---

title: Confidential Data Storage description: How to leverage confidential data to your advantage when building on SUAVE

---

import List from '@site/src/components/List/List.tsx';

Confidential storage works with opaque identifiers, which are generated by Kettles when data is put in storage during offchain/confidential computation. These identifiers are part of an object we call a DataRecord, which holds a reference to the data, its ID, who can store and retrieve it, and a few other pieces of metadata.

Interface

This is what these DataRecords look like in the suave-std library:

```solidity type DataId is bytes16;

struct DataRecord {
    DataId id;
    DataId salt;
    uint64 decryptionCondition;
    address[] allowedPeekers;
    address[] allowedStores;
    string version;
}

```

Practical Example

Consider the same Private OFA Suapp from the previous page. The OFA contract needs to accept user transactions, store them, and emit a hint that searchers can use. It then needs to accept backruns from searchers and match those with the user transactions in storage: a task it achieves by means of the recordId.

The end-to-end flow looks like this:

A user submits a transaction they wish to be included in the orderflow auction, calling the newOrder() function.

The contract defines logic which a Kettle can use to get the confidential data, simulate the results of including this transaction, and extract the hint to be shared with searchers. ```solidity bytes memory bundleData = Suave.confidentialInputs(); uint64 egp = Suave.simulateBundle(bundleData); bytes memory hint = Suave.extractHint(bundleData); ```
The contract then sets who can store and "peek" (i.e. retrieve) the data. In this case, both this Private OFA contract and the precompile contract deployed at the address `0x...43200001` are set as **allowedPeekers** and **allowedStores** (though you could have totally different addresses in these arrays in your own use case). If you consult the [`suave-std` library] (https://github.com/flashbots/suave-std/blob/main/src/suavelib/Suave.sol), you'll see that the precompile deployed at that address is the `FILL_MEV_SHARE_BUNDLE`, which is what we require for this particular contract. ```solidity address[] memory allowedList = new address[](2); allowedList[0] = address(this); allowedList[1] = 0x0000000000000000000000000000000043200001; ```
Then, we create the `dataRecord`. The `decryptionCondition` is passed in as an argument and can be anything (though `decryptionConditions` will be deprecated soon) and the version is left as a blank string. Then, we write both the transaction itself, and the results from its simulation (in this case, the effective gas price after it is included) into the confidential store at different keys, which we will later use to match bundles and backruns in order to build profitable blocks. ```solidity Suave.DataRecord memory dataRecord = Suave.newDataRecord(decryptionCondition, allowedList, allowedList, ""); Suave.confidentialStore(dataRecord.id, "mevshare:v0:ethBundles", bundleData); Suave.confidentialStore(dataRecord.id, "mevshare:v0:ethBundleSimResults", abi.encode(egp)); ```
The hint is not directly emitted on chain, as Confidential Compute Requests cannot directly change state, but rather is emitted as a callback once the order has been saved and the hint returned: ```solidity return abi.encodeWithSelector(this.emitHint.selector, hintOrder); ```

Searchers can then use the information in this event to construct and submit valid backruns, which are matched against the original transactions via the hintId.

Matched transactions and backruns are bundled together via the fillMevShareBundle() precompile.

These bundles are sent to predefined off chain builders via the submitBundleJsonRPC() precompile.

In this manner, the Confidential Data Store enables you to create applications on chain that can provide features credible pre-trade privacy (and more!) while nevertheless exposing their business logic in a verifiable and contestable manner.