File: ./modules/fhevm/docs/SUMMARY.md

Table of contents

File: ./modules/fhevm/docs/README.md

---

description: >- fhEVM is a technology that enables confidential smart contracts on the EVM using Fully

Homomorphic Encryption (FHE). layout: title: visible: true description: visible: true tableOfContents: visible: true outline: visible: true pagination: visible: false

Welcome to fhEVM

Get started

Learn the basics of fhEVM, set it up, and make it run with ease.

Overview	Explore the suite of fhEVM protocol.	start1.png	overview.md
Quick start with Remix	Learn and prototype in the in-browser IDE.	start4.png	overview.md
Get started with Hardhat	Develop in production-ready envrionment.	start5.png	hardhat.md

Develop a fhEVM smart contract

Start developing fhEVM smart contracts in Solidity by exploring its core features, discovering essential guides, and learning more with user-friendly tutorials.

Smart contract	Learn core Solidity library.	• Key features • Use encrypted types	build1.png
Frontend	Write a dAPP frontend.	• Set up • Build a web application	build2.png
Tutorials	Build quickly with tutorials.	• See all tutorials	build3.png

Explore more

Access to additional resources and join the Zama community.

Explanations

Explore the technical architecture of the fhEVM protocol and the underlying cryptographic principles that power it.

- Architecture overview
- FHE on blockchain
- fhEVM components
- Encryption, decryption re-encryption and computation

References

Refer to the API and access additional resources for in-depth explanations while working with fhEVM.

- API function specifications
- Repositories

Supports

Ask technical questions and discuss with the community. Our team of experts usually answers within 24 hours in working days.

- Community forum
- Discord channel
- Telegram

Developers

Collaborate with us to advance the FHE spaces and drive innovation together.

- Contribute to fhEVM
- Follow the development roadmap
- See the latest test release note
- Request a feature
- Report a bug

{% hint style="success" %} **Zama 5-Question Developer Survey**

We want to hear from you! Take 1 minute to share your thoughts and help us enhance our documentation and libraries. 👉 **Click here** to participate. {% endhint %}

File: ./modules/fhevm/docs/developer/contribute.md

Contributing

There are two ways to contribute to the Zama fhEVM:

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

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

Zama Bounty Program

Solve challenges and earn rewards:

- bounty-program - Zama's FHE Bounty Program

File: ./modules/fhevm/docs/developer/roadmap.md

Development roadmap

This document gives a preview of the upcoming features of fhEVM. In addition to what's listed here, you can submit your feature request on GitHub.

Features

Name	Description	ETA
Foundry template	Forge	Q1 '25

Operations

Name	Function name	Type	ETA
Signed Integers	`eintX`		Coming soon
Add w/ overflow check	`TFHE.safeAdd`	Binary, Decryption	Coming soon
Sub w/ overflow check	`TFHE.safeSub`	Binary, Decryption	Coming soon
Mul w/ overflow check	`TFHE.safeMul`	Binary, Decryption	Coming soon
Random signed int	`TFHE.randEintX()`	Random	-
Div	`TFHE.div`	Binary	-
Rem	`TFHE.rem`	Binary	-
Set inclusion	`TFHE.isIn()`	Binary	-

{% hint style="info" %} Random encrypted integers that are generated fully on-chain. Currently, implemented as a mockup by using a PRNG in the plain. Not for use in production! {% endhint %}

File: ./modules/fhevm/docs/frontend/cli.md

Using the CLI

The `fhevmjs` Command-Line Interface (CLI) tool provides a simple and efficient way to encrypt data for use with the blockchain's Fully Homomorphic Encryption (FHE) system. This guide explains how to install and use the CLI to encrypt integers and booleans for confidential smart contracts.

Installation

Ensure you have Node.js installed on your system before proceeding. Then, globally install the `fhevmjs` package to enable the CLI tool:

```
npm install -g fhevmjs
```

Once installed, you can access the CLI using the `fhevm` command. Verify the installation and explore available commands using:

```
fhevm help
```

To see specific options for encryption, run:

```
fhevm encrypt help
```

Encrypting Data

The CLI allows you to encrypt integers and booleans for use in smart contracts. Encryption is performed using the blockchain's FHE public key, ensuring the confidentiality of your data.

Syntax

```
fhevm encrypt --node <NODE_URL> <CONTRACT_ADDRESS> <USER_ADDRESS> <DATA
```

- **`--node`**: Specifies the RPC URL of the blockchain node (e.g., `http://localhost:8545`).
- **`<CONTRACT_ADDRESS>`**: The address of the contract interacting with the encrypted data.
- **`<USER_ADDRESS>`**: The address of the user associated with the encrypted data.
- **`<DATA:TYPE>`**: The data to encrypt, followed by its type:
  - `:64` for 64-bit integers
  - `:1` for booleans

Example Usage

Encrypt the integer `71721075` (64-bit) and the boolean `1` for the contract at `0x8Fdb26641d14a80FCCBE87BF455338Dd9C539a50` and the user at `0xa5e1defb98EFe38EBb2D958CEe052410247F4c80`:

```
fhevm encrypt --node http://localhost:8545 0x8Fdb26641d14a80FCCBE87BF45
```

File: ./modules/fhevm/docs/frontend/setup.md

Table of all addresses

Save this in your `.env` file:

Contract/Service	Address/Value
TFHE_EXECUTOR_CONTRACT	0x687408aB54661ba0b4aeF3a44156c616c6955E07
ACL_CONTRACT	0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5
PAYMENT_CONTRACT	0xFb03BE574d14C256D56F09a198B586bdfc0A9de2
KMS_VERIFIER_CONTRACT	0x9D6891A6240D6130c54ae243d8005063D05fE14b
GATEWAY_CONTRACT	0x33347831500F1e73f0ccCBb95c9f86B94d7b1123
PUBLIC_KEY_ID	0301c5dd3e2702992b7c12930b7d4defeaaa52cf
GATEWAY_URL	`https://gateway.sepolia.zama.ai/`

File: ./modules/fhevm/docs/frontend/node.md

Build with Node

This document provides instructions on how to build with `Node.js` using the `fhevmjs` library.

Install the library

First, you need to install the library:

```
# Using npm
npm install fhevmjs

# Using Yarn
yarn add fhevmjs

# Using pnpm
pnpm add fhevmjs
```

`fhevmjs` uses ESM format for web version and commonjs for node version. You need to set the [type to "commonjs" in your package.json](#) to load the correct version of fhevmjs. If your node project use `"type": "module"`, you can force the loading of the Node version by using `import { createInstance } from 'fhevmjs/node';`

Create an instance

An instance receives an object containing:

- `chainId` (optional): the chainId of the network
- `network` (optional): the Eip1193 object provided by `window.ethereum` (used to fetch the public key and/or chain id)
- `networkUrl` (optional): the URL of the network (used to fetch the public key and/or chain id)
- `publicKey` (optional): if the public key has been fetched separately (cache), you can provide it
- `gatewayUrl` (optional): the URL of the gateway to retrieve a reencryption

- `coprocessorUrl` (optional): the URL of the coprocessor

```
const { createInstance } = require("fhevmjs");

const createFhevmInstance = async () => {
  return createInstance({
    chainId: 11155111, // Sepolia chain ID
    networkUrl: "https://eth-sepolia.public.blastapi.io", // Sepolia RP
    gatewayUrl: "https://gateway.sepolia.zama.ai",
  });
};
createFhevmInstance().then((instance) => {
  console.log(instance);
});
```

You can now use your instance to encrypt parameters or do a reencryption.

File: ./modules/fhevm/docs/frontend/webpack.md

Common webpack errors

This document provides solutions for common Webpack errors encountered during the development process. Follow the steps below to resolve each issue.

Can't resolve 'tfhe_bg.wasm'

Error message: `Module not found: Error: Can't resolve 'tfhe_bg.wasm'`

Cause: In the codebase, there is a `new URL('tfhe_bg.wasm')` which triggers a resolve by Webpack.

Possible solutions: You can add a fallback for this file by adding a resolve configuration in your `webpack.config.js`:

```
resolve: {
  fallback: {
    'tfhe_bg.wasm': require.resolve('tfhe/tfhe_bg.wasm'),
  },
},
```

Buffer not defined

Error message: `ReferenceError: Buffer is not defined`

Cause: This error occurs when the Node.js `Buffer` object is used in a browser environment where it is not natively available.

Possible solutions: To resolve this issue, you need to provide browser-compatible fallbacks for Node.js core modules. Install the necessary browserified npm packages and configure

Webpack to use these fallbacks.

```
resolve: {
  fallback: {
    buffer: require.resolve('buffer/'),
    crypto: require.resolve('crypto-browserify'),
    stream: require.resolve('stream-browserify'),
    path: require.resolve('path-browserify'),
  },
},
```

Issue with importing ESM version

Error message: Issues with importing ESM version

Cause: With a bundler such as Webpack or Rollup, imports will be replaced with the version mentioned in the `"browser"` field of the `package.json`. This can cause issues with typing.

Possible solutions:

- If you encounter issues with typing, you can use this tsconfig.json using TypeScript 5.
- If you encounter any other issue, you can force import of the browser package.

Use bundled version

Error message: Issues with bundling the library, especially with SSR frameworks.

Cause: The library may not bundle correctly with certain frameworks, leading to errors during the build or runtime process.

Possible solutions: Use the prebundled version available with `fhevmjs/bundle`. Embed the library with a `<script>` tag and initialize it as shown below:

```
const start = async () => {
  await window.fhevm.initFhevm(); // load wasm needed
  const instance = window.fhevm
    .createInstance({
      kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
      aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
      network: window.ethereum,
      gatewayUrl: "https://gateway.sepolia.zama.ai/",
    })
    .then((instance) => {
      console.log(instance);
    });
};
```

File: ./modules/fhevm/docs/frontend/webapp.md

Build a web application

This document guides you through building a web application using the fhevmjs library. You can either start with a template or directly integrate the library into your project.

Using a template

`fhevmjs` is working out of the box and we recommend you to use it. We also provide three GitHub templates to start your project with everything set.

React + TypeScript

You can use [this template]() to start an application with fhevmjs, using Vite + React + TypeScript.

VueJS + TypeScript

You can also use [this template]() to start an application with fhevmjs, using Vite + Vue + TypeScript.

NextJS + Typescript

You can also use [this template]() to start an application with fhevmjs, using Next + TypeScript.

Using the mocked coprocessor for frontend

As an alternative to use the real coprocessor deployed on Sepolia to help you develop your dApp faster and without needing testnet tokens, you can use a mocked fhevm. Currently, we recommend you to use the `ConfidentialERC20` dApp example available on the `mockedFrontend` branch of the [React template](). Follow the README on this branch, and you will be able to deploy exactly the same dApp both on Sepolia as well as on the mocked coprocessor seamlessly.

Using directly the library

Step 1: Setup the library

`fhevmjs` consists of multiple files, including WASM files and WebWorkers, which can make packaging these components correctly in your setup cumbersome. To simplify this process, especially if you're developing a dApp with server-side rendering (SSR), we recommend using our CDN.

Using UMD CDN

Include this line at the top of your project.

```
<script src="https://cdn.zama.ai/fhevmjs/0.6.2/fhevmjs.umd.cjs" type="t
```

In your project, you can use the bundle import if you install `fhevmjs` package:

```
import { initFhevm, createInstance } from "fhevmjs/bundle";
```

Using ESM CDN

If you prefer You can also use the `fhevmjs` as a ES module:

```
<script type="module">
  import { initFhevm, createInstance } from "https://cdn.zama.ai/fhevmj

  await initFhevm();
  const instance = await createInstance({
    network: window.ethereum,
    kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
    aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
    gatewayUrl: "https://gateway.sepolia.zama.ai",
  });
</script>
```

Using npm package

Install the `fhevmjs` library to your project:

```
# Using npm
npm install fhevmjs

# Using Yarn
yarn add fhevmjs

# Using pnpm
pnpm add fhevmjs
```

`fhevmjs` uses ESM format. You need to set the type to "module" in your package.json. If your node project use `"type": "commonjs"` or no type, you can force the loading of the web version by using `import { createInstance } from 'fhevmjs/web';`

```
import { initFhevm, createInstance } from "fhevmjs";
```

Step 2: Initialize your project

To use the library in your project, you need to load the WASM of TFHE first with `initFhevm`.

```
import { initFhevm } from "fhevmjs/bundle";

const init = async () => {
  await initFhevm(); // Load needed WASM
};
```

Step 3: Create an instance

Once the WASM is loaded, you can now create an instance. An instance receives an object containing:

- `chainId` (optional): the chainId of the network
- `network` (optional): the Eip1193 object provided by `window.ethereum` (used to fetch the public key and/or chain id)
- `networkUrl` (optional): the URL of the network (used to fetch the public key and/or chain id)
- `publicKey` (optional): if the public key has been fetched separately (cache), you can provide it
- `gatewayUrl` (optional): the URL of the gateway to retrieve a reencryption
- `coprocessorUrl` (optional): the URL of the coprocessor

```
import { initFhevm, createInstance } from "fhevmjs/bundle";

const init = async () => {
  await initFhevm(); // Load TFHE
  return createInstance({
    kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
    aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
    network: window.ethereum,
    gatewayUrl: "https://gateway.sepolia.zama.ai/",
  });
};

init().then((instance) => {
  console.log(instance);
});
```

You can now use your instance to encrypt parameters or do a reencryption.

File: ./modules/fhevm/docs/references/repositories.md

Repositories

Explore our curated list of repositories to jumpstart your FHE development, contribute to the community, and access essential libraries and implementations.

Development templates

Quickly set up your development environment with these ready-to-use templates:

Repository	Description
fhevm-contracts	Example FHE-enabled smart contracts
fhevm-hardhat-template	Hardhat template for FHE smart contract development
fhevm-react-template	React.js template for building FHE dApps
fhevm-next-template	Next.js template for FHE-enabled dApps
fhevm-vue-template	Vue.js template for developing FHE dApps

IDE plugins

Repository	Description
fhevm-remix-plugin	Remix IDE plugin for FHE development

Zama Bounty Program

Contribute to the development of FHE technologies and earn rewards through Zama's bounty program:

- bounty-program - Explore open challenges and submit contributions to earn rewards.

Core libraries

Access the essential libraries for building and integrating FHE-enabled applications:

Repository	Description
fhevm	Solidity library for FHE operations
fhevmjs	JavaScript library for client-side FHE
fhevm-backend	Rust backend and go-ethereum modules for native and coprocessor

Core implementations

Explore the foundational implementations enabling FHE integration with blockchain systems:

Repository	Description
fhevm-go	Go implementation of the FHE Virtual Machine
zbc-go-ethereum	Modified go-ethereum with enhanced FHE support

Use these repositories to accelerate your development, integrate FHE into your applications, or contribute to the growing ecosystem.

File: ./modules/fhevm/docs/references/functions.md

Smart contracts - fhEVM API

This document provides an overview of the functions available in the `TFHE` Solidity library. The TFHE library provides functionality for working with encrypted types and performing operations on them. It implements fully homomorphic encryption (FHE) operations in Solidity.

Overview

The `TFHE` Solidity library provides essential functionality for working with encrypted data types and performing fully homomorphic encryption (FHE) operations in smart contracts. It is designed to streamline the developer experience while maintaining flexibility and performance.

Core Functionality

- **Homomorphic Operations**: Enables arithmetic, bitwise, and comparison operations on encrypted values.
- **Ciphertext-Plaintext Interoperability**: Supports operations that mix encrypted and plaintext operands, provided the plaintext operand's size does not exceed the encrypted operand's size.
    - Example: `add(uint8 a, euint8 b)` is valid, but `add(uint32 a, euint16 b)` is not.
    - Ciphertext-plaintext operations are generally faster and consume less gas than ciphertext-ciphertext operations.
- **Implicit Upcasting**: Automatically adjusts operand types when necessary to ensure compatibility during operations on encrypted data.

Key Features

- **Flexibility**: Handles a wide range of encrypted data types, including booleans, integers, addresses, and byte arrays.
- **Performance Optimization**: Prioritizes efficient computation by supporting optimized operator versions for mixed plaintext and ciphertext inputs.
- **Ease of Use**: Offers consistent APIs across all supported data types, enabling a smooth developer experience.

The library ensures that all operations on encrypted data follow the constraints of FHE while abstracting complexity, allowing developers to focus on building privacy-preserving smart contracts.

Types

Encrypted Data Types

Boolean

- `ebool`: Encrypted boolean value

Unsigned Integers

- `euint4`: Encrypted 4-bit unsigned integer
- `euint8`: Encrypted 8-bit unsigned integer
- `euint16`: Encrypted 16-bit unsigned integer
- `euint32`: Encrypted 32-bit unsigned integer
- `euint64`: Encrypted 64-bit unsigned integer
- `euint128`: Encrypted 128-bit unsigned integer
- `euint256`: Encrypted 256-bit unsigned integer

Addresses & Bytes

- `eaddress`: Encrypted Ethereum address
- `ebytes64`: Encrypted 64-byte value
- `ebytes128`: Encrypted 128-byte value
- `ebytes256`: Encrypted 256-byte value

Special Types

- `einput`: Input type for encrypted operations (bytes32)

Casting Types

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

`asEuint`

The `asEuint` functions serve three purposes:

- verify ciphertext bytes and return a valid handle to the calling smart contract;
- cast a `euintX` typed ciphertext to a `euintY` typed ciphertext, where `X != Y`;
- trivially encrypt a plaintext value.

The first case is used to process encrypted inputs, e.g. user-provided ciphertexts. Those are generally included in a transaction payload.

The second case is self-explanatory. When `X > Y`, the most significant bits are dropped. When `X < Y`, the ciphertext is padded to the left with trivial encryptions of `0`.

The third case is used to "encrypt" a public value so that it can be used as a ciphertext. Note that what we call a trivial encryption is **not** secure in any sense. When trivially encrypting a plaintext value, this value is still visible in the ciphertext bytes. More information about trivial encryption can be found here.

