intro.md:

---
sidebarposition: 1
displayedsidebar: docsSidebar

Overview

Welcome to the Fhenix documents! These docs should have everything you need to get started and create smart contracts that use FHE with encrypted data!

:::tip[Tip]
For questions & support join our Discord!
:::

Here we'll explain everything about how to use Fhenix and how to use FHE to create privacy-preserving Web3 applications. We include an extension to the Ethereum Virtual Machine (EVM) that introduces operations on encrypted data using Fully Homomorphic Encryption (FHE). We've added special precompiles to the EVM that allow computations on encrypted data without the need for decryption.

The integration of the FHE with Solidity means you can continue to write your smart contracts with familiar syntax while leveraging the capabilities of FHE.

In this documentation, you'll find guidance on operating on encrypted data, understanding patterns in FHE-friendly code writing, and access control in FHE-based smart contracts. Let's get started.

Quick links

fhenix-and-t-fhe.md

connecting-to-the-testnet.md

Get Started

We've put together some helpful guides for you to get setup quickly and easily.

[//]: '{% content-ref url="developer-guides/getting-started.md" %}'
[//]: "getting-started.md"
[//]: "{% endcontent-ref %}"
[//]: #
[//]: '{% content-ref url="developer-guides/fhenix-by-example/" %}'
[//]: "fhenix-by-example"
[//]: "{% endcontent-ref %}"

Permits-Access-Control.md:

▫ Permits & Access Control

In a Fully Homomorphic Encryption (FHE) framework, all data stored in a contract's storage is encrypted. Access control involves granting selective access to data by authorized parties while restricting access to unauthorized users.

Solidity contracts generally expose their data using view functions. However, permissioned data is a challenge, since Solidity view functions do not come with any in-built mechanism to allow the contract to verify cryptographically that callers are who they say they are (for transactions, this is done by verifying the signature on the data).

Fhenix handles this issue by implementing a seal function, which seals the data in a manner that only the intended recipient can decrypt and view (Fhenix uses the decrypt function for less sensitive data). This approach ensures that encrypted data remains confidential and only accessible to authorized users.

Permits and Access Control

Fhenix Solidity libraries (specifically, fhenix.js) are equipped with an in-built access control scheme.
This access control scheme enables contracts to perform a basic check of account ownership by adding authentication and authorization features to specific view functions.
(An added benefit of the Fhenix Solidity libraries is that developers save coding effort each time a project has cryptographic access control requirements.)

What is a Permit?

A permit is a mechanism that allows the contract to verify cryptographically the identity of callers, ensuring that they are who they claim to be.

In Fhenix, a permit is a signed message that contains the caller's public key, which the contract can use to verify the caller. The permit is a signed JSON object that follows the EIP-712 standard.
The permit contains the necessary information, including a public key, which allows data re-sealing in a smart contract environment.
The inclusion of this public key into the permit enables a secure process of data re-sealing within a smart contract after the JSON object is signed by the user.

How to Generate a Permit

Permits are generated using the getPermit method in fhenix.js. This method requires the following parameters:

contractAddress (required, string): The address of the contract.
provider (required): An ethers (or compatible) object that can sign EIP-712 formatted data. (Note that if you want to unseal data using your wallet's encryption key you can't use "JsonRpcProvider")

```javascript
const permit = await getPermit(contractAddress);
```

What is a Permission?

In Fhenix, a permission is that part of a permit that supplies proof that callers are who they say they are.
A permission contains the signature and corresponding public key.
In order to see how to verify a permission in a Solidity contract, please refer to our Permissioned.

How to Generate a Permission

The following is the syntax for generating a permission:

```javascript
const permission = client.extractPermitPermissions(permit);
```

Using a Permission

Once generated, the permission can be used and sent to the contract. It can also be used to unseal the output of the sealoutput function, assuming it was sealed using that same permission.

The following code snippet shows how to implement the added cryptographic functionality of Fhenix (specifically, permits and permissions) on Ethereum using the Fhenix library.

javascript
```
import { BrowserProvider } from "ethers";
import { FhenixClient, getPermit } from "fhenixjs";

const provider = new BrowserProvider(window.ethereum);
const client = new FhenixClient({ provider });
const permit = await getPermit(contractAddress, provider);
const permission = client.extractPemitPermissions(permit);
client.storePermit(permit); // Stores a permit for a specific contract address.
const response = await contract.connect(owner).getValue(permission); // Calling
"getValue" which is a view function in "contract"
const plaintext = await client.unseal(contractAddress, response);
```

Privacy-Web3.md:

🤫 Development Tips – Ensuring Privacy

Fhenix provides a secure and decentralized way to execute smart contracts on encrypted data; transactions and computations are fully encrypted. As such, Fhenix offers superior on-chain privacy. However, developers still need to be vigilant, because all blockchain privacy platforms have their idiosyncrasies and potential privacy risks.

Implement Best Practices
Fhenix ensures end-to-end encryption, but developers should be careful not to become complacent on matters of privacy. Developers should always prioritize best practices to ensure privacy and confidentiality.

Analyze Your Privacy Model
We recommend that Fhenix developers carefully analyze their smart contract privacy model (this applies to any blockchain platform with privacy features). Distinguish between the type of information that, if “leaked,” can affect contract privacy on the one hand, and the type of information that, if compromised, will not affect contract operation and user privacy on the other. Special attention should be given to the type of information that must remain confidential.

As a result of this analysis and the insights gained, structure your smart contracts in a way that safeguards the aspects that affect privacy, while ensuring that the smart contract continues to operate efficiently.

A Simple Example
A simple example of metadata leakage is gas usage. Consider a smart contract coded in Solidity that contains a conditional statement. In this case, the path taken by the condition, though encrypted, may still reveal information. A typical scenario is a conditional branch based on the value of a private variable, where gas usage, events, or other metadata could reveal the branch taken.

Javascript
```
function performActionBasedOnBalance(uint256 amount) public {
    if (balance[msg.sender] >= amount) {
        // perform some operation
    } else {
        // perform another operation
    }
```

}

In the above Solidity example, someone observing the transaction could potentially infer the chosen branch based on gas usage, events or metadata, which would, in turn, indirectly reveal whether the sender's balance was greater than or equal to the specified amount.

This example might seem insignificant, but it is important to remember that transactions can often be cheaply simulated with different input parameters. In the above example, performing a logarithmic search would reveal the exact balance fairly quickly.

Add Access Controls
It is important to provide access controls to functions that handle sensitive data. For instance, a function revealing a user’s balance should only be accessible to that specific user. We discuss this issue further in the section on access control

In Conclusion
Despite the embedded encryption protection provided by FHE, it is essential to understand and address potential risk areas that can compromise privacy. We will be updating this section and our other documentation as our product matures, so be sure to check back from time to time.

Examples-fheDapps.md:

---
sidebarposition: 2
title: Examples & fheDapps

Here you can find a list of some cool apps that you can use as a reference

```
<table>
    <thead>
        <tr>
            <th>App</th>
            <th>Repo</th>
            <th>UI</th>
            <th>Notes</th>
        </tr>
    </thead>
<tbody>
<tr>
<td>FHERC-20</td>
<td><a href="https://github.com/FhenixProtocol/example-contracts/blob/master/
wrapping-ERC20/WrappingERC20.sol">View on Github</a><br /></td>
<td><a href="http://fhenix-demo.pages.dev/">FHERC-20 Demo</a></td>
<td></td>
</tr>
<tr>
<td>Blind Auction</td>
<td><a href="https://github.com/FhenixProtocol/blind-auction-example">View on
Github</a></td>
<td><a href="https://github.com/FhenixProtocol/blind-auction-example/tree/main/
frontend">View on Github</a></td>
<td></td>
</tr>
```

[//]: (<tr>)

[//]: (<td>NFT + 128 bit key</td>)

```
[//]: (<td><a
href="https://github.com/FhenixProtocol/devnet-contracts/blob/main/
ERC721WithKey.sol">https://github.com/FhenixProtocol/devnet-contracts/blob/
main/ERC721WithKey.sol</a></td>)

[//]: (<td></td>)

[//]: (<td>This examples will need to be updated when using Fhenix's
FHE.sol</td>)

[//]: (</tr>)
<tr>
<td>Confidential Voting</td>
<td><a href="https://github.com/FhenixProtocol/confidential-voting">View on
Github</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple Lottery</td>
<td><a href="https://github.com/FhenixProtocol/example-contracts/blob/master/
lottery/Lottery.sol">View on Github</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contract Playground</td>
<td><a href="https://github.com/FhenixProtocol/contracts-playground">View on
Github</a></td>
<td></td>
<td>A monorepo with multiple examples of contracts in the same place</td>
</tr>
<tr>
<td>Fhevm Examples</td>
<td><a href="https://github.com/zama-ai/fhevm-solidity/tree/main/examples">View
on Github</a></td>
<td><a href="https://dapps.zama.ai/">https://dapps.zama.ai/</a><br /></td>
<td>NOTE: These examples are not directly compatible with Fhenix and must be
adapted</td>
</tr>


[//]: (<tr>)

[//]: (<td>NFT Event Ticket</td>)

[//]: (<td><a href="https://github.com/FhenixProtocol/ticketing-
contracts">https://github.com/FhenixProtocol/ticketing-contracts</a>)

[//]: (<a
href="https://github.com/FhenixProtocol/ticket-verifier">https://github.com/
FhenixProtocol/ticket-verifier</a></td>)

[//]: (<td><a href="https://ticket-manager.pages.dev/">https://ticket-
manager.pages.dev/</a><a href="https://ticket-manager.pages.dev/?
verifier=1">https://ticket-manager.pages.dev/?verifier=1</a></td>)

[//]: (<td>This examples will need to be updated when using Fhenix's
FHE.sol</td>)
[//]: (<td>FHE.sol Operation Examples</td>)

[//]: (<td><a
href="https://github.com/FhenixProtocol/fheos/tree/master/solidity/tests/
contracts">https://github.com/FhenixProtocol/fheos/tree/master/solidity/tests/
```

contracts</td>)

[//]: (<td><a
href="https://github.com/FhenixProtocol/fheos/blob/master/solidity/tests/
precompiles.test.ts">https://github.com/FhenixProtocol/fheos/blob/master/
solidity/tests/precompiles.test.ts</td>)

[//]: (<td>The UI link is for a javascript interface that uses hardhat in order to interact with the contracts</td>)

[//]: (</tr>)
</tbody>
</table>

Templates.md:

---
sidebarposition: 1
title: Templates

We compiled a list of a few templates that you can use as a reference to build your own dApp.

Hardhat + React

https://github.com/FhenixProtocol/fhenix-hardhat-example

Has a basic contract, some tasks and a simple frontend (TODO: copy over from playground).

Nuxt 3 + Fhenixjs + Ethers.js + Bootstrap Starter

With this template you can easily start developing your Fhenix front-end app using Nuxt 3 (vue3).

https://github.com/FhenixProtocol/fhenix-nuxt3-template

Connecting-To.md:

---
sidebarposition: 3

🔗 Connecting to Fhenix Helium Testnet

Fhenix Helium is the first publicly available FHE-based blockchain, and it is now live! Follow the instructions to connect to Fhenix Helium Testnet.

Configuring MetaMask

1. Open MetaMask in your browser and click on the Ethereum network.
2. Click Add Network.
3. Click Add a network manually.

4. Fill out the network details form. To add a custom network, fill in the following fields:
   1. Network Name: Fhenix Helium
   2. New RPC URL: https://api.helium.fhenix.zone
   3. Chain ID: 8008135
   4. Currency Symbol: tFHE
   5. Block Explorer URL: https://explorer.helium.fhenix.zone
5. Once you fill out all the details, click Save.

6. Now you are ready to switch to Fhenix Helium Testnet. Tokens are available from the testnet faucet. Start building!

API endpoints

```
<table>
   <thead>
      <tr>
         <th width="222">Type</th>
         <th>API</th>
      </tr>
   </thead>
   <tbody>
      <tr>
         <td>JSON-RPC</td>
         <td><a
href="https://api.helium.fhenix.zone"><strong>https://api.helium.fhenix.zone</
strong></a></td>
      </tr>
      <tr>
         <td>Chain ID</td>
         <td>8008135</td>
      </tr>
      <tr>
         <td>Websocket</td>
         <td>wss://api.helium.fhenix.zone:8548</td>
      </tr>
   </tbody>
</table>
```

Explorer

https://explorer.helium.fhenix.zone

Faucet

To get some test tokens, use the faucet at https://get-helium.fhenix.zone/.
You may receive 0.1 tokens once every five minutes. If you need more tokens, please reach out to us on Discord, or bridge some Sepolia!

Bridge

The Helium testnet is connected to the Sepolia testnet. You can use the bridge to move tokens between the two networks.
If you require more tokens, you can use the bridge to move tokens from Sepolia to Helium.

https://bridge.helium.fhenix.zone/

FHE-Overview.md:

```
---
sidebarposition: 4
title: ▦ FHE Schemes Overview
---
```

Fully Homomorphic Encryption (FHE) schemes are divided into three generations, each designed for different types of applications.

Each of these generations relies on solving complex problems like Learning with Errors (LWE) and its generalization Ring LWE (RLWE) to ensure security.

We believe that understanding the advantages and disadvantages of each scheme will be important in being able to provide developers with the right tool for the application that they are trying to create.

First Generation - Integer Arithmetic

BGV Scheme

The BGV scheme was the first practical, leveled homomorphic encryption method. It introduced a technique called "packing," allowing multiple plaintexts to be encrypted into a single ciphertext, making it efficient in handling multiple data points simultaneously (like SIMD in processors). It avoided the need for bootstrapping, although it also included a bootstrapping option to upgrade to a fully homomorphic scheme.

Second Generation - Binary Operations

GSW Scheme

The GSW scheme introduces a unique approach for performing homomorphic operations called the "approximate eigenvector method." This method eliminates the need for "modulus switching" and "key switching." Instead, it uses multiplication via tensoring, which is later formalized as using a "gadget matrix." This approach significantly reduces error growth, but it does result in larger ciphertexts and higher computational costs. Due to these drawbacks, computations are limited to a binary message space. There is also an RLWE version of this scheme.

FHEW Scheme

The FHEW scheme is an optimized version of the GSW scheme, focusing on bootstrapping efficiency. It treats decryption as an arithmetic function rather than a boolean circuit. This RLWE variant incorporates several optimizations, making GSW-based bootstrapping faster than the BGV scheme. Key improvements include:

1. Restricting computations to a binary message space and using a NAND gate for homomorphic operations.
2. Enabling the evaluation of arbitrary functions via lookup tables during bootstrapping, known as “programmable bootstrapping.”
3. Utilizing efficient Fast Fourier Transform (FFT) methods for faster computations.

TFHE Scheme

This scheme uses "Blind Rotation" to enable fast bootstrapping, which is the process of refreshing a ciphertext to prevent error accumulation from making it unusable.
It involves two layers of encryption: a basic Learning with Errors (LWE) encryption and a special ring-based encryption for secure and efficient computation.
The TFHE scheme builds on FHEW techniques and employs methods like "modulus switching" and "key switching" for improved performance.

Third Generation - Approximate Number Arithmetic

CKKS Scheme

CKKS introduces an innovative way to map real (or complex) numbers for encryption.
It includes a "rescaling" technique to manage noise during homomorphic computations, reducing ciphertext size while preserving most of the precision.
Originally a leveled scheme, it later incorporated efficient bootstrapping to become fully homomorphic and added support for packed ciphertexts.

Fhenix-T-FHE.md:

---
sidebarposition: 1

✨ Fhenix & FHE

Fhenix is revolutionizing the blockchain space by utilizing Fully Homomorphic Encryption (FHE) for confidential smart contracts on public blockchains. An urgent blockchain challenge is ensuring privacy, and FHE is a promising solution. By leveraging FHE's ability to process encrypted data, privacy concerns are effectively addressed, thereby creating a safer environment for Web3 applications.

FHE - Fully Homomorphic Encryption

FHE is a technology that enables processing data without decrypting it. With data encrypted both in transit and during processing, everything that is done online can now be encrypted end-to-end, not just digital messaging!

This means that companies can offer services, including operating on customer data, while ensuring customer privacy (since user data remains encrypted). Users can be confident that their data is private, and they experience no difference in functionality.

FHE makes it possible to write private smart contracts that keep on-chain data encrypted. You can create decentralized, permissionless blockchains with all data on-chain and auditable, while not actually visible.

To read more about different FHE schemes, see our FHE Overview Section.

Fhenix Helium Testnet

The current Fhenix Helium Testnet is the first public iteration of the Fhenix protocol. It is still an early build, and it has bugs (unfortunately) and many features that are still under development.

There are many challenges ahead and many problems to solve. However, we are excited to be working on this project, because it is potentially an innovative and disruptive technology in the blockchain space.

