected Output

```
  FHECounter
FHECounter has been deployed at address
0x7553CB9124f974Ee475E5cE45482F90d5B6076BC
    ✓ should be deployed
    ✓ encrypted count should be uninitialized after deployment
    ✓ increment the counter by 1
    ✓ decrement the counter by 1


  4 passing (7ms)
```

Congratulations! You've completed the full tutorial.

You have successfully written and tested your FHEVM-based counter smart contract. By now, your project should include the following files:

- `contracts/FHECounter.sol` ↗ — your Solidity smart contract
- `test/FHECounter.ts` ↗ — your Hardhat test suite written in TypeScript

Next step

If you would like to deploy your project on the testnet, or learn more about using FHEVM Hardhat Plugin, head to Deploy contracts and run tests.

Smart Contract

Configuration

This document explains how to enable encrypted computations in your smart contract by setting up the `fhevm` environment. Learn how to integrate essential libraries, configure encryption, and add secure computation logic to your contracts.

Core configuration setup

To utilize encrypted computations in Solidity contracts, you must configure the **FHE library** and **Oracle addresses**. The `fhevm` package simplifies this process with prebuilt configuration contracts, allowing you to focus on developing your contract's logic without handling the underlying cryptographic setup.

This library and its associated contracts provide a standardized way to configure and interact with Zama's FHEVM (Fully Homomorphic Encryption Virtual Machine) infrastructure on different Ethereum networks. It supplies the necessary contract addresses for Zama's FHEVM components (`ACL`, `FHEVMExecutor`, `KMSVerifier`, `InputVerifier`) and the decryption oracle, enabling seamless integration for Solidity contracts that require FHEVM support.

Key components configured automatically

1. **FHE library**: Sets up encryption parameters and cryptographic keys.
2. **Oracle**: Manages secure cryptographic operations such as public decryption.
3. **Network-specific settings**: Adapts to local testing, testnets (Sepolia for example), or mainnet deployment.

By inheriting these configuration contracts, you ensure seamless initialization and functionality across environments.

ZamaConfig.sol

The `ZamaConfig` library exposes functions to retrieve FHEVM configuration structs and oracle addresses for supported networks (currently only the Sepolia testnet).

Under the hood, this library encapsulates the network-specific addresses of Zama's FHEVM infrastructure into a single struct (`FHEVMConfigStruct`).

EthereumConfig

The `EthereumConfig` contract is designed to be inherited by a user contract. The constructor automatically sets up the FHEVM coprocessor and decryption oracle using the configuration provided by the library for the respective network. When a contract inherits from `EthereumConfig`, the constructor calls `FHE.setCoprocessor` with the appropriate addresses. This ensures that the inheriting contract is automatically wired to the correct FHEVM contracts and oracle for the target network, abstracting away manual address management and reducing the risk of misconfiguration.

Example

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

import { EthereumConfig } from "@fhevm/solidity/config/ZamaConfig.sol";

contract MyERC20 is EthereumConfig {
  constructor() {
    // Additional initialization logic if needed
  }
}
```

Using `isInitialized`

The `isInitialized` utility function checks whether an encrypted variable has been properly initialized, preventing unexpected behavior due to uninitialized values.

Function signature

```
function isInitialized(T v) internal pure returns (bool)
```

Purpose

- Ensures encrypted variables are initialized before use.
- Prevents potential logic errors in contract execution.

Example: Initialization Check for Encrypted Counter

```
require(FHE.isInitialized(counter), "Counter not initialized!");
```

Summary

By leveraging prebuilt a configuration contract like `EthereumConfig` in `ZamaConfig.sol`, you can efficiently set up your smart contract for encrypted computations. These tools abstract the complexity of cryptographic initialization, allowing you to focus on building secure, confidential smart contracts.

Contract addresses

Save this in your `.env` file.

These are Sepolia addresses.

Contract/Service	Address/Value
FHEVM_EXECUTOR_CONTRACT	0x848B0066793BcC60346Da1F49049357399B8D595
ACL_CONTRACT	0x687820221192C5B662b25367F70076A37bc79b6c
HCU_LIMIT_CONTRACT	0x594BB474275918AF9609814E68C61B1587c5F838
KMS_VERIFIER_CONTRACT	0x1364cBBf2cDF5032C47d8226a6f6FBD2AFCDacAC
INPUT_VERIFIER_CONTRACT	0xbc91f3daD1A5F19F8390c400196e58073B6a0BC4
DECRYPTION_ORACLE_CONTRACT	0xa02Cda4Ca3a71D7C46997716F4283aa851C28812
DECRYPTION_ADDRESS	0xb6E160B1ff80D67Bfe90A85eE06Ce0A2613607D1
INPUT_VERIFICATION_ADDRESS	0x7048C39f048125eDa9d678AEbaDfB22F7900a29F
RELAYER_URL	`https://relayer.testnet.zama.cloud`

Supported types

This document introduces the encrypted integer types provided by the `FHE` library in FHEVM and explains their usage, including casting, state variable declarations, and type-specific considerations.

Introduction

The `FHE` library offers a robust type system with encrypted integer types, enabling secure computations on confidential data in smart contracts. These encrypted types are validated both at compile time and runtime to ensure correctness and security.

Key features of encrypted types

- Encrypted integers function similarly to Solidity's native integer types, but they operate on **Fully Homomorphic Encryption (FHE)** ciphertexts.
- Arithmetic operations on `e(u)int` types are **unchecked**, meaning they wrap around on overflow. This design choice ensures confidentiality by avoiding the leakage of information through error detection.
- Future versions of the `FHE` library will support encrypted integers with overflow checking, but with the trade-off of exposing limited information about the operands.

> Encrypted integers with overflow checking will soon be available in the `FHE` library. These will allow reversible arithmetic operations but may reveal some information about the input values.

Encrypted integers in FHEVM are represented as FHE ciphertexts, abstracted using ciphertext handles. These types, prefixed with `e` (for example, `euint64`) act as secure wrappers over the ciphertext handles.

List of encrypted types

The `FHE` library currently supports the following encrypted types:

Type	Bit Length	Supported Operators	Aliases (with supported operators)
Ebool	2	and, or, xor, eq, ne, not, select, rand	
Euint8	8	add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, eq, ne, ge, gt, le, lt, min, max, neg, not, select, rand, randBounded	
Euint16	16	add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, eq, ne, ge, gt, le, lt, min, max, neg, not, select, rand, randBounded	
Euint32	32	add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, eq, ne, ge, gt, le, lt, min, max, neg, not, select, rand, randBounded	
Euint64	64	add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, eq, ne, ge, gt, le, lt, min, max, neg, not, select, rand, randBounded	
Euint128	128	add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, eq, ne, ge, gt, le, lt, min, max, neg, not, select, rand, randBounded	
Euint160	160		Eaddress (eq, ne, select)

Euint256	256	and, or, xor, shl, shr, rotl, rotr, eq, ne, neg, not, select, rand, randBounded	

Division (`div`) and remainder (`rem`) operations are only supported when the right-hand side (`rhs`) operand is a plaintext (non-encrypted) value. Attempting to use an encrypted value as `rhs` will result in a panic. This restriction ensures correct and secure computation within the current framework.

Higher-precision integer types are available in the `TFHE-rs` library and can be added to `fhevm` as needed.

Operations on encrypted types

This document outlines the operations supported on encrypted types in the `FHE` library, enabling arithmetic, bitwise, comparison, and more on Fully Homomorphic Encryption (FHE) ciphertexts.

Arithmetic operations

The following arithmetic operations are supported for encrypted integers (`euintX`):

Name	Function name	Symbol	Type
Add	`FHE.add`	`+`	Binary
Subtract	`FHE.sub`	`-`	Binary
Multiply	`FHE.mul`	`*`	Binary
Divide (plaintext divisor)	`FHE.div`		Binary
Reminder (plaintext divisor)	`FHE.rem`		Binary
Negation	`FHE.neg`	`-`	Unary
Min	`FHE.min`		Binary
Max	`FHE.max`		Binary

Division (FHE.div) and remainder (FHE.rem) operations are currently supported only with plaintext divisors.

Bitwise operations

The FHE library also supports bitwise operations, including shifts and rotations:

Name	Function name	Symbol	Type
Bitwise AND	`FHE.and`	`&`	Binary
Bitwise OR	`FHE.or`	`|`	Binary
Bitwise XOR	`FHE.xor`	`^`	Binary
Bitwise NOT	`FHE.not`	`~`	Unary
Shift Right	`FHE.shr`		Binary
Shift Left	`FHE.shl`		Binary
Rotate Right	`FHE.rotr`		Binary
Rotate Left	`FHE.rotl`		Binary

The shift operators `FHE.shr` and `FHE.shl` can take any encrypted type `euintX` as a first operand and either a `uint8` or a `euint8` as a second operand, however the second operand will always be computed modulo the number of bits of the first operand. For example, `FHE.shr(euint64 x, 70)` is equivalent to `FHE.shr(euint64 x, 6)` because `70 % 64 = 6`. This differs from the classical shift operators in Solidity, where there is no intermediate modulo operation, so for instance any `uint64` shifted right via `>>` would give a null result.

Comparison operations

Encrypted integers can be compared using the following functions:

Name	Function name	Symbol	Type
Equal	`FHE.eq`		Binary
Not equal	`FHE.ne`		Binary
Greater than or equal	`FHE.ge`		Binary
Greater than	`FHE.gt`		Binary
Less than or equal	`FHE.le`		Binary
Less than	`FHE.lt`		Binary

Ternary operation

The `FHE.select` function is a ternary operation that selects one of two encrypted values based on an encrypted condition:

Name	Function name	Symbol	Type
Select	`FHE.select`		Ternary

Random operations

You can generate cryptographically secure random numbers fully on-chain:

Name	Function Name	Symbol	Type
Random Unsigned Integer	`FHE.randEuintX()`		Random

For more details, refer to the Random Encrypted Numbers document.

Best Practices

Here are some best practices to follow when using encrypted operations in your smart contracts:

Use the appropriate encrypted type size

Choose the smallest encrypted type that can accommodate your data to optimize gas costs. For example, use `euint8` for small numbers (0-255) rather than `euint256`.

❌ Avoid using oversized types:

```
// Bad: Using euint256 for small numbers wastes gas
euint64 age = FHE.asEuint128(25);  // age will never exceed 255
euint64 percentage = FHE.asEuint128(75);  // percentage is 0-100
```

✅ Instead, use the smallest appropriate type:

```
// Good: Using appropriate sized types
euint8 age = FHE.asEuint8(25);  // age fits in 8 bits
euint8 percentage = FHE.asEuint8(75);  // percentage fits in 8 bits
```

Use scalar operands when possible to save gas

Some FHE operators exist in two versions: one where all operands are ciphertexts handles, and another where one of the operands is an unencrypted scalar. Whenever possible, use the scalar operand version, as this will save a lot of gas.

❌ For example, this snippet cost way more in gas:

```
euint32 x;
...
x = FHE.add(x,FHE.asEuint(42));
```

✅ Than this one:

```
euint32 x;
// ...
x = FHE.add(x,42);
```

Despite both leading to the same encrypted result!

Beware of overflows of FHE arithmetic operators

FHE arithmetic operators can overflow. Do not forget to take into account such a possibility when implementing FHEVM smart contracts.

❌ For example, if you wanted to create a mint function for an encrypted ERC20 token with an encrypted `totalSupply` state variable, this code is vulnerable to overflows:

```
function mint(externalEuint32 encryptedAmount, bytes calldata
inputProof) public {
  euint32 mintedAmount = FHE.asEuint32(encryptedAmount, inputProof);
  totalSupply = FHE.add(totalSupply, mintedAmount);
  balances[msg.sender] = FHE.add(balances[msg.sender], mintedAmount);
  FHE.allowThis(balances[msg.sender]);
  FHE.allow(balances[msg.sender], msg.sender);
}
```

✅ But you can fix this issue by using `FHE.select` to cancel the mint in case of an overflow:

```
function mint(externalEuint32 encryptedAmount, bytes calldata
inputProof) public {
  euint32 mintedAmount = FHE.asEuint32(encryptedAmount, inputProof);
  euint32 tempTotalSupply = FHE.add(totalSupply, mintedAmount);
  ebool isOverflow = FHE.lt(tempTotalSupply, totalSupply);
  totalSupply = FHE.select(isOverflow, totalSupply, tempTotalSupply);
  euint32 tempBalanceOf = FHE.add(balances[msg.sender], mintedAmount);
  balances[msg.sender] = FHE.select(isOverflow, balances[msg.sender],
tempBalanceOf);
  FHE.allowThis(balances[msg.sender]);
  FHE.allow(balances[msg.sender], msg.sender);
}
```

Notice that we did not check separately the overflow on `balances[msg.sender]` but only on `totalSupply` variable, because `totalSupply` is the sum of the balances of all the users, so `balances[msg.sender]` could never overflow if `totalSupply` did not.

Casting and trivial encryption

This documentation covers the `asEbool`, `asEuintXX`, and `asEaddress` operations provided by the FHE library for working with encrypted data in the FHEVM. These operations are essential for converting between plaintext and encrypted types, as well as handling encrypted inputs.

The operations can be categorized into two main use cases:

1. **Trivial encryption**: Converting plaintext values to encrypted types
2. **Type casting**: Converting between different encrypted types

1. Trivial encryption

Trivial encryption simply put is a plain text in a format of a ciphertext.

Overview

Trivial encryption is the process of converting plaintext values into encrypted types (ciphertexts) compatible with FHE operators. Although the data is in ciphertext format, it remains publicly visible on-chain, making it useful for operations between public and private values.

This type of casting involves converting plaintext (unencrypted) values into their encrypted equivalents, such as:

- `bool` → `ebool`
- `uint` → `euintXX`
- `address` → `eaddress`

> When doing trivial encryption, the data is made compatible with FHE operations but remains publicly visible on-chain unless explicitly encrypted.

Example

```
euint64 value64 = FHE.asEuint64(7262);  // Trivial encrypt a uint64
ebool valueBool = FHE.asEbool(true);   // Trivial encrypt a boolean
```

2. Casting between encrypted types

This type of casting is used to reinterpret or convert one encrypted type into another. For example:

- `euint32` → `euint64`

Casting between encrypted types is often required when working with operations that demand specific sizes or precisions.

> **Important**: When casting between encrypted types:
>
> - Casting from smaller types to larger types (e.g. `euint32` → `euint64`) preserves all information
> - Casting from larger types to smaller types (e.g. `euint64` → `euint32`) will truncate and lose information

The table below summarizes the available casting functions:

From type	To type	Function
`euintX`	`euintX`	`FHE.asEuintXX`
`ebool`	`euintX`	`FHE.asEuintXX`
`euintX`	`ebool`	`FHE.asEboolXX`

> Casting between encrypted types is efficient and often necessary when handling data with differing precision requirements.

Workflow for encrypted types

```
// Casting between encrypted types
euint32 value32 = FHE.asEuint32(value64); // Cast to euint32
ebool valueBool = FHE.asEbool(value32);   // Cast to ebool
```

Overall operation summary

Casting Type	Function	Input Type	Output Type
Trivial encryption	`FHE.asEuintXX(x)`	`uintX`	`euintX`
	`FHE.asEbool(x)`	`bool`	`ebool`
	`FHE.asEaddress(x)`	`address`	`eaddress`
Conversion between types	`FHE.asEuintXX(x)`	`euintXX` / `ebool`	`euintYY`
	`FHE.asEbool(x)`	`euintXX`	`ebool`

Generate random numbers

This document explains how to generate cryptographically secure random encrypted numbers fully on-chain using the `FHE` library in fhevm. These numbers are encrypted and remain confidential, enabling privacy-preserving smart contract logic.

Key notes on random number generation

- **On-chain execution**: Random number generation must be executed during a transaction, as it requires the pseudo-random number generator (PRNG) state to be updated on-chain. This operation cannot be performed using the `eth_call` RPC method.
- **Cryptographic security**: The generated random numbers are cryptographically secure and encrypted, ensuring privacy and unpredictability.

> Random number generation must be performed during transactions, as it requires the pseudo-random number generator (PRNG) state to be mutated on-chain. Therefore, it cannot be executed using the `eth_call` RPC method.

Basic usage

The `FHE` library allows you to generate random encrypted numbers of various bit sizes. Below is a list of supported types and their usage:

```
// Generate random encrypted numbers
ebool rb = FHE.randEbool();       // Random encrypted boolean
euint8 r8 = FHE.randEuint8();     // Random 8-bit number
euint16 r16 = FHE.randEuint16();  // Random 16-bit number
euint32 r32 = FHE.randEuint32();  // Random 32-bit number
euint64 r64 = FHE.randEuint64();  // Random 64-bit number
euint128 r128 = FHE.randEuint128(); // Random 128-bit number
euint256 r256 = FHE.randEuint256(); // Random 256-bit number
```

Example: Random Boolean

```
function randomBoolean() public returns (ebool) {
  return FHE.randEbool();
}
```

Bounded random numbers

To generate random numbers within a specific range, you can specify an **upper bound**. The specified upper bound must be a power of 2. The random number will be in the range `[0, upperBound - 1]`.

```
// Generate random numbers with upper bounds
euint8 r8 = FHE.randEuint8(32);      // Random number between 0-31
euint16 r16 = FHE.randEuint16(512);  // Random number between 0-511
euint32 r32 = FHE.randEuint32(65536); // Random number between 0-65535
```

Example: Random number with upper bound

```
function randomBoundedNumber(uint16 upperBound) public returns
(euint16) {
  return FHE.randEuint16(upperBound);
}
```

Security Considerations

- **Cryptographic security**:
  The random numbers are generated using a cryptographically secure pseudo-random number generator (CSPRNG) and remain encrypted until explicitly decrypted.
- **Gas consumption**:
  Each call to a random number generation function consumes gas. Developers should optimize the use of these functions, especially in gas-sensitive contracts.
- **Privacy guarantee**:
  Random values are fully encrypted, ensuring they cannot be accessed or predicted by unauthorized parties.

Encrypted inputs

This document introduces the concept of encrypted inputs in the FHEVM, explaining their role, structure, validation process, and how developers can integrate them into smart contracts and applications.

Encrypted inputs are a core feature of FHEVM, enabling users to push encrypted data onto the blockchain while ensuring data confidentiality and integrity.

What are encrypted inputs?

Encrypted inputs are data values submitted by users in ciphertext form. These inputs allow sensitive information to remain confidential while still being processed by smart contracts. They are accompanied by **Zero-Knowledge Proofs of Knowledge (ZKPoKs)** to ensure the validity of the encrypted data without revealing the plaintext.

Key characteristics of encrypted inputs:

1. **Confidentiality**: Data is encrypted using the public FHE key, ensuring that only authorized parties can decrypt or process the values.
2. **Validation via ZKPoKs**: Each encrypted input is accompanied by a proof verifying that the user knows the plaintext value of the ciphertext, preventing replay attacks or misuse.
3. **Efficient packing**: All inputs for a transaction are packed into a single ciphertext in a user-defined order, optimizing the size and generation of the zero-knowledge proof.

Parameters in encrypted functions

When a function in a smart contract is called, it may accept two types of parameters for encrypted inputs:

1. `externalEbool`, `externalEaddress`, `externalEuintXX`: Refers to the index of the encrypted parameter within the proof, representing a specific encrypted input handle.
2. `bytes`: Contains the ciphertext and the associated zero-knowledge proof used for validation.

Here’s an example of a Solidity function accepting multiple encrypted parameters:

```
function exampleFunction(
  externalEbool param1,
  externalEuint64 param2,
  externalEuint8 param3,
  bytes calldata inputProof
) public {
  // Function logic here
}
```

In this example, `param1`, `param2`, and `param3` are encrypted inputs for `ebool`, `euint64`, and `euint8` while `inputProof` contains the corresponding ZKPoK to validate their authenticity.

Input Generation using Hardhat

In the below example, we use Alice's address to create the encrypted inputs and submits the transaction.