Examples

```
// first case
function asEuint8(bytes memory ciphertext) internal view returns (euint
// second case
function asEuint16(euint8 ciphertext) internal view returns (euint16)
// third case
function asEuint16(uint16 value) internal view returns (euint16)
```

`asEbool`

The `asEbool` functions behave similarly to the `asEuint` functions, but for encrypted boolean values.

Core Functions

Configuration

```
function setFHEVM(FHEVMConfig.FHEVMConfigStruct memory fhevmConfig) int
```

Sets the FHEVM configuration for encrypted operations.

Initialization Checks

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

Returns true if the encrypted value is initialized, false otherwise. Supported for all encrypted types (T can be ebool, euintX, eaddress, ebytesX).

Arithmetic operations

Available for euint* types:

```
function add(T a, T b) internal returns (T)
function sub(T a, T b) internal returns (T)
function mul(T a, T b) internal returns (T)
```

- Arithmetic: `TFHE.add`, `TFHE.sub`, `TFHE.mul`, `TFHE.min`, `TFHE.max`, `TFHE.neg`, `TFHE.div`, `TFHE.rem`
  - Note: `div` and `rem` operations are supported only with plaintext divisors

Arithmetic operations (`add`, `sub`, `mul`, `div`, `rem`)

Performs the operation homomorphically.

Note that division/remainder only support plaintext divisors.

Examples

```
// a + b
function add(euint8 a, euint8 b) internal view returns (euint8)
function add(euint8 a, euint16 b) internal view returns (euint16)
function add(uint32 a, euint32 b) internal view returns (euint32)

// a / b
function div(euint8 a, uint8 b) internal pure returns (euint8)
function div(euint16 a, uint16 b) internal pure returns (euint16)
function div(euint32 a, uint32 b) internal pure returns (euint32)
```

Min/Max Operations - `min`, `max`

Available for euint* types:

```
function min(T a, T b) internal returns (T)
function max(T a, T b) internal returns (T)
```

Returns the minimum (resp. maximum) of the two given values.

Examples

```
// min(a, b)
function min(euint32 a, euint16 b) internal view returns (euint32)

// max(a, b)
function max(uint32 a, euint8 b) internal view returns (euint32)
```

Unary operators (`neg`, `not`)

There are two unary operators: `neg` (`-`) and `not` (`!`). Note that since we work with unsigned integers, the result of negation is interpreted as the modular opposite. The `not` operator returns the value obtained after flipping all the bits of the operand.

{% hint style="info" %} More information about the behavior of these operators can be found at the TFHE-rs docs. {% endhint %}

Bitwise operations

- Bitwise: `TFHE.and`, `TFHE.or`, `TFHE.xor`, `TFHE.not`, `TFHE.shl`, `TFHE.shr`, `TFHE.rotl`, `TFHE.rotr`

Bitwise operations (`AND`, `OR`, `XOR`)

Unlike other binary operations, bitwise operations do not natively accept a mix of ciphertext and plaintext inputs. To ease developer experience, the `TFHE` library adds function overloads for these operations. Such overloads implicitely do a trivial encryption before actually calling the operation function, as shown in the examples below.

Available for euint* types:

```
function and(T a, T b) internal returns (T)
function or(T a, T b) internal returns (T)
function xor(T a, T b) internal returns (T)
```

Examples

```
// a & b
function and(euint8 a, euint8 b) internal view returns (euint8)

// implicit trivial encryption of `b` before calling the operator
function and(euint8 a, uint16 b) internal view returns (euint16)
```

Bit shift operations (`<<`, `>>`)

Shifts the bits of the base two representation of `a` by `b` positions.

Examples

```
// a << b
function shl(euint16 a, euint8 b) internal view returns (euint16)
// a >> b
```

```
function shr(euint32 a, euint16 b) internal view returns (euint32)
```

Rotate operations

Rotates the bits of the base two representation of `a` by `b` positions.

Examples

```
function rotl(euint16 a, euint8 b) internal view returns (euint16)
function rotr(euint32 a, euint16 b) internal view returns (euint32)
```

Comparison operation (`eq`, `ne`, `ge`, `gt`, `le`, `lt`)

{% hint style="info" %} **Note** that in the case of ciphertext-plaintext operations, since our backend only accepts plaintext right operands, calling the operation with a plaintext left operand will actually invert the operand order and call the *opposite* comparison. {% endhint %}

The result of comparison operations is an encrypted boolean (`ebool`). In the backend, the boolean is represented by an encrypted unsinged integer of bit width 8, but this is abstracted away by the Solidity library.

Available for all encrypted types:

```
function eq(T a, T b) internal returns (ebool)
function ne(T a, T b) internal returns (ebool)
```

Additional comparisons for euint* types:

```
function ge(T a, T b) internal returns (ebool)
function gt(T a, T b) internal returns (ebool)
function le(T a, T b) internal returns (ebool)
function lt(T a, T b) internal returns (ebool)
```

Examples

```
// a == b
function eq(euint32 a, euint16 b) internal view returns (ebool)

// actually returns `lt(b, a)`
function gt(uint32 a, euint16 b) internal view returns (ebool)

// actually returns `gt(a, b)`
function gt(euint16 a, uint32 b) internal view returns (ebool)
```

Multiplexer operator (`select`)

```
function select(ebool control, T a, T b) internal returns (T)
```

If control is true, returns a, otherwise returns b. Available for ebool, eaddress, and ebytes* types.

This operator takes three inputs. The first input `b` is of type `ebool` and the two others of type `euintX`. If `b` is an encryption of `true`, the first integer parameter is returned. Otherwise, the second integer parameter is returned.

Example

```
// if (b == true) return val1 else return val2
function select(ebool b, euint8 val1, euint8 val2) internal view return
  return TFHE.select(b, val1, val2);
}
```

Generating random encrypted integers

Random encrypted integers can be generated fully on-chain.

That can only be done during transactions and not on an `eth_call` RPC method, because PRNG state needs to be mutated on-chain during generation.

Example

```
// Generate a random encrypted unsigned integer `r`.
euint32 r = TFHE.randEuint32();
```

Access control functions

The `TFHE` library provides a robust set of access control functions for managing permissions on encrypted values. These functions ensure that encrypted data can only be accessed or manipulated by authorized accounts or contracts.

Permission management

Functions

```
function allow(T value, address account) internal
function allowThis(T value) internal
function allowTransient(T value, address account) internal
```

Descriptions

- **`allow`**: Grants **permanent access** to a specific address. Permissions are stored persistently in a dedicated ACL contract.
- **`allowThis`**: Grants the **current contract** access to an encrypted value.
- **`allowTransient`**: Grants **temporary access** to a specific address for the duration of the transaction. Permissions are stored in transient storage for reduced gas costs.

Access control list (ACL) overview

The `allow` and `allowTransient` functions enable fine-grained control over who can access, decrypt, and reencrypt encrypted values. Temporary permissions (`allowTransient`) are ideal for minimizing gas usage in scenarios where access is needed only within a single transaction.

Example: granting access

```
// Store an encrypted value.
euint32 r = TFHE.asEuint32(94);

// Grant permanent access to the current contract.
TFHE.allowThis(r);

// Grant permanent access to the caller.
TFHE.allow(r, msg.sender);

// Grant temporary access to an external account.
TFHE.allowTransient(r, 0x1234567890abcdef1234567890abcdef12345678);
```

Permission checks

Functions

```
function isAllowed(T value, address account) internal view returns (boo
function isSenderAllowed(T value) internal view returns (bool)
```

Descriptions

- **`isAllowed`**: Checks whether a specific address has permission to access a ciphertext.
- **`isSenderAllowed`**: Similar to `isAllowed`, but automatically checks permissions for the `msg.sender`.

{% hint style="info" %} Both functions return `true` if the ciphertext is authorized for the specified address, regardless of whether the allowance is stored in the ACL contract or in transient storage. {% endhint %}

Verifying Permissions

These functions help ensure that only authorized accounts or contracts can access encrypted values.

Example: permission verification

```
// Store an encrypted value.
euint32 r = TFHE.asEuint32(94);

// Verify if the current contract is allowed to access the value.
bool isContractAllowed = TFHE.isAllowed(r, address(this)); // returns t

// Verify if the caller has access to the value.
bool isCallerAllowed = TFHE.isSenderAllowed(r); // depends on msg.sende
```

Storage Management

Function

```
function cleanTransientStorage() internal
```

Description

- **`cleanTransientStorage`**: Removes all temporary permissions from transient storage. Use this function at the end of a transaction to ensure no residual permissions remain.

Example

```
// Clean up transient storage at the end of a function.
function finalize() public {
  // Perform operations...

  // Clean up transient storage.
  TFHE.cleanTransientStorage();
}
```

Additional Notes

- **Underlying implementation**:
  All encrypted operations and access control functionalities are performed through the underlying `Impl` library.
- **Uninitialized values**:
  Uninitialized encrypted values are treated as `0` (for integers) or `false` (for booleans) in computations.
- **Implicit casting**:
  Type conversion between encrypted integers of different bit widths is supported through implicit casting, allowing seamless operations without additional developer intervention.

File: ./modules/fhevm/docs/references/table_of_addresses.md

Table of all addresses

Save this in your `.env` file:

Contract/Service	Address/Value
TFHE_EXECUTOR_CONTRACT	0x687408aB54661ba0b4aeF3a44156c616c6955E07
ACL_CONTRACT	0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5
PAYMENT_CONTRACT	0xFb03BE574d14C256D56F09a198B586bdfc0A9de2
KMS_VERIFIER_CONTRACT	0x9D6891A6240D6130c54ae243d8005063D05fE14b
GATEWAY_CONTRACT	0x33347831500F1e73f0ccCBb95c9f86B94d7b1123
PUBLIC_KEY_ID	0301c5dd3e2702992b7c12930b7d4defeaaa52cf
GATEWAY_URL	`https://gateway.sepolia.zama.ai/`

File: ./modules/fhevm/docs/references/fhevmjs.md

Frontend - fhevmjs lib

This document provides an overview of the `fhevmjs` library, detailing its initialization, instance creation, input handling, encryption, and re-encryption processes.

fhevmjs is designed to assist in creating encrypted inputs and retrieving re-encryption data off-chain through a gateway. The library works with any fhEVM and fhEVM Coprocessors.

Init (browser)

If you are using `fhevmjs` in a web application, you need to initialize it before creating an instance. To do this, you should call `initFhevm` and wait for the promise to resolve.

```
import { FhevmInstance, createInstance } from "fhevmjs/node";

initFhevm().then(() => {
  const instance = await createInstance({
    network: window.ethereum,
    gatewayUrl: "https://gateway.sepolia.zama.ai",
    kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
    aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
  });
});
```

Create instance

This function returns an instance of fhevmjs, which accepts an object containing:

- `kmsContractAddress`: the address of the KMSVerifier contract;
- `aclContractAddress`: the address of the ACL contract;
- `networkUrl` or `network`: the URL or Eip1193 object provided by `window.ethereum` - used to fetch chainId and KMS nodes' public key
- `gatewayUrl`: the URL of the gateway - used to retrieve the public key, ZKPoK public parameters and send inputs and get reencryption
- `chainId` (optional): the chainId of the network
- `publicKey` (optional): if the public key has been fetched separately or stored in cache, you can provide it
- `publicParams` (optional): if the public params has been fetched separately or stored in cache, you can provide it

```
import { createInstance } from "fhevmjs";

const instance = await createInstance({
  networkUrl: "https://eth-sepolia.public.blastapi.io",
  gatewayUrl: "https://gateway.sepolia.zama.ai",
  kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
  aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
});
```

Using `window.ethereum` object:

```
import { FhevmInstance, createInstance } from "fhevmjs/bundle";
```

```
const instance = await createInstance({
  network: window.ethereum,
  gatewayUrl: "https://gateway.sepolia.zama.ai",
  kmsContractAddress: "0x9D6891A6240D6130c54ae243d8005063D05fE14b",
  aclContractAddress: "0xFee8407e2f5e3Ee68ad77cAE98c434e637f516e5",
});
```

Input

This method creates an encrypted input and returns an input object. It requires both the user address and the contract address to ensure the encrypted input isn't reused inappropriately in a different context. An input can include **multiple values of various types**, resulting in a single ciphertext that packs these values.

```
const userAddress = "0xa5e1defb98EFe38EBb2D958CEe052410247F4c80";
const contractAddress = "0xfCefe53c7012a075b8a711df391100d9c431c468";

const input = instance.createEncryptedInput(contractAddress, userAddres
```

input.addBool, input.add8, ...

Input object has different method to add values:

- `addBool`
- `add4`
- `add8`
- `add16`
- `add32`
- `add64`
- `add128`
- `add256`
- `addBytes64`
- `addBytes128`
- `addBytes256`
- `addAddress`

```
const input = instance.createEncryptedInput(contractAddress, userAddres

input.addBool(true);
input.add16(239);
input.addAddress("0xa5e1defb98EFe38EBb2D958CEe052410247F4c80");
input.addBool(true);
```

input.encrypt and input.send

These methods process values and return the necessary data for use on the blockchain. The `encrypt` method encrypts these values and provides parameters for use. The `send` method encrypts, dispatches the ciphertext and proof to the coprocessor, and returns the required parameters.

```
input.addBool(true);
input.addBool(true);
input.add8(4);
const inputs = await input.encrypt(); // or input.send() if using a cop

contract.myExample(
  "0xa5e1defb98EFe38EBb2D958CEe052410247F4c80",
  inputs.handles[0],
  32,
  inputs.handles[1],
  inputs.handles[2],
  true,
  inputs.inputProof,
);
```

Re-encryption

Keypair

A keypair consists of a private key and a public key, both generated by the dApp. These keys are used to reencrypt a blockchain ciphertext, allowing it to be securely transferred to user-specific keypairs.

```
// Generate the private and public key, used for the reencryption
const { publicKey, privateKey } = instance.generateKeypair();
```

Verifying that the public key used in the reencryption process belongs to the user requires the user to sign the public key linked to a specific contract address. This signature allows any ciphertext allowed for the user and the contract can be reencrypted using the signed public key. To streamline user interaction during the signature process, we utilize the EIP712 standard as the object to be signed.

```
// Create an EIP712 object for the user to sign.
const eip712 = instance.createEIP712(publicKey, CONTRACT_ADDRESS);
```

This `eip712` can be signed using `eth_signTypedData_v4` for example in a browser:

```
const params = [USER_ADDRESS, JSON.stringify(eip712)];
const signature = await window.ethereum.request({ method: "eth_signType
```

Note: it is recommended to store the keypair and the signature in the user's browser to avoid re-requesting signature on every user connection.

Re-encryption

Reencrypt method will use the `gatewayUrl` to get the reencryption of a ciphertext and decrypt it.

```
const handle = await erc20.balanceOf(userAddress); // returns the handl
const myBalance = await instance.reencrypt(handle, privateKey, publicKe
```

File: ./modules/fhevm/docs/smart_contracts/contracts.md

fhevm-contracts

This guide explains how to use the fhEVM Contracts standard library. This library provides secure, extensible, and pre-tested Solidity templates designed for developing smart contracts on fhEVM using the TFHE library.

Overview

The **fhEVM Contracts standard library** streamlines the development of confidential smart contracts by providing templates and utilities for tokens, governance, and error management. These contracts have been rigorously tested by Zama's engineers and are designed to accelerate development while enhancing security.

Installation

Install the library using your preferred package manager:

```
# Using npm
npm install fhevm-contracts

# Using Yarn
yarn add fhevm-contracts

# Using pnpm
pnpm add fhevm-contracts
```

Example

Local testing with the mock network

When testing your contracts locally, you can use the `SepoliaZamaFHEVMConfig` which provides a mock configuration for local development and testing. This allows you to test your contracts without needing to connect to a real network:

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

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

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

Deploying to Ethereum Sepolia

When deploying to Sepolia, you can use the `SepoliaZamaFHEVMConfig` which provides the correct configuration for the Sepolia testnet:

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

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

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

Best practices for contract inheritance

When inheriting from configuration contracts, the order of inheritance is critical. Since constructors are evaluated from left to right in Solidity, you must inherit the configuration contract first to ensure proper initialization.

✓ **Correct Order**:

```
contract MyERC20 is SepoliaZamaFHEVMConfig, ConfidentialERC20 { ... }
```

✗ **Wrong order**:

```
contract MyERC20 is ConfidentialERC20, SepoliaZamaFHEVMConfig { ... }
```

Available contracts

For a list of all available contracts see the page [See all tutorials]

File: ./modules/fhevm/docs/smart_contracts/loop.md

If sentences

This document explains how to handle loops when working with Fully Homomorphic Encryption (FHE), specifically when a loop break is based on an encrypted condition.

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 = TFHE.asEuint(6); // Could be a value between 0 and 10
euint8 x = TFHE.asEuint(0);
// some code
while(TFHE.lt(x, maxValue)){
    x = TFHE.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 `TFHE.select` inside the loop.

