Create Precompiles

info Creating precompiles requires writing lower level code than the contracts and transactions we've been working with so far. You will need a local clone ofsuave-geth to follow this tutorial, and we'll be making changes to the core code itself. SUAVE usescustom precompiles to extend the EVM with specific MEV functions. Unless you have a very specific use case, building a SUAPP should not require writing precompiles.

In this tutorial, we will add a new precompile to thesuave-geth client that will be accessible in any builder solidity contract on SUAVE.

If you do want to create your own precompile, pleaseconsult our governance process along with this tutorial.

Understand the structure

We specify the inputs and outputs for each precompile inyaml . This automatically generates two bindings, one in Solidity (the client) and another one in Go (the server).

The Solidity binding is implemented in aSolidity library that Suave apps can use to call precompiles. TheGolang counterpart runs in the EVM and handles the Solidity calls to the SUAVE precompiles.

The bindings handle encoding/decoding and error management, providing a standard format for both runtimes to communicate with each other. This removes all the nitty-gritty work required such that you can focus on creating the precompiles you need without getting caught up in the implementation complexities.

Theyaml specification looks like this:

types : -

name : BidId type : bytes16 structs : -

name : Bid fields : -

name : id type : BidId -

name : decryptionCondition type : uint64

---

functions : -

name : confidentialInputs address :

'0x0000000000000000000000000000000042010001' output : plain :

true fields : -

name : output1 type : bytes -

name : newBid address :

'0x0000000000000000000000000000000042030000' input : -

name : decryptionCondition type : uint64 -

name : allowedPeekers type : address [ ] -

name : bidType type : string output : fields : -

name : bid type : Bid There are three top-level objects:types ,structs andfunctions . In this guide, we will focus on thefunctions , as adding a new precompile will most often entail writing a new function.

If you can specify the function's name, the address its logic is deployed at on SUAVE, and what form you expect the inputs and output to take, then our codegen tool will automatically generate both the Solidity and Go bindings required to make your precompile work.

The fields you can include when adding a newfunction to theyaml specification are:

- name
- : Name of the precompile.
- address
- : Address of the precompile.
- input

- : Configuration of the expected input.* name
  - : Name of the input field
  - type
  - : Type of the input field.
- output
- : Configuration of the expected output.* plain
  - : Boolean that specifies whether to pack the output. Only available if the function returns a single array of bytes.
  - fields
  - : Array of output fields for the precompile.* name
    - : Name of the output field
    - type
    - : Type of the output field.

Input and output types can be a basic Solidity type (address), a composite type (address[]), or a reference to any of the custom types and structs (i.e.Struct ,Struct[]). It must be written in the same format as it would be in Solidity.

Add your own

Now we can write a custom SUAVE precompile to perform the "add" operation in order to illustrate how to add your own. This is a two step process:

1. Edit theyaml
2. specification, adding your new precompile to thefunctions
3. block.
4. Implement the custom logic in the Go runtime that is required to get from the inputs to the outputs you specified.

You can edit theyaml specification here . We'll add a new entry in thefunctions section:

functions : -

name : add address :

'0x0000000000000000000000000000000042010009' input : -

name : a type : uint64 -

name : b type : uint64 output : fields : -

name : output1 type : uint64 Then, run our code generator:

go run suave/gen/main.go --write If youryaml additions have no errors and the--write flag is set, the bindings will be (re)generatedhere andhere .

A newAdd function will have been created in the interface, which looks like:

type SuaveRuntime interface

{ ... Add (a uint64 , b uint64)

(uint64 ,

error) ... } You will now need to write the logic required for your precompile to work as expected based on this generated skeleton.

Adapter code will be written intocontracts_suave_runtime_adapter.go , but you will need to add your method tosuave_contracts.go . In our case, the logic is very simple: just a straightforward addition of the values passed in as inputs. Your implementation may be arbitrarily more complex based on what you want the precompile to achieve.

func

(b * suaveRuntime)

Add (a uint64 , b uint64)

(uint64 ,

error)

{ return a + b ,

nil } You can find a worked example of how to add a more complicated precompile in this PR . Edit this page Previous Confidential Compute Requests Next Tools