```
import { fhevm } from "hardhat";

const input = fhevm.createEncryptedInput(contract.address,
signers.alice.address);
input.addBool(canTransfer); // at index 0
input.add64(transferAmount); // at index 1
input.add8(transferType); // at index 2
const encryptedInput = await input.encrypt();

const externalEboolParam1 = encryptedInput.handles[0];
const externalEuint64Param2 = encryptedInput.handles[1];
const externalEuint8Param3 = encryptedInput.handles[2];
const inputProof = encryptedInput.inputProof;

tx = await myContract
  .connect(signers.alice)
  [
    "exampleFunction(bytes32,bytes32,bytes32,bytes)"
  ](signers.bob.address, externalEboolParam1, externalEuint64Param2,
externalEuint8Param3, inputProof);

await tx.wait();
```

Input Order

Developers are free to design the function parameters in any order. There is no required correspondence between the order in which encrypted inputs are constructed in TypeScript and the order of arguments in the Solidity function.

Validating encrypted inputs

Smart contracts process encrypted inputs by verifying them against the associated zero-knowledge proof. This is done using the `FHE.asEuintXX`, `FHE.asEbool`, or `FHE.asEaddress` functions, which validate the input and convert it into the appropriate encrypted type.

Example validation

This example demonstrates a function that performs multiple encrypted operations, such as updating a user's encrypted balance and toggling an encrypted boolean flag:

```
function myExample(externalEuint64 encryptedAmount, externalEbool
encryptedToggle, bytes calldata inputProof) public {
  // Validate and convert the encrypted inputs
  euint64 amount = FHE.fromExternal(encryptedAmount, inputProof);
  ebool toggleFlag = FHE.fromExternal(encryptedToggle, inputProof);

  // Update the user's encrypted balance
  balances[msg.sender] = FHE.add(balances[msg.sender], amount);

  // Toggle the user's encrypted flag
  userFlags[msg.sender] = FHE.not(toggleFlag);

  // FHE permissions and function logic here
  ...
}

// Function to retrieve a user's encrypted balance
function getEncryptedBalance() public view returns (euint64) {
  return balances[msg.sender];
}

// Function to retrieve a user's encrypted flag
function getEncryptedFlag() public view returns (ebool) {
  return userFlags[msg.sender];
}
```

Example validation in the `ConfidentialERC20.sol` smart contract

Here's an example of a smart contract function that verifies an encrypted input before proceeding:

```
function transfer(
  address to,
  externalEuint64 encryptedAmount,
  bytes calldata inputProof
) public {
  // Verify the provided encrypted amount and convert it into an
encrypted uint64
  euint64 amount = FHE.fromExternal(encryptedAmount, inputProof);

  // Function logic here, such as transferring funds
  ...
}
```

How validation works

1. **Input verification**:
   The `FHE.fromExternal` function ensures that the input is a valid ciphertext with a corresponding ZKPoK.
2. **Type conversion**:
   The function transforms `externalEbool`, `externalEaddress`, `externalEuintXX` into the appropriate encrypted type (`ebool`, `eaddress`, `euintXX`) for further operations within the contract.

Best Practices

- **Input packing**: Minimize the size and complexity of zero-knowledge proofs by packing all encrypted inputs into a single ciphertext.
- **Frontend encryption**: Always encrypt inputs using the FHE public key on the client side to ensure data confidentiality.
- **Proof management**: Ensure that the correct zero-knowledge proof is associated with each encrypted input to avoid validation errors.

Encrypted inputs and their validation form the backbone of secure and private interactions in the FHEVM. By leveraging these tools, developers can create robust, privacy-preserving smart contracts without compromising functionality or scalability.

Access Control List

This document describes the Access Control List (ACL) system in FHEVM, a core feature that governs access to encrypted data. The ACL ensures that only authorized accounts or contracts can interact with specific ciphertexts, preserving confidentiality while enabling composable smart contracts. This overview provides a high-level understanding of what the ACL is, why it's essential, and how it works.

What is the ACL?

The ACL is a permission management system designed to control who can access, compute on, or decrypt encrypted values in fhevm. By defining and enforcing these permissions, the ACL ensures that encrypted data remains secure while still being usable within authorized contexts.

Why is the ACL important?

Encrypted data in FHEVM is entirely confidential, meaning that without proper access control, even the contract holding the ciphertext cannot interact with it. The ACL enables:

- **Granular permissions**: Define specific access rules for individual accounts or contracts.
- **Secure computations**: Ensure that only authorized entities can manipulate or decrypt encrypted data.
- **Gas efficiency**: Optimize permissions using transient access for temporary needs, reducing storage and gas costs.

How does the ACL work?

Types of access

- **Permanent allowance**:
  - Configured using `FHE.allow(ciphertext, address)`.

    - Grants long-term access to the ciphertext for a specific address.
    - Stored in a dedicated contract for persistent storage.
- **Transient allowance**:
    - Configured using `FHE.allowTransient(ciphertext, address)`.
    - Grants access to the ciphertext only for the duration of the current transaction.
    - Stored in transient storage, reducing gas costs.
    - Ideal for temporary operations like passing ciphertexts to external functions.
- **Permanent public allowance**:
    - Configured using `FHE.makePubliclyDecryptable(ciphertext)`.
    - Grants long-term access to the ciphertext for any user.
    - Stored in a dedicated contract for persistent storage.

Syntactic sugar:

- `FHE.allowThis(ciphertext)` is shorthand for `FHE.allow(ciphertext, address(this))`. It authorizes the current contract to reuse a ciphertext handle in future transactions.

Transient vs. permanent allowance

Allowance type	Purpose	Storage type	Use case
Transient	Temporary access during a transaction.	Transient storage ↗ (EIP-1153)	Calling external functions or computations with ciphertexts. Use when wanting to save on gas costs.
Permanent	Long-term access across multiple transactions.	Dedicated contract storage	Persistent ciphertexts for contracts or users requiring ongoing access.

Granting and verifying access

Granting access

Developers can use functions like `allow`, `allowThis`, and `allowTransient` to grant permissions:

- **`allow`**: Grants permanent access to an address.
- **`allowThis`**: Grants the current contract access to manipulate the ciphertext.
- **`allowTransient`**: Grants temporary access to an address for the current transaction.

Verifying access

To check if an entity has permission to access a ciphertext, use functions like `isAllowed` or `isSenderAllowed`:

- **`isAllowed`**: Verifies if a specific address has permission.
- **`isSenderAllowed`**: Simplifies checks for the current transaction sender.

Practical uses of the ACL

- **Confidential parameters**: Pass encrypted values securely between contracts, ensuring only authorized entities can access them.
- **Secure state management**: Store encrypted state variables while controlling who can modify or read them.
- **Privacy-preserving computations**: Enable computations on encrypted data with confidence that permissions are enforced.

For a detailed explanation of the ACL's functionality, including code examples and advanced configurations, see ACL examples.

ACL examples

This page provides detailed instructions and examples on how to use and implement the ACL (Access Control List) in FHEVM. For an overview of ACL concepts and their importance, refer to the access control list (ACL) overview.

Controlling access: permanent and transient allowances

The ACL system allows you to define two types of permissions for accessing ciphertexts:

Permanent allowance

- **Function**: `FHE.allow(ciphertext, address)`
- **Purpose**: Grants persistent access to a ciphertext for a specific address.
- **Storage**: Permissions are saved in a dedicated ACL contract, making them available across transactions.

Alternative Solidity syntax

You can also use method-chaining syntax for granting allowances since FHE is a Solidity library.

```
using FHE for *;
ciphertext.allow(address1).allow(address2);
```

This is equivalent to calling `FHE.allow(ciphertext, address1)` followed by `FHE.allow(ciphertext, address2)`.

Transient allowance

- **Function**: `FHE.allowTransient(ciphertext, address)`
- **Purpose**: Grants temporary access for the duration of a single transaction.
- **Storage**: Permissions are stored in transient storage to save gas costs.

- **Use Case**: Ideal for passing encrypted values between functions or contracts during a transaction.

Alternative Solidity syntax

Method chaining is also available for transient allowances since FHE is a Solidity library.

```
using FHE for *;
ciphertext.allowTransient(address1).allowTransient(address2);
```

Syntactic sugar

- **Function**: `FHE.allowThis(ciphertext)`
- **Equivalent To**: `FHE.allow(ciphertext, address(this))`
- **Purpose**: Simplifies granting permanent access to the current contract for managing ciphertexts.

Alternative Solidity syntax

You can also use method-chaining syntax for allowThis since FHE is a Solidity library.

```
using FHE for *;
ciphertext.allowThis();
```

Make publicly decryptable

To make a ciphertext publicly decryptable, you can use the `FHE.makePubliclyDecryptable(ciphertext)` function. This grants decryption rights to anyone, which is useful for scenarios where the encrypted value should be accessible by all.

```
// Grant public decryption right to a ciphertext
FHE.makePubliclyDecryptable(ciphertext);

// Or using method syntax:
ciphertext.makePubliclyDecryptable();
```

- **Function**: `FHE.makePubliclyDecryptable(ciphertext)`
- **Purpose**: Makes the ciphertext decryptable by anyone.
- **Use Case**: When you want to publish encrypted results or data.

> You can combine multiple allowance methods (such as `.allow()`, `.allowThis()`, `.allowTransient()`) directly on ciphertext objects to grant access to several addresses or contracts in a single, fluent statement.
>
> **Example**
>
> ```
> // Grant transient access to one address and permanent access to
> another address
> ciphertext.allowTransient(address1).allow(address2);
>
> // Grant permanent access to the current contract and another
> address
> ciphertext.allowThis().allow(address1);
> ```

Best practices

Verifying sender access

When processing ciphertexts as input, it’s essential to validate that the sender is authorized to interact with the provided encrypted data. Failing to perform this verification can expose the system to inference attacks where malicious actors attempt to deduce private information.

Example scenario: Confidential ERC20 attack

Consider an **Confidential ERC20 token**. An attacker controlling two accounts, **Account A** and **Account B**, with 100 tokens in Account A, could exploit the system as follows:

1. The attacker attempts to send the target user's encrypted balance from **Account A** to **Account B**.
2. Observing the transaction outcome, the attacker gains information:
   - **If successful**: The target's balance is equal to or less than 100 tokens.

- **If failed**: The target's balance exceeds 100 tokens.

This type of attack allows the attacker to infer private balances without explicit access.

To prevent this, always use the `FHE.isSenderAllowed()` function to verify that the sender has legitimate access to the encrypted amount being transferred.

Example: secure verification

```
function transfer(address to, euint64 encryptedAmount) public {
  // Ensure the sender is authorized to access the encrypted amount
  require(FHE.isSenderAllowed(encryptedAmount), "Unauthorized access to
encrypted amount.");

  // Proceed with further logic
  ...
}
```

By enforcing this check, you can safeguard against inference attacks and ensure that encrypted values are only manipulated by authorized entities.

ACL for user decryption

If a ciphertext can be decrypt by a user, explicit access must be granted to them. Additionally, the user decryption mechanism requires the signature of a public key associated with the contract address. Therefore, a value that needs to be decrypted must be explicitly authorized for both the user and the contract.

Due to the user decryption mechanism, a user signs a public key associated with a specific contract; therefore, the ciphertext also needs to be allowed for the contract.

Example: Secure Transfer in ConfidentialERC20

