[

image

948×949 371 KB

](https://collective.flashbots.net/uploads/default/original/2X/2/20fe0b8d6fb3355e01cc1dad2c6ee2914224425d.jpeg)

M.O.S.S Open Sequencing Substrate

M.O.S.S. is a contract pattern for SUAVE block construction, which intends to open up innovation and participation as widely as possible for originators and processors of order flow without sacrificing execution guarantees. We are in a research phase and are looking for feedback. The design is not optimised and still has some unresolved issues as can be found at the bottom of the doc. The post is Ethereum-centric, but the concepts can easily be generalised. Background on SUAVE architecture is assumed.

Moss

In the natural world, moss plays a vital role as a foundational element in various ecosystems. This simple yet resilient plant creates a substrate that supports a diverse array of emerging life forms, offering a stable base yet allowing for dynamic growth and evolution.

Similarly, the M.O.S.S Open Sequencing Substrate mirrors this natural phenomenon. In the current block building landscape, the absence of a unified foundation has led to significant challenges. New bundle types aiming to integrate into the existing ecosystem face considerable overhead and complexity. This fragmentation hinders innovation and efficiency, much like a garden without a common ground for plants to root.

Required properties

- Users stipulate the rules for their execution. It follows that SUAPPs must be able to stipulate rules for execution as users opt into the rules of SUAPPs
- Anyone can deploy a building algorithm without needing to campaign for special permissions
- Building algorithms need not be public

High-level description

The M.O.S.S system is built on two core layers:

1. The First Layer:

This layer hosts contracts such as CoWsuave, culminating in objects that implement the IBundle

interface defined later on. It also offers flexibility for users to directly send bundles, catering to a variety of needs and applications.

1. The Second Layer:

The block building layer, which assembles blocks from bundles. This process is facilitated by an block construction contracts, which apply bundles to form blocks. What “applies” means is defined by the bundle. In addition to the logic of the bundle itself, orderflow originators must carefully select block construction contracts which enforce acceptable rules such as no unwanted information leakage. The final step of the process entails constructed blocks being passed to a relay contract which privately stores the highest value block, waiting for the getHeader call. This allows built blocks to remain private until committed to on chain.

The bundle sequencing algorithm can be defined outside of the MEVM to protect proprietary code (although there are other ways of achieving this as well). Such an algorithm would not have direct access to all the information about generic bundles but would be able to call a getHints function and use the information returned to intelligently try orderings of bundles. Thus users are able to control the use of their information in block construction in a manner reminiscent of MEV-share.

The approach outlined above does not prevent the use of non-SUAVE flow in the construction of SUAVE blocks (can be passed in at the build() call).

[

image

1676×775 283 KB

](https://collective.flashbots.net/uploads/default/original/2X/f/f3affe8e78cf81de76c00361d49df26444e7238f.jpeg)

Core Interfaces

In M.O.S.S there are two core interfaces that map onto the two layers mentioned above.

Bundle Interface

To hook your SUAPP into M.O.S.S, the SUAPP needs to call a bundle contract which implements two core functions:

- emitHint
- this function selectively reveals information about the underlying bundle at the SUAPP developers discretion
- apply
- this function takes in a block state and simulates the bundle on top of it, choosing to return signed transactions representing the bundle based on the results of the simulation. This function must be gated so that only trusted contracts receive the legacy transactions.

interface IBundle { function emitHint() external; function apply(bytes calldata blockState) external returns (LegacyTransaction[] memory); }

Block Interface

An important difference between a block and a bundle is that appending to a bundle doesn't guarantee state (i.e. can prepend something as well). The M.O.S.S Block interface has one function:

- build
- this function takes a list of bundles and sequentially simulates them, only committing a bundle if the result of calling apply

on the bundle with the current block state is successful (returns transactions)

interface IBlock { function build(IBundle[] calldata bundles) external; }

Note: SUAPPs implementing IBundle will need to permission Block contracts

Example Flow

[

image

1820×752 341 KB

](https://collective.flashbots.net/uploads/default/original/2X/0/08dfa726259c1e6072db785bc1d37733a2bb770c.jpeg)

The M.O.S.S. block building process could happen across many kettles, but, for simplicity, let's think through an example of one block-building kettle. Apart from the kettle and its operator (the brewer), the relevant actors are the validator and the users. We assume that the brewer uses a piece of software known as the "proprietary sequencing algorithm"

to handle block building duties outside of MEVM/TEE execution. The pseudocode below this section is also a useful reference.