Suggested approach

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

```
euint8 maxValue = TFHE.asEuint(6); // Could be a value between 0 and 10
euint8 x;
// some code
for (uint32 i = 0; i < 10; i++) {
    euint8 toAdd = TFHE.select(TFHE.lt(x, maxValue), 2, 0);
    x = TFHE.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 `TFHE.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 encr
// ideally, the user already owns some amounts of both tokens and has p
function swapTokensForTokens(einput encryptedAmountAIn, einput encrypte
  euint32 encryptedAmountA = TFHE.asEuint32(encryptedAmountAIn, inputPr
  euint32 encryptedAmountB = TFHE.asEuint32(encryptedAmountBIn, inputPr

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

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

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

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

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

File: ./modules/fhevm/docs/smart_contracts/operations.md

Operations on encrypted types

This document outlines the operations supported on encrypted types in the `TFHE` 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	`TFHE.add`	`+`	Binary
Subtract	`TFHE.sub`	`-`	Binary
Multiply	`TFHE.mul`	`*`	Binary
Divide (plaintext divisor)	`TFHE.div`		Binary
Reminder (plaintext divisor)	`TFHE.rem`		Binary
Negation	`TFHE.neg`	`-`	Unary
Min	`TFHE.min`		Binary
Max	`TFHE.max`		Binary

{% hint style="info" %} Division (TFHE.div) and remainder (TFHE.rem) operations are currently supported only with plaintext divisors. {% endhint %}

Bitwise operations

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

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

The shift operators `TFHE.shr` and `TFHE.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, `TFHE.shr(euint64 x, 70)` is equivalent to `TFHE.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	`TFHE.eq`		Binary
Not equal	`TFHE.ne`		Binary

Greater than or equal	`TFHE.ge`		Binary
Greater than	`TFHE.gt`		Binary
Less than or equal	`TFHE.le`		Binary
Less than	`TFHE.lt`		Binary

Ternary operation

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

Name	**Function name**	**Symbol**	**Type**
Select	`TFHE.select`		Ternary

Random operations

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

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

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

Overload operators

The `TFHE` library supports operator overloading for encrypted integers (e.g., `+`, `-`, `*`, `&`) using the Solidity `using for` syntax. These overloaded operators currently perform unchecked operations, meaning they do not include overflow checks.

Example
Overloaded operators make code more concise:

```
euint64 a = TFHE.asEuint64(42);
euint64 b = TFHE.asEuint64(58);
euint64 sum = a + b; // Calls TFHE.add under the hood
```

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 = TFHE.euint256(25);  // age will never exceed 255
```

```
euint64 percentage = TFHE.euint256(75);  // percentage is 0-100
```

✓ Instead, use the smallest appropriate type:

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

Use scalar operands when possible to save gas

Some TFHE 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 = TFHE.add(x,TFHE.asEuint(42));
```

✓ Than this one:

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

Despite both leading to the same encrypted result!

Beware of overflows of TFHE arithmetic operators

TFHE 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(einput encryptedAmount, bytes calldata inputProof) public
  euint32 mintedAmount = TFHE.asEuint32(encryptedAmount, inputProof);
  totalSupply = TFHE.add(totalSupply, mintedAmount);
  balances[msg.sender] = TFHE.add(balances[msg.sender], mintedAmount);
  TFHE.allowThis(balances[msg.sender]);
  TFHE.allow(balances[msg.sender], msg.sender);
}
```

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

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

Additional Resources

- For detailed API specifications, visit the [fhEVM API Documentation].
- Check our [Roadmap] for upcoming features or submit a feature request on [GitHub].
- Join the discussion on the [Community Forum].

{% hint style="success" %} **Zama 5-Question Developer Survey**

We want to hear from you! Take 1 minute to share your thoughts and helping us enhance our documentation and libraries. 👉 **[Click here]** to participate. {% endhint %}

File: ./modules/fhevm/docs/smart_contracts/conditions.md

Branching in FHE

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.

Overview

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 `TFHE.select` function, which acts as a ternary operator for encrypted values.

Using `TFHE.select` for conditional logic

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

```
TFHE.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(einput encryptedValue, bytes calldata inputProof) external
  // Convert the encrypted input to an encrypted 64-bit integer
  euint64 bid = TFHE.asEuint64(encryptedValue, inputProof);

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

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

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

{% hint style="info" %} This is a simplified example to demonstrate the functionality. For a complete implementation with proper error handling and additional features, see the Blind Auction contract example. {% endhint %}

How It Works

- **Comparison**:
  - The `TFHE.lt` function compares `highestBid` and `bid`, returning an `ebool` (`isAbove`) that indicates whether the new bid is higher.
- **Selection**:
  - The `TFHE.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 `TFHE.allowThis`.

Key Considerations

- **Value change behavior:** Each time `TFHE.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 `TFHE.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., `TFHE.allowThis`, `TFHE.allow`) to ensure the updated ciphertexts are authorized for use in future computations or transactions.

Summary

- **`TFHE.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.

For more information on the supported operations, see the fhEVM API documentation.

File: ./modules/fhevm/docs/smart_contracts/error_handling.md

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

For a complete implementation of error handling, see our reference contracts:

- EncryptedErrors.sol - Base error handling contract
- ConfidentialERC20WithErrors.sol - Example usage in an ERC20 token

The following contract 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 = TFHE.asEuint8(0);           // Code 0: No error
  NOT_ENOUGH_FUNDS = TFHE.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 = TFHE.le(amount, balances[from]);

  // Log the error state: NO_ERROR or NOT_ENOUGH_FUNDS
  setLastError(TFHE.select(canTransfer, NO_ERROR, NOT_ENOUGH_FUNDS), ms

  // Perform the transfer operation conditionally
  balances[to] = TFHE.add(balances[to], TFHE.select(canTransfer, amount
  TFHE.allowThis(balances[to]);
  TFHE.allow(balances[to], to);

  balances[from] = TFHE.sub(balances[from], TFHE.select(canTransfer, am
  TFHE.allowThis(balances[from]);
  TFHE.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.

4. **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.

7. **Conditional updates**:

    - Use the `TFHE.select` function to update balances and log errors based on the transfer condition (`canTransfer`).

9. **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,
  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.

3. **Scalable error tracking**:

    - Logs errors per user, making it easy to identify and debug specific issues.

5. **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.

File: ./modules/fhevm/docs/

smart_contracts/inputs.md

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.

{% hint style="info" %} Understanding how encryption, decryption and reencryption works is a prerequisite before implementation, see [Encryption, Decryption, Re-encryption, and Computation]() {% endhint %}

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. **`einput`**: Refers to the index of the encrypted parameter, 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 myExample(
  address account,
  uint id,
  bool isAllowed,
  einput param1,
  einput param2,
  einput param3,
  bytes calldata inputProof
) public {
  // Function logic here
}
```

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

Client-Side implementation

To interact with such a function, developers can use the [fhevmjs] library to create and manage encrypted inputs. Below is an example implementation:

```
import { createInstances } from "../instance";
import { getSigners, initSigners } from "../signers";

await initSigners(); // Initialize signers
const signers = await getSigners();

const instance = await createInstances(this.signers);
// Create encrypted inputs
const input = instance.createEncryptedInput(contractAddress, userAddres
const inputs = input.add64(64).addBool(true).add8(4).encrypt(); // Encr

// Call the smart contract function with encrypted inputs
contract.myExample(
  "0xa5e1defb98EFe38EBb2D958CEe052410247F4c80", // Account address
  32, // Plaintext parameter
  true, // Plaintext boolean parameter
  inputs.handles[0], // Handle for the first parameter
  inputs.handles[1], // Handle for the second parameter
  inputs.handles[2], // Handle for the third parameter
  inputs.inputProof, // Proof to validate all encrypted inputs
);
```

In this example:

- **`add64`, `addBool`, and `add8`**: Specify the types and values of inputs to encrypt.
- **`encrypt`**: Generates the encrypted inputs and the zero-knowledge proof.

Validating encrypted inputs

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

Example validation that goes along the client-Side implementation

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(
    einput encryptedAmount,
    einput encryptedToggle,
    bytes calldata inputProof
  ) public {
    // Validate and convert the encrypted inputs
    euint64 amount = TFHE.asEuint64(encryptedAmount, inputProof);
    ebool toggleFlag = TFHE.asEbool(encryptedToggle, inputProof);

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

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

  // 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 `encryptedERC20.sol` smart contract

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

```
function transfer(
  address to,
  einput encryptedAmount,
  bytes calldata inputProof
) public {
  // Verify the provided encrypted amount and convert it into an encryp
  euint64 amount = TFHE.asEuint64(encryptedAmount, inputProof);

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

How validation works

1. **Input verification**:
   The `TFHE.asEuintXX` function ensures that the input is a valid ciphertext with a corresponding ZKPoK.
2. **Type conversion**:
   The function transforms the `einput` into the appropriate encrypted type (`euintXX`, `ebool`, etc.) 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.

File: ./modules/fhevm/docs/smart_contracts/architecture_overview.md

Architectural overview

The fhEVM architecture provides the foundation for confidential smart contracts on EVM-compatible blockchains. At its core is FHE, a cryptographic technique enabling computations directly on encrypted data, ensuring privacy at every stage.

This system relies on three key types:

- The **public key:** used for encrypting data.
- The **private key:** used for decryption and securely managed by the Key Management System or KMS
- The **evaluation key:** enabling encrypted computations performed by the coprocessor.

The fhEVM leverages Zama's TFHE library, integrating seamlessly with blockchain environments to address transparency, composability, and scalability challenges. Its hybrid architecture combines:

- **On-chain smart contracts** for encrypted state management and access controls.
- **Off-chain coprocessors** for resource-intensive FHE computations.
- **The Gateway** to coordinate between blockchain, KMS, and coprocessors.
- **The KMS** for secure cryptographic key management and proof validation.

This architecture enables developers to write private, composable smart contracts using symbolic execution and zero-knowledge proofs, ensuring data confidentiality and computational integrity.

For a detailed exploration of the fhEVM architecture, including components, workflows, and deployment models, see Architecture Overview.

File: ./modules/fhevm/docs/smart_contracts/asEXXoperators.md

asEbool, asEuintXX, asEaddress and asEbytesXX operations

This documentation covers the `asEbool`, `asEuintXX`, `asEaddress` and `asEbytesXX` operations provided by the TFHE 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 three main use cases:

1. **Trivial encryption**: Converting plaintext values to encrypted types
2. **Type casting**: Converting between different encrypted types
3. **Input handling**: Processing encrypted inputs with proofs

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 TFHE 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`
- `bytes` → `ebytesXX`
- `address` → `eaddress`

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

Example

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

Trivial encryption of `ebytesXX` types

The `TFHE.padToBytesXX` functions facilitate this trivial encryption process for byte arrays, ensuring compatibility with `ebytesXX` types. These functions:

- Pad the provided byte array to the appropriate length (`64`, `128`, or `256` bytes).
- Prevent runtime errors caused by improperly sized input data.
- Work seamlessly with `TFHE.asEbytesXX` for trivial encryption.

> **Important**: Trivial encryption does NOT provide any privacy guarantees. The

input data remains fully visible on the blockchain. Only use trivial encryption when working with public values that need to interact with actual encrypted data.

Workflow

1. **Pad Input Data**: Use the `padToBytesXX` functions to ensure your byte array matches the size requirements.
2. **Encrypt the Padded Data**: Use `TFHE.asEbytesXX` to encrypt the padded byte array into the corresponding encrypted type.
3. **Grant Access**: Use `TFHE.allowThis` and `TFHE.allow`optionally, if you want to persist allowance for those variables for later use.

Example: Trivial Encryption with `ebytesXX`

Below is an example demonstrating how to encrypt and manage `ebytes64`, `ebytes128`, and `ebytes256` types:

```
function trivialEncrypt() public {
  // Encrypt a 64-byte array
  ebytes64 yBytes64 = TFHE.asEbytes64(
    TFHE.padToBytes64(
      hex"19d179e0cc7e816dc944582ed4f5652f5951900098fc2e0a15a7ea4dc8cfa
    )
  );
  TFHE.allowThis(yBytes64);
  TFHE.allow(yBytes64, msg.sender);

  // Encrypt a 128-byte array
  ebytes128 yBytes128 = TFHE.asEbytes128(
    TFHE.padToBytes128(
      hex"13e7819123de6e2870c7e83bb764508e22d7c3ab8a5aee6bdfb26355ef0d3
    )
  );
  TFHE.allowThis(yBytes128);
  TFHE.allow(yBytes128, msg.sender);

  // Encrypt a 256-byte array
  ebytes256 yBytes256 = TFHE.asEbytes256(
    TFHE.padToBytes256(
      hex"d179e0cc7e816dc944582ed4f5652f5951900098fc2e0a15a7ea4dc8cfa4e
    )
  );
  TFHE.allowThis(yBytes256);
  TFHE.allow(yBytes256, msg.sender);
}
```

2. Casting between encrypted types

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

- `euint32` → `euint64`
- `ebytes128` → `ebytes256`

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`	`TFHE.asEuintXX`
`ebool`	`euintX`	`TFHE.asEuintXX`
`euintX`	`ebool`	`TFHE.asEboolXX`

{% hint style="info" %} Casting between encrypted types is efficient and often necessary when handling data with differing precision requirements. {% endhint %}

Workflow for encrypted types

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

3. Encrypted input

Overview

Encrypted input casting is the process of interpreting a handle (ciphertext reference) and its proof as a specific encrypted type. This ensures the validity of the input before it is used in computations.

Encrypted inputs is in depth explained in the following section: encrypted inputs

Example

```
euint64 encryptedValue = TFHE.asEuint64(einputHandle, inputProof); // I
```

Details

Encrypted input casting validates:

1. The input handle references a valid ciphertext.
2. The accompanying proof matches the expected type.

For more information, see the Encrypetd inputs documentation

Overall operation summary

Casting Type	Function	Input Type	Output Type
Trivial encryption	`TFHE.asEuintXX(x)`	`uintX`	`euintX`
	`TFHE.asEbool(x)`	`bool`	`ebool`
	`TFHE.asEbytesXX(x)`	`bytesXX`	`ebytesXX`
	`TFHE.asEaddress(x)`	`address`	`eaddress`
Conversion between types	`TFHE.asEuintXX(x)`	`euintXX`/`ebool`	`euintYY`
	`TFHE.asEbool(x)`	`euintXX`	`ebool`
Encrypted input	`TFHE.asEuintXX(x, y)`	`einput`, `bytes` proof	`euintX`
	`TFHE.asEbool(x, y)`	`einput`,`bytes` proof	`ebool`
	`TFHE.asEbytesXX(x, y)`	`einput`,`bytes` proof	`ebytesXX`
	`TFHE.asEaddress(x, y)`	`einput`, `bytes` proof	`eaddress`

File: ./modules/fhevm/docs/smart_contracts/configure.md

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 **TFHE library** and **Gateway 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.

Key components configured automatically

1. **TFHE library**: Sets up encryption parameters and cryptographic keys.
2. **Gateway**: Manages secure cryptographic operations, including reencryption and 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.

ZamaFHEVMConfig.sol

This configuration contract initializes the **fhEVM environment** with required encryption parameters.

Import based on your environment:

```
// For Ethereum Sepolia
import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so
```

Purpose:

- Sets encryption parameters such as cryptographic keys and supported ciphertext types.
- Ensures proper initialization of the FHEVM environment.

Example: using Sepolia configuration

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

import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so

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

ZamaGatewayConfig.sol

To perform decryption or reencryption, your contract must interact with the **Gateway**, which acts as a secure bridge between the blockchain, coprocessor, and Key Management System (KMS).

Import based on your environment

```
// For Ethereum Sepolia
import { SepoliaZamaGatewayConfig } from "fhevm/config/ZamaGatewayConfi
```

Purpose

- Configures the Gateway for secure cryptographic operations.
- Facilitates reencryption and decryption requests.

Example: Configuring the gateway with Sepolia settings

```
import "fhevm/lib/TFHE.sol";
import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so
import { SepoliaZamaGatewayConfig } from "fhevm/config/ZamaGatewayConfi
import "fhevm/gateway/GatewayCaller.sol";

contract Test is SepoliaZamaFHEVMConfig, SepoliaZamaGatewayConfig, Gate
  constructor() {
    // Gateway and FHEVM environment initialized automatically
  }
}
```

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

Summary

By leveraging prebuilt configuration contracts like `ZamaFHEVMConfig.sol` and `ZamaGatewayConfig.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.

File: ./modules/fhevm/docs/smart_contracts/gas.md

Gas estimation in fhEVM

This guide explains how to estimate gas costs for Fully Homomorphic Encryption (FHE) operations in your smart contracts on Zama's fhEVM. Understanding gas consumption is critical for designing efficient confidential smart contracts.

Overview

FHE operations in fhEVM are computationally intensive, resulting in higher gas costs compared to standard Ethereum operations. This is due to the complex mathematical operations required to ensure privacy and security.

Types of gas in fhEVM

1. **Native Gas**:

    - Standard gas used for operations on the underlying EVM chain.
    - On fhEVM, native gas consumption is approximately 20% higher than in mocked environments.

4. **FHEGas**:

- Represents gas consumed by FHE-specific computations.
- A new synthetic kind of gas consumed by FHE-specific computations.
- FHEGas is tracked in each block by the FHEGasLimit contract to prevent DDOS attacks.
- If too many FHE operations are requested in the same block, the transaction will revert once the FHEGas block limit is reached.
- FHEGas is consistent across both mocked and real fhEVM environments.

Note: Gas values provided are approximate and may vary based on network conditions, implementation details, and contract complexity.

Measuring gas consumption

To monitor gas usage during development, use the following tools:

- **`getFHEGasFromTxReceipt`**:
  - Extracts FHEGas consumption from a transaction receipt.
  - Works only in mocked fhEVM environments, but gives the exact same value as in non-mocked environments.
  - Import as: `import { getFHEGasFromTxReceipt } from "../coprocessorUtils";`
- **`.gasUsed` from ethers.js transaction receipt**:
  - Standard ethers.js transaction receipt property that returns the native gas used.
  - In mocked mode, this value underestimates real native gas usage by ~20%.
  - Works in both mocked and real fhEVM environments, as it's a standard Ethereum transaction property.

Example: gas measurement

The following code demonstrates how to measure both FHEGas and native gas during a transaction:

```
import { getFHEGasFromTxReceipt } from "../coprocessorUtils";

// ...