```
function transfer(address to, euint64 encryptedAmount) public {
  require(FHE.isSenderAllowed(encryptedAmount), "The caller is not
authorized to access this encrypted amount.");
  euint64 amount = FHE.asEuint64(encryptedAmount);
  ebool canTransfer = FHE.le(amount, balances[msg.sender]);

  euint64 newBalanceTo = FHE.add(balances[to], FHE.select(canTransfer,
amount, FHE.asEuint64(0)));
  balances[to] = newBalanceTo;
  // Allow this new balance for both the contract and the owner.
  FHE.allowThis(newBalanceTo);
  FHE.allow(newBalanceTo, to);

  euint64 newBalanceFrom = FHE.sub(balances[from],
FHE.select(canTransfer, amount, FHE.asEuint64(0)));
  balances[from] = newBalanceFrom;
  // Allow this new balance for both the contract and the owner.
  FHE.allowThis(newBalanceFrom);
  FHE.allow(newBalanceFrom, from);
}
```

By understanding how to grant and verify permissions, you can effectively manage access to encrypted data in your FHEVM smart contracts. For additional context, see the ACL overview.

Reorgs handling

This page provides detailed instructions on how to handle reorg risks on Ethereum when using FHEVM.

Since ACL events are propagated from the FHEVM host chain to the Gateway ↗ immediately after being included in a block, dApp developers must take special care when encrypted information is critically important. For example, if an encrypted handle conceals the private key of a Bitcoin wallet holding significant funds, we need to ensure that this information cannot inadvertently leak to the wrong person due to a reorg on the FHEVM host chain. Therefore, it's the responsibility of dApp developers to prevent such scenarios by implementing a two-step ACL authorization process with a timelock between the request and the ACL call.

Simple example: Handling reorg risk on Ethereum

On Ethereum, a reorg can be up to 95 slots deep in the worst case, so waiting for more than 95 blocks should ensure that a previously sent transaction has been finalized—unless more than 1/3 of the nodes are malicious and willing to lose their stake, which is highly improbable.

❌ **Instead of writing this contract:**

```
contract PrivateKeySale {
  euint256 privateKey;
  bool isAlreadyBought = false;

  constructor(externalEuint256 _privateKey, bytes inputProof) {
    privateKey = FHE.fromExternal(_privateKey, inputProof);
    FHE.allowThis(privateKey);
  }

  function buyPrivateKey() external payable {
    require(msg.value == 1 ether, "Must pay 1 ETH");
    require(!isBought, "Private key already bought");
    isBought = true;
    FHE.allow(encryptedPrivateKey, msg.sender);
  }
}
```

Since the `privateKey`` encrypted variable contains critical information, we don't want to mistakenly leak it for free if a reorg occurs. This could happen in the previous example because we immediately grant authorization to the buyer in the same transaction that processes the sale.

✅ **We recommend writing something like this instead:**

```
contract PrivateKeySale {
  euint256 privateKey;
  bool isAlreadyBought = false;
  uint256 blockWhenBought = 0;
  address buyer;

  constructor(externalEuint256 _privateKey, bytes inputProof) {
    privateKey = FHE.fromExternal(_privateKey, inputProof);
    FHE.allowThis(privateKey);
  }

  function buyPrivateKey() external payable {
    require(msg.value == 1 ether, "Must pay 1 ETH");
    require(!isBought, "Private key already bought");
    isBought = true;
    blockWhenBought = block.number;
    buyer = msg.sender;
  }

  function requestACL() external {
    require(isBought, "Private key has not been bought yet");
    require(block.number > blockWhenBought + 95, "Too early to request
ACL, risk of reorg");
    FHE.allow(privateKey, buyer);
  }
}
```

This approach ensures that at least 96 blocks have elapsed between the transaction that purchases the private key and the transaction that authorizes the buyer to decrypt it.

> ⓘ This type of contract worsens the user experience by adding a timelock before users can decrypt data, so it should be used sparingly: only when leaked information could be critically important and high-value.

Logics

Branching

This document explains how to implement conditional logic (if/else branching) when working with encrypted values in FHEVM. Unlike typical Solidity programming, working with Fully Homomorphic Encryption (FHE) requires specialized methods to handle conditions on encrypted data.

This document covers encrypted branching and how to move from an encrypted condition to a non-encrypted business logic in your smart contract.

What is confidential branching?

In FHEVM, when you perform comparison operations, the result is an encrypted boolean (`ebool`). Since encrypted booleans do not support standard boolean operations like `if` statements or logical operators, conditional logic must be implemented using specialized methods.

To facilitate conditional assignments, FHEVM provides the `FHE.select` function, which acts as a ternary operator for encrypted values.

Using `FHE.select` for conditional logic

The `FHE.select` function enables branching logic by selecting one of two encrypted values based on an encrypted condition (`ebool`). It works as follows:

```
FHE.select(condition, valueIfTrue, valueIfFalse);
```

- **`condition`**: An encrypted boolean (`ebool`) resulting from a comparison.
- **`valueIfTrue`**: The encrypted value to return if the condition is true.
- **`valueIfFalse`**: The encrypted value to return if the condition is false.

Example: Auction Bidding Logic

Here's an example of using conditional logic to update the highest winning number in a guessing game:

```
function bid(externalEuint64 encryptedValue, bytes calldata inputProof)
external onlyBeforeEnd {
  // Convert the encrypted input to an encrypted 64-bit integer
  euint64 bid = FHE.asEuint64(encryptedValue, inputProof);

  // Compare the current highest bid with the new bid
  ebool isAbove = FHE.lt(highestBid, bid);

  // Update the highest bid if the new bid is greater
  highestBid = FHE.select(isAbove, bid, highestBid);

  // Allow the contract to use the updated highest bid ciphertext
  FHE.allowThis(highestBid);
}
```

ⓘ This is a simplified example to demonstrate the functionality.

How Does It Work?

- **Comparison**:
  - The `FHE.lt` function compares `highestBid` and `bid`, returning an `ebool` (`isAbove`) that indicates whether the new bid is higher.
- **Selection**:
  - The `FHE.select` function updates `highestBid` to either the new bid or the previous highest bid, based on the encrypted condition `isAbove`.
- **Permission Handling**:
  - After updating `highestBid`, the contract reauthorizes itself to manipulate the updated ciphertext using `FHE.allowThis`.

Key Considerations

- **Value change behavior:** Each time `FHE.select` assigns a value, a new ciphertext is created, even if the underlying plaintext value remains unchanged. This behavior is inherent to FHE and ensures data confidentiality, but developers should account for it when designing their smart contracts.
- **Gas consumption:** Using `FHE.select` and other encrypted operations incurs additional gas costs compared to traditional Solidity logic. Optimize your code to minimize unnecessary operations.
- **Access control:** Always use appropriate ACL functions (e.g., `FHE.allowThis`, `FHE.allow`) to ensure the updated ciphertexts are authorized for use in future computations or transactions.

How to branch to a non-confidential path?

So far, this section only covered how to do branching using encrypted variables. However, there may be many cases where the "public" contract logic will depend on the outcome from a encrypted path.

To do so, there are only one way to branch from an encrypted path to a non-encrypted path: it requires a public decryption using the oracle. Hence, any contract logic that requires moving from an encrypted input to a non-encrypted path always requires an async contract logic.

Example: Auction Bidding Logic: Item Release

Going back to our previous example with the auction bidding logic. Let's assume that the winner of the auction can receive some prize, which is not confidential.

```
bool public isPrizeDistributed;
eaddress internal highestBidder;
euint64 internal highestBid;

function bid(externalEuint64 encryptedValue, bytes calldata inputProof)
external onlyBeforeEnd {
  // Convert the encrypted input to an encrypted 64-bit integer
  euint64 bid = FHE.asEuint64(encryptedValue, inputProof);

  // Compare the current highest bid with the new bid
  ebool isAbove = FHE.lt(highestBid, bid);

  // Update the highest bid if the new bid is greater
  highestBid = FHE.select(isAbove, bid, highestBid);

  // Update the highest bidder address if the new bid is greater
  highestBidder = FHE.select(isAbove, FHE.asEaddress(msg.sender),
currentBidder));

  // Allow the contract to use the highest bidder address
  FHE.allowThis(highestBidder);

  // Allow the contract to use the updated highest bid ciphertext
  FHE.allowThis(highestBid);
}

function revealWinner() external onlyAfterEnd {
  bytes32[] memory cts = new bytes32[](2);
  cts[0] = FHE.toBytes32(highestBidder);
  uint256 requestId = FHE.requestDecryption(cts,
this.transferPrize.selector);
}

function transferPrize(uint256 requestId, address auctionWinner, bytes
memory signatures) external {
  require(!isPrizeDistributed, "Prize has already been distributed");
  FHE.verifySignatures(requestId, signatures)

  isPrizeDistributed = true;
  // Business logic to transfer the prize to the auction winner
}
```

This is a simplified example to demonstrate the functionality.

As you can see the in the above example, the path to move from an encrypted condition to a decrypted business logic must be async and requires calling the decryption oracle contract to reveal the result of the logic using encrypted variables.

Summary

- **`FHE.select`** is a powerful tool for conditional logic on encrypted values.
- Encrypted booleans (`ebool`) and values maintain confidentiality, enabling privacy-preserving logic.
- Developers should account for gas costs and ciphertext behavior when designing conditional operations.

Dealing with branches and conditions

This document explains how to handle branches, loops or conditions when working with Fully Homomorphic Encryption (FHE), specifically when the condition / index is encrypted.

Breaking a loop

❌ In FHE, it is not possible to break a loop based on an encrypted condition. For example, this would not work:

```
euint8 maxValue = FHE.asEuint(6); // Could be a value between 0 and 10
euint8 x = FHE.asEuint(0);
// some code
while(FHE.lt(x, maxValue)){
    x = FHE.add(x, 2);
}
```

If your code logic requires looping on an encrypted boolean condition, we highly suggest to try to replace it by a finite loop with an appropriate constant maximum number of steps and use `FHE.select` inside the loop.

Suggested approach

✅ For example, the previous code could maybe be replaced by the following snippet:

```
euint8 maxValue = FHE.asEuint(6); // Could be a value between 0 and 10
euint8 x;
// some code
for (uint32 i = 0; i < 10; i++) {
    euint8 toAdd = FHE.select(FHE.lt(x, maxValue), 2, 0);
    x = FHE.add(x, toAdd);
}
```

In this snippet, we perform 10 iterations, adding 4 to `x` in each iteration as long as the iteration count is less than `maxValue`. If the iteration count exceeds `maxValue`, we add 0 instead for the remaining iterations because we can't break the loop.

Best practices

Obfuscate branching

The previous paragraph emphasized that branch logic should rely as much as possible on `FHE.select` instead of decryptions. It hides effectively which branch has been executed.

However, this is sometimes not enough. Enhancing the privacy of smart contracts often requires revisiting your application's logic.

For example, if implementing a simple AMM for two encrypted ERC20 tokens based on a linear constant function, it is recommended to not only hide the amounts being swapped, but also the token which is swapped in a pair.

✅ Here is a very simplified example implementation, we suppose here that the rate between tokenA and tokenB is constant and equals to 1:

```
// typically either encryptedAmountAIn or encryptedAmountBIn is an
encrypted null value
// ideally, the user already owns some amounts of both tokens and has
pre-approved the AMM on both tokens
function swapTokensForTokens(
  externalEuint32 encryptedAmountAIn,
  externalEuint32 encryptedAmountBIn,
  bytes calldata inputProof
) external {
  euint32 encryptedAmountA = FHE.asEuint32(encryptedAmountAIn,
inputProof); // even if amount is null, do a transfer to obfuscate
trade direction
  euint32 encryptedAmountB = FHE.asEuint32(encryptedAmountBIn,
inputProof); // even if amount is null, do a transfer to obfuscate
trade direction

  // send tokens from user to AMM contract
  FHE.allowTransient(encryptedAmountA, tokenA);
  IConfidentialERC20(tokenA).transferFrom(msg.sender, address(this),
encryptedAmountA);

  FHE.allowTransient(encryptedAmountB, tokenB);
  IConfidentialERC20(tokenB).transferFrom(msg.sender, address(this),
encryptedAmountB);

  // send tokens from AMM contract to user
  // Price of tokenA in tokenB is constant and equal to 1, so we just
swap the encrypted amounts here
  FHE.allowTransient(encryptedAmountB, tokenA);
  IConfidentialERC20(tokenA).transfer(msg.sender, encryptedAmountB);

  FHE.allowTransient(encryptedAmountA, tokenB);
  IConfidentialERC20(tokenB).transferFrom(msg.sender, address(this),
encryptedAmountA);
}
```

Notice that to preserve confidentiality, we had to make two inputs transfers on both tokens from the user to the AMM contract, and similarly two output transfers from the AMM to the user, even if technically most of the times it will make sense that one of the user inputs `encryptedAmountAIn` or `encryptedAmountBIn` is actually an encrypted zero.

This is different from a classical non-confidential AMM with regular ERC20 tokens: in this case, the user would need to just do one input transfer to the AMM on the token being sold, and receive only one output transfer from the AMM on the token being bought.

Avoid using encrypted indexes

Using encrypted indexes to pick an element from an array without revealing it is not very efficient, because you would still need to loop on all the indexes to preserve confidentiality.

However, there are plans to make this kind of operation much more efficient in the future, by adding specialized operators for arrays.

For instance, imagine you have an encrypted array called `encArray` and you want to update an encrypted value `x` to match an item from this list, `encArray[i]`, *without* disclosing which item you're choosing.

❌ You must loop over all the indexes and check equality homomorphically, however this pattern is very expensive in gas and should be avoided whenever possible.

```
euint32 x;
euint32[] encArray;

function setXwithEncryptedIndex(externalEuint32 encryptedIndex, bytes
calldata inputProof) public {
    euint32 index = FHE.asEuint32(encryptedIndex, inputProof);
    for (uint32 i = 0; i < encArray.length; i++) {
        ebool isEqual = FHE.eq(index, i);
        x = FHE.select(isEqual, encArray[i], x);
    }
    FHE.allowThis(x);
}
```

Error handling

This document explains how to handle errors effectively in FHEVM smart contracts. Since transactions involving encrypted data do not automatically revert when conditions are not met, developers need alternative mechanisms to communicate errors to users.

Challenges in error handling

In the context of encrypted data:

1. **No automatic reversion**: Transactions do not revert if a condition fails, making it challenging to notify users of issues like insufficient funds or invalid inputs.
2. **Limited feedback**: Encrypted computations lack direct mechanisms for exposing failure reasons while maintaining confidentiality.

Recommended approach: Error logging with a handler

To address these challenges, implement an **error handler** that records the most recent error for each user. This allows dApps or frontends to query error states and provide appropriate feedback to users.

Example implementation

The following contract snippet demonstrates how to implement and use an error handler:

```
struct LastError {
  euint8 error;      // Encrypted error code
  uint timestamp;    // Timestamp of the error
}

// Define error codes
euint8 internal NO_ERROR;
euint8 internal NOT_ENOUGH_FUNDS;

constructor() {
  NO_ERROR = FHE.asEuint8(0);           // Code 0: No error
  NOT_ENOUGH_FUNDS = FHE.asEuint8(1);   // Code 1: Insufficient funds
}

// Store the last error for each address
mapping(address => LastError) private _lastErrors;

// Event to notify about an error state change
event ErrorChanged(address indexed user);

/**
 * @dev Set the last error for a specific address.
 * @param error Encrypted error code.
 * @param addr Address of the user.
 */
function setLastError(euint8 error, address addr) private {
  _lastErrors[addr] = LastError(error, block.timestamp);
  emit ErrorChanged(addr);
}

/**
 * @dev Internal transfer function with error handling.
 * @param from Sender's address.
 * @param to Recipient's address.
 * @param amount Encrypted transfer amount.
 */
function _transfer(address from, address to, euint32 amount) internal {
  // Check if the sender has enough balance to transfer
  ebool canTransfer = FHE.le(amount, balances[from]);

  // Log the error state: NO_ERROR or NOT_ENOUGH_FUNDS
  setLastError(FHE.select(canTransfer, NO_ERROR, NOT_ENOUGH_FUNDS),
msg.sender);

  // Perform the transfer operation conditionally
  balances[to] = FHE.add(balances[to], FHE.select(canTransfer, amount,
FHE.asEuint32(0)));
  FHE.allowThis(balances[to]);
```

```
    FHE.allow(balances[to], to);

    balances[from] = FHE.sub(balances[from], FHE.select(canTransfer,
amount, FHE.asEuint32(0)));
    FHE.allowThis(balances[from]);
    FHE.allow(balances[from], from);
}
```

How It Works

1. **Define error codes**:
   - `NO_ERROR`: Indicates a successful operation.
   - `NOT_ENOUGH_FUNDS`: Indicates insufficient balance for a transfer.
2. **Record errors**:
   - Use the `setLastError` function to log the latest error for a specific address along with the current timestamp.
   - Emit the `ErrorChanged` event to notify external systems (e.g., dApps) about the error state change.
3. **Conditional updates**:
   - Use the `FHE.select` function to update balances and log errors based on the transfer condition (`canTransfer`).
4. **Frontend integration**:
   - The dApp can query `_lastErrors` for a user's most recent error and display appropriate feedback, such as "Insufficient funds" or "Transaction successful."

Example error query

The frontend or another contract can query the `_lastErrors` mapping to retrieve error details:

```
/**
 * @dev Get the last error for a specific address.
 * @param user Address of the user.
 * @return error Encrypted error code.
 * @return timestamp Timestamp of the error.
 */
function getLastError(address user) public view returns (euint8 error,
uint timestamp) {
  LastError memory lastError = _lastErrors[user];
  return (lastError.error, lastError.timestamp);
}
```

Benefits of this approach

1. **User feedback**:
   - Provides actionable error messages without compromising the confidentiality of encrypted computations.
2. **Scalable error tracking**:
   - Logs errors per user, making it easy to identify and debug specific issues.
3. **Event-driven notifications**:
   - Enables frontends to react to errors in real time via the `ErrorChanged` event.

By implementing error handlers as demonstrated, developers can ensure a seamless user experience while maintaining the privacy and integrity of encrypted data operations.

Decryption

This section explains how to handle decryption in fhevm. Decryption allows plaintext data to be accessed when required for contract logic or user presentation, ensuring confidentiality is maintained throughout the process.

Decryption is essential in two primary cases:

1. **Smart contract logic**: A contract requires plaintext values for computations or decision-making.
2. **User interaction**: Plaintext data needs to be revealed to all users, such as revealing the decision of the vote.

Overview

Decryption in FHEVM is an asynchronous process that involves the Relayer and Key Management System (KMS). Here’s an example of how to safely request decryption in a contract.

Example: asynchronous decryption in a contract

```
pragma solidity ^0.8.24;

import "@fhevm/solidity/lib/FHE.sol";
import { EthereumConfig } from "@fhevm/solidity/config/ZamaConfig.sol";

contract TestAsyncDecrypt is EthereumConfig {
  ebool xBool;
  bool public yBool;
  bool isDecryptionPending;
  uint256 latestRequestId;

  constructor() {
    xBool = FHE.asEbool(true);
    FHE.allowThis(xBool);
  }

  function requestBool() public {
    require(!isDecryptionPending, "Decryption is in progress");
    bytes32[] memory cts = new bytes32[](1);
    cts[0] = FHE.toBytes32(xBool);
    uint256 latestRequestId = FHE.requestDecryption(cts,
this.myCustomCallback.selector);

    /// @dev This prevents sending multiple requests before the first
callback was sent.
    isDecryptionPending = true;
  }

  function myCustomCallback(uint256 requestId, bytes memory cleartexts,
bytes memory decryptionProof) public returns (bool) {
    /// @dev This check is used to verify that the request id is the
expected one.
    require(requestId == latestRequestId, "Invalid requestId");
    FHE.checkSignatures(requestId, cleartexts, decryptionProof);

    (bool decryptedInput) = abi.decode(cleartexts, (bool));
    yBool = decryptedInput;
    isDecryptionPending = false;
    return yBool;
  }
}
```

Decryption in depth

This document provides a detailed guide on implementing decryption in your smart contracts using the `DecryptionOracle` in fhevm. It covers the setup, usage of the `FHE.requestDecryption` function, and testing with Hardhat.

`DecryptionOracle` setup

The `DecryptionOracle` is pre-deployed on the FHEVM testnet. It uses a default relayer account specified in the `.env` file.

Anyone can fulfill decryption requests but it is essential to add signature verification (and to include a logic to invalidate the replay of decryption requests). The role of the `DecryptionOracle` contract is to independently verify the KMS signature during execution. This ensures that the relayers cannot manipulate or send fraudulent decryption results, even if compromised.

There are two functions to consider: `requestDecryption` and `checkSignatures`.

`FHE.requestDecryption` function

You can call the function `FHE.requestDecryption` as such:

```
function requestDecryption(
  bytes32[] calldata ctsHandles,
  bytes4 callbackSelector
) external payable returns (uint256 requestId);
```

Function arguments

The first argument, `ctsHandles`, should be an array of ciphertexts handles which could be of different types, i.e `uint256` values coming from unwrapping handles of type either `ebool`, `euint8`, `euint16`, `euint32`, `euint64` or `eaddress`.

`ctsHandles` is the array of ciphertexts that are requested to be decrypted. The relayer will send the corresponding ciphertexts to the KMS for decryption before fulfilling the request.

`callbackSelector` is the function selector of the callback function, which will be called once the relayer fulfils the decryption request.