What we write here is not set in stone. We are still considering the best way to move forward, and we are excited to have you here with us as we embark on this journey. Please let us know if you have any suggestions, ideas or comments. Feedback is always welcome. We are looking for ways to improve and for people to join us and contribute.

Integration.md:

---
sidebarposition: 3
title: 🤲 3rd party Integrations

Are you a developer looking to integrate Fhenix into your project, or support Fhenix with your app? This section is for you!

Things to Know

APIs, RPCs and general compatibility

Fhenix is based on Arbitrum, with the Helium Testnet based on Arbitrum Nitro version 2.3.4 (ArbOS 20). This means that everything that is natively supported by Arbitrum Nitro is also supported by Fhenix (rpc calls, ABI, etc).

Please refer to the Arbitrum documentation for more information and specifics.

EVM Compatibility

Fhenix is fully EVM compatible, up to and including the Cancun Upgrade.
This means that any contract that runs on Ethereum should run on Fhenix as well.
We support Solidity compiler 0.8.26.

Public Endpoints

We have public endpoints available for the Helium Testnet, which can be used:

<table>
<thead>
<tr>
<th width="222">Type</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON-RPC</td>
<td><a
href="https://api.helium.fhenix.zone"><strong>https://api.helium.fhenix.zone</
strong></td>
</tr>
<tr>
<td>Chain ID</td>
<td>8008135</td>
</tr>
<tr>
<td>Websocket</td>
<td>wss://api.helium.fhenix.zone:8548</td>
</tr>
</tbody>
</table>

If you require specialized endpoints, or higher rate limits than the default please reach out to us on Discord or email.

Cross Chain Messaging Contracts

The following contracts are deployed on Ethereum Sepolia and may be used by developers that wish to interact with Fhenix in a similar way to Arbitrum

<table>
<tr>
<td>
Delayed Inbox
</td>
<td>
0xf993E10C83Fe26DddFc6cb5E82444C44201e8a9C
</td>
</tr>
<tr>
<td>
Bridge
</td>
<td>

```
            0xBAE4d0f2b685452450bfC29a920A82e1DBdcFdD1
        </td>
    </tr>
    <tr>
        <td>
            Outbox
        </td>
        <td>
            0x2635a570f9ae308618D0A340DCd1118fBF73B2E8
        </td>
    </tr>
</table>
```

Architecture.md:

---
sidebarposition: 2
title: 📐 Fhenix Architecture

Our goal with Fhenix is not only to provide the first FHE-based L2 solution, but also to create a platform that is modular, flexible, and can easily be changed, extended or improved as we see
traffic, use-cases and requirements evolve.

The Fhenix Protocol is composed of several components that work together to provide a secure and private environment for smart contracts. The main components are:

Core Chain (based on Arbitrum Nitro)
FheOS
Warp-Drive

These components are layered together to provide a modular approach, that allows for a flexible architecture

`![](/img/fhenix-stack.webp)`

Core Chain

The Core Blockchain is the base layer of the Fhenix Protocol. It is based on Arbitrum Nitro, which is a Layer 2 scaling solution for Ethereum. Arbitrum Nitro is a rollup chain that uses a combination of fraud proofs and optimistic rollups to provide a scalable and secure environment for smart contracts.

The Core Blockchain is responsible for processing transactions, executing smart contracts, and maintaining the state of the blockchain.

FheOS

FheOS is the heart of the FHE operations. Its goal is to be a modular & extendable component that can plug into the underlying blockchain and provide FHE capabilities to smart contracts.

It includes the relevant FHE function calls (precompiles), as well as the Solidity functions & ciphertext management that is required to interact with the FHE layer.

Warp-Drive

Warp-Drive is responsible for managing the FHE keys and the FHE operations. It includes multiple components - key management, FHE operation interfaces, encryption/decryption functions, and more.

The integration of Warp Drive as a separate component creates a separation of responsibilities, where the chain itself does not need to be aware of the FHE operations, nor depend on specific functionality.
This allows us to support multiple variants of FHE schemes, which can be used by developers according to their specific needs.

Warp Drive includes multiple components, which work together using shared interfaces to be easy to use and extend.

`![](/img/warp-drive-schema.webp)`

Changelog.md:

---
sidebarposition: 3
title: 🆕 Changelog

Here you can find a list of changes between different versions of the Fhenix Testnet(s) as we evolve and grow.

Helium - Latest Version

Added eaddress as a native type that can be found in FHE.sol directly
Added support for large integer sizes: euint64, euint128, euint256. Not all operations are supported for each type
at this time. See Types and Operators for more information.
Added support for solidity compiler version 0.8.25 and up
Performance has been greatly increased for all data types
Performance has been greatly increased for select operations
All client-side libraries and toolkits have been upgraded and will require an update to version 0.2 to work with Helium - FhenixJS, Remix plugin & hardhat plugins.
Refactored gas prices for FHE operations to better reflect the operational cost of the operation. See Gas and Benchmarks for more information.
Blocks are now posted to the Sepolia Testnet with support for EIP-4844 blobs.
Refactored gas cost for transaction data to be more in line with Ethereum.
LocalFhenix - Added support for Console.log functionality to allow debug logs during contract execution.
Many bug fixes and other improvements.

Frontier

Initial limited release!

Fhenix-Differences.md:

---
sidebarposition: 5
title: 🔀 Fhenix Differences For Developers

You might be familiar with fhevm, which is a fork of the Ethereum Virtual Machine that supports homomorphic encryption by Zama.

While Fhenix uses a similar FHE cryptography, it does not use fhevm.
However, in order to make the FHE ecosystem as accessible as possible, we have

chosen to maintain compatibility in most interfaces,
except when we felt that the developer experience was significantly improved by making changes.

In this page, we try and document the differences that developers should be aware of.

Differences

fhenix.js is the recommended Javascript library for interacting with Fhenix smart contracts.
FHE library is available at the npm repository @fhenixprotocol/contracts.
cmux is named select.
reencrypt is named sealoutput or seal.
Operations can be called directly as properties of encrypted types (e.g. euint32.add(euint32) instead of FHE.add(euint32, euint32)).
Operations between encrypted types expect the types to match (e.g. euint32 + euint32 instead of euint32 + euint64)).
In Fhenix, we recommend using the inEuintXX input types instead of raw bytes when receiving encrypted data.
Conversion to other encrypted types can be done using the .toUxx functions. E.g. euint32 b = a.toU32();
Division by zero will return a MAXUINT value instead of throwing an error (e.g. euint8(1) / euint8(0) will return euint8(255) instead of throwing an error).
Permits and Permissioned contracts are the recommended way to handle access to sensitive data in Fhenix. To read more about permits and access control, see Access Control.
Sealing and Decryption can be accessed using .seal and .decrypt respectively.
Large bit sizes are supported (including eaddress), with a limited instruction set

Limitations.md:

---
sidebarposition: 4
title: 🛑 Limitations

Decryption Key

Decryption key is stored locally on each node - until the addition of a more complete solution which will be a part of
future versions, the decryption keys are stored by the node for ease of use. This means that (obviously), you shouldn't store any
real sensitive data or private keys on the testnet.

Security

The current iteration of the network does not include multiple components (such as input knowledge proofs, threshold decryption, execution proofs, etc.) that are critical for the security of data and network keys.
These features will be added iteratively as we move towards full release - this should be obvious, but please do not store any valuable information on the network as long as it is in the testnet phase.

Randomness

Randomness as a service is planned as a future addition. Until we can guarantee a secure source of randomness, we do not
want to make such a function available as a network service. For demos and development that require a source of randomness, we encourage
the use of external oracles, or usage of a mock random number generator.

Gas Costs

All gas costs are subject to change, and are being evaluated for optimization. The current gas costs are not final, and may change.

Stability

The network is still in a beta phase, and may be subject to instability. Please do not rely on the network to store your contracts or data forever, or for any period of time.
Expect that we might have to reboot the network and wipe everything on it at any time.

Integer Bit Sizes

At the moment all integer bit sizes are supported, as well as eaddress, a 160-bit size for addresses. However, the instruction set is limited to a subset of operations for performance reasons.
When we move to full public testnet and mainnet we expect to be able to support a wider range of operations. See Types and Operators for more information.

Catching Errors.md:

---
sidebarposition: 100
title: Catching Errors

Catching Errors in Hardhat

There are some scenarios where handling errors in hardhat is not as straightforward as it seems.

Generally this simple ethers client would suffice to catch errors inside a try block:

```
javascript
try {
    await contract.method(params);
} catch (error) {
    console.log(error!);
}
```

However, if a contract calls a fails only on the commit of a transaction and not in the preceding gas estimation, then this will not raise an error.
This is because the transaction will be successfully added on-chain, but the result will be a failure.

The reason this happens is that during gas estimation the FHE operations are not actually performed, but rather the gas is estimated based on the size of the encrypted data.

Instead, when calling contracts that perform FHE operations, we recommend checking for the status of the transaction:

```
javascript
try {
    let tx = await contract.method(params);
    let receipt = await tx.wait();
    if (receipt?.status === 0) {
        throw Error(Transaction failed!)
    }
```

```
} catch (error) {
    console.log(error!);
}
```

:::note
This type of behaviour might be client and framework specific, and might change in the future - we're putting it here for now
because we've seen this behaviour in hardhat. We'll update in the future if this is only hardhat specific, ethers specific, or if it's a general behaviour.
:::

Decryption.md:

---
sidebarposition: 3
title: (Un)Sealing

When an app wants to read some piece of encrypted data from a Fhenix smart contract, that data must be converted from its encrypted form on chain to an encryption that the app or user can read.

The process of taking an FHE-encrypted ciphertext and converting it to standard encryption is called sealing.

The data is returned to the user using sealed box encryption from NaCL. The gist of it is that the user provides a public key to the contract during a view function call, which the contract then uses to encrypt the data in such a way that only the owner of the private key associated with the provided public key can decrypt and read the data.

:::tip[Don't Want to Seal?]
Fhenix supports standard decryption as well. Mostly suited for public data, an unsealed plaintext value can be returned from a contract.
You can read more about how to do this here.
:::

Encrypted Values & Permits

When reading encrypted values we can do one of two things:
 Receiving it as bytes calldata: 0x04000....
 RECOMMENDED: Receiving it as inEuint: ["0x04000"]

The main difference with inEuint is that you can be explicit with what is the exact parameter that you are looking for.

A Permit is a data structure that helps contracts know who is trying to call a specific function.

The fhenix.js Javascript library includes methods to support creating parameters for values that require Permits & Access Control. These methods can help creating ephemeral transaction keys, which are used by the smart contract to create a secure encryption channel to the caller.
Similarly to decryption, this usage can be implemented by any compliant library, but we include direct support in fhenix.js.

This is done in 3 steps: generating a permit, querying the contract and unsealing the data.

1. Creating a Permit

javascript

```
import { FhenixClient, getPermit } from 'fhenixjs';

const provider = new ethers.JsonRpcProvider('https://api.helium.fhenix.zone/');
const client = new FhenixClient({ provider });


const permit = await getPermit(contractAddress, provider);
client.storePermit(permit);
```

:::tip[Did you know?]
When you create a permit it gets stored in localstorage. This makes permits easily reusable and transferable
:::

2. Querying the Contract

We recommend that contracts implement the Permit/Permission interfaces (though this is not strictly required!).
In this case, we can easily inject our permit into the function call.

```javascript
const permission = client.extractPermitPermission(permit);
const response = await contract.balanceOf(permission);
```

3. Unsealing the Data

Now that we have the response data, we can use the unseal function to decipher the data

```javascript
client.unseal(contractAddress, response)
```

We have to provide the contract address so the fhenix client knows which permit to use for the unsealing function.

:::note
Permits are currently limited to support a single contract
:::

Putting it all Together

```typescript
import { FhenixClient, getPermit } from 'fhenixjs';
import { JsonRpcProvider } from 'ethers';

const provider = new ethers.JsonRpcProvider('https://api.helium.fhenix.zone/');
const client = new FhenixClient({provider});

const permit = await getPermit(contractAddress, provider);
client.storePermit(permit);

const permission = client.extractPermitPermission(permit);
const response = await contract.balanceOf(permission);

const plaintext = client.unseal(contractAddress, response);

console.log(My Balance: ${plaintext})
```

:::tip[Did you know?]
You have tools that can ease the process of interacting with the contract and decrypting values. If you want to use them please refer to

Tools and Utilities
:::

Encryption.md:

---
sidebarposition: 2

Encryption

fhenix.js provides an easy-to-use function to encrypt your inputs before sending them to the Fhenix blockchain.

:::tip
Encryption in Fhenix is done using the global chain key. This key is loaded when you create a fhenix.js client automatically
:::

When we perform encryption, we specify the type of euint (Encrypted Integer) we want to create. This should match the expected type in the Solidity contract we are working with.

First, initialize the library -

Typescript
```
import { FhenixClient } from 'fhenixjs';
import { BrowserProvider } from "ethers";

const provider = new BrowserProvider(window.ethereum);

const client = new FhenixClient({provider});
```

Then, you can use the created client to encrypt

Typescript

```
import { FhenixClient, EncryptedType, EncryptedUint8 } from 'fhenixjs';

let result: EncryptedUint8 = await client.encrypt(number,
EncryptionTypes.uint8);
let result: EncryptedUint16 = await client.encrypt(number,
EncryptionTypes.uint16);
let result: EncryptedUint32 = await client.encrypt(number,
EncryptionTypes.uint32);
let result: EncryptedUint64 = await client.encrypt(number,
EncryptionTypes.uint64);
let result: EncryptedUint128 = await client.encrypt(number,
EncryptionTypes.uint128);
let result: EncryptedUint256 = await client.encrypt(number,
EncryptionTypes.uint256);
let result: EncryptedAddress = await client.encrypt(address,
EncryptionTypes.address);
```

Or, we can use the lower-level type specific functions

javascript
```
const resultUint8 = await client.encryptuint8(number);
const resultUint16 = await client.encryptuint16(number);
const resultUint32 = await client.encryptuint32(number);
```

```
const resultUint64 = await client.encryptuint64(number);
const resultUint128 = await client.encryptuint128(number);
const resultUint256 = await client.encryptuint256(number);
const resultAddress = await client.encryptaddress(address);
```

The returned types from the encrypt function will be of the type EncryptedUint8, EncryptedUint16 or EncryptedUint32 (or 64/128/256 etc.) depending on the type you specified.

The EncryptedUint types sound scary, but are actually pretty simple. It's just a

```typescript
export interface EncryptedNumber {
  data: Uint8Array;
}

export interface EncryptedUint8 extends EncryptedNumber {}
```

These types exist in order to enable type checking when interacting with Solidity contracts, and to make it easier to work with encrypted data. However, feel free to use the data field directly if you prefer.

Permits.md:

---
sidebarposition: 4
title: Permits

Permits & Permissions

Overview

Permits are a mechanism that allows the contract to cryptographically verify that the caller is who he says he is.

Simply, they are a signed message that contains the caller's public key, which the contract can then use to verify that the caller is who he says he is.

Usage

Permits are meant to be used together with the interfaces exposed by Permissioned.Sol. If a contract expects a Signature
parameter, that's a good sign that we should use a permit to manage and create user permissions.

Out-of-the-box, Fhenix Solidity libraries come with a basic access control scheme. This helps contracts perform a basic check for ownership of an account.

To confirm whether the recipient is authorized, EIP712 signatures are employed. EIP712 is a standard for Ethereum signed messages that makes it easier to understand the information being signed. This allows us to verify that the signer of a given piece of data is the owner of the account they claim to be.

:::tip[Did You Know?]
When signing EIP712 typed data, wallets such as MetaMask provide a more transparent, safe interface for users to understand what they are signing
:::

Let's see this concept in action using an example. In an encrypted ERC20 token contract, a user would want to query their token balance. Since the balance is stored as encrypted data, the contract must first verify that the query is indeed from the token owner before revealing the information. This is where the EIP712 signatures step in.

Below is a function from an EncryptedERC20 contract:

```javascript
function balanceOf(
    Permission calldata perm
)
    public
    view
    onlySender(perm)
    returns (bytes memory)
{
    return FHE.sealoutput(balances[msg.sender], perm.publicKey);
}
```

In this function, onlySender is a modifier that verifies if the EIP712 signature is valid. If the signature corresponds to the account that is making the call (msg.sender), then the function will execute. If not, it will revert.

Here's what the onlySender modifier looks like:

```javascript
struct Permission {
    bytes32 publicKey;
    bytes signature;
}

modifier onlySender(Permission memory permission) {
    bytes32 digest = hashTypedDataV4(keccak256(abi.encode(
        keccak256("Permissioned(bytes32 publicKey)"),
        permission.publicKey
    )));
    address signer = ECDSA.recover(digest, permission.signature);
    if (signer != msg.sender)
        revert SignerNotMessageSender();
    ;
    }
```

The onlySender modifier takes a Permission. It then calculates the digest from the publicKey. The signer's address is recovered from the digest using the ECDSA.recover function. If the recovered address matches msg.sender, it means that the caller is indeed the owner of the account and is allowed to access the data.