1. Users seeking to interact with SUAPPs send confidential compute requests (CCRs) to the kettle
2. The kettle executes the CCRs according to the SUAPP logic (e.g. executing a batch auction). After the SUAPP executes it must outputs an encrypted bundle which is stored in the confidential store. The BundleID

is also output to alert the brewer that a new bundle has arrived.

1. After having received a critical number of bundles, the brewer's sequencing algorithm calls getHint on these bundles. What information is returned depends on the type of the bundle (i.e. which bundle contract is allowed to interact with the encrypted bundle data). Examples of bundles are included below.
2. Using the information returned from calling getHint()

on the bundles, the sequencing algorithm passes in a list of bundleID

s to the building contract. Depending on the implementation of the building algorithm, additional information may be passed in that determines block bids, bundle ordering logic etc. For now let's stick with a simple version in which the sequencing algo just passes in an ordered list of bundles.

1. The building algorithm, which must be explicitly granted permissions by bundles, calls apply()

on these bundles. apply

takes in information from the building algorithm (e.g. information about eth state at a point in the block) and returns an Ethereum transaction(s) to be used by the building contract. The building contract can clearly misuse these transactions which is why the building contract must be specifically permissioned by the bundles.

1. Upon completion, the building contract forwards the block to the relay contract which compares it to the prev highest value block and stores an encrypted version of the updated highest value block.
2. The validator eventually engages in a scheme with the relay contract (e.g. PBS commit-reveal) which ends with the highest value block being decrypted.

Note that in this design innovation on bundle formats by users/searchers and relay contracts by validators (in PEPC style) is permissionless. While anyone could deploy a building contract, new building contracts would require bundles to explicitly permission it, meaning that there are significant barriers to deploying a new building contract.

The design gives bundle submitters guarantees that

- bundle logic will be enforced (e.g. revert protection)
- sequencing logic only has access to information about your bundle which you explicitly grant it (caveat: side channels)

It also allows any brewer to deploy their own sequencing algorithm, which can also be used to incorporate flow from outside of SUAVE.

Pseudo-Code

Basic Bundle

A basic example of an IBundle

instantiation

contract BasicBundle is IBundle { LegacyTransaction[] private transactions;

```
event HintEvent(bytes Hint);

function emitHint() external override {
        // some arbitrary hint extraction logic
    bytes hint = extractHint(transactions)
    emit HintEvent(hint);
}

function apply(bytes calldata blockState) external override returns (LegacyTransaction[] memory) {
        // its important to make sure the caller contract behaves
        //  in an expected way
        // this will require delegatecall in a real implementation
    assert(msg.sender==blockConstructionContract)

        // Arbitrary logic like bundle simulation

        return transactions;
}
```

}

Basic Block

This defines a block object which allows for bundles to be appended. In the MOSS pattern, IBundle.apply()

should only be callable from within IBlock.build()

implementations which enforce acceptable behaviour.

contract BasicBlock is IBlock { callBuild(client, blockBuilder, bundleOrder) // Internal representation of the block state // Most likely this looks like a list of block session IDs bytes private blockState;

```
// Function to build block from given bundles
function build(IBundle[] calldata bundles) external override {
    //its important this function doesn't leak any unintended info
    for (uint i = 0; i < bundles.length; i++) {
        IBundle bundle = bundles[i];

        // Apply the bundle to the current block state and update the state
        LegacyTransaction[] memory txs = bundle.apply(blockState);
        // Update blockState based on transactions - placeholder logic
        success = commitTransactions(txs, blockState);
    }
            return true
}

// Function to retrieve the current block state
function getBlockState() public view returns (bytes memory) {
    return blockState;
}

    // this part is the most pseudo code
function commitTransactions(LegacyTransaction[] memory txs, bytes memory currentState) private pure returns (bool memory) {
     // Update the block
        blockState = blockState.ApplyTransaction()
        return true;
}

}
```

Example Blind Bundle Sequencer

This golang BlindSequencer

:

- maintains a set of bundles to extract hints on
- listens for new SUAVE blocks
- creates some sequence from those hints
- calls buildBlock

with sequence

Note: This program only constructs one sequence per block, but you could imagine it grinding many different orderings and cache'ing sim results.

package BlindSequencer

// BundleContract represents the ABI of the bundle contract type BundleContract struct { Address common.Address HintABI abi.ABI }