```
function [callbackName](uint256 requestID, bytes memory cleartexts,
bytes memory decryptionProof) external;
```

`cleartexts` is the bytes array corresponding to the ABI encoding of all requested decrypted values. Each of these decrypted values' type should be a native Solidity type corresponding to the original ciphertext type, following this table of conventions:

Ciphertext type	Decrypted type
ebool	bool
euint8	uint8
euint16	uint16
euint32	uint32
euint64	uint64
euint128	uint128
euint256	uint256
eaddress	address

Here `callbackName` is a custom name given by the developer to the callback function, `requestID` will be the request id of the decryption (could be commented if not needed in the logic, but must be present) and `cleartexts` is an ABI encoded byte array of the results of the decryption of the `ct` array values, i.e their number should be the size of the `ct` array. `decryptionProof` is a byte array containing the KMS signatures and extra data.

`msgValue` is the value in native tokens to be sent to the calling contract during fulfillment, i.e when the callback will be called with the results of decryption.

> Notice that the callback should always verify the signatures and implement a replay protection mechanism (see below).

`FHE.checkSignatures` function

You can call the function `FHE.checkSignatures` as such:

```
function checkSignatures(uint256 requestId, bytes memory cleartexts,
bytes[] memory signatures);
```

Function arguments

- `requestID`, is the value that was returned in the `requestDecryption` function.
- `cleartexts`, is an ABI encoding of the decrypted values associated to the handles (using `abi.encode`). This can contain one or multiple values, depending on the number of handles requested in the `requestDecryption` function. Each of these values' type must match the type of the corresponding handle.
- `decryptionProof`, is a byte array containing the KMS signatures and extra data.

This function reverts if the signatures are invalid.

Development Guide

Hardhat plugin

This section will guide you through writing and testing FHEVM smart contracts in Solidity using Hardhat ↗.

The FHEVM Hardhat Plugin

To write FHEVM smart contracts using Hardhat, you need to install the FHEVM Hardhat Plugin ↗ in your Hardhat project.

This plugin enables you to develop, test, and interact with FHEVM contracts right out of the box.

It extends Hardhat's functionality with a complete FHEVM API that allows you:

- Encrypt data
- Decrypt data
- Run tests using various FHEVM execution modes
- Write FHEVM-enabled Hardhat Tasks

Where to go next

🟧 Go to **Setup Hardhat** ↗ to initialize your FHEVM Hardhat project.

🟧 Go to **Write FHEVM Tests in Hardhat** for details on writing tests of FHEVM smart contracts using Hardhat.

🟧 Go to **Run FHEVM Tests in Hardhat** to learn how to execute those tests in different FHEVM environments.

🟧 Go to **Write FHEVM Hardhat Task** to learn how to write your own custom FHEVM Hardhat task.

Write FHEVM tests in Hardhat

In this section, you'll find everything you need to set up a new Hardhat ↗ project and start developing FHEVM smart contracts from scratch using the FHEVM Hardhat Plugin ↗

Enabling the FHEVM Hardhat Plugin in your Hardhat project

Like any Hardhat plugin, the FHEVM Hardhat Plugin ↗ must be enabled by adding the following `import` statement to your `hardhat.config.ts` file:

```
import "@fhevm/hardhat-plugin";
```

> Without this import, the Hardhat FHEVM API will **not** be available in your Hardhat runtime environment (HRE).

Accessing the Hardhat FHEVM API

The plugin extends the standard Hardhat Runtime Environment ↗ (or `hre` in short) with the new `fhevm` Hardhat module.

You can access it in either of the following ways:

```
import { fhevm } from "hardhat";
```

or

```
import * as hre from "hardhat";

// Then access: hre.fhevm
```

Encrypting Values Using the Hardhat FHEVM API

Suppose the FHEVM smart contract you want to test has a function called `foo` that takes an encrypted `uint32` value as input. The Solidity function `foo` should be declared as follows:

```
function foo(externalEunit32 value, bytes calldata memory inputProof);
```

Where:

- `externalEunit32 value` : is a `bytes32` representing the encrypted `uint32`
- `bytes calldata memory inputProof` : is a `bytes` array representing the zero-knowledge proof of knowledge that validates the encryption

To compute these arguments in TypeScript, you need:

- The **address of the target smart contract**
- The **signer's address** (i.e., the account sending the transaction)

1. **Create a new encryted input**

```
// use the `fhevm` API module from the Hardhat Runtime Environment
const input = fhevm.createEncryptedInput(contractAddress,
signers.alice.address);
```

2. **Add the value you want to encrypt.**

```
input.add32(12345);
```

3. **Perform local encryption.**

```
const encryptedInputs = await input.encrypt();
```

4 **Call the Solidity function**

```
const externalUint32Value = encryptedInputs.handles[0];
const inputProof = encryptedInputs.proof;

const tx = await input.foo(externalUint32Value, inputProof);
await tx.wait();
```

Encryption examples

- Basic encryption examples ↗
- FHECounter ↗

Decrypting values using the Hardhat FHEVM API

Suppose user **Alice** wants to decrypt a `euint32` value that is stored in a smart contract exposing the following Solidity `view` function:

```
function getEncryptedUint32Value() public view returns (euint32) {
returns _encryptedUint32Value; }
```

> For simplicity, we assume that both Alice's account and the target smart contract already have the necessary FHE permissions to decrypt this value. For a detailed explanation of how FHE permissions work, see the `initializeUint32()` ↗ function in DecryptSingleValue.sol ↗.

1 **Retrieve the encrypted value (a `bytes32` handle) from the smart contract:**

```
const encryptedUint32Value = await
contract.getEncryptedUint32Value();
```

2 Perform the decryption using the FHEVM API:

```
const clearUint32Value = await fhevm.userDecryptEuint(
  FhevmType.euint32, // Encrypted type (must match the Solidity
type)
  encryptedUint32Value, // bytes32 handle Alice wants to decrypt
  contractAddress, // Target contract address
  signers.alice, // Alice's wallet
);
```

> If either the target smart contract or the user does **NOT** have FHE permissions, then the decryption call will fail!

Supported Decryption Types

Use the appropriate function for each encrypted data type:

Type	Function
`euintXXX`	`fhevm.userDecryptEuint(...)`
`ebool`	`fhevm.userDecryptEbool(...)`
`eaddress`	`fhevm.userDecryptEaddress(...)`

Decryption examples

- Basic decryption examples ↗
- FHECounter ↗

Deploy contracts and run tests

In this section, you'll find everything you need to test your FHEVM smart contracts in your Hardhat ↗ project.

FHEVM Runtime Modes

The FHEVM Hardhat plugin provides three **FHEVM runtime modes** tailored for different stages of contract development and testing. Each mode offers a trade-off between speed, encryption, and persistence.

1. The **Hardhat (In-Memory)** default network: 💉 *Uses mock encryption.* Ideal for regular tests, CI test coverage, and fast feedback during early contract development. No real encryption is used.
2. The **Hardhat Node (Local Server)** network: 💉 *Uses mock encryption.* Ideal when you need persistent state - for example, when testing frontend interactions, simulating user flows, or validating deployments in a realistic local environment. Still uses mock encryption.
3. The **Sepolia Testnet** network: 🔐 *Uses real encryption.* Use this mode once your contract logic is stable and validated locally. This is the only mode that runs on the full FHEVM stack with **real encrypted values**. It simulates real-world production conditions but is slower and requires Sepolia ETH.

> **Zama Testnet** is not a blockchain itself. It is a protocol that enables you to run confidential smart contracts on existing blockchains (such as Ethereum, Base, and others) with the support of encrypted types. See the FHE on blockchain ↗ guide to learn more about the protocol architecture.
>
> Currently, **Zama Protocol** is available on the **Sepolia Testnet**. Support for additional chains will be added in the future. See the roadmap↗ ↗

Summary

Mode	Encryption	Persistent	Chain	Speed	Usage
Hardhat (default)	🧪 Mock	❌ No	In-Memory	⚡⚡ Very Fast	Fast local testing and coverage
Hardhat Node	🧪 Mock	✅ Yes	Server	⚡ Fast	Frontend integration and local persistent testing
Sepolia Testnet	🔐 Real Encryption	✅ Yes	Server	🐢 Slow	Full-stack validation with real encrypted data

The FHEVM Hardhat Template

To demonstrate the three available testing modes, we'll use the fhevm-hardhat-template ↗, which comes with the FHEVM Hardhat Plugin pre-installed, a basic `FHECounter` smart contract, and ready-to-use tasks for interacting with a deployed instance of this contract.

Run on Hardhat (default)

To run your tests in-memory using FHEVM mock values, simply run the following:

```
npx hardhat test --network hardhat
```

Run on Hardhat Node

You can also run your tests against a local Hardhat node, allowing you to deploy contract instances and interact with them in a persistent environment.

1. **Launch the Hardhat Node server:**

   - Open a new terminal window.
   - From the root project directory, run the following:

```
npx hardhat node
```

2. **Run your test suite (optional):**

   From the root project directory:

```
npx hardhat test --network localhost
```

3. **Deploy the `FHECounter` smart contract on Hardhat Node**

   From the root project directory:

```
npx hardhat deploy --network localhost
```

   Check the deployed contract FHEVM configuration:

```
npx hardhat fhevm check-fhevm-compatibility --network localhost --
address <deployed contract address>
```

4 Interact with the deployed `FHECounter` smart contract

From the root project directory:

1. Decrypt the current counter value:

```
npx hardhat --network localhost task:decrypt-count
```

2. Increment the counter by 1:

```
npx hardhat --network localhost task:increment --value 1
```

3. Decrypt the new counter value:

```
npx hardhat --network localhost task:decrypt-count
```

Run on Sepolia Ethereum Testnet

To test your FHEVM smart contract using real encrypted values, you can run your tests on the Sepolia Testnet.

1 Rebuild the project for Sepolia

From the root project directory:

```
npx hardhat clean
npx hardhat compile --network sepolia
```

2 Deploy the `FHECounter` smart contract on Sepolia

```
npx hardhat deploy --network sepolia
```

3 Check the deployed `FHECounter` contract FHEVM configuration

From the root project directory:

```
npx hardhat fhevm check-fhevm-compatibility --network sepolia --
address <deployed contract address>
```

If an internal exception is raised, it likely means the contract was not properly compiled for the Sepolia network.

4 Interact with the deployed `FHECounter` contract

From the root project directory:

1. Decrypt the current counter value (⌛ wait...):

```
npx hardhat --network sepolia task:decrypt-count
```

2. Increment the counter by 1 (⌛ wait...):

```
npx hardhat --network sepolia task:increment --value 1
```

3. Decrypt the new counter value (⌛ wait...):

```
npx hardhat --network sepolia task:decrypt-count
```

Write FHEVM-enabled Hardhat Tasks

In this section, you'll learn how to write a custom FHEVM Hardhat task.

Writing tasks is a gas-efficient and flexible way to test your FHEVM smart contracts on the Sepolia network. Creating a custom task is straightforward.

Prerequisite

- You should be familiar with Hardhat tasks. If you're new to them, refer to the Hardhat Tasks official documentation ↗.
- You should have already **completed** the FHEVM Tutorial ↗.
- This page provides a step-by-step walkthrough of the `task:decrypt-count` tasks included in the file tasks/FHECounter.ts ↗ file, located in the fhevm-hardhat-template ↗ repository.

A Basic Hardhat Task

1

A Basic Hardhat Task

Let’s start with a simple example: fetching the current counter value from a basic `Counter.sol` contract.

If you're already familiar with Hardhat and custom tasks, the TypeScript code below should look familiar and be easy to follow:

```
task("task:get-count", "Calls the getCount() function of Counter
Contract")
  .addOptionalParam("address", "Optionally specify the Counter
contract address")
  .setAction(async function (taskArguments: TaskArguments, hre) {
    const { ethers, deployments } = hre;

    const CounterDeployement = taskArguments.address
      ? { address: taskArguments.address }
      : await deployments.get("Counter");
    console.log(`Counter: ${CounterDeployement.address}`);

    const counterContract = await ethers.getContractAt("Counter",
CounterDeployement.address);

    const clearCount = await counterContract.getCount();

    console.log(`Clear count    : ${clearCount}`);
});
```

Now, let’s modify this task to work with FHEVM encrypted values.

Comment Out Existing Logic and rename

2

Comment Out Existing Logic and Rename

First, comment out the existing logic so we can incrementally add the necessary changes for FHEVM integration.

```
task("task:get-count", "Calls the getCount() function of Counter
Contract")
  .addOptionalParam("address", "Optionally specify the Counter
contract address")
  .setAction(async function (taskArguments: TaskArguments, hre) {
    // const { ethers, deployments } = hre;

    // const CounterDeployement = taskArguments.address
    //   ? { address: taskArguments.address }
    //   : await deployments.get("Counter");
    // console.log(`Counter: ${CounterDeployement.address}`);

    // const counterContract = await
ethers.getContractAt("Counter", CounterDeployement.address);

    // const clearCount = await counterContract.getCount();

    // console.log(`Clear count    : ${clearCount}`);
});
```

Next, rename the task by replacing:

```
task("task:get-count", "Calls the getCount() function of Counter
Contract")
```

With:

```
task("task:decrypt-count", "Calls the getCount() function of
Counter Contract")
```

This updates the task name from `task:get-count` to `task:decrypt-count`, reflecting that it now includes decryption logic for FHE-encrypted values.

Initialize FHEVM CLI API

3 Initialize FHEVM CLI API

Replace the line:

```
// const { ethers, deployments } = hre;
```

With:

```
const { ethers, deployments, fhevm } = hre;

await fhevm.initializeCLIApi();
```

> Calling `initializeCLIApi()` is essential. Unlike built-in Hardhat tasks like `test` or `compile`, which automatically initialize the FHEVM runtime environment, custom tasks require you to call this function explicitly. **Make sure to call it at the very beginning of your task** to ensure the environment is properly set up.

Call the view function `getCount` from the FHECounter contract

4 Call the view function `getCount` from the FHECounter contract

Replace the following commented-out lines:

```
    // const CounterDeployement = taskArguments.address
    //   ? { address: taskArguments.address }
    //   : await deployments.get("Counter");
    // console.log(`Counter: ${CounterDeployement.address}`);

    // const counterContract = await
ethers.getContractAt("Counter", CounterDeployement.address);

    // const clearCount = await counterContract.getCount();
```

With the FHEVM equivalent:

```
    const FHECounterDeployement = taskArguments.address
      ? { address: taskArguments.address }
      : await deployments.get("FHECounter");
    console.log(`FHECounter: ${FHECounterDeployement.address}`);

    const fheCounterContract = await
ethers.getContractAt("FHECounter", FHECounterDeployement.address);

    const encryptedCount = await fheCounterContract.getCount();
    if (encryptedCount === ethers.ZeroHash) {
      console.log(`encrypted count: ${encryptedCount}`);
      console.log("clear count    : 0");
      return;
    }
```

Here, `encryptedCount` is an FHE-encrypted `euint32` primitive. To retrieve the actual value, we need to decrypt it in the next step.

Decrypt the encrypted count value

5 **Decrypt the encrypted count value:**

Now replace the following commented-out line:

```
    // console.log(`Clear count    : ${clearCount}`);
```

With the decryption logic:

```
    const signers = await ethers.getSigners();
    const clearCount = await fhevm.userDecryptEuint(
      FhevmType.euint32,
      encryptedCount,
      FHECounterDeployement.address,
      signers[0],
    );
    console.log(`Encrypted count: ${encryptedCount}`);
    console.log(`Clear count    : ${clearCount}`);
```

At this point, your custom Hardhat task is fully configured to work with FHE-encrypted values and ready to run!

6 Step 6: Run your custom task using Hardhat Node

Start the Local Hardhat Node:

- Open a new terminal window.
- From the root project directory, run the following:

```
npx hardhat node
```

Deploy the FHECounter smart contract on the local Hardhat Node

```
npx hardhat deploy --network localhost
```

Run your custom task

```
npx hardhat task:decrypt-count --network localhost
```

Step 7: Run your custom task using Sepolia

7

Step 7: Run your custom task using Sepolia

Deploy the FHECounter smart contract on Sepolia Testnet (if not already deployed)

```
npx hardhat deploy --network sepolia
```

Execute your custom task

```
npx hardhat task:decrypt-count --network sepolia
```

Foundry

This guide explains how to use Foundry with FHEVM for developing smart contracts.

While a Foundry template is currently in development, we strongly recommend using the Hardhat template) for now, as it provides a fully tested and supported development environment for FHEVM smart contracts.

However, you could still use Foundry with the mocked version of the FHEVM, but please be aware that this approach is **NOT** recommended, since the mocked version is not fully equivalent to the real FHEVM node's implementation (see warning in hardhat). In order to do this, you will need to rename your `FHE.sol` imports from `@fhevm/solidity/lib/FHE.sol` to `fhevm/mocks/FHE.sol` in your solidity source files.

HCU

This guide explains how to use Fully Homomorphic Encryption (FHE) operations in your smart contracts on FHEVM. Understanding HCU is critical for designing efficient confidential smart contracts.

Overview

FHE operations in FHEVM are computationally intensive compared to standard Ethereum operations, as they require complex mathematical computations to maintain privacy and security. To manage computational load and prevent potential denial-of-service attacks, FHEVM implements a metering system called **Homomorphic Complexity Units ("HCU")**.

To represent this complexity, we introduced the **Homomorphic Complexity Unit ("HCU")**. In Solidity, each FHE operation consumes a set amount of HCU based on the operational computational complexity for hardware computation. Since FHE transactions are symbolic, this helps preventing resource exhaustion outside of the blockchain.

To do so, there is a contract named `HCULimit`, which monitors HCU consumption for each transaction and enforces two key limits:

- **Sequential homomorphic operations depth limit per transaction**: Controls HCU usage for operations that must be processed in order.
- **Global homomorphic operations complexity per transaction**: Controls HCU usage for operations that can be processed in parallel.

If either limit is exceeded, the transaction will revert.

HCU limit

The current devnet has an HCU limit of **20,000,000** per transaction and an HCU depth limit of **5,000,000** per transaction. If either HCU limit is exceeded, the transaction will revert.

To resolve this, you must do one of the following:

- Refactor your code to reduce the number of FHE operations in your transaction.
- Split your FHE operations across multiple independent transactions.

HCU costs for common operations

Boolean operations (`ebool`)

Function name	HCU (scalar)	HCU (non-scalar)
`and`	22,000	25,000
`or`	22,000	24,000
`xor`	2,000	22,000
`not`	-	2
`select`	-	55,000
`randEbool`	-	19,000

Unsigned integer operations

HCU increase with the bit-width of the encrypted integer type. Below are the detailed costs for various operations on encrypted types.

8-bit Encrypted integers (`euint8`)

Function name	HCU (scalar)	HCU (non-scalar)
`add`	84,000	88,000
`sub`	84,000	91,000
`mul`	122,000	150,000
`div`	210,000	-
`rem`	440,000	-
`and`	31,000	31,000
`or`	30,000	30,000
`xor`	31,000	31,000
`shr`	32,000	91,000
`shl`	32,000	92,000
`rotr`	31,000	93,000
`rotl`	31,000	91,000
`eq`	55,000	55,000
`ne`	55,000	55,000
`ge`	52,000	63,000
`gt`	52,000	59,000
`le`	58,000	58,000
`lt`	52,000	59,000
`min`	84,000	119,000
`max`	89,000	121,000
`neg`	-	79,000
`not`	-	9
`select`	-	55,000
`randEuint8`	-	23,000

16-bit Encrypted integers (`euint16`)

Function name	HCU (scalar)	HCU (non-scalar)
`add`	93,000	93,000
`sub`	93,000	93,000
`mul`	193,000	222,000
`div`	302,000	-
`rem`	580,000	-
`and`	31,000	31,000
`or`	30,000	31,000
`xor`	31,000	31,000
`shr`	32,000	123,000
`shl`	32,000	125,000
`rotr`	31,000	125,000
`rotl`	31,000	125,000
`eq`	55,000	83,000
`ne`	55,000	83,000
`ge`	55,000	84,000
`gt`	55,000	84,000
`le`	58,000	83,000
`lt`	58,000	84,000
`min`	88,000	146,000
`max`	89,000	145,000
`neg`	-	93,000
`not`	-	16
`select`	-	55,000
`randEuint16`	-	23,000

32-bit Encrypted Integers (`euint32`)

Function name	HCU (scalar)	HCU (non-scalar)
`add`	95,000	125,000
`sub`	95,000	125,000
`mul`	265,000	328,000
`div`	438,000	-
`rem`	792,000	-
`and`	32,000	32,000
`or`	32,000	32,000
`xor`	32,000	32,000
`shr`	32,000	163,000
`shl`	32,000	162,000
`rotr`	32,000	160,000
`rotl`	32,000	163,000
`eq`	82,000	86,000
`ne`	83,000	85,000
`ge`	84,000	118,000
`gt`	84,000	118,000
`le`	84,000	117,000
`lt`	83,000	117,000
`min`	117,000	182,000
`max`	117,000	180,000
`neg`	-	131,000
`not`	-	32
`select`	-	55,000
`randEuint32`	-	24,000

64-bit Encrypted integers (`euint64`)

Function name	HCU (scalar)	HCU (non-scalar)
`add`	133,000	162,000
`sub`	133,000	162,000
`mul`	365,000	596,000
`div`	715,000	-
`rem`	1,153,000	-
`and`	34,000	34,000
`or`	34,000	34,000
`xor`	34,000	34,000
`shr`	34,000	209,000
`shl`	34,000	208,000
`rotr`	34,000	209,000
`rotl`	34,000	209,000
`eq`	83,000	120,000
`ne`	84,000	118,000
`ge`	116,000	152,000
`gt`	117,000	152,000
`le`	119,000	149,000
`lt`	118,000	146,000
`min`	150,000	219,000
`max`	149,000	218,000
`neg`	-	131,000
`not`	-	63
`select`	-	55,000
`randEuint64`	-	24,000

128-bit Encrypted integers (`euint128`)

Function name	HCU (scalar)	HCU (non-scalar)
`add`	172,000	259,000
`sub`	172,000	260,000
`mul`	696,000	1,686,000
`div`	1,225,000	-
`rem`	1,943,000	-
`and`	37,000	37,000
`or`	37,000	37,000
`xor`	37,000	37,000
`shr`	37,000	272,000
`shl`	37,000	272,000
`rotr`	37,000	283,000
`rotl`	37,000	278,000
`eq`	117,000	122,000
`ne`	117,000	122,000
`ge`	149,000	210,000
`gt`	150,000	218,000
`le`	150,000	218,000
`lt`	149,000	215,000
`min`	186,000	289,000
`max`	180,000	290,000
`neg`	-	168,000
`not`	-	130
`select`	-	57,000
`randEuint128`	-	25,000

256-bit Encrypted integers (`euint256`)

Function name	HCU (scalar)	HCU (non-scalar)
`and`	38,000	38,000
`or`	38,000	38,000
`xor`	39,000	39,000
`shr`	38,000	369,000
`shl`	39,000	378,000
`rotr`	40,000	375,000
`rotl`	38,000	378,000
`eq`	118,000	152,000
`ne`	117,000	150,000
`neg`	-	269,000
`not`	-	130
`select`	-	108,000
`randEuint256`	-	30,000

Encrypted addresses (`euint160`)

When using `eaddress` (internally represented as `euint160`), the HCU costs for equality and inequality checks and select are as follows:

Function name	HCU (scalar)	HCU (non-scalar)
`eq`	115,000	125,000
`ne`	115,000	124,000
`select`	-	83,000

Additional Operations

Function name	HCU
`cast`	32
`trivialEncrypt`	32
`randBounded`	23,000-30,000

Migrate to v0.7

This document provides instructions on migrating from FHEVM v0.6 to v0.7.

From 0.6.x

Package and library

The package is now `@fhevm/solidity` instead of `FHEVM` and the library name has changed from `TFHE` to `FHE`