You can use this helpful contract out-of-the-box by importing it from @fhenixprotocol/contracts/access and can be easily imported to integrate into your contracts.

```javascript
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@fhenixprotocol/contracts/Fhe.sol";
import { Permissioned } from
"@fhenixprotocol/contracts/access/Permissioned.sol";
```

```
contract WrappingERC20 is Permissioned, ERC20 {

    function balanceOfEncrypted(Permission memory perm)
    public
    view
    onlySender(perm)
    returns (bytes memory) {
        return FHE.sealoutput(encBalances[msg.sender], perm.publicKey);
    }
}
```

For a full example what this looks like - see EncryptedERC20.sol or our getting started tutorial for a full example, including client-side integration.

Advanced Access Control

While the above-mentioned access control scheme leveraging EIP712 signatures provides a robust mechanism for verifying the identity of users querying encrypted data, it does have some limitations. One of the primary missing pieces is the absence of roles and permissions associated with those roles. The scheme as described validates that a user querying a balance, for example, is indeed the owner of that account, but it doesn't provide a mechanism for defining different levels of access or permissions.

For instance, in more complex scenarios, you might want to allow certain users to only view specific pieces of data, or perhaps perform certain actions based on their role (admin, user, auditor, etc.). Moreover, there's no provision for dynamic access control in which permissions could be granted or revoked at runtime.

Additionally, this scheme doesn't cover collective authority, where, for example, an action might require the approval of multiple participants to be executed. Such advanced access control mechanisms are not built into this scheme and would need to be implemented separately based on specific application needs.

Lastly, the EIP-712 standard mostly considers messages targeted at a single smart contract. Some use-cases, however, benefit from allowing the user to provide access to multiple contracts concurrently. For example, consider a DeX (decentralized exchange). Allowing such an app to be able to display the balances of all the user's different tokens would be a UX challenge if the user had to approve each one individually.

Standardization

While we recommend and provide Permits as a basic access control mechanism, we do not enforce any particular standard for them.
We feel that as the ecosystem evolves, different standards will emerge and we do not want to limit the ecosystem by enforcing a particular standard at this stage.

In other words, if you think there is a better way to do it, feel free to do so!

Sending-a-Transaction.md:

---
sidebarposition: 4

End-to-End Example

In this section, we'll explore how to use fhenix.js to send transactions on the Fhenix blockchain.

To send transactions with fhenix.js, we'll first establish a connection to the blockchain, then interact with it using a contract method. For this process, we'll also need to encrypt the transaction data.

Here's a step-by-step explanation, using ethers, though other libraries like web3can also be used in a similar way.

Let's assume we have a deployed ERC20 contract, only this one uses encrypted inputs and outputs (you can find the solidity code here. Let's see how we can transfer some of our tokens to another address, while keeping the amount hidden.

1. Import fhenixjs and ethers

:::danger
OUTDATED
:::

```javascript
import { FhenixClient } from "fhenixjs";
import { BrowserProvider } from "ethers";
```

2. Define the Smart Contract Address and Provider: The smart contract address is the Ethereum address of the deployed contract. provider allows you to interact with the Ethereum blockchain.

```javascript
const CONTRACTADDRESS = "0x1c786b8ca49D932AFaDCEc00827352B503edf16c";
const provider = new BrowserProvider(window.ethereum);
```

3. Create a Client to Interact With Fhenix: The constructor of FhenixClient is used to create an instance of the client with the given provider.

```javascript
const client = new FhenixClient({ provider });
```

4. Create the Transfer Function: The transfer function is used to send a transaction on the blockchain. It requires the recipient address and the amount to be sent as parameters.

```javascript
const transfer = async (to, amount) => {
  // Create client
  const client = new FhenixClient({ provider });

  // get contract
  const contract = await ethers.getContractAt(CONTRACTNAME, CONTRACTADDRESS);
  const encryptedAmount = await client.encrypt(number, EncryptionTypes.uint32);

  const response = await contract
    .connect(SENDERACCOUNT)
    .transfer(address, encryptedAmount);
  return response;
};
```

Hardhat.md:

👷 Hardhat

Prerequisites

- Docker
- pnpm

Clone Hardhat Template

We provide a hardhat template available that comes "batteries included", with everything you need to hit the ground running. The template is available here. You can create a new repository, or clone it locally:

```
git clone https://github.com/fhenixprotocol/fhenix-hardhat-example
```

You'll also probably want to set an .env file with your mnemonics:

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

Install Dependencies

Once you've cloned the repository, you can install the dependencies with pnpm:

```sh
pnpm install
```

Start LocalFhenix

LocalFhenix is a complete Fhenix local testnet and ecosystem containerized with Docker. It simplifies the way contract developers test their contracts in a sandbox before they deploy them on a testnet or mainnet - similar to Ganache, or other local network environments.

To start a LocalFhenix instance, run the following command:

```sh
pnpm localfhenix:start
```

This will start a LocalFhenix instance in a docker container, managed by the fhenix-hardhat-docker plugin for Hardhat.
If this worked you should see a LocalFhenix started message in your console.

You've now officially created a LocalFhenix testnet. 🎉

After you're done, you can stop the LocalFhenix instance with:

```sh
pnpm localfhenix:stop
```

Deploy the contracts

To deploy the contracts to LocalFhenix, run the following command:

```sh
pnpm hardhat deploy
```

This will compile the contracts in the contracts directory and deploy them to the LocalFhenix network.

(note: if you want to deploy to a different network, you can specify the network with the --network flag)

Tasks

We've included a few tasks in the tasks directory to help you get started. You can run them with the pnpm task command.

```
sh
pnpm task:getCount => 0
pnpm task:addCount
pnpm task:getCount => 1
pnpm task:addCount --amount 5
pnpm task:getCount => 6
```

intro.md:

---
sidebarposition: 1
title: Overview
description: Different ways to set up your development environment for Fhenix
---

Overview

There are a few different ways to set up an environment for development on Fhenix. All the tools you know from Solidity are mostly supported, though the addition of FHE means that a few custom tools are helpful. Here we'll describe the different ways you can set up your development environment.

The following environments are recommended for development on Fhenix:

- Fhenix Hardhat Example
- Fhenix with Remix
- Gitpod Environment

:::note[Note]
The main developer tools are all based on Javascript & Solidity, but we have open bounties to add support for Python & Vyper!
:::

If you just want to utilize one of the tools in your own environment, take a look at:

- Fhenix-Remix-Plugin
- Fhenix-Encryption-UI
- Hardhat-Plugin

Remix.md:

🎧 Remix

To get up and running with Remix, you should follow a few easy steps:

1. Add Fhenix to Metamask
2. Import FHE.sol from your contract
3. Optionally, use the Fhenix Remix Plugin to make your life a bit easier

1. Add Fhenix to Metamask

Follow the instructions in the Fhenix Helium Testnet to add Fhenix to Metamask.

2. Import FHE.sol

All you need is to include is the FHE.sol solidity library to your project so the compiler will know what to do.

```solidity
import "@fhenixprotocol/contracts/FHE.sol";
```

3. Fhenix Remix Plugin

The Fhenix Remix Plugin is a browser extension that adds Fhenix support to Remix. It allows you to encrypt and decrypt data in your contracts, and to interact with the Fhenix testnet.

See the Fhenix Remix Plugin for more information on how to install and use the plugin!

FHE.md:

FHE.sol

isInitialized

```solidity
function isInitialized(ebool v) internal pure returns (bool)
```

isInitialized

```solidity
function isInitialized(euint8 v) internal pure returns (bool)
```

isInitialized

```solidity
function isInitialized(euint16 v) internal pure returns (bool)
```

isInitialized

```solidity
function isInitialized(euint32 v) internal pure returns (bool)
```

isInitialized

```solidity
function isInitialized(euint64 v) internal pure returns (bool)
```

isInitialized

```solidity
function isInitialized(euint128 v) internal pure returns (bool)
```

isInitialized

solidity
```
function isInitialized(euint256 v) internal pure returns (bool)
```

isInitialized

solidity
```
function isInitialized(eaddress v) internal pure returns (bool)
```

mathHelper

solidity
```
function mathHelper(uint8 utype, uint256 lhs, uint256 rhs, function
(uint8,bytes,bytes) pure external returns (bytes) impl) internal pure returns
(uint256 result)
```

add

solidity
```
function add(euint8 lhs, euint8 rhs) internal pure returns (euint8)
```

This function performs the add operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

add

solidity
```
function add(euint16 lhs, euint16 rhs) internal pure returns (euint16)
```

This function performs the add operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

add

solidity
```
function add(euint32 lhs, euint32 rhs) internal pure returns (euint32)
```

This function performs the add operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

add

solidity
```
function add(euint64 lhs, euint64 rhs) internal pure returns (euint64)
```

This functions performs the add operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

add

solidity
```
function add(euint128 lhs, euint128 rhs) internal pure returns (euint128)
```

This functions performs the add operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

sealoutput

solidity
```
function sealoutput(ebool value, bytes32 publicKey) internal pure returns (string)
```

performs the sealoutput function on a ebool ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	ebool	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
```
function sealoutput(euint8 value, bytes32 publicKey) internal pure returns (string)
```

performs the sealoutput function on a euint8 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	euint8	Ciphertext to decrypt and seal

| publicKey | bytes32 | Public Key that will receive the sealed plaintext |

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(euint16 value, bytes32 publicKey) internal pure returns (string)

performs the sealoutput function on a euint16 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	euint16	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(euint32 value, bytes32 publicKey) internal pure returns (string)

performs the sealoutput function on a euint32 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	euint32	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(euint64 value, bytes32 publicKey) internal pure returns

(string)

performs the sealoutput function on a euint64 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	euint64	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(euint128 value, bytes32 publicKey) internal pure returns (string)

performs the sealoutput function on a euint128 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	euint128	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(euint256 value, bytes32 publicKey) internal pure returns (string)

performs the sealoutput function on a euint256 ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

| Name | Type | Description |

| ---- | ---- | ----------- |
| value | euint256 | Ciphertext to decrypt and seal |
| publicKey | bytes32 | Public Key that will receive the sealed plaintext |

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

sealoutput

solidity
function sealoutput(eaddress value, bytes32 publicKey) internal pure returns (string)

performs the sealoutput function on a eaddress ciphertext. This operation returns the plaintext value, sealed for the public key provided

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
value	eaddress	Ciphertext to decrypt and seal
publicKey	bytes32	Public Key that will receive the sealed plaintext

Return Values

Name	Type	Description
[0]	string	Plaintext input, sealed for the owner of publicKey

decrypt

solidity
function decrypt(ebool input1) internal pure returns (bool)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	ebool	the input ciphertext

decrypt

solidity
function decrypt(euint8 input1) internal pure returns (uint8)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint8	the input ciphertext

decrypt

solidity
function decrypt(euint16 input1) internal pure returns (uint16)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint16	the input ciphertext

decrypt

solidity
function decrypt(euint32 input1) internal pure returns (uint32)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint32	the input ciphertext

decrypt

solidity
function decrypt(euint64 input1) internal pure returns (uint64)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint64	the input ciphertext

decrypt

solidity
function decrypt(euint128 input1) internal pure returns (uint128)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint128	the input ciphertext

decrypt

solidity
function decrypt(euint256 input1) internal pure returns (uint256)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint256	the input ciphertext

decrypt

solidity
function decrypt(eaddress input1) internal pure returns (address)

Performs the decrypt operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	eaddress	the input ciphertext

lte

solidity
function lte(euint8 lhs, euint8 rhs) internal pure returns (ebool)

This function performs the lte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lte

solidity
```
function lte(euint16 lhs, euint16 rhs) internal pure returns (ebool)
```

This function performs the lte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lte

solidity
```
function lte(euint32 lhs, euint32 rhs) internal pure returns (ebool)
```

This function performs the lte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description

| [0] | ebool | The result of the operation |

lte

solidity
```
function lte(euint64 lhs, euint64 rhs) internal pure returns (ebool)
```

This functions performs the lte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lte

solidity
```
function lte(euint128 lhs, euint128 rhs) internal pure returns (ebool)
```

This functions performs the lte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

sub

solidity
```
function sub(euint8 lhs, euint8 rhs) internal pure returns (euint8)
```

This function performs the sub operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

sub

solidity
function sub(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the sub operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

sub

solidity
function sub(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the sub operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

sub

solidity
function sub(euint64 lhs, euint64 rhs) internal pure returns (euint64)

This functions performs the sub operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

sub

solidity
function sub(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the sub operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

mul

solidity
function mul(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the mul operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

mul

solidity
function mul(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the mul operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

mul

solidity
function mul(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the mul operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

mul

solidity
```
function mul(euint64 lhs, euint64 rhs) internal pure returns (euint64)
```

This functions performs the mul operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

lt

solidity
```
function lt(euint8 lhs, euint8 rhs) internal pure returns (ebool)
```

This function performs the lt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lt

solidity
```
function lt(euint16 lhs, euint16 rhs) internal pure returns (ebool)
```

This function performs the lt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lt

solidity
function lt(euint32 lhs, euint32 rhs) internal pure returns (ebool)

This function performs the lt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lt

solidity
function lt(euint64 lhs, euint64 rhs) internal pure returns (ebool)

This functions performs the lt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input

| rhs | euint64 | The second input |

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

lt

solidity
```
function lt(euint128 lhs, euint128 rhs) internal pure returns (ebool)
```

This functions performs the lt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

select

solidity
```
function select(ebool input1, ebool input2, ebool input3) internal pure returns (ebool)
```

select

solidity
```
function select(ebool input1, euint8 input2, euint8 input3) internal pure returns (euint8)
```

select

solidity
```
function select(ebool input1, euint16 input2, euint16 input3) internal pure returns (euint16)
```

select

solidity
```
function select(ebool input1, euint32 input2, euint32 input3) internal pure returns (euint32)
```

select

solidity
function select(ebool input1, euint64 input2, euint64 input3) internal pure returns (euint64)

select

solidity
function select(ebool input1, euint128 input2, euint128 input3) internal pure returns (euint128)

select

solidity
function select(ebool input1, euint256 input2, euint256 input3) internal pure returns (euint256)

select

solidity
function select(ebool input1, eaddress input2, eaddress input3) internal pure returns (eaddress)

req

solidity
function req(ebool input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	ebool	the input ciphertext

req

solidity
function req(euint8 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint8	the input ciphertext

req

solidity
function req(euint16 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint16	the input ciphertext

req

solidity
function req(euint32 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint32	the input ciphertext

req

solidity
function req(euint64 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint64	the input ciphertext

req

solidity
function req(euint128 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint128	the input ciphertext

req

solidity
function req(euint256 input1) internal pure

Performs the req operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint256	the input ciphertext

div

solidity
function div(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the div operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

div

solidity
function div(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the div operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

div

solidity
```
function div(euint32 lhs, euint32 rhs) internal pure returns (euint32)
```

This function performs the div operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

gt

solidity
```
function gt(euint8 lhs, euint8 rhs) internal pure returns (ebool)
```

This function performs the gt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gt

solidity
function gt(euint16 lhs, euint16 rhs) internal pure returns (ebool)

This function performs the gt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gt

solidity
function gt(euint32 lhs, euint32 rhs) internal pure returns (ebool)

This function performs the gt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gt

solidity
function gt(euint64 lhs, euint64 rhs) internal pure returns (ebool)

This functions performs the gt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gt

solidity
function gt(euint128 lhs, euint128 rhs) internal pure returns (ebool)

This functions performs the gt operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gte

solidity
function gte(euint8 lhs, euint8 rhs) internal pure returns (ebool)

This function performs the gte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description

[0]	ebool	The result of the operation

gte

solidity
```
function gte(euint16 lhs, euint16 rhs) internal pure returns (ebool)
```

This function performs the gte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gte

solidity
```
function gte(euint32 lhs, euint32 rhs) internal pure returns (ebool)
```

This function performs the gte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gte

solidity
```
function gte(euint64 lhs, euint64 rhs) internal pure returns (ebool)
```

This functions performs the gte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

gte

solidity
```
function gte(euint128 lhs, euint128 rhs) internal pure returns (ebool)
```

This functions performs the gte operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

rem

solidity
```
function rem(euint8 lhs, euint8 rhs) internal pure returns (euint8)
```

This function performs the rem operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

rem

solidity
```
function rem(euint16 lhs, euint16 rhs) internal pure returns (euint16)
```

This function performs the rem operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

rem

solidity
```
function rem(euint32 lhs, euint32 rhs) internal pure returns (euint32)
```