// BlockBuildingContract represents the ABI of the block building contract type BlockBuildingContract struct { Address common.Address MOSSABI abi.ABI }

func BlindSequencer() { SuaveClient, err := ethclient.Dial("SUAVE_NODE_URL")

```
// Initialize the bundle contracts and block building contract
bundleContracts := []BundleContract{
    // Initialize with actual contract addresses and ABIs
}
blockBuildingContract := BlockBuildingContract{
    // Initialize with actual contract address and ABI
}

// Listen to new blocks
headers := make(chan *types.Header)
sub, err := client.SubscribeNewHead(context.Background(), headers)

    // listen for new head events
for {
    select {
    case header := <-headers:
        handleNewBlock(client, header, bundleContracts, blockBuildingContract)
    }
}

}
```

func handleNewBlock(client *ethclient.Client, header* types.Header, bundles []BundleContract, blockBuilder BlockBuildingContract) { var hints []byte

```
// Call emitHint on all bundles and collect hints
for _, bundle := range bundles {
    hint := callEmitHint(client, bundle)
    hints = append(hints, hint...)
}

// Process hints through someFunction
bundleOrder := someFunction(hints)

// Call build on the block building contract with the bundle order
callBuild(client, blockBuilder, bundleOrder)
```

}

Happy Path Bundle Grinding

Note: This example only shows one example sequence but you can use your imagination to envision it trying 3! combinations of BasicBundle 1, 2, and 3.

[

image

580×857 42.7 KB

](https://collective.flashbots.net/uploads/default/original/2X/f/ffb53f195a62bb027cfcfc1d4997542fb1f08427.png)

Example Bundle Types

A few quick examples of bundle types and how their emitHint

and apply

would be structured, they get more pseudo-code-ish the further down you go.

Revert Protection Bundle

This bundle contract is meant to replicate the functionality of today’s bundles under the M.O.S.S. pattern by implementing revert protection.

contract StandardBundle is IBundle {

// Array to store transactions LegacyTransaction[] private transactions; // which tx can revert uint[] private canRevert;

```
// Event to emit hints
event Hint(LegacyTransaction[] transactions);

// Extract hint
function extractHint() external override {
            // do some fancier extracting logic
    emit Hint(transactions);
}

function apply(blockState) external returns (LegacyTransaction[] memory) {
        //to keep track of which functions didn't revert
        LegacyTransaction[] private doReturn;
        uint cnt = 0;

        // iterate through
        for (uint i = 0; i<=transactions.length; i++) {
            didRevert = blockState.applyTransactions(transactions[i])
            if didRevert{
                !canRevert[i]{
                    // don't include
                    return LegacyTransaction[](0) // pseudo code
                }
            }else{
                doReturn[cnt] = transactions[i]
            }
  }
        return doReturn
}
```

```
}
```

MEV Share Bundle

This implements the basic logic needed to conform to themev-share protocol.

```
contract MevShareBundle is IBundle {

    BlockStateSimulator simulator;

// Store user transaction
LegacyTransaction private userTransaction;

// Structure to store the original transaction and its match
struct TransactionPair {
    LegacyTransaction original;
    LegacyTransaction match;
    LegacyTransaction refund;
}

// Array to store transaction pairs
TransactionPair[] private transactionPairs;

// Event to emit hints
event HintEvent(bytes hint, uint index);

// Function to send in a transaction and emit a hint
function submitTransaction(LegacyTransaction memory transaction) external {
    bool res = SUAVE.simulateTransaction(transaction);
    require(res == true, "Transaction Invalid");
    userTransaction = transaction;
    bytes memory hint = extractHint(transaction);
    emit HintEvent(hint, transactionPairs.length);
}

// Function to submit a match for a transaction
function submitMatch(LegacyTransaction memory matchTransaction) external {
    SimulationResult res = SUAVE.simulateBundle(
        userTransaction,
        matchTransaction
    );
    require(res == true, "Transaction Pair Invalid");
    require(res.CoinbaseDiff > 0, "Match Does Not Refund");

    LegacyTransaction memory refundTransaction = SUAVE.ConstructRefund(
        res,
        userTransaction
    );

    // Adding the transaction pair to the array
    transactionPairs.push(TransactionPair({
        original: userTransaction,
        match: matchTransaction,
        refund: refundTransaction
    }));
}

// Extract hint function
function emitHint() external override {
    // Logic for extracting hint
    // ...
}

// Apply function
function apply(blockState) external override returns (LegacyTransaction[] memory) {
            blockStateSimulator = simulator(blockState)

    uint highestRefundIndex = 0;
    uint highestRefundValue = 0;

    // Iterate through all transaction pairs
    for (uint i = 0; i < transactionPairs.length; i++) {
        SimulationResult res = blockStateSimulator.simulateBundle(
            transactionPairs[i].original,
            transactionPairs[i].match
        );

        // Check for the highest UserRefundValue
        if (res.UserRefundValue > highestRefundValue) {
            highestRefundValue = res.UserRefundValue;
            highestRefundIndex = i;
        }
    }

    // Return the transaction pair with the highest UserRefundValue
    TransactionPair memory highestRefundPair = transactionPairs[highestRefundIndex];
    return new LegacyTransaction[](3) {
        highestRefundPair.original,
        highestRefundPair.match,
        highestRefundPair.refund
    };
}

}
```