```
import { FHE } from "@fhevm/solidity";
```

Configuration

Configuration has been renamed from `SepoliaZamaConfig` to `SepoliaConfig`.

```
import { SepoliaConfig } from "@fhevm/solidity/config/ZamaConfig.sol";
```

Also, the function to define manually the Coprocessor has been renamed from `setFHEVM` to `setCoprocessor`, and the function to define the oracle is now integrated into `setCoprocessor`.

```
import { ZamaConfig } from "@fhevm/solidity/config/ZamaConfig.sol";
constructor () {
  FHE.setCoprocessor(ZamaConfig.getSepoliaConfig());
}
```

You can read more about Configuration on the dedicated page.

Decryption Oracle

Previously, an abstract contract `GatewayCaller` was used to request decryption. It has been replaced by `FHE.requestDecryption`:

```
function requestBoolInfinite() public {
  bytes32[] memory cts = new bytes32[](1);
  cts[0] = FHE.toBytes32(myEncryptedValue);
  FHE.requestDecryption(cts, this.myCallback.selector);
}
```

You can read more about Decryption Oracle on the dedicated page.

Deprecation of ebytes

`ebytes` has been deprecated and removed from FHEVM.

Block gas limit

Block gas limit has been removed in favor of HCU (Homomorphic Complexity Unit) limit. FHEVM 0.7.0 includes two limits:

- **Sequential homomorphic operations depth limit per transaction**: Controls HCU usage for operations that must be processed in order. This limit is set to **5,000,000** HCU.
- **Global homomorphic operations complexity per transaction**: Controls HCU usage for operations that can be processed in parallel. This limit is set to **20,000,000** HCU.

You can read more about HCU on the dedicated page.

How to Transform Your Smart Contract into a FHEVM Smart Contract?

This short guide will walk you through converting a standard Solidity contract into one that leverages Fully Homomorphic Encryption (FHE) using FHEVM. This approach lets you develop your contract logic as usual, then adapt it to support encrypted computation for privacy.

For this guide, we will focus on a voting contract example.

---

1. Start with a Standard Solidity Contract

Begin by writing your voting contract in Solidity as you normally would. Focus on implementing the core logic and functionality.

```
// Standard Solidity voting contract example
pragma solidity ^0.8.0;

contract SimpleVoting {
    mapping(address => bool) public hasVoted;
    uint64 public yesVotes;
    uint64 public noVotes;
    uint256 public voteDeadline;

    function vote(bool support) public {
        require(block.timestamp <= voteDeadline, "Too late to vote");
        require(!hasVoted[msg.sender], "Already voted");
        hasVoted[msg.sender] = true;

        if (support) {
            yesVotes += 1;
        } else {
            noVotes += 1;
        }
    }

    function getResults() public view returns (uint64, uint64) {
        return (yesVotes, noVotes);
    }
}
```

---

2. Identify Sensitive Data and Operations

Review your contract and determine which variables, functions, or computations require privacy. In this example, the vote counts (`yesVotes`, `noVotes`) and individual votes should be encrypted.

---

3. Integrate FHEVM and update your business logic accordingly.

Replace standard data types and operations with their FHEVM equivalents for the identified sensitive parts. Use encrypted types and FHEVM library functions to perform computations on encrypted data.

```
pragma solidity ^0.8.0;

import "@fhevm/solidity/lib/FHE.sol";
import {EthereumConfig} from "@fhevm/solidity/config/ZamaConfig.sol";

contract EncryptedSimpleVoting is EthereumConfig {
    enum VotingStatus {
        Open,
        DecryptionInProgress,
        ResultsDecrypted
    }
    mapping(address => bool) public hasVoted;

    VotingStatus public status;

    uint64 public decryptedYesVotes;
    uint64 public decryptedNoVotes;

    uint256 public voteDeadline;

    euint64 private encryptedYesVotes;
    euint64 private encryptedNoVotes;

    constructor() {
        encryptedYesVotes = FHE.asEuint64(0);
        encryptedNoVotes = FHE.asEuint64(0);

        FHE.allowThis(encryptedYesVotes);
        FHE.allowThis(encryptedNoVotes);
    }

    function vote(externalEbool support, bytes memory inputProof)
public {
        require(block.timestamp <= voteDeadline, "Too late to vote");
        require(!hasVoted[msg.sender], "Already voted");
        hasVoted[msg.sender] = true;
        ebool isSupport = FHE.fromExternal(support, inputProof);
        encryptedYesVotes = FHE.select(isSupport,
FHE.add(encryptedYesVotes, 1), encryptedYesVotes);
        encryptedNoVotes = FHE.select(isSupport, encryptedNoVotes,
FHE.add(encryptedNoVotes, 1));
        FHE.allowThis(encryptedYesVotes);
        FHE.allowThis(encryptedNoVotes);

    }

    function requestVoteDecryption() public {
        require(block.timestamp > voteDeadline, "Voting is not
```

```
finished");
        bytes32[] memory cts = new bytes32[](2);
        cts[0] = FHE.toBytes32(encryptedYesVotes);
        cts[1] = FHE.toBytes32(encryptedNoVotes);
        uint256 requestId = FHE.requestDecryption(cts,
this.callbackDecryptVotes.selector);
        status = VotingStatus.DecryptionInProgress;
    }

    function callbackDecryptVotes(uint256 requestId, bytes memory
cleartexts, bytes memory decryptionProof) public {
        FHE.checkSignatures(requestId, cleartexts, decryptionProof);

        (uint64 yesVotes, uint64 noVotes) = abi.decode(cleartexts,
(uint64, uint64));
        decryptedYesVotes = yesVotes;
        decryptedNoVotes = noVotes;
        status = VotingStatus.ResultsDecrypted;
    }

    function getResults() public view returns (uint64, uint64) {
        require(status == VotingStatus.ResultsDecrypted, "Results were
not decrypted");
        return (
            decryptedYesVotes,
            decryptedNoVotes
        );
    }
}
```

Adjust your contract's code to accept and return encrypted data where necessary. This may involve changing function parameters and return types to work with ciphertexts instead of plaintext values, as shown above.

- The `vote` function now has two parameters: `support` and `inputProof`.
- The `getResults` can only be called after the decryption occurred. Otherwise, the decrypted results are not visible to anyone.

However, it is far from being the main change. As this example illustrates, working with FHEVM often requires re-architecting the original logic to support privacy.

In the updated code, the logic becomes async; results are hidden until a request (to the oracle) explicitely has to be made to decrypt publically the vote results.

Conclusion

As this short guide showed, integrating with FHEVM not only requires integration with the FHEVM stack, it also requires refactoring your business logic to support mechanism to swift between encrypted and non-encrypted components of the logic.