This function performs the rem operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

and

solidity
```
function and(ebool lhs, ebool rhs) internal pure returns (ebool)
```

This function performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	ebool	The first input
rhs	ebool	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

and

solidity
function and(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

and

solidity
function and(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

and

solidity
```
function and(euint32 lhs, euint32 rhs) internal pure returns (euint32)
```

This function performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

and

solidity
```
function and(euint64 lhs, euint64 rhs) internal pure returns (euint64)
```

This functions performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

and

solidity
```
function and(euint128 lhs, euint128 rhs) internal pure returns (euint128)
```

This functions performs the and operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

or

solidity
function or(ebool lhs, ebool rhs) internal pure returns (ebool)

This function performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	ebool	The first input
rhs	ebool	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

or

solidity
function or(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input

| rhs | euint8 | The second input |

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

or

solidity
```
function or(euint16 lhs, euint16 rhs) internal pure returns (euint16)
```

This function performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

or

solidity
```
function or(euint32 lhs, euint32 rhs) internal pure returns (euint32)
```

This function performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

or

solidity
```
function or(euint64 lhs, euint64 rhs) internal pure returns (euint64)
```

This functions performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

or

solidity
function or(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the or operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

xor

solidity
function xor(ebool lhs, ebool rhs) internal pure returns (ebool)

This function performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description

| ---- | ---- | ----------- |
| lhs | ebool | The first input |
| rhs | ebool | The second input |

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

xor

solidity
```
function xor(euint8 lhs, euint8 rhs) internal pure returns (euint8)
```

This function performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

xor

solidity
```
function xor(euint16 lhs, euint16 rhs) internal pure returns (euint16)
```

This function performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

xor

solidity
function xor(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

xor

solidity
function xor(euint64 lhs, euint64 rhs) internal pure returns (euint64)

This functions performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

xor

solidity
function xor(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the xor operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

eq

solidity
```
function eq(ebool lhs, ebool rhs) internal pure returns (ebool)
```

This function performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	ebool	The first input
rhs	ebool	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
```
function eq(euint8 lhs, euint8 rhs) internal pure returns (ebool)
```

This function performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
function eq(euint16 lhs, euint16 rhs) internal pure returns (ebool)

This function performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
function eq(euint32 lhs, euint32 rhs) internal pure returns (ebool)

This function performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
function eq(euint64 lhs, euint64 rhs) internal pure returns (ebool)

This functions performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
```
function eq(euint128 lhs, euint128 rhs) internal pure returns (ebool)
```

This functions performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

eq

solidity
```
function eq(euint256 lhs, euint256 rhs) internal pure returns (ebool)
```

This functions performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint256	The first input
rhs	euint256	The second input

Return Values

Name	Type	Description

| [0] | ebool | The result of the operation |

eq

solidity
function eq(eaddress lhs, eaddress rhs) internal pure returns (ebool)

This functions performs the eq operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	eaddress	The first input
rhs	eaddress	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(ebool lhs, ebool rhs) internal pure returns (ebool)

This function performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	ebool	The first input
rhs	ebool	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint8 lhs, euint8 rhs) internal pure returns (ebool)

This function performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext

Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint16 lhs, euint16 rhs) internal pure returns (ebool)

This function performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint32 lhs, euint32 rhs) internal pure returns (ebool)

This function performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint64 lhs, euint64 rhs) internal pure returns (ebool)

This functions performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint128 lhs, euint128 rhs) internal pure returns (ebool)

This functions performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(euint256 lhs, euint256 rhs) internal pure returns (ebool)

This functions performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint256	The first input
rhs	euint256	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

ne

solidity
function ne(eaddress lhs, eaddress rhs) internal pure returns (ebool)

This functions performs the ne operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	eaddress	The first input
rhs	eaddress	The second input

Return Values

Name	Type	Description
[0]	ebool	The result of the operation

min

solidity
function min(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the min operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

min

solidity
function min(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the min operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

min

solidity
function min(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the min operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

min

solidity
function min(euint64 lhs, euint64 rhs) internal pure returns (euint64)

This functions performs the min operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

min

solidity
function min(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the min operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

max

solidity
function max(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the max operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input

| rhs | euint8 | The second input |

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

max

solidity
function max(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the max operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

max

solidity
function max(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the max operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

max

solidity
function max(euint64 lhs, euint64 rhs) internal pure returns (euint64)

This functions performs the max operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

max

solidity
function max(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the max operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

shl

solidity
function shl(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the shl operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

| Name | Type | Description |

lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

shl

solidity
function shl(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the shl operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

shl

solidity
function shl(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the shl operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

shl

solidity
function shl(euint64 lhs, euint64 rhs) internal pure returns (euint64)

This functions performs the shl operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

shl

solidity
function shl(euint128 lhs, euint128 rhs) internal pure returns (euint128)

This functions performs the shl operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

shr

solidity
function shr(euint8 lhs, euint8 rhs) internal pure returns (euint8)

This function performs the shr operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint8	The first input
rhs	euint8	The second input

Return Values

Name	Type	Description
[0]	euint8	The result of the operation

shr

solidity
function shr(euint16 lhs, euint16 rhs) internal pure returns (euint16)

This function performs the shr operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint16	The first input
rhs	euint16	The second input

Return Values

Name	Type	Description
[0]	euint16	The result of the operation

shr

solidity
function shr(euint32 lhs, euint32 rhs) internal pure returns (euint32)

This function performs the shr operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint32	The first input
rhs	euint32	The second input

Return Values

Name	Type	Description
[0]	euint32	The result of the operation

shr

solidity
```
function shr(euint64 lhs, euint64 rhs) internal pure returns (euint64)
```

This functions performs the shr operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint64	The first input
rhs	euint64	The second input

Return Values

Name	Type	Description
[0]	euint64	The result of the operation

shr

solidity
```
function shr(euint128 lhs, euint128 rhs) internal pure returns (euint128)
```

This functions performs the shr operation

If any of the inputs are expected to be a ciphertext, it verifies that the value matches a valid ciphertext
Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
lhs	euint128	The first input
rhs	euint128	The second input

Return Values

Name	Type	Description
[0]	euint128	The result of the operation

not

solidity
```
function not(ebool value) internal pure returns (ebool)
```

Performs the "not" for the ebool type

Implemented by a workaround due to ebool being a euint8 type behind the scenes, therefore xor is needed to assure that not(true) = false and vise-versa

Parameters

Name	Type	Description
value	ebool	input ebool ciphertext

Return Values

Name	Type	Description
[0]	ebool	Result of the not operation on value

not

solidity
function not(euint8 input1) internal pure returns (euint8)

Performs the not operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint8	the input ciphertext

not

solidity
function not(euint16 input1) internal pure returns (euint16)

Performs the not operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint16	the input ciphertext

not

solidity
function not(euint32 input1) internal pure returns (euint32)

Performs the not operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint32	the input ciphertext

not

solidity
```
function not(euint64 input1) internal pure returns (euint64)
```

Performs the not operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint64	the input ciphertext

not

solidity
```
function not(euint128 input1) internal pure returns (euint128)
```

Performs the not operation on a ciphertext

Verifies that the input value matches a valid ciphertext. Pure in this function is marked as a hack/workaround - note that this function is NOT pure as fetches of ciphertexts require state access

Parameters

Name	Type	Description
input1	euint128	the input ciphertext

asEbool

solidity
```
function asEbool(struct inEbool value) internal pure returns (ebool)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an ebool

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	ebool	a ciphertext representation of the input

asEuint8

solidity
```
function asEuint8(ebool value) internal pure returns (euint8)
```

Converts a ebool to an euint8

asEuint16

solidity
```
function asEuint16(ebool value) internal pure returns (euint16)
```

Converts a ebool to an euint16

asEuint32

solidity
```
function asEuint32(ebool value) internal pure returns (euint32)
```

Converts a ebool to an euint32

asEuint64

solidity
```
function asEuint64(ebool value) internal pure returns (euint64)
```

Converts a ebool to an euint64

asEuint128

solidity
```
function asEuint128(ebool value) internal pure returns (euint128)
```

Converts a ebool to an euint128

asEuint256

solidity
```
function asEuint256(ebool value) internal pure returns (euint256)
```

Converts a ebool to an euint256

asEaddress

solidity
```
function asEaddress(ebool value) internal pure returns (eaddress)
```

Converts a ebool to an eaddress

asEbool

solidity
```
function asEbool(euint8 value) internal pure returns (ebool)
```

Converts a euint8 to an ebool

asEuint8

solidity
```
function asEuint8(struct inEuint8 value) internal pure returns (euint8)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint8

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint8	a ciphertext representation of the input

asEuint16

solidity
```
function asEuint16(euint8 value) internal pure returns (euint16)
```

Converts a euint8 to an euint16

asEuint32

solidity
```
function asEuint32(euint8 value) internal pure returns (euint32)
```

Converts a euint8 to an euint32

asEuint64

solidity
```
function asEuint64(euint8 value) internal pure returns (euint64)
```

Converts a euint8 to an euint64

asEuint128

solidity
```
function asEuint128(euint8 value) internal pure returns (euint128)
```

Converts a euint8 to an euint128

asEuint256

solidity
```
function asEuint256(euint8 value) internal pure returns (euint256)
```

Converts a euint8 to an euint256

asEaddress

solidity
```
function asEaddress(euint8 value) internal pure returns (eaddress)
```

Converts a euint8 to an eaddress

asEbool

solidity
```
function asEbool(euint16 value) internal pure returns (ebool)
```

Converts a euint16 to an ebool

asEuint8

solidity
function asEuint8(euint16 value) internal pure returns (euint8)

Converts a euint16 to an euint8

asEuint16

solidity
function asEuint16(struct inEuint16 value) internal pure returns (euint16)

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint16

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint16	a ciphertext representation of the input

asEuint32

solidity
function asEuint32(euint16 value) internal pure returns (euint32)

Converts a euint16 to an euint32

asEuint64

solidity
function asEuint64(euint16 value) internal pure returns (euint64)

Converts a euint16 to an euint64

asEuint128

solidity
function asEuint128(euint16 value) internal pure returns (euint128)

Converts a euint16 to an euint128

asEuint256

solidity
function asEuint256(euint16 value) internal pure returns (euint256)

Converts a euint16 to an euint256

asEaddress

solidity
function asEaddress(euint16 value) internal pure returns (eaddress)

Converts a euint16 to an eaddress

asEbool

```solidity
function asEbool(euint32 value) internal pure returns (ebool)
```

Converts a euint32 to an ebool

asEuint8

```solidity
function asEuint8(euint32 value) internal pure returns (euint8)
```

Converts a euint32 to an euint8

asEuint16

```solidity
function asEuint16(euint32 value) internal pure returns (euint16)
```

Converts a euint32 to an euint16

asEuint32

```solidity
function asEuint32(struct inEuint32 value) internal pure returns (euint32)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint32

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint32	a ciphertext representation of the input

asEuint64

```solidity
function asEuint64(euint32 value) internal pure returns (euint64)
```

Converts a euint32 to an euint64

asEuint128

```solidity
function asEuint128(euint32 value) internal pure returns (euint128)
```

Converts a euint32 to an euint128

asEuint256

solidity

```solidity
function asEuint256(euint32 value) internal pure returns (euint256)
```

Converts a euint32 to an euint256

asEaddress

```solidity
function asEaddress(euint32 value) internal pure returns (eaddress)
```

Converts a euint32 to an eaddress

asEbool

```solidity
function asEbool(euint64 value) internal pure returns (ebool)
```

Converts a euint64 to an ebool

asEuint8

```solidity
function asEuint8(euint64 value) internal pure returns (euint8)
```

Converts a euint64 to an euint8

asEuint16

```solidity
function asEuint16(euint64 value) internal pure returns (euint16)
```

Converts a euint64 to an euint16

asEuint32

```solidity
function asEuint32(euint64 value) internal pure returns (euint32)
```

Converts a euint64 to an euint32

asEuint64

```solidity
function asEuint64(struct inEuint64 value) internal pure returns (euint64)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint64

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint64	a ciphertext representation of the input

asEuint128

```solidity
function asEuint128(euint64 value) internal pure returns (euint128)
```

Converts a euint64 to an euint128

asEuint256

```solidity
function asEuint256(euint64 value) internal pure returns (euint256)
```

Converts a euint64 to an euint256

asEaddress

```solidity
function asEaddress(euint64 value) internal pure returns (eaddress)
```

Converts a euint64 to an eaddress

asEbool

```solidity
function asEbool(euint128 value) internal pure returns (ebool)
```

Converts a euint128 to an ebool

asEuint8

```solidity
function asEuint8(euint128 value) internal pure returns (euint8)
```

Converts a euint128 to an euint8

asEuint16

```solidity
function asEuint16(euint128 value) internal pure returns (euint16)
```

Converts a euint128 to an euint16

asEuint32

```solidity
function asEuint32(euint128 value) internal pure returns (euint32)
```

Converts a euint128 to an euint32

asEuint64

```solidity
function asEuint64(euint128 value) internal pure returns (euint64)
```

Converts a euint128 to an euint64

asEuint128

solidity
function asEuint128(struct inEuint128 value) internal pure returns (euint128)

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint128

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint128	a ciphertext representation of the input

asEuint256

solidity
function asEuint256(euint128 value) internal pure returns (euint256)

Converts a euint128 to an euint256

asEaddress

solidity
function asEaddress(euint128 value) internal pure returns (eaddress)

Converts a euint128 to an eaddress

asEbool

solidity
function asEbool(euint256 value) internal pure returns (ebool)

Converts a euint256 to an ebool

asEuint8

solidity
function asEuint8(euint256 value) internal pure returns (euint8)

Converts a euint256 to an euint8

asEuint16

solidity
function asEuint16(euint256 value) internal pure returns (euint16)

Converts a euint256 to an euint16

asEuint32

solidity
function asEuint32(euint256 value) internal pure returns (euint32)

Converts a euint256 to an euint32

asEuint64

solidity
```
function asEuint64(euint256 value) internal pure returns (euint64)
```

Converts a euint256 to an euint64

asEuint128

solidity
```
function asEuint128(euint256 value) internal pure returns (euint128)
```

Converts a euint256 to an euint128

asEuint256

solidity
```
function asEuint256(struct inEuint256 value) internal pure returns (euint256)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint256

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint256	a ciphertext representation of the input

asEaddress

solidity
```
function asEaddress(euint256 value) internal pure returns (eaddress)
```

Converts a euint256 to an eaddress

asEbool

solidity
```
function asEbool(eaddress value) internal pure returns (ebool)
```

Converts a eaddress to an ebool

asEuint8

solidity
```
function asEuint8(eaddress value) internal pure returns (euint8)
```

Converts a eaddress to an euint8

asEuint16

solidity
```
function asEuint16(eaddress value) internal pure returns (euint16)
```

Converts a eaddress to an euint16

asEuint32

solidity
```
function asEuint32(eaddress value) internal pure returns (euint32)
```

Converts a eaddress to an euint32

asEuint64

solidity
```
function asEuint64(eaddress value) internal pure returns (euint64)
```

Converts a eaddress to an euint64

asEuint128

solidity
```
function asEuint128(eaddress value) internal pure returns (euint128)
```

Converts a eaddress to an euint128

asEuint256

solidity
```
function asEuint256(eaddress value) internal pure returns (euint256)
```

Converts a eaddress to an euint256

asEaddress

solidity
```
function asEaddress(struct inEaddress value) internal pure returns (eaddress)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an eaddress

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	eaddress	a ciphertext representation of the input

asEbool

solidity
```
function asEbool(uint256 value) internal pure returns (ebool)
```

Converts a uint256 to an ebool

asEuint8

solidity

```solidity
function asEuint8(uint256 value) internal pure returns (euint8)
```

Converts a uint256 to an euint8

asEuint16

```solidity
function asEuint16(uint256 value) internal pure returns (euint16)
```

Converts a uint256 to an euint16

asEuint32

```solidity
function asEuint32(uint256 value) internal pure returns (euint32)
```

Converts a uint256 to an euint32

asEuint64

```solidity
function asEuint64(uint256 value) internal pure returns (euint64)
```

Converts a uint256 to an euint64

asEuint128

```solidity
function asEuint128(uint256 value) internal pure returns (euint128)
```

Converts a uint256 to an euint128

asEuint256

```solidity
function asEuint256(uint256 value) internal pure returns (euint256)
```

Converts a uint256 to an euint256

asEaddress

```solidity
function asEaddress(uint256 value) internal pure returns (eaddress)
```

Converts a uint256 to an eaddress

asEbool

```solidity
function asEbool(bytes value) internal pure returns (ebool)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an ebool

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	ebool	a ciphertext representation of the input

asEuint8

solidity
```
function asEuint8(bytes value) internal pure returns (euint8)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint8