const tx = await this.erc20["transfer(address,bytes32,bytes)"](
  this.signers.bob.address,
  encryptedTransferAmount.handles[0],
  encryptedTransferAmount.inputProof,
);
const receipt = await tx.wait();
expect(receipt?.status).to.eq(1);

if (network.name === "hardhat") {
  // The getFHEGasFromTxReceipt function only works in mocked mode (har
  // but returns the exact same FHEGas value that would be consumed on
  const FHEGasConsumed = getFHEGasFromTxReceipt(receipt);
  console.log("FHEGas Consumed:", FHEGasConsumed);
}

console.log("Native Gas Consumed:", transaction.gasUsed);
```

FHEGas limit

The current devnet has a FHEGas limit of **10,000,000** per block. Here's what you need to know:

- If you send a transaction that exceeds this limit or if the FHEGas block limit is exceeded, depending on other previous transaction in the same block:
  - The transaction will revert
  - Any native gas fees (but not FHEGas) will still be charged
  - You should do one of the following:
    - Reduce the number of FHE operations in your transaction
    - Wait for the next block when the FHEGas limit resets
    - Split your operations across multiple transactions

FHEGas costs for common operations

Boolean operations (`ebool`)

Function Name	FHEGas Cost
`and/or/xor`	26,000
`not`	30,000

Unsigned integer operations

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

4-bit Encrypted Integers (`euint4`)

function name	FHEGas
`add/sub`	65,000
`add/sub` (scalar)	65,000
`mul`	150,000
`mul` (scalar)	88,000
`div` (scalar)	139,000
`rem` (scalar)	286,000
`and/or/xor`	32,000
`shr/shl`	116,000
`shr/shl` (scalar)	35,000
`rotr/rotl`	116,000

`rotr/rotl` (scalar)	35,000
`eq/ne`	51,000
`ge/gt/le/lt`	70,000
`min/max`	121,000
`min/max` (scalar)	121,000
`neg`	60,000
`not`	33,000
`select`	45,000

8-bit Encrypted integers (`euint8`)

Function name	FHEGas
`add/sub`	94,000
`add/sub` (scalar)	94,000
`mul`	197,000
`mul` (scalar)	159,000
`div` (scalar)	238,000
`rem` (scalar)	460,000
`and/or/xor`	34,000
`shr/shl`	133,000
`shr/shl` (scalar)	35,000
`rotr/rotl`	133,000
`rotr/rotl` (scalar)	35,000
`eq/ne`	53,000
`ge/gt/le/lt`	82,000
`min/max`	128,000
`min/max` (scalar)	128,000
`neg`	95,000
`not`	34,000
`select`	47,000
`randEuint8()`	100,000

16-bit Encrypted integers (`euint16`)

Function name	FHEGas
`add/sub`	133,000
`add/sub` (scalar)	133,000
`mul`	262,000
`mul` (scalar)	208,000
`div` (scalar)	314,000
`rem` (scalar)	622,000
`and/or/xor`	34,000

`shr/shl`	153,000
`shr/shl` (scalar)	35,000
`rotr/rotl`	153,000
`rotr/rotl` (scalar)	35,000
`eq/ne`	54,000
`ge/gt/le/lt`	105,000
`min/max`	153,000
`min/max` (scalar)	150,000
`neg`	131,000
`not`	35,000
`select`	47,000
`randEuint16()`	100,000

32-bit Encrypted Integers (`euint32`)

Function name	FHEGas
`add/sub`	162,000
`add/sub` (scalar)	162,000
`mul`	359,000
`mul` (scalar)	264,000
`div` (scalar)	398,000
`rem` (scalar)	805,000
`and/or/xor`	35,000
`shr/shl`	183,000
`shr/shl` (scalar)	35,000
`rotr/rotl`	183,000
`rotr/rotl` (scalar)	35,000
`eq/ne`	82,000
`ge/gt/le/lt`	128,000
`min/max`	183,000
`min/max` (scalar)	164,000
`neg`	160,000
`not`	36,000
`select`	50,000
`randEuint32()`	100,000

64-bit Encrypted integers (`euint64`)

Function name	FHEGas
`add/sub`	188,000
`add/sub` (scalar)	188,000
`mul`	641,000

`mul` (scalar)	356,000
`div` (scalar)	584,000
`rem` (scalar)	1,095,000
`and/or/xor`	38,000
`shr/shl`	227,000
`shr/shl` (scalar)	38,000
`rotr/rotl`	227,000
`rotr/rotl` (scalar)	38,000
`eq/ne`	86,000
`ge/gt/le/lt`	156,000
`min/max`	210,000
`min/max` (scalar)	192,000
`neg`	199,000
`not`	37,000
`select`	53,000
`randEuint64()`	100,000

128-bit Encrypted integers (`euint128`)

Function name	FHEGas
`add/sub`	218,000
`add/sub` (scalar)	218,000
`mul`	1,145,000
`mul` (scalar)	480,000
`div` (scalar)	857,000
`rem` (scalar)	1,499,000
`and/or/xor`	41,000
`shr/shl`	282,000
`shr/shl` (scalar)	41,000
`rotr/rotl`	282,000
`rotr/rotl` (scalar)	41,000
`eq/ne`	88,000
`ge/gt/le/lt`	190,000
`min/max`	241,000
`min/max` (scalar)	225,000
`neg`	248,000
`not`	38,000
`select`	70,000

256-bit Encrypted integers (`euint256`)

function name	FHEGas

`add/sub`	253,000
`add/sub` (scalar)	253,000
`mul`	2,045,000
`mul` (scalar)	647,000
`div` (scalar)	1,258,000
`rem` (scalar)	2,052,000
`and/or/xor`	44,000
`shr/shl`	350,000
`shr/shl` (scalar)	44,000
`rotr/rotl`	350,000
`rotr/rotl` (scalar)	44,000
`eq/ne`	100,000
`ge/gt/le/lt`	231,000
`min/max`	277,000
`min/max` (scalar)	264,000
`neg`	309,000
`not`	39,000
`select`	90,000

eAddress

Function name	FHEGas
`eq/ne`	90,000

Additional Operations

Function name	FHEGas
`cast`	200
`trivialEncrypt` (basic)	100-800
`trivialEncrypt` (extended)	1,600-6,400
`randBounded`	100,000
`select`	43,000-300,000
`rand`	100,000-400,000

Fixing Failed Transactions in MetaMask

To resolve a failed transaction due to gas limits:

1. Open MetaMask and go to Settings
2. Navigate to Advanced Settings
3. Enable "Customize transaction nonce"
4. When resending the transaction:

    - Use the same nonce as the failed transaction

- Set an appropriate gas limit under 10M
- Adjust other parameters as needed

This allows you to "replace" the failed transaction with a valid one using the correct gas parameters.

File: ./modules/fhevm/docs/smart_contracts/mocked.md

Mocked mode

This document provides an overview of mocked mode in the fhEVM framework, explaining how it enables faster development and testing of smart contracts that use Fully Homomorphic Encryption (FHE).

Overview

Mocked mode is a development and testing feature provided in the `fhEVM` framework that allows developers to simulate the behavior of Fully Homomorphic Encryption (FHE) without requiring the full encryption and decryption processes to be performed. This makes development and testing cycles faster and more efficient by replacing actual cryptographic operations with mocked values, which behave similarly to encrypted data but without the computational overhead of true encryption.

How to use mocked mode

1. Hardhat template

Mocked mode is currently supported in the Zama Hardhat template. Developers can enable mocked mode to simulate encrypted operations while building and testing smart contracts locally. The Hardhat template includes pre-configured scripts and libraries to simplify the setup process for mocked mode.

Refer to the [Quick start - Hardhat] guide for instructions on using the Hardhat template.

2. Foundry (coming soon)

Mocked mode support is planned for Foundry in future releases.

How mocked mode works

For faster testing iterations, instead of launching all the tests on the local fhEVM node, which could last several minutes, you can use a mocked version of the fhEVM by running `pnpm test`. The same tests should (almost always) pass as is, without any modification; neither the JavaScript files nor the Solidity files need to be changed between the mocked and the real version.

The mocked mode does **not** actually perform real encryption for encrypted types and instead runs the tests on a local Hardhat node, which is implementing the original EVM (i.e., non-fhEVM).

Additionally, the mocked mode will let you use all the Hardhat-related special testing and debugging methods, such as `evm_mine`, `evm_snapshot`, `evm_revert`, etc., which are very helpful for testing.

Development workflow with mocked mode

When developing confidential contracts, we recommend to use first the mocked version of fhEVM for faster testing with `pnpm test` and coverage computation via `pnpm coverage`, this will lead to a better developer experience.

It's essential to run tests of the final contract version using the real fhEVM. You can do this by running `pnpm test` before deployment.

To run the mocked tests use either:

```
pnpm test
```

Or equivalently:

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

In mocked mode, all tests should pass in few seconds instead of few minutes, allowing a better developer experience. Furthermore, getting the coverage of tests is only possible in mocked mode. Just use the following command:

```
pnpm coverage
```

Or equivalently:

```
npx hardhat coverage
```

Then open the file `coverage/index.html` to see the coverage results. This will increase security by pointing out missing branches not covered yet by the current test suite.

{% hint style="info" %} Due to limitations in the `solidity-coverage` package, test coverage computation does not work with tests that use the `evm_snapshot` Hardhat testing method. {% endhint %}

If you are using Hardhat snapshots in your tests, we recommend adding the `[skip-on-coverage]` tag at the end of your test description. Here's an example:

```
import { expect } from 'chai';
import { ethers, network } from 'hardhat';

import { createInstances, decrypt8, decrypt16, decrypt32, decrypt64 } f
import { getSigners, initSigners } from '../signers';
import { deployRandFixture } from './Rand.fixture';

describe('Rand', function () {
  before(async function () {
    await initSigners();
    this.signers = await getSigners();
```

```
});

beforeEach(async function () {
  const contract = await deployRandFixture();
  this.contractAddress = await contract.getAddress();
  this.rand = contract;
  this.instances = await createInstances(this.signers);
});

it('64 bits generate with upper bound and decrypt', async function ()
  const values: bigint[] = [];
  for (let i = 0; i < 5; i++) {
    const txn = await this.rand.generate64UpperBound(262144);
    await txn.wait();
    const valueHandle = await this.rand.value64();
    const value = await decrypt64(valueHandle);
    expect(value).to.be.lessThanOrEqual(262141);
    values.push(value);
  }
  // Expect at least two different generated values.
  const unique = new Set(values);
  expect(unique.size).to.be.greaterThanOrEqual(2);
});

it('8 and 16 bits generate and decrypt with hardhat snapshots [skip-o
  if (network.name === 'hardhat') {
    // snapshots are only possible in hardhat node, i.e in mocked mod
    this.snapshotId = await ethers.provider.send('evm_snapshot');
    const values: number[] = [];
    for (let i = 0; i < 5; i++) {
      const txn = await this.rand.generate8();
      await txn.wait();
      const valueHandle = await this.rand.value8();
      const value = await decrypt8(valueHandle);
      expect(value).to.be.lessThanOrEqual(0xff);
      values.push(value);
    }
    // Expect at least two different generated values.
    const unique = new Set(values);
    expect(unique.size).to.be.greaterThanOrEqual(2);

    await ethers.provider.send('evm_revert', [this.snapshotId]);
    const values2: number[] = [];
    for (let i = 0; i < 5; i++) {
      const txn = await this.rand.generate8();
      await txn.wait();
      const valueHandle = await this.rand.value8();
      const value = await decrypt8(valueHandle);
      expect(value).to.be.lessThanOrEqual(0xff);
      values2.push(value);
    }
    // Expect at least two different generated values.
    const unique2 = new Set(values2);
```

```
      expect(unique2.size).to.be.greaterThanOrEqual(2);
    }
  });
});
```

- **The first test** always runs in both mocked mode (`pnpm test`) and coverage mode (`pnpm coverage`).
- **The second test** runs only while testing mocked mode (`pnpm test`) since it uses snapshots which are only available there. The test is skipped in coverage mode due to the `[skip-on-coverage]` suffix in its description. It also checks that the network is 'hardhat' to avoid failures in non-mocked environments.

File: ./modules/fhevm/docs/smart_contracts/d_re_ecrypt_compute.md

Encryption, decryption, re-encryption, and computation

This document introduces the core cryptographic operations in the fhEVM system, including how data is encrypted, decrypted, re-encrypted and computed upon while maintaining privacy.

The fhEVM system ensures end-to-end confidentiality by leveraging Fully Homomorphic Encryption (FHE). The encryption, decryption, re-encryption, and computation processes rely on a coordinated flow of information and cryptographic keys across the fhEVM components. This section details how these operations work and outlines the role of the FHE keys in enabling secure and private processing.

FHE keys and their locations

1. **Public Key**:

   - **Location**: Directly accessible from the frontend or the smart contract.
   - **Role**: Used to encrypt plaintext data before submission to the blockchain or during contract execution.

4. **Private Key**:

   - **Location**: Stored securely in the Key Management System (KMS).
   - **Role**: Used for decrypting ciphertexts when necessary, either to verify results or enable user-specific plaintext access.

7. **Evaluation Key**:

   - **Location**: Stored on the coprocessor.
   - **Role**: Enables operations on ciphertexts (e.g., addition, multiplication) without decrypting them.

[image]

Workflow: encryption, decryption, and processing

Encryption

Encryption is the starting point for any interaction with the fhEVM system, ensuring that data is protected before it is transmitted or processed.

- **How It Works**:
    1. The **frontend** or client application uses the **public key** to encrypt user-provided plaintext inputs.
    2. The encrypted data (ciphertext) is submitted to the blockchain as part of a transaction or stored for later computation.
- **Data Flow**:
    - **Source**: Frontend or smart contract.
    - **Destination**: Blockchain (for storage and symbolic execution) or coprocessor (for processing).

[image]

You can read about the implementation details in our encryption guide.

Computation

Encrypted computations are performed using the **evaluation key** on the coprocessor.

- **How it works**:
    1. The blockchain initiates symbolic execution, generating a set of operations to be performed on encrypted data.
    2. These operations are offloaded to the **coprocessor**, which uses the **evaluation key** to compute directly on the ciphertexts.
    3. The coprocessor returns updated ciphertexts to the blockchain for storage or further use.
- **Data flow**:
    - **Source**: Blockchain smart contracts (via symbolic execution).
    - **Processing**: Coprocessor (using the evaluation key).
    - **Destination**: Blockchain (updated ciphertexts).

[image]

Decryption

Decryption is used when plaintext results are required for contract logic or for presentation to a user. After the decryption is performed on the blockchain, the decrypted result is exposed to everyone who has access to the blockchain.

[image]

- **How it works**:
  Validators on the blockchain do not possess the private key needed for decryption. Instead, the **Key Management System (KMS)** securely holds the private key. If plaintext values are needed, the process is facilitated by a service called the **Gateway**, which provides two options:

  1. **For Smart contract logic**:
     The Gateway acts as an oracle service, listening for decryption request events emitted by the blockchain. Upon receiving such a request, the Gateway interacts with the KMS to decrypt the ciphertext and sends the plaintext back to the smart contract via a callback function.
  2. **For dApps**:
     If a dApp needs plaintext values, the Gateway enables re-encryption of the ciphertext. The KMS securely re-encrypts the ciphertext with the dApp's public key, ensuring that only the dApp can decrypt and access the plaintext.

- **Data flow**:

  - **Source**: Blockchain or dApp (ciphertext).
  - **Processing**: KMS performs decryption or re-encryption via the Gateway.
  - **Destination**: Plaintext is either sent to the smart contract or re-encrypted and delivered to the dApp.

[image]

re-encryption

You can read about the implementation details in our decryption guide.

---

4. Re-encryption

Re-encryption enables encrypted data to be securely shared or reused under a different encryption key without ever revealing the plaintext. This process is essential for scenarios where data needs to be accessed by another contract, dApp, or user while maintaining confidentiality.

- **How it work:** Re-encryption is facilitated by the **Gateway** in collaboration with the **Key Management System (KMS).**

  1. The Gateway receives a re-encryption request, which includes details of the original ciphertext and the target public key.
  2. The KMS securely decrypts the ciphertext using its private key and re-encrypts the data with the recipient's public key.
  3. The re-encrypted ciphertext is then sent to the intended recipient.

- **Data flow:**

  1. **Source**:

     1. The process starts with an original ciphertext retrieved from the blockchain or a dApp.

2. **Processing**:

    1. The Gateway forwards the re-encryption request to the KMS.
    2. The KMS handles decryption and re-encryption using the appropriate keys.

3. **Destination:**

    1. The re-encrypted ciphertext is delivered to the target entity, such as a dApp, user, or another contract.

[image]

re-encryption process

Client-side implementation

Re-encryption is initiated on the client side via the **Gateway service** using the [`fhevmjs`](fhevmjs) library. Here's the general workflow:

1. **Retrieve the ciphertext**:

    - The dApp calls a view function (e.g., `balanceOf`) on the smart contract to get the handle of the ciphertext to be re-encrypted.

3. **Generate and sign a keypair**:

    - The dApp generates a keypair for the user.
    - The user signs the public key to ensure authenticity.

6. **Submit re-encryption request**:

    - The dApp calls the Gateway, providing the following information:

        - The ciphertext handle.
        - The user's public key.
        - The user's address.
        - The smart contract address.
        - The user's signature.

13. **Decrypt the re-encrypted ciphertext**:

    - The dApp receives the re-encrypted ciphertext from the Gateway.
    - The dApp decrypts the ciphertext locally using the private key.

You can read our re-encryption guide explaining how to use it.

Tying It All Together

The flow of information across the fhEVM components during these operations highlights how the system ensures privacy while maintaining usability:

Operation		
Encryption	Public Key	Frontend encrypts plaintext → Ciphertext submitted to blockchain or coprocessor.

Computation	Evaluation Key	Blockchain initiates computation → Coprocessor processes ciphertext using evaluation key → Updated ciphertext returned to blockchain.
Decryption	Private Key	Blockchain or Gateway sends ciphertext to KMS → KMS decrypts using private key → Plaintext returned to authorized requester (e.g., frontend or specific user).
Re-encryption	Private and Target Keys	Blockchain or Gateway sends ciphertext to KMS → KMS re-encrypts using private key and target key → Updated ciphertext returned to blockchain, frontend, or other contract/user.

This architecture ensures that sensitive data remains encrypted throughout its lifecycle, with decryption or re-encryption only occurring in controlled, secure environments. By separating key roles and processing responsibilities, fhEVM provides a scalable and robust framework for private smart contracts.

File: ./modules/fhevm/docs/smart_contracts/debug_decrypt.md

Debugging with `debug.decrypt[XX]`

This guide explains how to use the `debug.decrypt[XX]` functions for debugging encrypted data in mocked environments during development with fhEVM.

{% hint style="warning" %} The `debug.decrypt[XX]` functions should not be used in production as they rely on private keys. {% endhint %}

Overview

The `debug.decrypt[XX]` functions allow you to decrypt encrypted handles into plaintext values. This feature is useful for debugging encrypted operations such as transfers, balance checks, and other computations involving FHE-encrypted data.

Key points

- **Environment**: The `debug.decrypt[XX]` functions work **only in mocked environments** (e.g., `hardhat` network).
- **Production limitation**: In production, decryption is performed asynchronously via the Gateway and requires an authorized onchain request.
- **Encrypted types**: The `debug.decrypt[XX]` functions supports various encrypted types, including integers, booleans, and `ebytesXX`.
- **Bypass ACL authorization**: The `debug.decrypt[XX]` functions allow decryption without ACL authorization, useful for verifying encrypted operations during development and testing.

Supported functions

Integer decryption

Decrypts encrypted integers of different bit-widths (`euint4`, `euint8`, ..., `euint256`).

Function Name	Returns	Encrypted Type
`decrypt4`	`bigint`	`euint4`
`decrypt8`	`bigint`	`euint8`
`decrypt16`	`bigint`	`euint16`
`decrypt32`	`bigint`	`euint32`
`decrypt64`	`bigint`	`euint64`
`decrypt128`	`bigint`	`euint128`
`decrypt256`	`bigint`	`euint256`

Boolean decryption

Decrypts encrypted booleans (`ebool`).

Function Name	Returns	Encrypted Type
`decryptBool`	`boolean`	`ebool`

Byte array decryption

Decrypts encrypted byte arrays of various sizes (`ebytesXX`).

Function Name	Returns	Encrypted Type
`decryptEbytes64`	`string`	`ebytes64`
`decryptEbytes128`	`string`	`ebytes128`
`decryptEbytes256`	`string`	`ebytes256`

Address decryption

Decrypts encrypted addresses.

Function Name	Returns	Encrypted Type
`decryptAddress`	`string`	`eaddress`

Function usage

Example: decrypting encrypted values