CowSUAVE Bundle

This is one of the most interesting examples because it addresses a problem that currently exists in CowSwap which is that solutions are being scored based on top of block sims not behind transactions which they will actually exist behind.

contract CowSUAVEBundle is IBundle { struct Bid { address bidder; uint amount; // Other relevant fields }

```
struct Solution {
    LegacyTransaction transaction;
    // Other relevant fields
}

Bid[] private bids;
Solution[] private solutions;

event BidHintEmitted(Bid bid);
event SolutionSubmitted(uint index);

// Function for users to submit bids
function submitBid(Bid memory bid) external {
    bids.push(bid);
    emit BidHintEmitted(bid);
}

// Function to submit a solution (transaction)
function submitSolution(LegacyTransaction memory transaction) external {
    // Simulate the transaction to ensure it's valid
    uint res = SUAVE.simulateTransaction(transaction);

            require(res == true, "Solution Invalid");

    solutions.push(Solution({
        transaction: transaction
    }));
    emit SolutionSubmitted(solutions.length - 1);
}

function apply(blockState) external override returns (LegacyTransaction[] memory) {
            blockStateSimulator = simulator(blockState)
    // add some check that enough time has passed for the bids

            uint highestWelfareScore = 0;
    Solution memory bestSolution;

    // Iterate through each submitted solution
    for (uint i = 0; i < solutions.length; i++) {
        // Simulate each solution on the block state
        simResult res = blockStateSimulator(solutions[i].transaction);
        // Check if this solution has the highest welfare score
        if (res.UserWelfareScore > highestWelfareScore) {
            highestWelfareScore = res.UserWelfareScore;
            bestSolution = solutions[i];
        }
    }

    // Return the solution with the highest welfare score
    return new LegacyTransaction[](1) { bestSolution.transaction };
}

}
```

4337 UserOp Bundle

This is a high level sketch of how a 4337 userOp Bundle would work according to EIP 4337.

contract UserOpBundle is IBundle {

```
BlockStateSimulator simulator;

// Structure to store User Operations
struct UserOp {
    // UserOp Values, use your imagination
}

// Array to store UserOps
UserOp[] private userOps;

// Address of the EntryPoint
address public entryPoint;

// Event to emit when a UserOp is submitted
event UserOpSubmitted(uint index);

// Constructor to set the entry point address
constructor(address _entryPoint) {
    entryPoint = _entryPoint;
}

// Function to validate a UserOp (empty for now)
function validateUserOp(UserOp memory userOp) private {
    // Validation logic
}

// Function to submit a UserOp
function submitUserOp(UserOp memory userOp) external {
    validateUserOp(userOp);
    userOps.push(userOp);
    emit UserOpSubmitted(userOps.length - 1);
}

// Apply function
function apply(blockState) external override returns (LegacyTransaction[] memory) {
    blockStateSimulator = simulator(blockState);

    // Create a transaction targeted at the entry point
    LegacyTransaction memory userOpTx = LegacyTransaction({
        // add userOps to call data in specified format
        to: entryPoint
    });

    // Simulate the transaction
    bool res = blockStateSimulator.simulateTransaction(userOpTx);
    require(res == true, "Transaction Simulation Failed");

    // Return the transaction
    return new LegacyTransaction[](1) { tx };
}
```

}

Routing Bundle

this bundle routes a trade perfectly given some state (i.e. does not have to guess what prices will be when routing). If this feels very inefficient to you, have a look at the open questions at the bottom.