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint8	a ciphertext representation of the input

asEuint16

solidity
```
function asEuint16(bytes value) internal pure returns (euint16)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint16

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint16	a ciphertext representation of the input

asEuint32

solidity
```
function asEuint32(bytes value) internal pure returns (euint32)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint32

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint32	a ciphertext representation of the input

asEuint64

solidity

```solidity
function asEuint64(bytes value) internal pure returns (euint64)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint64

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint64	a ciphertext representation of the input

asEuint128

solidity
```solidity
function asEuint128(bytes value) internal pure returns (euint128)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint128

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint128	a ciphertext representation of the input

asEuint256

solidity
```solidity
function asEuint256(bytes value) internal pure returns (euint256)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an euint256

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	euint256	a ciphertext representation of the input

asEaddress

solidity
```solidity
function asEaddress(bytes value) internal pure returns (eaddress)
```

Parses input ciphertexts from the user. Converts from encrypted raw bytes to an eaddress

Also performs validation that the ciphertext is valid and has been encrypted using the network encryption key

Return Values

Name	Type	Description
[0]	eaddress	a ciphertext representation of the input

asEaddress

solidity
```
function asEaddress(address value) internal pure returns (eaddress)
```

Converts a address to an eaddress
Allows for a better user experience when working with eaddresses

asEbool

solidity
```
function asEbool(bool value) internal pure returns (ebool)
```

Converts a plaintext boolean value to a ciphertext ebool

Privacy: The input value is public, therefore the ciphertext should be considered public and should be used
only for mathematical operations, not to represent data that should be private

Return Values

Name	Type	Description
[0]	ebool	A ciphertext representation of the input

Permissioned.md:

Permissioned.Sol

Permission

Used to pass both the public key and signature data within transactions

Should be used with Signature-based modifiers for access control

solidity
```
struct Permission {
  bytes32 publicKey;
  bytes signature;
}
```

Abstract contract that provides EIP-712 based signature verification for access control. To learn more about why this can be important, and what EIP712 is, refer to our Permits & Access Control.

This contract should be inherited by other contracts to provide EIP-712 signature validated access control

SignerNotMessageSender

solidity
```
error SignerNotMessageSender()
```

Emitted when the signer is not the message sender

SignerNotOwner

solidity
error SignerNotOwner()

Emitted when the signer is not the specified owner

constructor

solidity
constructor() internal

Constructor that initializes EIP712 domain separator with a name and version
solhint-disable-next-line func-visibility, no-empty-blocks

onlySender

solidity
modifier onlySender(struct Permission permission)

Modifier that requires the provided signature to be signed by the message sender

Parameters

Name	Type	Description
permission	struct Permission	Data structure containing the public key and the signature to be verified

onlyPermitted

solidity
modifier onlyPermitted(struct Permission permission, address owner)

Modifier that requires the provided signature to be signed by a specific owner address

Parameters

Name	Type	Description
permission	struct Permission	Data structure containing the public key and the signature to be verified
owner	address	The expected owner of the public key to match against the recovered signer

FHERC20.md:

FHERC20

encBalances

solidity
mapping(address => euint32) encBalances

constructor

solidity
```
constructor(string name, string symbol) public
```

allowanceEncrypted

solidity
```
function allowanceEncrypted(address owner, address spender) public view virtual
returns (euint32)
```

allowanceEncrypted

solidity
```
function allowanceEncrypted(address spender, struct Permission permission)
public view virtual returns (bytes)
```

Returns the remaining number of tokens that spender will be
allowed to spend on behalf of owner through transferFromEncrypted. This is
zero by default.

This value changes when approveEncrypted or transferFromEncrypted are called.

approveEncrypted

solidity
```
function approveEncrypted(address spender, struct inEuint32 value) public
virtual returns (bool)
```

Sets a value amount of tokens as the allowance of spender over the
caller's tokens.

Returns a boolean value indicating whether the operation succeeded.

IMPORTANT: Beware that changing an allowance with this method brings the risk
that someone may use both the old and the new allowance by unfortunate
transaction ordering. One possible solution to mitigate this race
condition is to first reduce the spender's allowance to 0 and set the
desired value afterwards:
https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

Emits an ApprovalEncrypted event.

approve

solidity
```
function approve(address owner, address spender, euint32 value) internal
```

spendAllowance

solidity
```
function spendAllowance(address owner, address spender, euint32 value) internal
virtual returns (euint32)
```

transferFromEncrypted

solidity
```
function transferFromEncrypted(address from, address to, euint32 value) public
```

virtual returns (euint32)

transferFromEncrypted

solidity
```
function transferFromEncrypted(address from, address to, struct inEuint32 value)
public virtual returns (euint32)
```

Moves a value amount of tokens from from to to using the allowance mechanism. value is then deducted from the caller's allowance.

Returns a boolean value indicating whether the operation succeeded.

Emits a TransferEncrypted event.

wrap

solidity
```
function wrap(uint32 amount) public
```

unwrap

solidity
```
function unwrap(uint32 amount) public
```

mintEncrypted

solidity
```
function mintEncrypted(address to, struct inEuint32 encryptedAmount) internal
```

transferEncrypted

solidity
```
function transferEncrypted(address to, struct inEuint32 encryptedAmount) public
returns (euint32)
```

transferEncrypted

solidity
```
function transferEncrypted(address to, euint32 amount) public returns (euint32)
```

transferImpl

solidity
```
function transferImpl(address from, address to, euint32 amount) internal returns
(euint32)
```

balanceOfEncrypted

solidity
```
function balanceOfEncrypted(address account, struct Permission auth) public view
virtual returns (bytes)
```

Returns the value of tokens owned by account, sealed and encrypted for the caller.

IFHERC20.md:

IFHERC20

Interface of the ERC-20 standard as defined in the ERC.

TransferEncrypted

solidity
event TransferEncrypted(address from, address to)

Emitted when value tokens are moved from one account (from) to another (to).

Note that value may be zero.

ApprovalEncrypted

solidity
event ApprovalEncrypted(address owner, address spender)

Emitted when the allowance of a spender for an owner is set by a call to approveEncrypted. value is the new allowance.

balanceOfEncrypted

solidity
function balanceOfEncrypted(address account, struct Permission auth) external view returns (bytes)

Returns the value of tokens owned by account, sealed and encrypted for the caller.

transferEncrypted

solidity
function transferEncrypted(address to, struct inEuint32 value) external returns (euint32)

Moves a value amount of tokens from the caller's account to to.

Returns a boolean value indicating whether the operation succeeded.

Emits a TransferEncrypted event.

transferEncrypted

solidity
function transferEncrypted(address to, euint32 value) external returns (euint32)

allowanceEncrypted

solidity
function allowanceEncrypted(address spender, struct Permission permission) external view returns (bytes)

Returns the remaining number of tokens that spender will be
allowed to spend on behalf of owner through transferFromEncrypted. This is
zero by default.

This value changes when approveEncrypted or transferFromEncrypted are called.

approveEncrypted

solidity
function approveEncrypted(address spender, struct inEuint32 value) external
returns (bool)

Sets a value amount of tokens as the allowance of spender over the
caller's tokens.

Returns a boolean value indicating whether the operation succeeded.

IMPORTANT: Beware that changing an allowance with this method brings the risk
that someone may use both the old and the new allowance by unfortunate
transaction ordering. One possible solution to mitigate this race
condition is to first reduce the spender's allowance to 0 and set the
desired value afterwards:
https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

Emits an ApprovalEncrypted event.

transferFromEncrypted

solidity
function transferFromEncrypted(address from, address to, struct inEuint32 value)
external returns (euint32)

Moves a value amount of tokens from from to to using the
allowance mechanism. value is then deducted from the caller's
allowance.

Returns a boolean value indicating whether the operation succeeded.

Emits a TransferEncrypted event.

transferFromEncrypted

solidity
function transferFromEncrypted(address from, address to, euint32 value) external
returns (euint32)

Fhenix-Encryption-UI.md:

Fhenix Encryption UI

Fhenix encryption UI can be found in the following link
This UI is useful for those who are not using remix or using remix without using
the plugin

`![](/img/encui.png)`

Encryption

In order to encrypt a number you can simply write the number you want to encrypt instead of the "Enter a number" text.
You can choose what Euint\ type you want as an output and eventually you can choose one of the two options:

1. Encrypt (Plain) - Will output hex encoded bytes (0x04000...) that can be used as "bytes calldata" input or as the input for the remix plugin
2. Encrypt (InEuint) - Will output hex encoded bytes in square brackets ([0x04000...]) that can be used in remix (not with the plugin) for function that receive inEuint\

All output will be copied to your clipboard and a notification will pop up telling you that the output was copied.

Unsealing

You can only unseal data that was sealed using your wallet public encryption key.
In order to get your wallets public encryption key you can click on "Get Public Key" that will use metamask in order to retrieve the key. The key will be shown as a notification on which you can click in order to copy the value to your clipboard.

Decryption can be done by simply pasting the encrypted value instead of the "Enter sealed value" text and clicking on the Unseal button which will use metamask to decrypt the value.

Permit Generation

This tool can also be used to generate a permit for a contract. Enter a contract address, and click generate permit.
The permit will be generated and copied to your clipboard. You can save the permit to fhenix.js, or use the signature field to interact with contracts using the Permission structure.

Fhenix-Hardhat-Plugin.md:

👷 Fhenix Hardhat Plugin

Fhenix Hardhat Plugin is designed to extend your Hardhat environment with additional capabilities focused on Fhenix. It integrates seamlessly with your Hardhat projects to provide a local Fhenix environment, including customized network configuration and utilities for managing funds and permits within your blockchain applications.

Features

- Local Fhenix Environment: Automatically sets up a local Fhenix network configuration within Hardhat, allowing for easy deployment and interaction with Fhenix contracts.
- Faucet Integration: Enables developers to easily obtain funds for testing purposes through a simple API call to a local faucet.
- Permit Management: Simplifies the process of creating and storing permit signatures required for transactions, reducing the complexity of interacting with contracts that require permissions.

If you want to see a full example in action, check out our Hardhat Example Template!

Installation

To use FhenixJS in your Hardhat project, first install the plugin via npm (or your favorite package manager):

```sh
pnpm install fhenix-hardhat-plugin
```

If you wish to run your own local dev environment, please install the fhenix-hardhat-docker plugin as well.

```sh
pnpm install fhenix-hardhat-docker
```

Setup

After installation, import the plugin in your Hardhat configuration file (e.g., hardhat.config.js):

```javascript
require("fhenix-hardhat-plugin");
// if using the docker plugin
require("fhenix-hardhat-docker");
```

or if you are using TypeScript, in your hardhat.config.ts:

```typescript
import "fhenix-hardhat-plugin";
// if using the docker plugin
import "fhenix-hardhat-docker";
```

Configuration

Network Configuration

The plugin automatically adds a localfhenix network configuration to your Hardhat project. This configuration is designed for local development and includes settings such as gas estimates, accounts, and the local network URL.

This network is chosen as the default once the plugin is imported.
If you want to use a different network, simply add --network <customnetwork> to your hardhat commands, or set it as the default.

If you want to use Fhenix Helium Testnet (or a custom Fhenix network), you can add a new network configuration to your hardhat.config.js file:

```typescript
const config: HardhatUserConfig = {
    networks: {
        fhenixHelium: {
            url: "https://api.helium.fhenix.zone",
            chainId: 8008135,
            accounts: mnemonic,
        },
    },
};

export default config;
```

Using FhenixJS from Hardhat Runtime Environment

After importing fhenix-hardhat-plugin hardhat will automatically extend the

Hardhat Runtime Environment (HRE) with a fhenixjs object, providing access to Fhenix-specific functionality:

- Use the fhenixjs object directly to encrypt, unseal or manage permits.
- getFunds(address: string): Request funds from the local faucet for the specified address.
- createPermit(contractAddress: string, provider?: SupportedProvider): Create and store a permit for interacting with a contract.

Usage

Local Dev Environment

To set up a localfhenix instance, simply import fhenix-hardhat-docker. This will add two new hardhat tasks:

- localfhenix:start To start a local dev environment using docker. By default, the instance will listen for rpc connections on port 42069
- localfhenix:stop Stops the docker container

To start the container:

```sh
pnpm hardhat localfhenix:start
```

If starting the instance was successful, you should see the message: Started LocalFhenix successfully at 127.0.0.1:42069.

To stop the running container:

```sh
pnpm hardhat localfhenix:stop
```

Which will result in Successfully shut down LocalFhenix

Requesting Funds

To request funds from the local faucet for an address, use the getFunds method:

```javascript
await hre.fhenixjs.getFunds("yourwalletaddress");
```

Encryption

```javascript
const encyrptedAmount = await fhenixjs.encryptuint32(15);
```

Creating a Permit

To create a permit for a contract, use the createPermit method:

```javascript
const permit = await hre.fhenixjs.createPermit("contractaddress");
```

Support

For issues, suggestions, or contributions, please open an issue or pull request in the Hardhat Fhenix Plugin GitHub repository.

Fhenix-Remix-Plugin.md:

🎧 Fhenix Remix Plugin

Fhenix created a plugin to ease the interaction with the contracts.

Adding the Plugin

In order to add the plugin you can simply click on the Plugin Manager button in remix (left bottom side), then click on the Connect to a Local Plugin link.
Set the Plugin Name value to be Fhenix and the URL value to be https://remix.helium.fhenix.zone

`![](/img/install-plugin.webp)`

Key Features

Interact with the contract - On contract interaction you should use the values that were encrypted by the plugin for encrypted inputs. For contracts that are returning an output of a sealOutput function, the plugin will already generate a public address and it will decrypt the output for you.
Encrypt numbers
Show permit information of a contract (to manually interact with it)

Using the Plugin

After deploying a contract (the plugin is only aware of contracts that are deployed while it is active), MetaMask will request that you sign a message.
This message is a permit that allows you to interact with the contract from the plugin.
After the message is signed, the contract will be saved to the list.

`![](/img/interact-info-1.webp)`

1. Select the contract you wish to interact with.
2. Remove the selected contract from the list
3. Click to encrypt a number - If the field has a defined type (inEuint8, inEuint16, or inEuint32), it will automatically encrypt it correctly. If the field is of a generic bytes type, you will be prompted to select the required encryption.
4. Autofilled "permission" type - The field detects the unique type and fills it for you based on the created permit.
5. Autofilled "publicKey" - If a publicKey field is detected, it will be auto-filled with the public key from the permit.

Additional Tools

`![](/img/tools-info.webp)`

1. Switch to the Fhenix network or add it to MetaMask if it is not already present.
2. Select the desired encryption type.
3. Select the contract to display permit information.

Your-First-FHE-Contract.md:

---
sidebarposition: 2
displayedsidebar: docsSidebar

Your First FHE Contract

In this short guide, we'll demonstrate how simple it is to enable confidentiality in your smart contracts using Fhenix.