```
import { debug } from "../utils";

// Decrypt a 64-bit encrypted integer
const handle64: bigint = await this.erc20.balanceOf(this.signers.alice)
const plaintextValue: bigint = await debug.decrypt64(handle64);
console.log("Decrypted Balance:", plaintextValue);
```

{% hint style="info" %} To utilize the debug functions, import the utils.ts file. {% endhint %}

For a more complete example, refer to the [ConfidentialERC20 test file].

Example: decrypting byte arrays

```
// Decrypt a 128-byte encrypted value
const ebytes128Handle: bigint = ...; // Get handle for the encrypted by
const decryptedBytes: string = await debug.decryptEbytes128(ebytes128Ha
console.log("Decrypted Bytes:", decryptedBytes);
```

How it works

Verifying types

Each decryption function includes a **type verification step** to ensure the provided handle matches the expected encrypted type. If the type is mismatched, an error is thrown.

```
function verifyType(handle: bigint, expectedType: number) {
  const typeCt = handle >> 8n;
  if (Number(typeCt % 256n) !== expectedType) {
    throw "Wrong encrypted type for the handle";
  }
}
```

Environment checks

{% hint style="danger" %} The functions only work in the `hardhat` network. Attempting to use them in a production environment will result in an error. {% endhint %}

```
if (network.name !== "hardhat") {
  throw Error("This function can only be called in mocked mode");
}
```

Best practices

- **Use only for debugging**: These functions require access to private keys and are meant exclusively for local testing and debugging.
- **Production decryption**: For production, always use the asynchronous Gateway-based decryption.

File: ./modules/fhevm/docs/smart_contracts/random.md

Generate random numbers

This document explains how to generate cryptographically secure random encrypted numbers fully on-chain using the `TFHE` 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.

{% hint style="info" %} 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. {% endhint %}

Basic usage

The `TFHE` 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 = TFHE.randEbool();       // Random encrypted boolean
euint4 r4 = TFHE.randEuint4();     // Random 4-bit number
euint8 r8 = TFHE.randEuint8();     // Random 8-bit number
euint16 r16 = TFHE.randEuint16();  // Random 16-bit number
euint32 r32 = TFHE.randEuint32();  // Random 32-bit number
euint64 r64 = TFHE.randEuint64();  // Random 64-bit number
euint128 r128 = TFHE.randEuint128(); // Random 128-bit number
euint256 r256 = TFHE.randEuint256(); // Random 256-bit number
```

Example: Random Boolean

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

Bounded random numbers

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

```
// Generate random numbers with upper bounds
euint8 r8 = TFHE.randEuint8(100);      // Random number between 0-99
euint16 r16 = TFHE.randEuint16(1000);  // Random number between 0-999
euint32 r32 = TFHE.randEuint32(1000000); // Random number between 0-999
```

Example: Random bumber with upper bound

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

Random encrypted bytes

To generate larger random values, you can use encrypted bytes. These are ideal for scenarios requiring high-precision or high-entropy data.

```
// Generate random encrypted bytes
ebytes64 rb64 = TFHE.randEbytes64();    // 64 bytes (512 bits)
ebytes128 rb128 = TFHE.randEbytes128(); // 128 bytes (1024 bits)
ebytes256 rb256 = TFHE.randEbytes256(); // 256 bytes (2048 bits)
```

Example: Random Bytes

```
function randomBytes256() public returns (ebytes256) {
  return TFHE.randEbytes256();
}
```

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.

File: ./modules/fhevm/docs/smart_contracts/types.md

Supported types

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

Introduction

The `TFHE` 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 `TFHE` library will support encrypted integers with overflow checking, but with the trade-off of exposing limited information about the operands.

{% hint style="info" %} Encrypted integers with overflow checking will soon be available in the `TFHE` library. These will allow reversible arithmetic operations but may reveal some information about the input values. {% endhint %}

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 `TFHE` library currently supports the following encrypted types:

Type	Supported
`ebool`	Yes
`euint4`	Yes
`euint8`	Yes
`euint16`	Yes
`euint32`	Yes
`euint64`	Yes
`euint128`	Yes
`euint256`	Yes
`eaddress`	Yes
`ebytes64`	Yes
`ebytes128`	Yes
`ebytes256`	Yes
`eint8`	No, coming soon
`eint16`	No, coming soon
`eint32`	No, coming soon
`eint64`	No, coming soon
`eint128`	No, coming soon
`eint256`	No, coming soon

{% hint style="info" %} Higher-precision integer types are available in the `TFHE-rs` library and can be added to `fhEVM` as needed. {% endhint %}

File: ./modules/fhevm/docs/smart_contracts/key_concepts.md

Key features

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
- **Gateway configuration**: Configure secure gateway 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**: `euint4`, `euint8`, `euint16`, `euint32`, `euint64`, `euint128`, `euint256`
- **Addresses**: `eaddress`
- **Bytes**: `ebytes64`, `ebytes128`, `ebytes256`
- **Input**: `einput` 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**: `TFHE.asEbool` converts encrypted integers to encrypted booleans
- **Casting to encrypted types**: `TFHE.asEuintX` converts plaintext values to encrypted types
- **Casting to encrypted addresses**: `TFHE.asEaddress` converts plaintext addresses to encrypted addresses
- **Casting to encrypted bytes**: `TFHE.asEbytesX` converts plaintext bytes to encrypted bytes

For more information see use of encrypted types.

Confidential computation

fhEVM enables symbolic execution of encrypted operations, supporting:

- **Arithmetic:** `TFHE.add`, `TFHE.sub`, `TFHE.mul`, `TFHE.min`, `TFHE.max`, `TFHE.neg`, `TFHE.div`, `TFHE.rem`
  - Note: `div` and `rem` operations are supported only with plaintext divisors
- **Bitwise:** `TFHE.and`, `TFHE.or`, `TFHE.xor`, `TFHE.not`, `TFHE.shl`, `TFHE.shr`, `TFHE.rotl`, `TFHE.rotr`
- **Comparison:** `TFHE.eq`, `TFHE.ne`, `TFHE.lt`, `TFHE.le`, `TFHE.gt`, `TFHE.ge`

- **Advanced:** `TFHE.select` for branching on encrypted conditions, `TFHE.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**: `TFHE.allow`, `TFHE.allowThis` grants permanent permissions for ciphertexts.
- **Transient access**: `TFHE.allowTransient` provides temporary access for specific transactions.
- **Validation**: `TFHE.isSenderAllowed` ensures that only authorized entities can interact with ciphertexts.

For more information see ACL.

File: ./modules/fhevm/docs/smart_contracts/architecture_overview/fhe-on-blockchain.md

FHE on blockchain

This page gives an overview of Fully Homomorphic Encryption (FHE) and its implementation on the blockchain by fhEVM. It provides the essential architectural concepts needed to start building with fhEVM.

FHE overview

FHE is an advanced cryptographic technique that allows computations to be performed directly on encrypted data, without the need for decryption. This ensures that data remains confidential throughout its entire lifecycle, even during processing.

With FHE:

- Sensitive data can be securely encrypted while still being useful for computations.
- The results of computations are encrypted, maintaining end-to-end privacy.

FHE operates using three types of keys, each playing a crucial role in its functionality:

Private key

- **Purpose**: - for securely decrypting results - Decrypts ciphertexts to recover the original

plaintext.
- **Usage in fhEVM**: Managed securely by the Key Management System (KMS) using a threshold MPC protocol. This ensures no single entity ever possesses the full private key.

Public key

- **Purpose**: - for encrypting data. - Encrypts plaintexts into ciphertexts.
- **Usage in fhEVM**: Shared globally to allow users and smart contracts to encrypt inputs or states. It ensures that encrypted data can be processed without revealing the underlying information.

Evaluation key

- **Purpose**: - for performing encrypted computations - Enables efficient homomorphic operations (e.g., addition, multiplication) on ciphertexts.
- **Usage in fhEVM**: Provided to FHE nodes (on-chain validators or off-chain coprocessors) to perform computations on encrypted data while preserving confidentiality.

These three keys work together to facilitate private and secure computations, forming the foundation of FHE-based systems like fhEVM.

[image]

Overview of FHE Keys and their roles

FHE to Blockchain: From library to fhEVM

Building on Zama's FHE library

At its core, the fhEVM is built on Zama's high-performance FHE library, **TFHE-rs**, written in Rust. This library implements the TFHE (Torus Fully Homomorphic Encryption) scheme and is designed to perform secure computations on encrypted data efficiently.

> **Info**: For detailed documentation and implementation examples on the `tfhe-rs` library, visit the TFHE-rs documentation.

However, integrating a standalone FHE library like TFHE-rs into a blockchain environment involves unique challenges. Blockchain systems demand efficient processing, public verifiability, and seamless interoperability, all while preserving their decentralized nature. To address these requirements, Zama designed the fhEVM, a system that bridges the computational power of TFHE-rs with the transparency and scalability of blockchain technology.

Challenges in blockchain integration

Integrating FHE into blockchain systems posed several challenges that needed to be addressed to achieve the goals of confidentiality, composability, and scalability:

1. **Transparency and privacy**: Blockchains are inherently transparent, where all on-chain data is publicly visible. FHE solves this by keeping all sensitive data encrypted, ensuring privacy without sacrificing usability.

2. **Public verifiability**: On-chain computations need to be verifiable by all participants. This required a mechanism to confirm the correctness of encrypted computations without revealing their inputs or outputs.
3. **Composability**: Smart contracts needed to interact seamlessly with each other, even when operating on encrypted data.
4. **Performance and scalability**: FHE computations are resource-intensive, and blockchain systems require high throughput to remain practical.

To overcome these challenges, Zama introduced a hybrid architecture for fhEVM that combines:

- **On-chain** functionality for managing state and enforcing access controls.
- **Off-chain** processing via a coprocessor to execute resource-intensive FHE computations.

File: ./modules/fhevm/docs/smart_contracts/architecture_overview/fhevm-components.md

fhEVM components

This document gives a detailed explanantion of each component of fhEVM and illustrate how they work together to perform computations.

Overview

The fhEVM architecture is built around four primary components, each contributing to the system's functionality and performance. These components work together to enable the development and execution of private, composable smart contracts on EVM-compatible blockchains. Below is an overview of these components and their responsibilities:

fhEVM Smart Contracts	Smart contracts deployed on the blockchain to manage encrypted data and interactions.	Includes the Access Control List (ACL) contract, `TFHE.sol` Solidity library, `Gateway.sol` and other FHE-enabled smart contracts.
Gateway	An off-chain service that bridges the blockchain with the cryptographic systems like KMS and coprocessor.	Acts as an intermediary to forward the necessary requests and results between the blockchain, the KMS, and users.
Coprocessor	An off-chain computational engine designed to execute resource-intensive FHE operations.	Executes symbolic FHE operations, manages ciphertext storage, and ensures efficient computation handling.
Key Management System (KMS)	A decentralized cryptographic service that securely manages FHE keys and validates operations.	Manages the global FHE key (public, private, evaluation), performs threshold decryption, and validates ZKPoKs.

[image]

High level overview of the fhEVM Architecture

Developer workflow:

As a developer working with fhEVM, your workflow typically involves two key elements:

1. **Frontend development**:
   You create a frontend interface for users to interact with your confidential application. This includes encrypting inputs using the public FHE key and submitting them to the blockchain.
2. **Smart contract development**:
   You write Solidity contracts deployed on the same blockchain as the fhEVM smart contracts. These contracts leverage the `TFHE.sol` library to perform operations on encrypted data. Below, we explore the major components involved.

fhEVM smart contracts

fhEVM smart contracts include the Access Control List (ACL) contract, `TFHE.sol` library, and related FHE-enabled contracts.

Symbolic execution in Solidity

fhEVM implements **symbolic execution** to optimize FHE computations:

- **Handles**: Operations on encrypted data return "handles" (references to ciphertexts) instead of immediate results.
- **Lazy Execution**: Actual computations are performed asynchronously, offloading resource-intensive tasks to the coprocessor.

This approach ensures high throughput and flexibility in managing encrypted data.

Zero-Knowledge proofs of knowledge (ZKPoKs)

fhEVM incorporates ZKPoKs to verify the correctness of encrypted inputs and outputs:

- **Validation**: ZKPoKs ensure that inputs are correctly formed and correspond to known plaintexts without revealing sensitive data.
- **Integrity**: They prevent misuse of ciphertexts and ensure the correctness of computations.

By combining symbolic execution and ZKPoKs, fhEVM smart contracts maintain both privacy and verifiability.

Coprocessor

The coprocessor is the backbone for handling computationally intensive FHE tasks.

Key functions:

1. **Execution**: Performs operations such as addition, multiplication, and comparison on

encrypted data.
2. **Ciphertext management**: Stores encrypted inputs, states, and outputs securely, either off-chain or in a dedicated on-chain database.

Gateway

The Gateway acts as the bridge between the blockchain, coprocessor, and KMS.

Key functions:

- **API for developers**: Exposes endpoints for submitting encrypted inputs, retrieving outputs, and managing ciphertexts.
- **Proof validation**: Forwards ZKPoKs to the KMS for verification.
- **Off-chain coordination**: Relays encrypted data and computation results between on-chain and off-chain systems.

The Gateway simplifies the development process by abstracting the complexity of cryptographic operations.

Key management system (KMS)

The KMS securely manages the cryptographic backbone of fhEVM by maintaining and distributing the global FHE keys.

Key functions:

- **Threshold decryption**: Uses Multi-Party Computation (MPC) to securely decrypt ciphertexts without exposing the private key to any single entity.
- **ZKPoK validation**: Verifies proofs of plaintext knowledge to ensure that encrypted inputs are valid.
- **Key distribution**: Maintains the global FHE keys, which include:
  - **Public key**: Used for encrypting data (accessible to the frontend and smart contracts).
  - **Private key**: Stored securely in the KMS and used for decryption.
  - **Evaluation key**: Used by the coprocessor to perform FHE computations.

The KMS ensures robust cryptographic security, preventing single points of failure and maintaining public verifiability.

In the next section, we will dive deeper into encryption, re-encryption, and decryption processes, including how they interact with the KMS and Gateway services. For more details, see Decrypt and re-encrypt.

File: ./modules/fhevm/docs/smart_contracts/decryption/reencryption.md

Re-encryption

This document explains how to perform re-encryption. Re-encryption is required when you want a user to access their private data without it being exposed to the blockchain.

Re-encryption in fhEVM enables the secure sharing or reuse of encrypted data under a new public key without exposing the plaintext. This feature is essential for scenarios where encrypted data must be transferred between contracts, dApps, or users while maintaining its confidentiality.

{% hint style="info" %} Before implementing re-encryption, ensure you are familiar with the foundational concepts of encryption, re-encryption and computation. Refer to [Encryption, Decryption, Re-encryption, and Computation](). {% endhint %}

When to use re-encryption

Re-encryption is particularly useful for **allowing individual users to securely access and decrypt their private data**, such as balances or counters, while maintaining data confidentiality.

Overview

The re-encryption process involves retrieving ciphertext from the blockchain and performing re-encryption on the client-side. In other words we take the data that has been encrypted by the KMS, decrypt it and encrypt it with the users private key, so only he can access the information.

This ensures that the data remains encrypted under the blockchain's FHE key but can be securely shared with a user by re-encrypting it under the user's NaCl public key.

Re-encryption is facilitated by the **Gateway** and the **Key Management System (KMS)**. The workflow consists of the following:

1. Retrieving the ciphertext from the blockchain using a contract's view function.
2. Re-encrypting the ciphertext client-side with the user's public key, ensuring only the user can decrypt it.

