Useful Tips

Trivial Encryption

When we are usingFHE.asEuint*(PLAINTEXT_NUMBER) 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 aeuint *variable uninitialized it will be considered as 0. Every FHE function that will receive an uninitialized euint* will assume it isFHE.asEuint*(0) . You can assume now thatFHE.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 was changes 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 theFHE.req condition met or not and understand a lot about what the encrypted values represent. Example:

function

f (euint8 a , euint8 b)

public

{ FHE . req (a . eq (b)) ; // Do some heavy logic } In this case, ifa andb won't be equal it will fail immediately and take less gas than the case whena andb are equal which means that one who checks the gas can easily know the equality ofa andb it won't leak their values but it will leak confidential data. The rule of thumb that we are suggesting is to useFHE.req only inview functions while the logic ofFHE.req in txs can be implemented usingFHE.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 usingFHE.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 ofENCRYPTED_UINT_32 * FHE.asEuint32(2) you can useFHE.shl(ENCRYPTED_UINT_32, FHE.asEuint32(1)) in some casesFHE.div(ENCRYPTED_UINT_32, FHE.asEuint32(2)) can be replaced byFHE.shr(ENCRYPTED_UINT_32, 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. 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); } } Edit this page

Previous ⚕ Types and Operations Next Gas and Benchmarks