```javascript
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "@fhenixprotocol/contracts/FHE.sol";

contract EarlyWin {
    uint8 plaintext;
    euint8 public cipherText;

    function setCipherText(inEuint8 calldata encryptedNumber) public {
        // convert inEuint8 type structure to euint8
        cipherText = FHE.asEuint8(encryptedNumber);
    }

    function setPlainText(uint8 number) public {
        // set standard plaintext
        plaintext = number;
    }

    function decrypt() public view returns (uint8) {
        return FHE.decrypt(cipherText);
    }
}
```

Code Walkthrough

First, FHE is imported directly into your contract with a single line of code. Next, we establish two unsigned integers, with cipherText being encrypted. This means it will not be publicly accessible by anyone other than the intended viewer. The standard plaintext uint8 represents a number that is public for all to view.

Step By Step

1. Importing FHE

   ```javascript
   import "@fhenixprotocol/contracts/FHE.sol";
   ```

   We can import the FHE precompiles directly into the smart contract with a single line of code. The power of FHE in one single line of copy-paste.

2. Declaring Variables

   ```javascript
   uint8 plaintext;
   euint8 public cipherText;
   ```

   Line 8 is a familiar way of setting a number in Solidity. However, this unsigned integer will be publicly queryable by everyone with access to the network. The number set on line 9 as the encrypted unsigned integer will not be.

3. Setting the Encrypted Number

```javascript
function setCipherText(inEuint8 calldata encryptedNumber) public {
    // convert inEuint8 type structure to euint8
    cipherText = FHE.asEuint8(encryptedNumber);
}
```

Here, we set the encrypted number via the setter function. We pass an inEuint8 as the ciphertext, which represents the number we want to set.

4. Setting the Plaintext Number

```javascript
function setPlainText(uint8 number) public {
    // set standard plaintext
    plaintext = number;
}
```

This is the standard way of setting a number via a function call in plaintext Solidity.

5. Decrypting the Encrypted Number

```javascript
function decrypt() public view returns (uint8) {
    return FHE.decrypt(cipherText);
}
```

Finally, we call the decrypt function to convert the private number to a public one. The method on line 21 represents an example of synchronous decryption. Fhenix will eventually move to an asynchronous decryption call. Don't worry, it will still be possible, and we will update you when the implementation is ready.

Next Steps

If you want to learn more about working with Fhenix, please check out docs for a development tutorial. Here, you will learn how to set up your local dev environment and create an encrypted ERC-20 token!

[//]: (Or, [click here to check out part 2 of our easy win guide](#), where we go over Fhenix principles 101 on Remix. Learn how to handle operations, conditional logic, and permissions (viewing encrypted fields).)

Have Questions?

Hop into our Discord and ask questions in the #dev-general or #tech-questions channels!

Adding View Functions.md:

---
sidebarposition: 4

Adding Encrypted Balance Retrieval

To enhance our contract with secure balance viewing, we're going to implement a getBalanceEncrypted() function. This function will employ permissions to enforce access control, ensuring that only the rightful owner can retrieve and decrypt their encrypted balance.

Defining the Function

We'll start by adding a new function to our WrappingERC20 contract. This function will use the onlySender(perm) modifier from the Permissioned contract to ensure that only the message sender, validated through a signature, can access their encrypted balance.

solidity
```
    function getBalanceEncrypted(Permission calldata perm)
    public
    view
    onlySender(perm)
    returns (uint256) {
        return FHE.decrypt(encBalances[msg.sender]);
    }
```

Off-Chain Signature Generation

Users will need to generate a signature off-chain, using EIP-712 to sign their balance retrieval request. This signature proves that the user has authorized the retrieval of their encrypted balance.

Executing the Function

When calling getBalanceEncrypted(), the user includes their off-chain generated signature as a parameter. The function will execute only if the signature is valid and matches the msg.sender, returning the user's encrypted balance.

Putting it All Together

javascript
```
pragma solidity ^0.8.20;

import "@fhenixprotocol/contracts/access/Permissioned.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@fhenixprotocol/contracts/FHE.sol";

contract WrappingERC20 is ERC20, Permissioned {

    mapping(address => euint32) internal encBalances;

    constructor(string memory name, string memory symbol) ERC20(name, symbol) {
        mint(msg.sender, 100  10  uint(decimals()));
    }

    function wrap(uint32 amount) public {
        // Make sure that the sender has enough of the public balance
        require(balanceOf(msg.sender) >= amount);
        // Burn public balance
        burn(msg.sender, amount);

        // convert public amount to shielded by encrypting it
        euint32 shieldedAmount = FHE.asEuint32(amount);
        // Add shielded balance to his current balance
        encBalances[msg.sender] = encBalances[msg.sender] + shieldedAmount;
    }

    function unwrap(inEuint32 memory amount) public {
        euint32 amount = FHE.asEuint32(amount);
        // verify that our shielded balance is greater or equal than the
```

```
requested amount
        FHE.req(encBalances[msg.sender].gte(amount));
        // subtract amount from shielded balance
        encBalances[msg.sender] = encBalances[msg.sender] - amount;
        // add amount to caller's public balance by calling the mint function
        mint(msg.sender, FHE.decrypt(amount));
    }

    function transferEncrypted(address to, inEuint32 calldata encryptedAmount)
public {
        euint32 amount = FHE.asEuint32(encryptedAmount);
        // Make sure the sender has enough tokens.
        FHE.req(amount.lte(encBalances[msg.sender]));

        // Add to the balance of to and subract from the balance of from.
        encBalances[to] = encBalances[to] + amount;
        encBalances[msg.sender] = encBalances[msg.sender] - amount;
    }

    function getBalanceEncrypted(Permission calldata perm)
    public
    view
    onlySender(perm)
    returns (uint256) {
        return FHE.decrypt(encBalances[msg.sender]);
    }
}
```

intro.md:

---
sidebarposition: 1

Overview

In this guide, we'll be creating a shielded ERC20 token using Solidity. Our token will be unique in that it will offer encrypted token balances, thereby enhancing privacy for token holders.

We'll be making use of the FHE library and Fhenix Helium Testnet to enable this functionality - it allows us to perform computations on encrypted data without first having to decrypt it, which is vital for preserving privacy.

You can find all the completed code in our example project repository. You can just skip there if you just want to see the final code.

[//]: (This example focuses on Javascript. If you're more of a python fan, check out the workshop available here: https://github.com/zama-ai/ethcc23-workshop)

What We'll Be Building

We'll be building a contract for a new token that extends the standard ERC20 token functionality. Our contract will introduce an additional layer of privacy by encrypting token balances. This means that even though transactions are public on the blockchain, it will be impossible to know the balance of a user's account without having the corresponding decryption key.

Our token will offer the ability to 'wrap' and 'unwrap' tokens, where wrapping refers to the conversion of regular tokens into their encrypted form and unwrapping refers to the conversion back into regular tokens.

In addition, our contract will also support the transfer of encrypted tokens from one account to another. The balance of encrypted tokens can also be queried by the token holder, keeping their balance private from others on the network.

Why Is This Useful?

Traditional ERC20 tokens operate transparently, meaning that balances and transactions are publicly visible on the blockchain. This transparency can lead to issues around privacy. For example, once an address is linked to an individual, anyone can view their token balance and see all incoming and outgoing transactions.

By using encryption, we can offer the same functionality while greatly enhancing user privacy. Encrypted balances ensure that no one can determine a user's token balance without the appropriate decryption key. This type of token can be beneficial for users who want the benefits of transacting on the blockchain but with an additional layer of privacy.

This could be particularly useful in a range of applications, from privacy-preserving DeFi applications to personal tokens where the individual does not want their total supply public.

The shielded ERC20 token we will build offers the right balance between transparency, needed for the operation of the blockchain, and privacy, providing individuals the discretion they need over their own finances.

By just extending (and not replacing) the basic functionality of the ERC20 standard we can also maintain compatibility with applications that support the ERC20 token, such as wallets and DeFi applications.

Testing.md:

---
sidebarposition: 6

Testing on Fhenix

During this phase, we will focus on deploying the contract, wrapping tokens, and executing transactions using the FhenixJS library and Hardhat.

FhenixJS is injected by the Fhenix Hardhat plugin and can be used automatically by tests.

We will break down each step, providing code snippets and explanations to ensure you understand how to test the contract effectively.

:::note
At the moment, only Hardhat is supported as a full testing environment. Stay tuned for Foundry support in the future.
:::

Step-by-Step Guide

1. Set Up the Test Environment

First, import the necessary modules and define the initial variables.

javascript
```
import { WrappingERC20 } from "../types/contracts/WrappingERC20";
import hre, { ethers } from 'hardhat';
```

```
import { Permit } from "fhenixjs";

describe('Test WERC20', () =>  {
  let contractAddr: string;
  let contract: WrappingERC20;
  let permit: Permit;
  let owner: string;
  let destination: string = "0x1245dD4AdB920c460773a105e1B3345707B4834A";

  const amountToSend = BigInt(1);
```

2. Test Contract Deployment

In this phase, we will deploy the WrappingERC20 contract and initialize the permit using FhenixJS.

javascript
```
  it(Test Contract Deployment, async () => {
    const { ethers, fhenixjs } = hre;
    const { deploy } = hre.deployments;
    const [signer] = await ethers.getSigners();

    // Set the owner to the signer's address
    owner = signer.address;

    // Deploy the WrappingERC20 contract
    const token = await deploy("WrappingERC20", {
      from: signer.address,
      args: ["Test Token", "TST"],
      log: true,
      skipIfAlreadyDeployed: false,
    });

    // Get the deployed contract address
    contractAddr = token.address;

    // Generate the permit using FhenixJS
    permit = await fhenixjs.generatePermit(contractAddr, undefined, signer);
    contract = (await ethers.getContractAt("WrappingERC20", contractAddr)) as
unknown as WrappingERC20;

    console.log(contractAddr: , contractAddr);
  });
```

Explanation:

- FhenixJS Injection: The fhenixjs object is automatically available through Hardhat's runtime environment (hre). This means you don't need to explicitly import or initialize it.
- Permit Generation: The generatePermit function from FhenixJS is used to create a permit for interacting with the contract. This permit is essential for performing private operations on the contract, such as viewing encrypted balances.

3. Wrap Tokens

Now, we will test the wrapping functionality of the contract.

javascript
```
  it(Wrap Tokens, async () => {
    // Get the balance before wrapping
    let balanceBefore = await contract.balanceOf(owner);
```

```javascript
    let privateBalanceBefore = await contract.getBalanceEncrypted(permit);
    console.log(Public Balance before wrapping: ${balanceBefore});
    console.log(Private Balance before wrapping: ${privateBalanceBefore});

    // Wrap the tokens
    await contract.wrap(amountToSend);

    // Get the balance after wrapping
    let balanceAfter = await contract.balanceOf(owner);
    let privateBalanceAfter = await contract.getBalanceEncrypted(permit);
    console.log(Public Balance after wrapping: ${balanceAfter.toString()});
    console.log(Private Balance after wrapping: $
{privateBalanceAfter.toString()});
  });
```

Explanation:

- Public and Private Balances: Before wrapping tokens, we check both the public balance (visible on the blockchain) and the private balance (encrypted and only visible with the permit).
- Wrapping Tokens: The wrap function is called on the contract to wrap the specified amount of tokens.
- Encrypted Balances: After wrapping, we again check both balances to ensure the wrapping process worked as expected.

4. Execute Transaction

Finally, we will test the transaction execution using encrypted amounts.

```javascript
  it(Execute Transaction, async () => {
    // Get the private balance before sending
    let privateBalanceBefore = await contract.getBalanceEncrypted(permit);
    console.log(Private Balance before sending: ${privateBalanceBefore});

    // Encrypt the amount to send
    const encrypted = await hre.fhenixjs.encryptuint32(Number(amountToSend));

    // Transfer the encrypted amount
    await contract.transferEncrypted(destination, encrypted);

    // Get the private balance after sending
    let privateBalanceAfter = await contract.getBalanceEncrypted(permit);
    console.log(Private Balance after sending: ${privateBalanceAfter});
  });
});
```

Explanation:

- Private Balance Check: Before sending tokens, we check the private balance to verify the initial state.
- Encryption: The amount to send is encrypted using the encryptuint32 function from FhenixJS. This ensures that the amount is securely transmitted.
- Encrypted Transfer: The transferEncrypted function is called on the contract to transfer the encrypted amount to the destination address.
- Balance Verification: After the transfer, we check the private balance again to confirm the transaction.

Conclusion

This guide provided a step-by-step explanation of how to test a contract on Fhenix using Hardhat. By following these steps, you should be able to deploy a contract, wrap tokens, and execute transactions using the FhenixJS library. FhenixJS simplifies handling encrypted operations and permits, making it easier to integrate privacy features into your smart contracts.

Writing-The-Contract.md:

```
---
sidebarposition: 3
---
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
```

Writing the Contract

Let's get started with writing our first FHE powered contract.

Let's create a new file in contracts/, and call that WrappingERC20.sol.

shell
```
touch contracts/WrappingERC20.sol
```

Our goal is to create an ERC20 contract that supports shielded balances. Let's run through the different functions, step-by-step and show how we can implement each. We'll also link to more detailed explanations about the custom functionality we make use of.

Importing FHE Libraries

javascript
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@fhenixprotocol/contracts/FHE.sol";
```

The OpenZeppelin ERC20 contract will provide the basic functionality of the ERC20 token, while FHE.sol is necessary to create and use FHE.

We'll also have to install the OpenZeppelin contracts, since they are not part of the default template.

```
<Tabs groupId="package-managers">
    <TabItem value="npm" label="npm">
        bash
        npm install @openzeppelin/contracts

    </TabItem>
    <TabItem value="yarn" label="yarn">
        bash
        yarn install @openzeppelin/contracts

    </TabItem>
    <TabItem value="pnpm" label="pnpm">
        bash
        pnpm install @openzeppelin/contracts

    </TabItem>
</Tabs>
```

Creating the Contract

Inherit from ERC20

The contract WrappingERC20 is an ERC20 contract. It uses function calls from FHE.sol for encryption and to keep balances private and only viewable by the holder of the correct decryption key.

```javascript
contract WrappingERC20 is ERC20 {

}
```

Create Encrypted Balances

An encrypted balance is initialized for each address, encBalances, which will hold encrypted token balances for users. The euintXXX are encrypted data types that represent FHE-encrypted unsigned integers of various bit lengths.

```javascript
mapping(address => euint32) internal encBalances;
```

Constructor

The constructor function sets the name and symbol of the token, and then mints an initial 100 tokens to the address that deploys the contract.

```javascript
constructor(string memory name, string memory symbol) ERC20(name, symbol) {
     mint(msg.sender, 100  10  uint(decimals()));
}
```

Wrap

First, let's define a function wrap(uint32 amount) that allows users to convert (wrap) their tokens into encrypted form.
The function will burn a specified amount from the user's balance and add the same amount to their encrypted balance.

```javascript
function wrap(uint32 amount) public {
    // Make sure that the sender has enough of the public balance
    require(balanceOf(msg.sender) >= amount);
    // Burn public balance
    burn(msg.sender, amount);

    // convert public amount to shielded by encrypting it
    euint32 shieldedAmount = FHE.asEuint32(amount);
    // Add shielded balance to his current balance
    encBalances[msg.sender] = encBalances[msg.sender] + shieldedAmount;
}
```

Breaking this down, the following logic is performed:

1. Verify that the user has enough public tokens to wrap
2. Burn public tokens
3. Add shielded tokens to the caller's balance

There are two main FHE operations that happened here:

FHE.asEuint32(amount) - this converted a standard, public uint to an FHE-encrypted number
encBalances[msg.sender] + shieldedAmount - this performs homomorphic addition between the two encrypted numbers shieldedAmount and encBalances[msg.sender]

:::note
Even though we called FHE.asEuint32() on a public value and encrypted it we did not actually hide any information - the plaintext value was already known beforehand
:::

Unwrap

Next, let's define unwrap(inEuint32 amount). This function will allow users to convert (unwrap) their encrypted tokens back into public tokens.
The function will remove the specified amount from the user's encrypted balance and add the same amount to the user's public balance.

javascript
```
function unwrap(inEuint32 memory amount) public {
    euint32 amount = FHE.asEuint32(amount);
    // verify that our shielded balance is greater or equal than the requested
amount. (gte = greater-than-or-equal)
    FHE.req(encBalances[msg.sender].gte(amount));
    // subtract amount from shielded balance
    encBalances[msg.sender] = encBalances[msg.sender] - amount;
    // add amount to caller's public balance by calling the mint function
    mint(msg.sender, FHE.decrypt(amount));
}
```

Here we can see a few interesting things:

FHE.req (stands for FHE require) verifies that a statement is true, or reverts the function. We use this to verify that we have enough shielded amount.
encBalances[msg.sender].gte(amount) checks that encBalances[msg.sender] is greater or equal than amount
inEuint32 is a data type specifically for input parameters. You can read more about it here.

Encrypted Transfers

transferEncrypted(address to, bytes calldata encryptedAmount) is a public function that transfers encrypted tokens from the function caller to the to address. It converts the encrypted amount into the encrypted integer form euint32 using the FHE.asEuint32(encryptedAmount) function and then calls transferEncrypted.
The function transferEncrypted(address to, euint32 amount) is an internal function that just calls transferImpl.
transferImpl(address from, address to, euint32 amount) performs the actual transfer. It checks if the sender has enough tokens, then adds the amount to the to address encrypted balance and subtracts the same amount from the from address encrypted balance.

javascript
```
function transferEncrypted(address to, inEuint32 calldata encryptedAmount)
public {
    euint32 amount = FHE.asEuint32(encryptedAmount);
    // Make sure the sender has enough tokens. (lte = less-then-or-equal)
    FHE.req(amount.lte(encBalances[msg.sender]));

    // Add to the balance of to and subract from the balance of msg.sender.
    encBalances[to] = encBalances[to] + amount;
    encBalances[msg.sender] = encBalances[msg.sender] - amount;
}
```

And that's it! To recap, we just created a contract that allows users to wrap and unwrap their tokens, and transfer them in encrypted form.