Step 1: retrieve the ciphertext

To retrieve the ciphertext that needs to be re-encrypted, you can implement a view function in your smart contract. Below is an example implementation:

```
import "fhevm/lib/TFHE.sol";

contract ConfidentialERC20 {
  ...
  function balanceOf(account address) public view returns (bytes euint6
    return balances[msg.sender];
  }
  ...
}
```

Here, `balanceOf` allows retrieval of the user's encrypted balance stored on the blockchain.

Step 2: re-encrypt the ciphertext

Re-encryption is performed client-side using the `fhevmjs` library. Refer to the guide to learn how to include `fhevmjs` in your project. Below is an example of how to implement reencryption in a dApp:

```
import { createInstances } from "../instance";
import { getSigners, initSigners } from "../signers";
import abi from "./abi.json";
import { Contract, BrowserProvider } from "ethers";
import { createInstance } from "fhevmjs/bundle";

const CONTRACT_ADDRESS = "";

const provider = new BrowserProvider(window.ethereum);
const accounts = await provider.send("eth_requestAccounts", []);
const USER_ADDRESS = accounts[0];

await initSigners(); // Initialize signers
const signers = await getSigners();

const instance = await createInstances(this.signers);
// Generate the private and public key, used for the reencryption
const { publicKey, privateKey } = instance.generateKeypair();

// Create an EIP712 object for the user to sign.
const eip712 = instance.createEIP712(publicKey, CONTRACT_ADDRESS);

// Request the user's signature on the public key
const params = [USER_ADDRESS, JSON.stringify(eip712)];
const signature = await window.ethereum.request({ method: "eth_signType

// Get the ciphertext to reencrypt
const ConfidentialERC20 = new Contract(CONTRACT_ADDRESS, abi, signer).c
const encryptedBalance = ConfidentialERC20.balanceOf(userAddress);

// This function will call the gateway and decrypt the received value w
const userBalance = instance.reencrypt(
  encryptedBalance, // the encrypted balance
  privateKey, // the private key generated by the dApp
  publicKey, // the public key generated by the dApp
  signature, // the user's signature of the public key
  CONTRACT_ADDRESS, // The contract address where the ciphertext is
  USER_ADDRESS, // The user address where the ciphertext is
);

console.log(userBalance);
```

This code retrieves the user's encrypted balance, re-encrypts it with their public key, and decrypts it on the client-side using their private key.

Key additions to the code

- **`instance.generateKeypair()`**: Generates a public-private keypair for the user.
- **`instance.createEIP712(publicKey, CONTRACT_ADDRESS)`**: Creates an EIP712 object for signing the user's public key.
- **`instance.reencrypt()`**: Facilitates the re-encryption process by contacting the Gateway and decrypting the data locally with the private key.

File: ./modules/fhevm/docs/smart_contracts/decryption/decrypt_details.md

Decryption in depth

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

`GatewayContract` set up

The `GatewayContract` is pre-deployed on the fhEVM testnet. It uses a default relayer account specified in the `PRIVATE_KEY_GATEWAY_RELAYER` or `ADDRESS_GATEWAY_RELAYER` environment variable in the `.env` file.

Relayers are the only accounts authorized to fulfill decryption requests. The role of the `GatewayContract`, however, is to independently verify the KMS signature during execution. This ensures that the relayers cannot manipulate or send fraudulent decryption results, even if compromised. However, the relayers are still trusted to forward decryption requests on time.

`Gateway.requestDecryption` function

The interface of the `Gateway.requestDecryption` function from previous snippet is the following:

```
function requestDecryption(
    uint256[] calldata ctsHandles,
    bytes4 callbackSelector,
    uint256 msgValue,
    uint256 maxTimestamp,
    bool passSignaturesToCaller
) external virtual returns (uint256 initialCounter) {
```

Parameters

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`, `euint4`, `euint8`, `euint16`, `euint32`, `euint64` or `eaddress`.

`ct` is the list of ciphertexts that are requested to be decrypted. Calling `requestDecryption` will emit an `EventDecryption` on the `GatewayContract` contract which will be detected by a relayer. Then, 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 by the `GatewayContract` contract once the relayer fulfils the decryption request. Notice that the callback function should always follow this convention, if `passSignaturesToCaller` is set to `false`:

```
function [callbackName](uint256 requestID, XXX x_0, XXX x_1, ..., XXX x
```

Or, alternatively, if `passSignaturesToCaller` is set to `true`:

```
function [callbackName](uint256 requestID, XXX x_0, XXX x_1, ..., XXX x
```

Notice that `XXX` should be the decrypted type, which is a native Solidity type corresponding to the original ciphertext type, following this table of conventions:

Ciphertext type	Decrypted type
ebool	bool
euint4	uint8
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 `x_0`, `x_1`, ... `x_N-1` are the results of the decryption of the `ct` array values, i.e their number should be the size of the `ct` array.

`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.

`maxTimestamp` is the maximum timestamp after which the callback will not be able to receive the results of decryption, i.e the fulfillment transaction will fail in this case. This can be used for time-sensitive applications, where we prefer to reject decryption results on too old, out-of-date, values.

`passSignaturesToCaller` determines whether the callback needs to transmit signatures from the KMS or not. This is useful if the dApp developer wants to remove trust from the Gateway service and prefers to check the KMS signatures directly from within his dApp smart contract. A concrete example of how to verify the KMS signatures inside a dApp is available here in the `requestBoolTrustless` function.

> ***WARNING:*** Notice that the callback should be protected by the `onlyGateway` modifier to ensure security, as only the `GatewayContract` contract should be able to call it.

Finally, if you need to pass additional arguments to be used inside the callback, you could use any of the following utility functions during the request, which would store additional values in the storage of your smart contract:

```
function addParamsEBool(uint256 requestID, ebool _ebool) internal;

function addParamsEUint4(uint256 requestID, euint4 _euint4) internal;

function addParamsEUint8(uint256 requestID, euint8 _euint8) internal;

function addParamsEUint16(uint256 requestID, euint16 _euint16) internal

function addParamsEUint32(uint256 requestID, euint32 _euint32) internal

function addParamsEUint64(uint256 requestID, euint64 _euint64) internal

function addParamsEAddress(uint256 requestID, eaddress _eaddress) inter

function addParamsAddress(uint256 requestID, address _address) internal

function addParamsUint256(uint256 requestID, uint256 _uint) internal;
```

With their corresponding getter functions to be used inside the callback:

```
function getParamsEBool(uint256 requestID) internal;

function getParamsEUint4(uint256 requestID) internal;

function getParamsEUint8(uint256 requestID) internal;

function getParamsEUint16(uint256 requestID) internal;

function getParamsEUint32(uint256 requestID) internal;

function getParamsEUint64(uint256 requestID) internal;

function getParamsEAddress(uint256 requestID) internal;

function getParamsAddress(uint256 requestID) internal;

function getParamsUint256(uint256 requestID) internal;
```

For example, see this snippet where we add two `uint256`s during the request call, to make them available later during the callback:

```
pragma solidity ^0.8.24;

import "fhevm/lib/TFHE.sol";
import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so
import { SepoliaZamaGatewayConfig } from "fhevm/config/ZamaGatewayConfi
import "fhevm/gateway/GatewayCaller.sol";

contract TestAsyncDecrypt is SepoliaZamaFHEVMConfig, SepoliaZamaGateway
  euint32 xUint32;
  uint32 public yUint32;

  constructor() {
      xUint32 = TFHE.asEuint32(32);
      TFHE.allowThis(xUint32);
  }

  function requestUint32(uint32 input1, uint32 input2) public {
      uint256[] memory cts = new uint256[](1);
      cts[0] = Gateway.toUint256(xUint32);
      uint256 requestID = Gateway.requestDecryption(cts, this.callbackU
      addParamsUint256(requestID, input1);
      addParamsUint256(requestID, input2);
  }

  function callbackUint32(uint256 requestID, uint32 decryptedInput) pub
    uint256[] memory params = getParamsUint256(requestID);
    unchecked {
        uint32 result = uint32(params[0]) + uint32(params[1]) + decrypt
        yUint32 = result;
        return result;
    }
}
```

When the decryption request is fufilled by the relayer, the `GatewayContract` contract, when calling the callback function, will also emit the following event:

```
event ResultCallback(uint256 indexed requestID, bool success, bytes res
```

The first argument is the `requestID` of the corresponding decryption request, `success` is a boolean assessing if the call to the callback succeeded, and `result` is the bytes array corresponding to the return data from the callback.

In your hardhat tests, if you sent some transactions which are requesting one or several decryptions and you wish to await the fulfillment of those decryptions, you should import the two helper methods `initGateway` and `awaitAllDecryptionResults` from the `asyncDecrypt.ts` utility file. This would work both when testing on a fhEVM node or in mocked mode. Here is a simple hardhat test for the previous `TestAsyncDecrypt` contract (more examples can be seen here):

```
import { initGateway, awaitAllDecryptionResults } from "../asyncDecrypt
import { getSigners, initSigners } from "../signers";
import { expect } from "chai";
import { ethers } from "hardhat";

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

  beforeEach(async function () {
    const contractFactory = await ethers.getContractFactory("TestAsyncD
    this.contract = await contractFactory.connect(this.signers.alice).d
  });

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

You should initialize the gateway by calling `initGateway` at the top of the `before` block - more specifically, before doing any transaction which could involve a decryption request. Notice that when testing on the fhEVM, a decryption is fulfilled usually 2 blocks after the request, while in mocked mode the fulfillment will always happen as soon as you call the `awaitAllDecryptionResults` helper function. A good way to standardize hardhat tests is hence to always call the `awaitAllDecryptionResults` function which will ensure that all pending decryptions are fulfilled in both modes.

File: ./modules/fhevm/docs/smart_contracts/decryption/README.md

Decryption

File: ./modules/fhevm/docs/smart_contracts/decryption/decrypt.md

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.

{% hint style="info" %} Understanding how encryption, decryption and reencryption works is a prerequisit before implementation, see [Encryption, Decryption, Re-encryption, and Computation](). {% endhint %}

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.

To learn how decryption works see [Encryption, Decryption, Re-encryption, and Computation]()

Overview

Decryption in fhEVM is an asynchronous process that involves the Gateway and Key Management System (KMS). Contracts requiring decryption must extend the GatewayCaller contract, which imports the necessary libraries and provides access to the Gateway.

Here's an example of how to request decryption in a contract:

Example: asynchronous decryption in a contract

```
pragma solidity ^0.8.24;

import "fhevm/lib/TFHE.sol";
import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so
import { SepoliaZamaGatewayConfig } from "fhevm/config/ZamaGatewayConfi
import "fhevm/gateway/GatewayCaller.sol";

contract TestAsyncDecrypt is SepoliaZamaFHEVMConfig, SepoliaZamaGateway
  ebool xBool;
  bool public yBool;

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

  function requestBool() public {
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(xBool);
    Gateway.requestDecryption(cts, this.myCustomCallback.selector, 0, b
  }

  function myCustomCallback(uint256 /*requestID*/, bool decryptedInput)
    yBool = decryptedInput;
    return yBool;
  }
```

Key additions to the code

1. **Configuration imports**: The configuration contracts are imported to set up the FHEVM environment and Gateway.

   ```
   import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfi
   import { SepoliaZamaGatewayConfig } from "fhevm/config/ZamaGatewayC
   ```

2. **`GatewayCaller` import**:
   The `GatewayCaller` contract is imported to enable decryption requests.

   ```
   import "fhevm/gateway/GatewayCaller.sol";
   ```

Next steps

Explore advanced decryption techniques and learn more about re-encryption:

- Decryption in depth
- Re-encryption

File: ./modules/fhevm/docs/smart_contracts/write_contract/foundry.md

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 `TFHE.sol` imports from `fhevm/lib/TFHE.sol` to `fhevm/mocks/TFHE.sol` in your solidity source files.

File: ./modules/fhevm/docs/smart_contracts/acl/README.md

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 `TFHE.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 `TFHE.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.

Syntactic sugar:

- `TFHE.allowThis(ciphertext)` is shorthand for `TFHE.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.md).

File: ./modules/fhevm/docs/smart_contracts/acl/acl_examples.md

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**: `TFHE.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.

Transient allowance

- **Function**: `TFHE.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.

Syntactic sugar

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

Example: granting permissions in a multi-contract setup

```
import "fhevm/lib/TFHE.sol";
import { SepoliaZamaFHEVMConfig } from "fhevm/config/ZamaFHEVMConfig.so

contract SecretGiver is SepoliaZamaFHEVMConfig {
  SecretStore public secretStore;

  constructor() {
    secretStore = new SecretStore();
  }

  function giveMySecret() public {
    // Create my secret - asEuint16 gives automatically transient allow
    euint16 mySecret = TFHE.asEuint16(42);

    // Allow temporarily the SecretStore contract to manipulate `mySecr
    TFHE.allowTransient(mySecret, address(secretStore));

    // Call `secretStore` with `mySecret`
    secretStore.storeSecret(mySecret);
  }
}

contract SecretStore is SepoliaZamaFHEVMConfig {
  euint16 public secretResult;

  function storeSecret(euint16 callerSecret) public {
    // Verify that the caller has also access to this ciphertext
    require(TFHE.isSenderAllowed(callerSecret), "The caller is not auth

    // do some FHE computation (result is automatically put in the ACL
    euint16 computationResult = TFHE.add(callerSecret, 3);

    // then store the resulting ciphertext handle in the contract stora
    secretResult = computationResult;

    // Make the temporary allowance for this ciphertext permanent to le
    TFHE.allowThis(secretResult); // this is strictly equivalent to `TF
  }
}
```

Automatic transient allowance

Some functions automatically grant transient allowances to the calling contract, simplifying workflow. These include:

- **Type Conversion**:
  - `TFHE.asEuintXX()`, `TFHE.asEbool()`, `TFHE.asEaddress()`
- **Random Value Generation**:
  - `TFHE.randXX()`
- **Computation Results**:
  - `TFHE.add()`, `TFHE.select()`

Example: random value generation

```
function randomize() public {
  // Generate a random encrypted value with transient allowance
  euint64 random = TFHE.randEuint64();

  // Convert the transient allowance into a permanent one
  TFHE.allowThis(random);
}
```

🔧 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: Encrypted ERC20 attack

Consider an **Encrypted 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 `TFHE.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, bytes calldata i
  // Ensure the sender is authorized to access the encrypted amount
  require(TFHE.isSenderAllowed(encryptedAmount), "Unauthorized access t

  // Proceed with further logic
  euint64 amount = TFHE.asEuint64(encryptedAmount);
  ...
}
```

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

ACL for reencryption

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

Due to the reencryption 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 Encrypted ERC-20

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

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

  euint64 newBalanceFrom = TFHE.sub(balances[from], TFHE.select(canTran
  balances[from] = newBalanceFrom;
  // Allow this new balance for both the contract and the owner.
  TFHE.allowThis(newBalanceFrom);
  TFHE.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](acl).

File: ./modules/fhevm/docs/getting-started/overview.md

Overview

[image]

To **integrate FHE at the protocol level** or operate an **FHE-enabled network**, fhEVM offers the fhevm backend modules. These repositories include the foundational implementations that enables FHE in blockchain systems, ensuring that privacy remains at the core of your network architecture.

Repository	Description
fhevm-backend	Rust backend & Go-Ethereum modules, enabling native or coprocessor-based FHE.
fhevm-go	Go implementation of the FHE Virtual Machine
zbc-go-ethereum	Modified go-ethereum with enhanced FHE support

File: ./modules/fhevm/docs/getting-started/overview-1/overview.md

Quick Start

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 20 minutes, you will:

- Build your first **confidential ERC20** contract that leverages FHE.
- Deploy the contract on the **Sepolia** Network.
- **Mint tokens** and **perform transactions** in FHE.
- Build a **frontend application** for your contract.

Prerequisite

- A basic understanding of **Solidity** library and **Ethereum**.
- A certain amount of **Sepolia ETH** available.
    - If you don't have enough ETH, use a Sepolia faucet to request free SepoliaETH for testing such as Alchemy Faucet or QuickNode Faucet.

What is Confidenetial ERC20

The contract that you will build with this tutorial is called `ConfidentialERC20Mintable` — a privacy-preserving ERC20 implementation that leverages **FHE** to keep balances and transactions confidential. To understand this contract, let's first introduce the foundational concepts.

RC20

ERC20 is a widely used token standard on Ethereum that defines a set of rules for creating and managing fungible tokens. These tokens are efficient but lack privacy — balances and transactions are visible to anyone on the blockchain.

Confidential ERC20

Zama's `ConfidentialERC20` introduces privacy to ERC20 tokens by storing balances and transactions in an encrypted format using FHE.

The `ConfidentialERC20` contract still supports standard ERC20 functions such as `transfer`, `approve`, `transferFrom`, `balanceOf`, and `totalSupply` but ensures these operations are processed securely with encrypted data.

To explore the implementation details of ConfidentialERC20, check out the Zama blog post.

Confidential ERC-20 Mintable

The contract that we will build in this tutorial is `ConfidentialERC20Mintable` . It's built on top of `ConfidentialERC20` by adding secure minting capabilities. This allows authorized accounts to create new tokens, while maintaining the privacy guarantees of encrypted balances and transactions.

The `ConfidentialERC20Mintable` contract ensures:

- **Enhanced privacy**: Balances are stored as encrypted values (`euint64`), preventing public inspection of account balances.
- **Secure transactions**: Token transfers are processed securely, maintaining confidentiality of amounts.
- **Owner visibility**: Only account owners can decrypt and view their balances.

Next steps

Choose your path and get started:

- **Remix Guide** – Rapid in-browser setup, great for **learning** and fast **prototyping**.
- **Hardhat Guide** – Full-fledged development environment, suitable for **production**.

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/5.-interacting-with-the-contract.md

5. Interacting with the contract

After deploying your first **fhEVM** contract using **Hardhat**, this guide shows you how to interact with it using Hardhat tasks.

