Confidential Computation

Confidential computation enables you to handle orderflow privately and securely.

In SUAVE, we achieve this with Kettles performing compute offchain, but according to smart contracts written onchain. In this way, offchain compute is not constrained by chain consensus. The Kettles will eventually run in TEEs , which provide bothenhanced privacy (no-one, not even the host OS, can see unencrypted data) andintegrity (you can be sure the correct code, and only that code, is running at all times).

info For practical examples of how Confidential Compute Requests (CCRs) work, please follow this tutorial . The below will explain CCRs conceptually, without code.

Interface

SUAVE exposes several precompiles to help you with confidential computation. The first is a simple check, often used inrequire statements in smart contracts:

function

isConfidential ( )

internal

view

returns

(bool b) If whoever calls a function in the contract has passed in confidential inputs to be computed offchain, then those inputs can be accessed by the MEVM using:

function

confidentialInputs ( )

internal

view

returns

(bytes

memory) Once the MEVM has the confidential inputs, there is any number of things you could tell it to do with them. For instance, in the Private OFA Suapp we'll look at in greater detail below, we use the below precompile to extract any information about a transaction that the user has agreed to share in order for it to be included timeously in a block on Ethereum L1, thereby replicating MEV-Share in a smart contract on SUAVE:

function

extractHint (bytes

memory bundleData)

internal

view

returns

(bytes

memory) You can also consult our technical specs for further information about confidential computation if required.

Computing over Confidential Data

How to index, store, and use confidential data is left up to each SUAPP.

For example, in the Private OFA Suapp , to submit a valid backrun, a searcher must include therecordId of the user transaction in order for the Suapp to match them. Therefore, the Suapp emits the user transactionrecordId as a log on chain, which searchers can listen for and use to construct valid backruns.

However, the NFTEE example demonstrates how to store a private key in the confidential store. In order to get it to sign a transaction intended for Ethereum L1, we store therecordId associated with that private key in the contract's memory, which ends up onchain. In this context, this is not a concern, since it's gated, so only that Suapp can access the key for signing purposes.

Restricting Access

You need not use confidential requests in the ways which our examples illustrate. For instance, if you wish to restrict access to methods with a modifier requiring some confidential secret, you can do so. This can be achieved by following the below steps, contributed by Miha:

1. When you initialize a contract, pass it a secret key which is stored in confidential storage.
2. Use this key with the local nonce to derive the next secret.
3. The hash of the present secret is in public (evm) storage.
4. Whenever a restricted method is accessed, the present secret needs to be provided.1. This is only accessible through confidential execution.

You can find Miha's implementation ofConfidentialControl here , along with agood example which illustrates its use .

This particular approach, and the reason it currently works in this manner, is being discussed in thissuave-geth issue . Edit this page Previous Confidential Data Store Next Block Building