Let's see what the entire code looks like:

```javascript
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@fhenixprotocol/contracts/FHE.sol";

contract WrappingERC20 is ERC20 {

    mapping(address => euint32) internal encBalances;

    constructor(string memory name, string memory symbol) ERC20(name, symbol) {
        mint(msg.sender, 100  10  uint(decimals()));
    }

    function wrap(uint32 amount) public {
        // Make sure that the sender has enough of the public balance
        require(balanceOf(msg.sender) >= amount);
        // Burn public balance
        burn(msg.sender, amount);

        // convert public amount to shielded by encrypting it
        euint32 shieldedAmount = FHE.asEuint32(amount);
        // Add shielded balance to his current balance
        encBalances[msg.sender] = encBalances[msg.sender] + shieldedAmount;
    }

    function unwrap(inEuint32 memory amount) public {
        euint32 amount = FHE.asEuint32(amount);
        // verify that our shielded balance is greater or equal than the
requested amount
        FHE.req(encBalances[msg.sender].gte(amount));
        // subtract amount from shielded balance
        encBalances[msg.sender] = encBalances[msg.sender] - amount;
        // add amount to caller's public balance by calling the mint function
        mint(msg.sender, FHE.decrypt(amount));
    }

    function transferEncrypted(address to, inEuint32 calldata encryptedAmount)
public {
        euint32 amount = FHE.asEuint32(encryptedAmount);
        // Make sure the sender has enough tokens.
        FHE.req(amount.lte(encBalances[msg.sender]));

        // Add to the balance of to and subract from the balance of from.
        encBalances[to] = encBalances[to] + amount;
        encBalances[msg.sender] = encBalances[msg.sender] - amount;
    }
}
```

Note that in a real use case the actual code would include more functionality, and structure things a bit differently.
If you want to see what such a contract looks like, you can check out the FHERC20 contract in the Fhenix contracts repository.

Wait a second...

But what about viewing the encrypted balances? Well, we'll cover that in the next section, where we'll be adding viewing functionality to our contract, and see how we can utilize Permissions to manage access to encrypted data.

Converting-between-Types.md:

---
sidebarposition: 7
title: 🧙 Type Conversions
description: Converting between different FHE types

Converting Between Types

As a user of FHE.sol, you'll often need to convert between various encrypted types or from plaintext to encrypted form within your contracts. This documentation illustrates how you can leverage the type conversion functions provided in FHE.sol to manipulate encrypted data effectively.

Using Conversion Functions

1. Converting Encrypted to Other Encrypted Types:

Suppose you have a voting contract and you want to convert an encrypted boolean vote to an encrypted integer before tallying.

- Contract Example:
    Javascript
```
contract Voting {
    function convertVote(ebool encryptedVote) public {
        // Convert the encrypted boolean vote to an encrypted 32-bit integer
        euint32 encryptedVoteInt = encryptedVote.toU32();
        // Further processing with encryptedVoteInt
    }
}
```

2. Converting from Plaintext to Encrypted Type:

If you're initializing an encrypted counter in a contract, you might start with a plaintext value that needs to be encrypted.

- Contract Example:
    Javascript
```
contract Counter {
    euint32 private encryptedCount;

    function initializeCount(uint256 initialCount) public {
        // Convert a plaintext count to an encrypted count
        encryptedCount = FHE.asEuint32(initialCount);
    }
}
```

> Note that when converting from plaintext to encrypted, the value is still exposed in plaintext to the contract and on the public blockchain.
> This pattern should only be used when the plaintext data is not sensitive and can be exposed publicly.
>
> After encrypting from plaintext, the resulting encrypted number should be considered public until it is involved in an FHE operation that
> included an input which was sent encrypted to the contract.

Conversion Functions

Tips for Users

- Understand the Types: Know the types you are working with and the implications of converting between them. Ensure the conversion is logical and secure. For example, you can only convert addresses to and from the euint256 type.

- Monitor Gas Usage: Be aware of the gas costs associated with type conversions, especially if they occur within functions that are called frequently.

- Test Thoroughly: Always test your contracts with various scenarios to ensure that type conversions are behaving as expected.

As a user of FHE.sol, understanding and utilizing type conversions is essential for manipulating encrypted data within your smart contracts. By following the examples and best practices provided, you can ensure your contracts are efficient, secure, and functional. Remember to test thoroughly and consider the implications of each conversion to maintain the integrity and reliability of your contract's operations.

Debug-Logging.md:

---
sidebarposition: 8
title: ▫ Console.log
description: How to debug your contracts using Console.log

Console.log in Fhenix's Localfhenix Environment

In Fhenix's Localfhenix environment, the Console.log function and its variants serve as essential debugging tools for Solidity developers. These logs are directed to the Docker log output, aiding in monitoring and troubleshooting smart contract execution in real-time.

Public Functions

The Console library provides two primary public logging functions:

1. log(int256 p0): Logs an integer value.
2. logBytes(bytes memory p0): Logs a byte array.

Usage Examples

Here is how you can use these logging functions in your smart contracts:

Logging an Integer:

```
solidity
import { Console } from "@fhenixprotocol/contracts/utils/debug/Console.sol";

contract ExampleContract {
    function logIntExample() public pure {
        Console.log(123);  // Contract Log: 123
    }
}
```

Logging a Byte Array:

```
solidity
import { Console } from "@fhenixprotocol/contracts/utils/debug/Console.sol";
```

```
contract ExampleContract {
    function logBytesExample() public pure {
        bytes memory data = "Hello, Fhenix!";
        Console.logBytes(data);  // Contract Log: Hello, Fhenix!
    }
}
```

Usefulness in Encrypted Number Handling

When working with encrypted numbers in smart contracts, having robust logging mechanisms is indispensable. Encrypted computations can be complex and opaque, making it difficult to trace issues or verify the correctness of computations. Here's how the logging functions provided by the Console library can be particularly useful:

1. Transparency and Debugging:
   Encrypted numbers typically undergo various transformations and operations. Logging these values at different stages helps verify that transformations are accurate and that no data corruption occurs. For instance, if an encrypted number is not decrypting correctly, logs can help trace back to the point where an issue might have arisen.

2. Validation:
   Smart contracts that operate with encryption often involve sensitive data and critical operations. Logging intermediate values ensures that all operations are performed correctly, and their outcomes match expected results, providing an additional layer of validation.

Here's an example demonstrating how logging might be used in the context of encrypted number operations:

```
solidity
import "@fhenixprotocol/contracts/utils/debug/Console.sol";
import { FHE } from "@fhenixprotocol/contracts/FHE.sol";

contract EncryptedNumberContract {
    using EncryptedNumberLibrary for EncryptedNumber;

    function computeWithEncryptedNumbers(inEuint64 encryptedA, inEuint64
encryptedB) public {

        // Perform some operations
        euint64 result = FHE.asEuint64(encryptedA) + FHE.asEuint64(encryptedB);

        // DEBUG: Log the intermediate result
        uint256 debugResult = FHE.decrypt(result);
        // Log the result
        Console.log(result);

        // Perform more operations
        euint64 finalResult = result  FHE.asEuint64(encryptedA);

        return finalResult;
    }
}
```

By strategically placing logs, developers can gain insights into the operations and transformations performed on encrypted numbers, greatly simplifying debugging and ensuring the integrity of computations.

Viewing Logs in the Localfhenix Docker Container

Logging in the Localfhenix environment is directed to the Docker log output. To view these logs, follow these steps:

:::note
Logging is not available on the Fhenix Testnet or Mainnet. It is only available in the Localfhenix development environment.
:::

If you are running Localfhenix using the Hardhat plugin, you can view the logs by running the following command:

```sh
docker logs localfhenixhhplugin -f 2>&1 | grep "Contract Log:"
```

If you are running LocalFhenix using the Docker image directly, you must first identify the container name using the docker ps command and then view the logs:

```sh
docker logs <containername> -f 2>&1 | grep "Contract Log:"
```

Gas-and-Benchmarks.md:

🔥 Gas and Benchmarks

This section will list the gas costs for every operation based on it's inputs. The gas prices are subject to change based on usage and performance.

:::tip
The current gas limit for a transaction is set to be 50 million
:::

New for Fhenix Kimchi Testnet we changed the calculation of TX data, which previously was heavily discounted artificially.
The new calculation should be similar to default EVM for most transactions.

The new formula offers a discount of 75% for any data over 64KB, with default EVM costs per byte otherwise (64 gas units per non-zero byte, or 4 gas for zero).

The gas costs for the FHE operations are as follows:

FHE.sol function	euint8	euint16	euint32	euint64	euint128	euint256	ebool	eaddress
add, sub	50,000	65,000	120,000	175,000	290,000	n/a	n/a	n/a
asEuint (inEuint)	65,000	65,000	65,000	300,000	300,000	300,000	n/a	300,000
asEuint (euint)	75,000	85,000	105,000	120,000	140,000	175,000	n/a	150,000
asEuint (uint)	20,000	20,000	30,000	35,000	65,000	70,000	n/a	70,000
sealOutput	150,000	150,000	150,000	150,000	150,000	150,000	150,000	150,000
decrypt	25,000	150,000	150,000	150,000	150,000	150,000	150,000	150,000
mul	40,000	70,000	125,000	280,000	n/a	n/a	n/a	n/a
lt, lte, gt, gte	40,000	50,000	75,000	125,000	190,000	n/a	n/a	n/a

select	55,000	55,000	85,000	125,000	225,000	n/a	35,000	n/a
require	150,000	150,000	150,000	150,000	150,000	150,000	150,000	150,000
div, rem	125,000	335,000	1,003,000	n/a	n/a	n/a	n/a	n/a
and, or, xor	40,000	50,000	70,000	130,000	200,000	n/a	35,000	n/a
ne, eq	40,000	50,000	65,000	120,000	180,000	260,000	35,000	210,000
min, max	45,000	55,000	100,000	145,000	250,000	n/a	n/a	n/a
shl, shr	65,000	90,000	130,000	210,000	355,000	n/a	n/a	n/a
not	42,000	35,000	49,000	85,000	120,000	n/a	28,000	n/a

Permissions.md:

---
sidebarposition: 6
title: 🕵️ Permissions
description: Managing access to sensitive data & Permissioned contracts
---

Overview

The Permissioned contract is an abstract Solidity contract that leverages EIP-712 standard signatures to enforce access controls. It's designed to be used by developers who require signature verification to restrict access to certain contract functions. While it can be used to restrict any kind of function, it's particularly useful for creating access-controlled view functions where data should only be visible to entities with a verified signature.

Use Cases

One of the common use cases for such access control is in scenarios where sensitive information must be retrieved from a contract but should not be publicly accessible. For example, a contract managing private user data may implement view functions which require a signature to confirm the identity of the requester. This ensures that only the user or an authorized party can access that user's data.

How to Use

To utilize the Permissioned contract, you would inherit it in your own contract and apply the custom modifiers to the functions you want to protect. To implement access-controlled view functions, follow these steps:

1. Define a view function in your contract. For example, to retrieve sensitive data:

```
    javascript
    function getSensitiveData(Permission calldata perm) public view
onlySender(perm) returns (string memory) {
        // Logic to return sensitive data
    }
```

2. Off-chain, the user generates a signature over their request using EIP-712 signed with their private key. This process typically involves structured data that lists the types of variables involved and their values. The result is a signature that proves the user consents to the requested operation.

3. Call the view function with the generated signature as one of the parameters. Only if the signature is verified and corresponds to the msg.sender will the view function execute and return the sensitive data.

Example Scenario 1

Imagine a contract holding medical records. You want to create a secure method for patients to view their records:

```javascript
pragma solidity ^0.8.20;

import "@fhenixprotocol/contracts/access/Permissioned.sol";

contract MedicalRecords is Permissioned {

    mapping(address => string) private records;

    function viewMedicalRecord(Permission calldata perm) public view
onlySender(perm) returns (string memory) {
        return records[msg.sender];
    }
}
```

The patient, after obtaining the appropriate signature using their private key, would submit it along with their request to view their records. The contract verifies the signature against the caller's address, and if it matches, returns the patient’s medical record.

:::danger
In this example we are just showcasing the usage of permissions. string and address are still public data types and can be read directly from the chain!
:::

Example Scenario 2

```javascript
pragma solidity ^0.8.20;
import {FHE, euint8, inEuint8} from "@fhenixprotocol/contracts/FHE.sol";
contract Test {
    euint8 output;

    function setOutput(inEuint8 calldata encryptedNumber) public  {
        // convert inEuint8 type structure to euint8
        output = FHE.asEuint8(encryptedNumber);
    }

    function getSealedOutput(Permission memory signature) public view  returns
(string memory) {
        // Seal the output for a specific publicKey
        return FHE.sealoutput(output, signature.publicKey);
    }
}
```

Notes

- Permissioned view functions only allow access upon successful signature verification, enhancing contract's data privacy.
- Users need to protect their private keys used to generate EIP-712 signatures to maintain the integrity of the access control system.
- Developers must integrate off-chain EIP-712 compliant signing processes to

ensure users can generate valid signatures for contract interactions.
- EIP-712 signatures provide strong assurances of user intention, making them ideal for sensitive operations.

Requires.md:

---
sidebarposition: 5
title: 🚦Require Statements
description: How to perform assertions on Encrypted data

Require Statement

Encrypted require statements (req) are analogous to standard Solidity require statements. Given an encrypted Boolean predicate, the statement forces the transaction execution to halt if the predicate evaluates to false. Evaluating the encrypted Boolean predicate implies a (threshold) decryption.

Example

In the following code, the function failingRequire is intended to revert the transaction if the equality condition between val and val2 is not met.

Javascript
```
// A transaction calling this function will revert.
function failingRequire(euint8 a) public {
    euint8 val = FHE.asEuint8(4);
    euint8 val2 = FHE.asEuint8(5);
    FHE.req(FHE.eq(val, val2));
}
```

Returning-Data.md:

---
sidebarposition: 3
title: 👈 Outputs
description: Sealing & Decryption - how data from a contract is returned
---

Sealing and Decrypting

When an application reads encrypted data from a Fhenix smart contract, that data must first be converted from its encrypted on-chain form to an encrypted form that the application can read and the user can decrypt.

There are two ways to return encrypted data to the user:

1. Sealed Box Encryption

    The data is returned to the user using sealed box encryption from NaCL. The gist of it is that the user provides a public key to the contract during a view function call, which the contract then uses to encrypt the data in such a way that only the owner of the private key associated with the provided public key can decrypt and read the data.

    From a contract perspective, this is done by using the FHE.sealoutput (or .seal) function, which takes the data to be sealed and the public key of the user, and returns an encrypted blob.

    The encrypted data is then stored in a JSON structure, which is described in a later section.

This data can then be decrypted using fhenix.js, manually by using the caller's private key or using Metamask or compatible APIs.

2. Standard Decryption

Alternatively, Fhenix supports standard decryption as well. If some data needs to be decrypted for public access, that can be done as well and a plaintext value is returned to the caller.
This can be done using the FHE.decrypt function.

Sealed Data Format

:::note
If using fhenixjs, parsing the raw sealed data that is returned from sealoutput or seal is unnecessary.
:::

The following JSON structure shows the components of the encrypted data returned by the seal function:

```json
{
  "version": "x25519-xsalsa20-poly1305",
  "nonce": "<base64 bytes of a nonce used for encrypted>",
  "ephemPublicKey": "<base64 bytes of the target public key>",
  "ciphertext": "<base64 string of a big-endian number>"
}
```

Metamask Compatability

The encryption schema and structure matches the one used by Metamask's ethdecrypt function.
This means that we can consume sealed data directly from Metamask, which provides a more engaging experience for a dApp user.

Fetch an address's public key using the ethgetEncryptionPublicKey method, seal the data for that specific public key (either as a permit or by using the public key directly), and then use Metamask's ethdecrypt call to provide a guided decryption experience.

:::danger[Warning]
Metamask's ethgetEncryptionPublicKey and ethdecrypt methods are deprecated. We provide these examples to demonstrate compatibility with native wallet encryption/decryption procedures. We aim to maintain compatibility as new standards emerge for encryption on Ethereum.
:::

Examples

Sealed Box Encryption

```solidity
import {FHE} from "@fhenixprotocol/contracts";

function sealoutputExample(bytes32 pubkey) public pure returns (bytes memory reencrypted) {
    euint8 memory foo = asEuint8(100);
    return foo.seal(pubkey);
}
```

Decryption

```
Javascript
import {FHE} from "@fhenixprotocol/contracts";

function sealoutputExample() public pure returns (uint8 decrypted) {
    euint8 memory foo = asEuint8(100);
    return FHE.decrypt(foo);
}
```

Metamask Unsealing

```
Javascript
async getPub() {
    const provider = new BrowserProvider(window.ethereum);
    const client = new FhenixClient({provider});
    const accounts = await window.ethereum.request({ method:
'ethrequestAccounts' });
    const keyResult = await provider.send('ethgetEncryptionPublicKey',
[accounts[0]]);
    const pk = 0x${this.base64ToHex(keyResult)};
    this.showNotification(pk);
}
async unseal() {
    const provider = new BrowserProvider(window.ethereum);
    const client = new FhenixClient({provider});
    const accounts = await window.ethereum.request({ method:
'ethrequestAccounts' });
    const result = await provider.send('ethdecrypt', [this.sealedInput,
accounts[0]]);
    const plaintext = this.toString(result);
    this.showNotification(Unsealed result: ${plaintext});
}
```

Taken from the encryption & unsealing tool

Select.md:

---
sidebarposition: 4
title: 🌴 Select vs If...else
description: Why if..else is not possible and what are the alternatives
---

Writing code with Fully Homomorphic Encryption (FHE) introduces a fundamental shift in how we handle conditionals or branches in our code. As you already know, with FHE, we're operating on encrypted data. This means we can't use typical if...else branching structures, because we don't have visibility into the actual values we're comparing.