Prerequisites

Before interacting with your deployed contract, ensure the following:

- **Deployment completed**: You have successfully deployed the `MyConfidentialERC20` contract (see previous section)
- **Contract address**: You have saved the deployed contract address
- **Sepolia ETH**: You have some Sepolia ETH in your wallet for transaction fees
- **Environment setup**: Your `.env` file is properly configured with your private key

Step 1: Mint tokens to your account

First, let's mint some confidential tokens to your account:

```
# Mint 1000 tokens to your account by replacing recipient-address with
npx hardhat mint --to <recipient-address> --amount 1000 --network sepol
```

Once successful, you'll see a transaction confirmation in the terminal that looks like this:

```
Starting mint process...
Contract address: 0x1...26
Minting 1000 tokens to address: 0x1...26
Transaction submitted, waiting for confirmation...
✓ Mint transaction successful!
Transaction hash: 0x1...26
1000 tokens were minted to 0x1....26
```

Step 2: Verify total supply

You can check the total supply of tokens to confirm your mint was successful:

```
# Check the total supply
npx hardhat totalSupply --network sepolia
```

Once successful, you'll see a transaction confirmation in the terminal:

```
✓ Retrieved total supply successfully
----------------------------------------
Total Supply: 1000 tokens
----------------------------------------
```

Step 3: Check your balance

To verify your account balance:

```
# Check your encrypted balance by replacing <private-key> with the priv
# You can get the private key by running 'npx hardhat get-accounts --nu
npx hardhat balance --privatekey <private-key> --network sepolia
```

Note to remind yourself what was are the private keys of the accounts of the MNEMONIC SEED you can always check it by running:

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

Once successful, you'll see a transaction confirmation in the terminal.

```
✓ Retrieved balance handle successfully
```

```
----------------------------------------
Address: 0x1..59
Balance: 1000 tokens
----------------------------------------
```

Step 4: Transfer tokens

To transfer confidential tokens to another account:

```
# Transfer tokens from your account to another address
# Replace <private-key> with the private key of the sending account
# Replace <recipient-address> with the destination wallet address
# Adjust the amount (100) as needed
npx hardhat transfer --privatekey <private-key> --to <recipient-address
```

Once successful, you'll see a transaction confirmation in the terminal.

```
Starting transfer process...
Contract address: 0x1...26
From: 0x1...59
To: 0xA...4D
Amount: 100 tokens
Encrypting and proving in 9.4s
Verifying in 8.7s
Submitting transfer transaction...
Waiting for confirmation...
----------------------------------------
✓ Transfer successful!
Transaction hash: 0xc..13
Transferred 100 tokens to 0xA..4D
```

Step 5: Verify updated balances

After the transfer, you can check both accounts balances:

```
# Check sender's updated balance after transfer
# Replace <private-key> with the private key of the senders account
npx hardhat balance --privatekey <private-key> --network sepolia

# Check recipient's updated balance after receiving transfer
# Replace <private-key> with the private key of the recipient account
npx hardhat balance --privatekey <private-key> --network sepolia
```

If both balances have changed accordingly the transaction was successful.

Available Tasks

Here's a complete list of available Hardhat tasks for interacting with your contract:

- `mint`: Mint new encrypted tokens
- `transfer`: Transfer encrypted tokens to another address
- `balance`: Check encrypted balance of an account

- `totalSupply`: Get the total token supply

For detailed help on any task, run:

```
npx hardhat help <task-name>
```

Next steps

🎉 **Congratulations on completing this tutorial!** You've taken the first step in building confidential smart contracts using **fhEVM**. It's time now to take the next step and build your own confidential dApps!

1. Resources

To continue your journey and deepen your knowledge, explore the resources below.

- Read the white paper: Understand the core technology behind fhEVM, including its cryptographic foundations and use cases.
- See more demos and tutorials: Expand your skills with hands-on demos and tutorials crafted to guide you through various real-world scenarios.
- Try out AI coding assistant: If you have a ChatGPT plus account, try out our custom ChatGPT model tailored for Solidity and fhEVM developers.

2. Tools

Use out-of-box templates and frameworks designed for developers to build confidential dApps easily.

Smart contract development

- Hardhat Template: A developer-friendly starting point for building and testing smart contracts on fhEVM.
- fhEVM Contracts Library: Access standardized contracts for encrypted operations.

Frontend development

- React.js Template: Quickly develop FHE-compatible dApps using a clean React.js setup.
- Next.js Template: Build scalable, server-rendered dApps with FHE integration.
- Vue.js Template: Develop responsive and modular dApps with FHE support in Vue.js.

3. Community

Join the community to shape the future of blockchain together with us.

- Discord: Join the community to get the latest update, have live discussion with fellow developers and Zama team.
- Community Forum: Get support on all technical questions related to fhEVM
- Zama Bounty Program: Participate to tackle challenges and earn rewards in cash.

File: ./modules/fhevm/docs/getting-

started/overview-1/hardhat/1.-setting-up-hardhat.md

1. Setting up Hardhat

This guide walks you through setting up your Hardhat environment using Zama's fhEVM Hardhat template.

Step 1: Clone the Hardhat template

1. Open the repository: [Zama fhEVM Hardhat Template]()
2. Click "Use this template" to create a new repository under your GitHub account.
3. Clone the repository you have created to your local environment
4. Copy the repository URL of your newly created repository.
5. Open a terminal and run:

```
git clone <your-new-repo-url>
cd <your-new-repo-name>
```

{% embed url="https://scribehow.com/embed/Step_1__M1Gjr6SAQuOsPyT7luekmw?removeLogo=true&skipIntro=true" %}

Step 2: Configure the environment

1. Copy the environment configuration template:

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

2. Install project dependencies: Depending on your package manager, run one of the following:

   ```
   # Using npm
   npm install

   # Using yarn
   yarn install

   # Using pnpm
   pnpm install
   ```

{% embed url="https://scribehow.com/embed/Step_2_Copy__dIIvxIkrTPWTMD-A9s3Twg?removeLogo=true&skipIntro=true" %}

Project structure overview

- **`contracts/`**: Write your Solidity smart contracts here.
- **`test/`**: Place your test scripts for smart contract testing.
- **`deploy/`**: Deployment scripts for deploying your contracts.
- **`hardhat.config.js`**: The pre-configured Hardhat setup file for deploying on Sepolia.

- **`.env`:** The environment file that stores sensitive or environment-specific variables such as private keys and API keys.

To learn more about Hardhat, refer to the official Hardhat documentation.

You are now ready to start building your confidential smart contracts with fhEVM!

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/3.-testing-in-mocked-mode.md

3. Testing in mocked mode

This tutorial walks you through performing tests in the **mocked** mode provided by the fhEVM Hardhat template.

Prerequisites

Before proceeding, ensure you have:

- A configured Hardhat project using the fhEVM Hardhat template. (See the previous section)
- Basic knowledge of Solidity and Hardhat testing. (See the Hardhat testing documentation)

{% hint style="info" %} fhEVM provides a **mocked mode** in Hardhat that allows for:

- Faster testing on a local Hardhat network.
- The ability to analyze code coverage.
- A simulated version of encrypted types (they are not truly encrypted).
- Access to Hardhat features such as snapshots (`evm_snapshot`), time manipulation (`evm_increaseTime`), and debugging (`console.log`).

To learn more about the fhEVM **mocked** mode, refer to the README in the fhEVM Hardhat template repository. {% endhint %}

Running your tests

To run tests in **mocked mode**, open a terminal in your project's root directory and execute:

```
# Using npm
npm run test

# Using yarn
yarn test

# Using pnpm
pnpm test
```

This command runs all tests locally in mocked mode. You should see the test results in your console.

{% embed url="https://scribehow.com/embed/Test__GPNkyWV-T3Kz3wOu14xWQA?removeLogo=true&skipIntro=true" %}

Refer to the [Smart contract - Mocked mode] guide for more detailed instructions on how mocked mode works.

Next steps

For most development and demonstration scenarios, mocked mode is sufficient. However, for production-ready development and a real testing environment, you need to run your tests on a real network where the coprocessor is deployed for example Sepolia. Refer to the next section on how to deploy your contract on Sepolia test network.

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/prerequisites.md

Prerequisites

This guide will walk you through installing all necessary dependencies from scratch, preparing you for developing and deploying fhEVM smart contracts using Hardhat.

To use Hardhat to build and deploy confidential smart contract, you need to have the following:

- **Node.js (v20 or later)**
- **A package manager**: `npm`, `yarn`, or `pnpm`
- **Git** for version control
- **Foundry** for using the command `cast`

If you already have these installed, skip ahead to 1. Setting Up Hardhat.

Step 1: Open your terminal

To run the installation commands, open a terminal:

- Use your system's built-in terminal (e.g., Terminal on macOS, Command Prompt or PowerShell on Windows, or a Linux shell).
- Alternatively, use the integrated terminal in a IDE such as Visual Studio Code (VS Code).

{% hint style="info" %} VS code provide the official Hardhat extension for an enhanced development experience with Hardhat. {% endhint %}

Step 2. Install Node.js and a package manager

Instead of installing Node.js directly, we recommend using Node Version Manager (`nvm`), which allows you to manage multiple versions of Node.js:

1. **Install `nvm`** following the recommended tutorials:
   - Linux
   - Windows
   - Mac OS

5. **Install Node.js** using nvm. This installs the latest version of Node.js:

```
nvm install node
```

1. **Install pnpm** (optional but recommended for better speed and efficiency). After installing Node.js, you automatically have npm, so run:

```
npm install -g pnpm
```

Alternatively, you can continue using `npm` or `yarn` if you prefer.

Step 3. Install Git

Git is required for managing source code, cloning repositories, and handling version control. Install Git based on your operating system:

- **Linux**: Use your distribution's package manager (e.g., `sudo apt-get install git` on Ubuntu/Debian).
- **Windows**: Download Git for Windows.
- **macOS**: Install via Homebrew with `brew install git` or download Git.

Step 4. Install Foundry

We will need the command `cast` in our tutorial. This command is available with installing Foundry.

1. **Install Foundry** using the official installer:

```
curl -L https://foundry.paradigm.xyz | bash
```

1. **Reload your terminal configuration**:

```
source ~/.bashrc   # Linux/WSL
# or
source ~/.zshrc    # macOS/zsh
```

1. **Run the Foundry installer**:

```
foundryup
```

This will install all Foundry components, including `cast`, `forge`, `anvil`, and `chisel`.

To verify the installation, run:

```
cast --version
```

Next Steps

You now have all the necessary dependencies installed. Continue to **1. Setting Up Hardhat** to begin creating and deploying your FHE smart contracts!

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/README.md

Hardhat

This tutorial covers the essential steps to quickly write and deploy a confidential ERC20 smart contract using the fhEVM Hardhat template.

Hardhat is a flexible, extensible development environment for Ethereum, providing a robust toolkit for compiling, testing, and deploying smart contracts. With Zama's fhEVM integration, you can develop confidential ERC20 contracts locally and then seamlessly deploy them to a real fhEVM node.

In this tutorial, you will learn to:

1. Set up a Hardhat project with the **fhEVM Hardhat Template**.
2. Write confidential contracts utilizing encrypted data types.
3. Test your contracts in **mocked mode** (for rapid feedback and coverage).
4. Deploy your confidential ERC20 contract to the Sepolia test network.
5. Interact with your deployed contract, including performing encrypted transfers.

By following these steps, you'll get familiar with the complete development workflow using Hardhat, and be able to build, test and deploy confidential ERC20 smart contracts on Sepolia testnet.

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/2.-writing-contracts.md

2. Writing contracts

This document explains how to write confidential smart contract using fhEVM in Hardhat projects.

Prerequisites

Before proceeding, ensure you have:

- A working Hardhat environment set up (see previous section).
- Basic knowledge of Solidity.
- An understanding of ERC20 tokens.

Understanding the example contract

The Hardhat template includes an example contract in the `contracts/` folder - `MyConfidentialERC20.sol`. This contract enables:

- Private ERC20 token transfers.
- Encrypted balances.
- Minting functionality for authorized addresses.

Let's break down the contract.

{% embed url="https://scribehow.com/embed/Contract_WHzYcFfeQp-wamyKwJVDhw?removeLogo=true&skipIntro=true" %}

Step 1. Importing required libraries and contracts.

```
pragma solidity ^0.8.24;

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

- **`TFHE.sol`**: The core Solidity library of Zama's fhEVM. It enables encrypted data type like `euint64`, secures encrypted operations, such as addition and comparison and allows access control.
- **`SepoliaZamaFHEVMConfig`**: A configuration contract that automatically sets up the required configurations for real-time encrypted operations on the Sepolia testnet.
- **`ConfidentialERC20Mintable.sol`** : The confidential smart contract that allows for full ERC20 compatibility with FHE encryption.

Step 2. Contract construction

```
contract MyConfidentialERC20 is SepoliaZamaFHEVMConfig, ConfidentialERC
  constructor(string memory name_, string memory symbol_) ConfidentialE
}
```

- This contract inherits `SepoliaZamaFHEVMConfig` and `ConfidentialERC20Mintable`.
- The constructor initializes the ERC20 token with a name and symbol, setting the deployer as the initial owner.

Going further

This is a simple basic contract that we will deploy and use in this tutorial. To write more complex confidential smart contracts or customize your own functions:

- Explore the full range of fhEVM capabilities in the **Smart Contract** section.
- Use the **`fhevm-contracts`** library and extend from the basic contract templates.

{% hint style="info" %} The **fhevm-contracts** is a Solidity library designed for developers to easily develop confidential smart contracts using fhEVM. It provides:

- **Ready-to-use confidential contracts**: Pre-built implementations of common token standards with FHE capabilities
- **Base contracts**: Foundational building blocks for creating custom confidential smart contracts
- **Extensions**: Additional features and utilities that can be added to base contracts
- **Testing utilities**: Tools to help test FHE-enabled smart contracts

See more details in the fhEVM-contracts documentation. {% endhint %}

Your contract is ready! Let's move on to testing and deployment.

File: ./modules/fhevm/docs/getting-started/overview-1/hardhat/4.-deploying-the-contract.md

4. Deploying the contract

This guide walks you through deploying your Confidential ERC20 smart contract on the Sepolia test network.

Prerequisites

Before proceeding, ensure you have:

- A configured Hardhat project using the fhEVM Hardhat Template (see previous sections).
- A crypto wallet installed (e.g., Metamask).
- Some Sepolia ETH available for testing. If you don’t have enough ETH, use a Sepolia faucet to request free SepoliaETH for testing:
  - Alchemy Faucet
  - QuickNode Faucet

Step 1. Preparing for deployment

1. Generate a mnemonic seed for accounts using `cast`:
   - (`cast` is a Foundry function. If you don't have Foundry installed, run `curl -L https://foundry.paradigm.xyz | bash` first):

```
cast wallet new-mnemonic
```

1. Obtain a Sepolia RPC URL:
   - Sign up for a node provider such as Alchemy - `https://www.alchemy.com/` or

    Infura - `https://infura.io/`. Copy your Sepolia RPC URL.

3. Open the `.env` file:
    - Past your mnemonic in the code:
        - MNEMONIC=`<Your mnemonic generated>`
    - Paste a Sepolia RPC URL:
        - SEPOLIA_RPC_URL=`<Your node provider URL>`
8. Verify generated accounts:

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

{% embed url="https://scribehow.com/embed/Step_3_Copy__EGYS53LkSP-Pr2WB7yQ1vw?removeLogo=true&skipIntro=true" %}

Step 2. Funding your wallet

1. Open your wallet (e.g, MetaMask)
2. Import the first 2 accounts (e.g., Alice and Bob) into your wallet with their private keys.
3. Fund these 2 accounts with some Sepolia ETH.

{% embed url="https://scribehow.com/embed/Step_3_2__Hbg4nSgdR3KMcCkB4aw8Jw?removeLogo=true&skipIntro=true" %}

Step 3. Deploying the contract

In the `deploy/` directory, there is a preconfigured `deploy.ts` file that handles the deployment process. You can customize it or add your own scripts.

To deploy the contracts to Sepolia, run:

```
# Using npm
npm run deploy-sepolia

# Using yarn
yarn deploy-sepolia

# Using pnpm
pnpm deploy-sepolia
```

Step 4. Verify the deployment

Once deployment is successful, you should see a console output that includes the contract address such as:

```
MyConfidentialERC20: 0x1234........ABCD
```

You can verify this contract on Sepolia Etherscan by searching for the deployed address.

{% embed url="https://scribehow.com/embed/Step_4__b3lGkybMS3ihZa8FklQo5A?removeLogo=true&skipIntro=true" %}

Congratulations! 🎉 You have deployed your first confidential ERC20 smart contract. Let's mint a few tokens and perform some encrypted transactions!

File: ./modules/fhevm/docs/getting-started/overview-1/remix/deploying_cerc20.md

3. Deploying ConfidentialERC20

In this tutorial, you'll learn how to deploy a confidential token contract using Zama's **fhEVM**. We'll create `MyConfidentialERC20.sol` to demonstrate the essential features.

Prerequisites

Ensure the following before deploying the smart contract:

- The **Zama Plugin** installed is installed in the Remix IDE(see Step 1).
- Your **wallet** is connected to the **Sepolia testnet**(see Step 2).

Step 1. Setting up the contract file

First, let's create a file for our confidential ERC20 contract:

1. Open the **File explorer** from the side menu.
2. Navigate to the **contracts** folder.
3. Click the **Create new file** icon.
4. Name the file `MyConfidentialERC20.sol` and press Enter.

{% embed url="https://scribehow.com/embed/31_Wg2FlRX2T-WPJ1qPnLuObA?removeLogo=true&skipIntro=true" %}

Step 2. Writing contract

Step 2.1 Basic contract structure

The foundational structure includes importing Zama's libraries and connecting to Sepolia's fhEVM configuration.

Copy the following code in the `MyConfidentialERC20.sol` that you just created:

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

pragma solidity ^0.8.24;
```

```
import "fhevm/lib/TFHE.sol";
import "fhevm/config/ZamaFHEVMConfig.sol";

contract MyConfidentialERC20 is SepoliaZamaFHEVMConfig {}
```

It should appear as follows:

[image]

Remix automatically saves any changes as you type. Upon saving, it imports the following libraries:

- **`TFHE.sol`**: The core Solidity library of Zama's fhEVM. It enables encrypted data type like `euint64`, secures encrypted operations, such as addition and comparison and allows access control.
- **`SepoliaZamaFHEVMConfig`**: A configuration contract that automatically sets up the required configurations for real-time encrypted operations on the Sepolia testnet.

Step 2.2 Enhancing the functionality

Next, we'll enhance our contract by importing the `fhevm-contracts` library.

{% hint style="info" %} The **fhevm-contracts** is a Solidity library designed for developers to easily develop confidential smart contracts using fhEVM. It provides:

- **Ready-to-use confidential contracts**: Pre-built implementations of common token standards with FHE capabilities
- **Base contracts**: Foundational building blocks for creating custom confidential smart contracts
- **Extensions**: Additional features and utilities that can be added to base contracts
- **Testing utilities**: Tools to help test FHE-enabled smart contracts

See more details in the fhEVM-contracts documentation. {% endhint %}

The `fhevm-contracts` library includes the `ConfidentialERC20Mintable` contract, which is an extention of `ConfidentialERC20` with minting capabilities, providing:

- Private token transfers and encrypted balances
- Minting functionality for authorized addresses
- Full ERC20 compatibility

It inherits all base `ConfidentialERC20` features while adding secure token creation and distribution capabilities.

To use `ConfidentialERC20Mintable` contract, simply update your `MyConfidentialERC20.sol` with the following code:

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

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

contract MyConfidentialERC20 is SepoliaZamaFHEVMConfig, ConfidentialERC
```

```
  constructor(string memory name_, string memory symbol_) ConfidentialE
}
```

It should appear as follows:

[image]

Step 3. Compiling the contract

Now the contract is ready, the next step is to compile it:

1. Select `MyConfidentialERC20.sol`.
2. Go to **Solidity compiler** in Remix.
3. Click **Compile**.

If successful, you will see a green checkmark on the Solidity Compiler, indicating "Compilation successful"

{% embed url="https://scribehow.com/embed/33_ScX1aJqhRMy3nnLhezsKDw?removeLogo=true&skipIntro=true" %}

Step 4. Deploying the contract

Now the contract is ready to be deployed:

1. Make sure that the envrionment is set up properly

    1. **Envrionment:** Injected Provider - Metamask
    2. **Account:** Your wallet address

2. Expand the **Deploy** section.
3. Fill the constructor parameters:

    - **Name**: Your token’s name (e.g., "My Private Token").
    - **Symbol**: Token symbol (e.g., "MPT").

6. Click **Transact** and confirm the transaction in MetaMask.

Once successfully deployed, your contract will appear under **Deployed Contracts**. You can also view your contract on Etherscan by clicking the contract address.

{% embed url="https://scribehow.com/embed/34_FKpOfgAWTzKX_e9VlakvHA?removeLogo=true&skipIntro=true" %}

By following these steps, you’ve successfully created and deployed an confidential ERC-20 token using Zama's fhEVM! 🎉 Let's see how the transaction works in the next chapter.

File: ./modules/fhevm/docs/getting-started/overview-1/remix/interact.md

4. Interacting with the contract

After deploying your first **fhEVM** contract using **Remix**, this guide shows you how to interact with it directly in Remix using the **Zama Plugin**.

Prerequisite

Before interacting with your deployed contract, ensure the following:

- **Deployment completed**: You have successfully deployed the `MyConfidentialERC20` contract (see Step 3).
- **MetaMask wallet**: Your MetaMask wallet is connected to the Sepolia testnet(see Step 2). You might want to prepare an additional wallet as the receiver to mock the transfer function.
- **Zama Plugin in Remix**: The Zama Plugin is installed and accessible in Remix (see Step 1).

Step 1. Connecting the deployed contract

To perform transactions directly in Remix, your contract needs to be connected to the **Zama Plugin**:

1. Open Deploy & run transaction from the side bar
2. In "**Deployed Contract**", copy the address of the `MYCONFIDENTIALERC20` contract that you just deployed.
3. Open the **Zama Plugin** from the side menu.
4. Paste the contract address into the "**At address**" field under the **Deploy** section.
5. Click **At address**.

If the address was entered correctly, the `MyConfidentialERC20.sol` contract will be displayed in the "**Deployed Contract**" inside the **Zama Plugin**.

{% embed url="https://scribehow.com/embed/Load_MyConfidentialERC20_Contract_in_Remix__tJ1PmbA4TuGQ2fj6kMdMtQ?removeLogo=true&skipIntro=true" %}

Click to expand the contract, you'll see the interface to interact with all the functions of your contract. `ConfidentialERC20Mintable` supports all standard ERC-20 functions, but adapted for encrypted values, including:

- `transfer`: Securely transfers encrypted tokens.
- `approve`: Approves encrypted amounts for spending.
- `transferFrom`: Transfers tokens on behalf of another address.
- `balanceOf`: Returns the encrypted balance of an account.
- `totalSupply`: Returns the total token supply.

Step 2. Mint tokens to your account

From here, you can mint confidential ERC20 token to your account:

1. Copy your wallet address from **MetaMask.**

2. Inside **Zama Plugin,** click to expand the **mint** function of your contract.
3. Enter your wallet address and the amount of tokens to mint (e.g., `1000`).
4. Click `Submit`.
5. Confirm the transaction in **MetaMask**.

Once sccussful, you should see the message in the terminal.

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow_I0epwZMASXuUfaqe3iU3jg?removeLogo=true&skipIntro=true" %}

Step 3. Verify total supply

After a successful mint transaction, click the **totalSupply** to reflect the minted tokens (e.g., `1000`).

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow_uV00AdSrSpO33EVGfI2LQg?removeLogo=true&skipIntro=true" %}

Step 4. Check your balance

To verify your account balance:

1. Click to expand the **balanceOf** function.
2. Enter your wallet address.
3. Click `Submit` to verify your account balance.

Your balance is stored as encrypted data using FHE. No one else can view if except you.

To view the balance in plaintext:

- Click the **re-encrypt** option
- Confirm the transaction in Metamask

You can see that the ciphertext is decrypted and your balance is the amount that you just minted.

{% embed url="https://scribehow.com/embed/44_Q0zdu_sETU2e9tVRkvfxkQ?removeLogo=true&skipIntro=true" %}

Step 5. Transfer tokens

To transfer confidential ERC20 tokens to another account:

1. Copy the address of the receiver's wallet.
2. Click **transfer** to expand the function, set the following parameters:

    - **To**: Enter the wallet address of **reveiver.**
    - **encryptedAmount**: Specify the amount that you want to transfer (e.g.`1000`). Choose `euint64`.
    - **inputProof**: Check the **input** box.

6. Click **Submit** to proceed.
7. Confirm the transaction in **MetaMask**.

If successful, you should see the message in the terminal.

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow__3owHVNoyTQmsgEOoLWcB5Q?removeLogo=true&skipIntro=true" %}

Step 6. Verify updated balance

After making a transfer, you can verify your updated account balance:

1. Use the **balanceOf** function again to check the updated balance of your **original account** (see the Step 5: Check your balance.)
2. Perform **re-encryption** to confirm the changes, you should see the remaining token in your account.(e.g., `900` tokens remaining).

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow__SB36tWugQLG2PdPwG-3iHw?removeLogo=true&skipIntro=true" %}

{% hint style="info" %} Always re-encrypt to validate ciphertext transformations and confirm operations. {% endhint %}

Next steps

🎉 **Congratulations on completing this tutorial!** You've taken the first step in building confidential smart contracts using **fhEVM**. It's time now to take the next step and build your own confidential dApps!

1. Resources

To continue your journey and deepen your knowledge, explore the resources below.

- **Read the Whitepaper**: Understand the core technology behind fhEVM, including its cryptographic foundations and use cases.
- **See more demos and tutorials**: Expand your skills with hands-on demos and tutorials crafted to guide you through various real-world scenarios.
- **Try out AI coding assistant**: If you have a chatGPT plus account, try out our custom ChatGPT model tailored for Solidity and fhEVM developers.

2. Tools

Use out-of-box templates and frameworks designed for developers to build confidential dapps easily.

Smart contract development

- **Hardhat Template**: A developer-friendly starting point for building and testing smart contracts on fhEVM.
- **fhEVM Contracts Library**: Access standardized contracts for encrypted operations.

Frontend development

- React.js Template: Quickly develop FHE-compatible dApps using a clean React.js setup.
- Next.js Template: Build scalable, server-rendered dApps with FHE integration.
- Vue.js Template: Develop responsive and modular dApps with FHE support in Vue.js.

3. Community

Join the community to shape the future of blockchain together with us.

- Discord: Join the community to get the latest update, have live discussion with fellow developers and Zama team.
- Community Forum: Get support on all technical questions related to fhEVM
- Zama Bounty Program: Participate to tackle challenges and earn rewards in cash.

File: ./modules/fhevm/docs/getting-started/overview-1/remix/README.md

Remix

This tutorial covers the essentials steps to quickly write and deploy confidential ERC20 smart contract using the Zama Plug in.

Remix is a powerful in-browser IDE for Ethereum smart contract development. It offers a fast, intuitive way to write and deploy confidential ERC20 contracts using Zama's fhEVM plugin—no local setup required.

In this tutorial, you will learn to:

1. Set up Remix for fhEVM development.
2. Connect your wallet (e.g., MetaMask) to Remix.
3. Deploy a **ConfidentialERC20** contract.
4. Interact with your deployed contract directly in the Remix interface.

By the end, you'll have a working confidential ERC20 token on Sepolia network and know how to perform encrypted transactions.

File: ./modules/fhevm/docs/getting-started/overview-1/remix/connect_wallet.md

2. Connect your wallet to Remix

In this guide, you'll learn how to connect your wallet and the **Zama Plugin** in Remix IDE to interact with fhEVM smart contracts.

Prerequisites

Before starting, ensure you have the following:

- **MetaMask** or another Ethereum-compatible wallet installed.
- **Zama Plugin** installed in Remix IDE (See Setting up Remix)

{% hint style="danger" %} Note thate when using Remix to connect a wallet, issues may arise if **multiple** wallet extensions are installed (e.g., MetaMask, Phantom). This is a known issue of Remix that can affect wallet connection functionality.

If you encounter errors, consider keeping **only MetaMask** as the active wallet extension, removing other wallet extensions, and refreshing Remix cookies and try to reconnect. {% endhint %}

Step 1. Setting up Sepolia testnet

If you're using Metamask, the Sepolia testnet should be pre-configured. Follow the steps to set it up:

1. Open **MetaMask**.
2. Click the **network dropdown** at the top left corner, and select **Sepolia Test Network**.
3. Ensure you have **Sepolia ETH** available. If you don't have enough ETH, use a **Sepolia faucet** to request free SepoliaETH for testing:
   - Alchemy Faucet
   - QuickNode Faucet

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow__hfRvx_T1To-YVlnj28WYng?removeLogo=true&skipIntro=true" %}

{% hint style="info" %} If Sepolia isn't visible:

1. Go to **Settings** > **Advanced**.
2. Toggle **Show test networks** to **ON**. {% endhint %}

{% hint style="info" %} If Sepolia isn't pre-configured in your wallet, add it manually:

1. Open your wallet's **network settings**.
2. Click **Add Network** or **Add Network Manually**.
3. Enter the following details:
   - **Network Name**: `Sepolia`
   - **RPC URL**: (provided by your node provider, e.g., Alchemy or Infura)
   - **Chain ID**: `11155111`
   - **Currency Symbol**: `SepoliaETH`
   - **Block Explorer URL**: `https://sepolia.etherscan.io` {% endhint %}

Step 2. Connecting to Zama Plugin

Zama Plugin provides the **Zama Coprocessor - Sepolia configuration** that ensures Remix and the wallet are properly set up to interact with fhEVM smart contracts.

To complete the configuration:

1. Open the **Zama Plugin** in Remix from the side pannel.
2. Click **Connect your wallet.**
3. **Confirm** the connection in **MetaMask.**
4. In the Zama Plugin, select **Zama Coprocessor - Sepolia**.
5. Click **Use this configuration** to finalize the setup.

Once successful, you should see the green text in the terminal indicating that the configuration is ready.

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow__pdS0_bEDRNGRMLcJ02A9Dw?removeLogo=true&skipIntro=true" %}

Step 3. Connecting wallet to Remix

Follow the steps to connect your wallet to Remix:

1. Open Remix and navigate to **Deploy & run transactions**.
2. Under **Environment**, select **Injected Provider - MetaMask**.
3. MetaMask will prompt a connection request. Click **Connect** to proceed.
4. Choose your wallet address in **Account.**

{% embed url="https://scribehow.com/embed/Google_Chrome_Workflow__b4xdbTivQrCjPelyPxI0AQ?removeLogo=true&skipIntro=true" %}

Now that your wallet is connected and your SepoliaETH balance is ready, you can proceed to deploy the `ConfidentialERC20Mintable` contract!

File: ./modules/fhevm/docs/getting-started/overview-1/remix/remix.md

1. Setting up Remix

This guide helps you set up the **Zama Plugin** in the official Remix IDE, enabling seamless development and management of smart contracts using the **fhEVM**.

Prerequisites

Before starting, make sure you have the following:

- A web **browser** (such as Chrome, Firefox).
- Basic familiarity with **Ethereum smart contracts**.
- A crypto **wallet** (we recommend using **MetaMask** for this tutorial).

Step 1. Connecting to the Zama Plugin in Remix

Zama Plugin allows you to interact with confidential contracts directly within Remix. To set it up:

1. Open the **Remix IDE** by navigating to https://remix.ethereum.org.
2. In the left sidebar, click on the **Plugin Manager**.
3. In the Plugin Manager, select **Connect to a Local Plugin**.

{% embed url="https://scribehow.com/embed/11_nqfYD8nFSVmwCZFm0hGggQ?removeLogo=true&skipIntro=true" %}

Step 2. Installing the Zama Plugin

Use the following configurations:

- **Plugin Name** (required) : Enter `Zama`.
- **URL**(required): Enter `https://remix.zama.ai/`.
- **Type of connection**(required): Select `Iframe`
- **Location in remix** (required):Select `Side Panel`
- Click `OK`.

{% embed url="https://scribehow.com/embed/How_to_Add_Zama_Plugin_in_Remix_Ethereum__yIcEiiToSpOmfRq0kwyiyA?removeLogo=true&skipIntro=true" %}

Now that you've configured the plugin, you are able to deploy and interact with **fhEVM** encrypted contracts directly directly via Remix interface. Next, let's dive into the contract deployment.

File: ./modules/fhevm/docs/tutorials/see-all-tutorials.md

See all tutorials

Solidity smart contracts templates - `fhevm-contracts`

The fhevm-contracts repository provides a comprehensive collection of secure, pre-tested Solidity templates optimized for fhEVM development. These templates leverage the TFHE library to enable encrypted computations while maintaining security and extensibility.

The library includes templates for common use cases like tokens and governance, allowing developers to quickly build confidential smart contracts with battle-tested components. For detailed implementation guidance and best practices, refer to the contracts standard library guide.

Token

- ConfidentialERC20: Standard ERC20 with encryption.
- ConfidentialERC20Mintable: ERC20 with minting capabilities.

- ConfidentialERC20WithErrors: ERC20 with integrated error handling.
- ConfidentialERC20WithErrorsMintable: ERC20 with both minting and error handling.

Governance

- ConfidentialERC20Votes: Confidential ERC20 governance token implementation. It is based on Comp.sol.
- ConfidentialGovernorAlpha: A governance contract for managing proposals and votes. It is based on GovernorAlpha.sol.

Utils

- EncryptedErrors: Provides error management utilities for encrypted contracts.

Code examples on GitHub

- Blind Auction: A smart contract for conducting blind auctions where bids are encrypted and the winning bid remains private.
- Decentralized ID: A blockchain-based identity management system using smart contracts to store and manage encrypted personal data.
- FheWordle: A privacy-preserving implementation of the popular word game Wordle where players guess a secret encrypted word through encrypted letter comparisons.
- Cipherbomb: A multiplayer game where players must defuse an encrypted bomb by guessing the correct sequence of numbers.
- Voting example: Suffragium is a secure, privacy-preserving voting system that combines zero-knowledge proofs (ZKP) and Fully Homomorphic Encryption (FHE) to create a trustless and tamper-resistant voting platform.

Frontend examples

- Cipherbomb UI: A multiplayer game where players must defuse an encrypted bomb by guessing the correct sequence of numbers.

Blog tutorials

- Suffragium: An Encrypted Onchain Voting System Leveraging ZK and FHE Using Zama's fhEVM - Nov 2024

Video tutorials

- How to do Confidential Transactions Directly on Ethereum? - Nov 2024
- Zama - FHE on Ethereum (Presentation at The Zama CoFHE Shop during EthCC 7) - Jul 2024

{% hint style="success" %} **Zama 5-Question Developer Survey**

We want to hear from you! Take 1 minute to share your thoughts and helping us enhance our documentation and libraries. 👉 **Click here** to participate. {% endhint %}

Legacy - Not compatible with latest fhEVM

- Build an Encrypted Wordle Game Onchain using FHE and Zama's fhEVM - February 2024
- Programmable Privacy and Onchain Compliance using Homomorphic Encryption - November 2023
- Confidential DAO Voting Using Homomorphic Encryption - October 2023
- On-chain Blind Auctions Using Homomorphic Encryption and the fhEVM - July 2023
- Confidential ERC-20 Tokens Using Homomorphic Encryption and the fhEVM - June 2023
- Using asynchronous decryption in Solidity contracts with fhEVM - April 2024
- Accelerate your code testing and get code coverage using fhEVM mocks - January 2024
- Use the CMUX operator on Zama’s fhEVM - October 2023
- [Video tutorial] How to Write Confidential Smart Contracts Using Zama's fhEVM - October 2023
- Workshop during ETHcc: Homomorphic Encryption in the EVM - July 2023