For example, this will not work:

```
Javascript
euint32 a = FHE.asEuint32(10);
euint32 b = FHE.asEuint32(20);
if (a.lt(b)) {
   return FHE.decrypt(a);
} else {
   return FHE.decrypt(b);
}
```

When writing Solidity contracts for our blockchain, you'll need to consider all possible branches of a conditional at the same time. It's somewhat akin to writing constant-time cryptographic code, where you want to avoid timing attacks that could leak information about secret data.

To handle these conditionals, we use a concept called a "selector".
A selector is a function that takes in a control and two branches, and returns the result of the branch that corresponds to the condition. A selector is like a traffic signal that decides which traffic to let through based on the color of the light (control signal).

In Fhenix, we utilize this by calling the select function. It's a function that takes in a condition and two inputs, and returns the input that corresponds to the state of the condition. You can think of this like a ternary boolean conditional (condition ? "yes" : "no"), but for encrypted data.

Let's take a look at an example of select usage from a Blind Auction Smart Contract: TBD(ADD LINK):

```
Javascript
ebool isHigher = existingBid.lt(value);
bids[msg.sender] = FHE.select(isHigher, value, existingBid);
```

In this snippet, the bidder is trying to place a new bid that is higher than their existing one. The lt function checks if the existing bid is less than the new value and assigns the result to isHigher (the result is of type ebool).

Then FHE.select takes over. If isHigher is true (remember, this is still an encrypted boolean, not a plaintext one), it returns the value (the new bid), otherwise, it returns existingBid. This gets assigned to bids[msg.sender], effectively replacing the old bid with the new one if the new one is higher.

The crucial part here is that all these operations take place on encrypted data, so the actual value of the bids and the result of the comparison stay concealed. It's a powerful pattern to handle conditional execution in the context of FHE data, maintaining privacy without sacrificing functionality.

Types-and-Operators.md:

---
sidebarposition: 100
title: 👨⚕ Types and Operations
description: List of supported types and different operations
---

Supported Types and Operations

The library exposes utility functions for FHE operations. The goal of the library is to provide a seamless developer experience for writing smart contracts that can operate on confidential data.

Types

The library provides a type system that is checked both at compile time and at run time. The structure and operations related to these types are described in this sections.

We currently support encrypted integers of bit length up to 256 bits and special types such as ebool and eaddress.

The encrypted integers behave as much as possible as Solidity's integer types.

However, behaviour such as "revert on overflow" is not supported as this would leak some information of the encrypted integers. Therefore, arithmetic on euint types is unchecked, i.e. there is wrap-around on overlow.

In the back-end, encrypted integers are FHE ciphertexts. The library abstracts away the ciphertexts and presents pointers to ciphertexts, or ciphertext handles, to the smart contract developer. The euint, ebool and eaddress types are wrappers over these handles.
<table>
<tr><th colspan="2"> Supported types </th></tr>
<tr><td>

name	Bit Size	Usage
euint8	8	Compute
euint16	16	Compute
euint32	32	Compute
euint64	64	Compute
euint128	128	Compute
euint256	256	Compute
ebool	8	Compute
eaddress	160	Compute

</td><td>

name	Bit Size	Usage
inEuint8	8	Input
inEuint16	16	Input
inEuint32	32	Input
inEuint64	64	Input
inEuint128	128	Input
inEuint256	256	Input
inEbool	8	Input
inEaddress	160	Input

</td></tr> </table>

Operations

There are three ways to perform operations with FHE.sol:

Using Direct Function Calls

Direct function calls are the most straightforward way to perform operations with FHE.sol. For example, if you want to add two encrypted 8-bit integers (euint8), you can do so as follows:

javascript
```
euint8 result = FHE.add(lhs, rhs);
```

Here, lhs and rhs are your euint8 variables, and result will store the outcome of the addition.

Using Library Bindings

FHE.sol also provides library bindings, allowing for a more natural syntax. To use this, you first need to include the library for your specific data type. For euint8, the usage would look like this:

javascript
```
euint8 result = lhs.add(rhs);
```

In this example, lhs.add(rhs) performs the addition, using the library function implicitly.

Utilizing Operator Overloading

For an even more intuitive approach, FHE.sol supports operator overloading. This means you can use standard arithmetic operators like +, -, \, etc., directly on encrypted types. Here's how you can use it for adding two euint8 values:

```javascript
euint8 result = lhs + rhs;
```

With operator overloading, lhs + rhs performs the addition seamlessly.

Comparisons

Unlike other operations in FHE.sol, comparison operations do not support their respective operators (e.g. >, < etc.).
This is because solidity expects these operators to return a boolean value, which is not possible with FHE.
Intuitively, this is because returning a boolean value would leak information about the encrypted data.

Instead, comparison operations are implemented as functions that return an ebool type.

:::tip
The ebool type is not a real boolean type. It is implemented as a euint8
:::

Supported Operations

:::tip
A documented documentation of each and every function in FHE.sol (including inputs and outputs) can be found in FHE.sol
:::

All operations supported by FHE.sol are listed in the table below. For performance reasons, not all operations are supported for all types.

Please refer to the table below for a comprehensive list of supported operations. This list will evolve as the network matures.

Note that all functions are supported in both direct function calls and library bindings. However, operator overloading is only supported for the operations listed in the table (solidity please support operator overloading for boolean return types!).

name	FHE.sol function	Operator	euint8	euint16	euint32	euint64	euint128	euint256	ebool	eaddress
Addition	add	+	✓	✓	✓	✓	✓	n/a	n/a	n/a
Subtraction	sub	-	✓	✓	✓	✓	✓	n/a	n/a	n/a
Multiplication	mul	\	✓	✓	✓	✓	x	n/a	n/a	n/a
Bitwise And	and	&	✓	✓	✓	✓	✓	n/a	✓	n/a
Bitwise Or	or	|	✓	✓	✓	✓	✓	n/a	✓	n/a
Bitwise Xor	xor	^	✓	✓	✓	✓	✓	n/a	✓	n/a

Division	div	/	✓	✓	✓	x	x	n/a	n/a	n/a
Remainder	rem	%	✓	✓	✓	x	x	n/a	n/a	n/a
Shift Right	shr	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Shift Left	shl	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Equal	eq	n/a	✓	✓	✓	✓	✓	✓	✓	✓
Not equal	ne	n/a	✓	✓	✓	✓	✓	✓	✓	✓
Greater than or equal	gte	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Greater than	gt	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Less than or equal	lte	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Less than	lt	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Min	min	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Max	max	n/a	✓	✓	✓	✓	✓	n/a	n/a	n/a
Not	not	n/a	✓	✓	✓	✓	✓	n/a	✓	n/a
Select	select	n/a	✓	✓	✓	✓	✓	✓	✓	✓
Require	req	n/a	✓	✓	✓	✓	✓	✓	✓	✓
Decrypt	decrypt	n/a	✓	✓	✓	✓	✓	✓	✓	✓
Seal Output	sealOutput	n/a	✓	✓	✓	✓	✓	✓	✓	✓

:::danger
At the moment it is not possible to do ebool result = (lhs == rhs) and others that return a boolean result. This is because FHE.sol expects a ebool, while Solidity only allows overloading to return a regular boolean.
Instead, we recommend ebool result = lhs.eq(rhs).
:::

:::danger
Using require and decrypt in a TX is dangerous as it can break the confidentiality of the data. Please refer to Useful-Tips to read some more
:::

:::tip
Division and Remainder by 0 will output with an encrypted representation of the maximal value of the uint that is used (Ex. encrypted 255 for euint8)
:::

Useful-Tips.md:

---
sidebarposition: 900
title: 💯 Useful Tips
description: Tidbits of wisdom for working with FHE

Trivial Encryption

When we are using FHE.asEuint(PLAINTEXTNUMBER) we are actually using a trivial encryption of our FHE scheme. Unlike normal FHE encryption trivial encryption is a deterministic encryption. The meaning is that if you will do it twice you will still get the same result

Default Value of a Euint

When having a euint variable uninitialized it will be considered as 0. Every FHE function that will receive an uninitialized euint will assume it is FHE.asEuint(0).
You can assume now that FHE.asEuint(0)is used quite often - Luckily we realized this and decided to have the values ofFHE.asEuint(0)pre-calculated on node initialization so when you useFHE.asEuint\(0) we will just return those values.

Re-encrypting a Value

To explain this tip we will use an example. Let's assume we want to develop a confidential voting and let's say we have 4 candidates.
Assuming that on each vote we increase (cryptographically with FHE.add) the tally, one can just monitor the key in the DB that represents this specific tally and once the key is changed he will know who we voted for.
An ideal solution for this issue is to change all keys no matter who we voted for, but how?!

In order to understand how we will first need to understand that FHE encryption is a non-deterministic encryption means that encrypting (non-trivial encryption) a number twice will result with 2 different encrypted outputs.

Now that we know that, we can add 0 (cryptographically with FHE.add) to all of those tallies that shouldn't be changed and they will be changed in the DB!

FHE.req()

All the operations are supported both in TXs and in Queries. That being said we strongly advise to think twice before you use those operations inside a TX.
FHE.req is actually exposing the value of your encrypted data. Assuming we will send the transaction and monitor the gas usage we can probably identify whether the FHE.req condition met or not and understand a lot about what the encrypted values represent.
Example:

```javascript
function f(euint8 a, euint8 b) public {
    FHE.req(a.eq(b));
    // Do some heavy logic
}
```

In this case, if a and b won't be equal it will fail immediately and take less gas than the case when a and b are equal which means that one who checks the gas can easily know the equality of a and b it won't leak their values but it will leak confidential data.
The rule of thumb that we are suggesting is to use FHE.req only in view functions while the logic of FHE.req in txs can be implemented using FHE.select

FHE.decrypt()

Generally speaking, the idea of Fhenix and having FHE in place is the ability to have your values encrypted throughout the whole lifetime of the data (since you can operate on encrypted data). When using FHE.decrypt you should always consider the following:
a. On mainnet (and future testnet versions) the decryption process will be done on a threshold network and the operation might not be fully deterministic (network issues for example)
b. Assuming malicious node runner have DMA (direct memory access) or any other

way to read the process' memory he can see what is the decrypted value while it is being executed and use MEV techniques.

We recommended a rule of thumb to when to decrypt:
a. In view functions
b. In TXs when you are 100% confident that the data is not confidential anymore (For example in poker game when the transaction is a roundup transaction so you can reveal the cards without being afraid of data leakage)

Performance and Gas Usage

Currently, we support many FHE operations. Some of them might take a lot of time to compute, some good examples are: Div (5 seconds for euint32), Mul, Rem, and the time will grow depends on the value types you are using.

When writing FHE code we encourage you to use the operations wisely and choose what operation should be used.
Example: Instead of ENCRYPTEDUINT32 FHE.asEuint32(2) you can use FHE.shl(ENCRYPTEDUINT32, FHE.asEuint32(1)) in some cases
FHE.div(ENCRYPTEDUINT32, FHE.asEuint32(2)) can be replaced by FHE.shr(ENCRYPTEDUINT32, FHE.asEuint32(1))

For more detailed benchmarks please refer to: Gas-and-Benchmarks

Randomness

Confidentiality is a crucial step in order to achieve on-chain randomness. Fhenix, as a chain that implements confidentiality, is a great space to implement and use on-chain random numbers and this is part of our roadmap.
We know that there are some #BUIDLers that are planning to implement dapps that leverage both confidentiality and random numbers so until we will have on-chain true random, we are suggesting to use the following implementation as a MOCKUP.

:::danger
PLEASE NOTE THAT THIS RANDOM NUMBER IS VERY PREDICTABLE AND SHOULD NOT BE USED IN PRODUCTION.
:::

```
solidity
library RandomMock {
    function getFakeRandom() internal returns (uint256) {
        uint blockNumber = block.number;
        uint256 blockHash = uint256(blockhash(blockNumber));

        return blockHash;
    }

    function getFakeRandomU8() public view returns (euint8) {
        uint8 blockHash = uint8(getFakeRandom());
        return FHE.asEuint8(blockHash);
    }

    function getFakeRandomU16() public view returns (euint16) {
        uint16 blockHash = uint16(getFakeRandom());
        return FHE.asEuint16(blockHash);
    }

    function getFakeRandomU32() public view returns (euint32) {
        uint32 blockHash = uint32(getFakeRandom());
        return FHE.asEuint32(blockHash);
    }
}
```

User-Inputs.md:

---
sidebarposition: 2.5
title: 👉 Inputs
description: How to handle encrypted data coming from the user
---

Handling Encrypted Inputs

Overview

Fhenix’s Fully Homomorphic Encryption (FHE) smart contracts handle encrypted data input differently from standard Solidity smart contracts.

First, Fhenix has different data types: boolean, integer and user input.
Second, inEuint and inEbool are used for handling input data, whereas euint and Ebool are used for already processed data within the contract.
Third, conversion is required from inEuint to euint to ensure that only correctly formatted encrypted user input is processed. This is done using a helper function: FHE.asEuintxx.
Finally, follow best practices. Try to minimize storing large quantities of encrypted data on-chain & optimize computation to lower gas costs; process data as needed. Also, use structured types, and avoid using raw bytes to handle encrypted data input.

Encrypted Data Types
Different types of encrypted data can be defined:

- inEbool: Encrypted boolean.
- inEuint8: Encrypted unsigned 8-bit integer.
- inEuint16: Encrypted unsigned 16-bit integer.
- inEuint32: Encrypted unsigned 32-bit integer.
- inEuint64: Encrypted unsigned 64-bit integer.
- inEuint128: Encrypted unsigned 128-bit integer.
- inEuint256: Encrypted unsigned 256-bit integer.
- inEaddress: Encrypted address.

Receiving Encrypted Inputs

Two methods can be used to receive encrypted inputs: inEuintXX structs or raw bytes.

The following code snippets show how to use the two methods for an encrypted transfer to a specific Contract on the blockchain:

inEuintXX Structs

solidity
```
   function transferEncryptedToAccount(address to, inEuint32 calldata
encryptedBalance) public {
     updateAccountBalance(to, FHE.asEuint32(encryptedBalance));
   }
```

Raw Bytes
solidity
```
    function transferEncryptedData(address to, bytes calldata encryptedData)
public {
        storeEncryptedData(to, FHE.asEuint32(encryptedData));
    }
```

As you can see, the advantage of using inEuint over raw bytes is that it ensures type safety and readability. It also provides a structured approach that integrates well with the FHE.sol and fhenix.js library's functions.

Advantages of inEuint, inEbool and inEaddress Over Raw Bytes

Fhenix strongly recommends using inEuintxx (and/or inEbool, inEaddress) structs instead of raw bytes to ensure type safety and readability. These structs provide a structured approach that integrates well with FHE.sol library functions. We believe that the advantages of inEuintxx, inEbool and inEaddress structs are more compatible with handling encrypted data and ensuring application safety, even though raw bytes may result in very slightly lower gas costs.

Examples

Voting in a Poll

solidity
```
   function castEncryptedVote(address poll, inEbool calldata encryptedVote)
public {
     submitVote(poll, FHE.asEbool(encryptedVote));
   }
```

Setting Encrypted User Preferences

solidity
```
    function updateUserSetting(address user, inEuint8 calldata encryptedSetting)
public {
        applyUserSetting(user, FHE.asEuint8(encryptedSetting));
    }
```

inExxx vs. exxx Types

- inExxx types, such as all of inEuint types, inEbool and inEaddress types are used for handling incoming encrypted data.
- exxx types such as all of euint types, ebool and eaddress are used for data already processed and in use within the contract.

Conversion Requirement

Conversion from inEuint (or inEbool, inEaddress) to euint (ebool, eaddress) is required to ensure that only correctly formatted encrypted data is processed.

This is done using the FHE.asEuintXX, FHE.asEbool or FHE.asEaddress functions, where XX is the bit size of the encrypted data. The example above uses the FHE.asEuint8 helper function.

Gas Cost Implications

Attempting to store inEuint, inEbool or inEaddress types directly in storage can lead to prohibitively high gas costs due to the large size of encrypted data. It's generally recommended to avoid storing these directly and instead process them as needed.

Best Practices – Use Structured Types

Ensure data integrity and security of smart contract operation when handling encrypted input. Use the structured inEuint, inEbool or inEaddress types for clearer and safer code, and be mindful of gas costs when designing your contract's data handling strategies. Thorough testing and consideration of security implications are essential in maintaining the robustness and reliability of your FHE-based smart contracts.