contract RoutingBundle is IBundle { // Array to store transactions EIP712 private limitOrder;

```
// Event to emit hints
event Hint(LegacyTransaction[] transactions);

// Extract hint
function extractHint() external override {
        // do some fancier extracting logic
    emit Hint(transactions);
}

function fancyRouting(BlockState blockState) internal returns (LegaceTransaction{
    assetA, assetB := limitOrder.getAssets();
    limit := limitOrder.getLimit();
    // bipartite graph sth sth
}

function apply(blockState) external returns (LegacyTransaction[] memory){
    LegacyTransaction tx := fancyRouting(blockState);
    if tx == nil{ return nil;}
    return tx;//pretend its an array
}

}
```

Batch Auction

contract EIP712AuctionBundle is IBundle { struct Bid { address bidder; uint amount; // Other fields as per EIP712 standard }

```
Bid[] private bids;
bool private auctionActive;
uint private auctionEndTime;
Bid private winningBid;

event AuctionStarted(uint endTime);
event BidReceived(address bidder, uint amount);
event AuctionSettled(Bid winningBid);

function startAuction(uint duration) external {
    require(!auctionActive, "Auction already in progress");
    auctionActive = true;
    auctionEndTime = block.timestamp + duration;
    emit AuctionStarted(auctionEndTime);
}

function submitBid(Bid memory bid) external {
    require(auctionActive, "No active auction");
    require(block.timestamp < auctionEndTime, "Auction has ended");
    bids.push(bid);
    emit BidReceived(bid.bidder, bid.amount);
}

function settleAuction() external {
    require(block.timestamp >= auctionEndTime, "Auction not yet ended");
    require(auctionActive, "No active auction to settle");

    uint highestAmount = 0;
    for (uint i = 0; i < bids.length; i++) {
        if (bids[i].amount > highestAmount) {
            highestAmount = bids[i].amount;
            winningBid = bids[i];
        }
    }

    auctionActive = false;
    emit AuctionSettled(winningBid);
}

function emitHint() external override {
    // Hint extraction logic
            emit HintEvent(hint);
}

function apply(bytes calldata blockState) external override returns (LegacyTransaction[] memory) {
    blockStateSimulator = simulator(blockState);

            // Ensure auction is settled before applying
    require(!auctionActive, "Auction not yet settled");

    // Convert winning bid to a transaction
    LegacyTransaction memory winningTransaction = convertBidToTransaction(winningBid);

            bool res = blockStateSimulator.simulateTransaction(winningTransaction);
    require(res == true, "Transaction Simulation Failed");

    return new LegacyTransaction[](1) { winningTransaction };
}

// Pseudo function to convert a bid to a transaction
function convertBidToTransaction(Bid memory bid) private returns (LegacyTransaction memory) {
    // Convert the bid to a LegacyTransaction
}

}
```

NFT Auction Bundle

contract NFTAuctionBundle is IBundle { struct NFTBid { address bidder; uint bidAmount; uint nftId; }

```
NFTBid[] private bids;
bool private auctionActive;
uint private auctionEndTime;

event AuctionStarted(uint endTime);
event BidReceived(address bidder, uint bidAmount, uint nftId);
event AuctionSettled(uint nftId, NFTBid winningBid);

function startAuction(uint duration, uint nftId) external {
    require(!auctionActive, "Auction already in progress");
    auctionActive = true;
    auctionEndTime = block.timestamp + duration;
    emit AuctionStarted(auctionEndTime);
}

function submitBid(NFTBid memory bid) external {
    require(auctionActive, "No active auction");
    require(block.timestamp < auctionEndTime, "Auction has ended");
    bids.push(bid);
    emit BidReceived(bid.bidder, bid.bidAmount, bid.nftId);
}

function settleAuction(uint nftId) external {
    require(block.timestamp >= auctionEndTime, "Auction not yet ended");
    require(auctionActive, "No active auction to settle");

    // Logic to determine the winning bid for the specified NFT
    NFTBid memory winningBid;
    // ... Winning bid logic

    auctionActive = false;
    emit AuctionSettled(nftId, winningBid);
}

function extractHint() external override {
    // Hint extraction logic
}

function apply(bytes calldata blockState) external override returns (LegacyTransaction[] memory) {
    // Logic to apply the auction results to the block state
    // Return the array of transactions including the NFT transfer
}

}
```

More ideas

- HeuristicRoutingBundleInstead of routing in-place like the RoutingBundle, which is slow, a bundle can hold N transactions. Each transaction executes the same trade with a different route. The bundle also holds a binary tree which checks storage slots. Leaves are transactions.

Open Questions

Security

Can this pattern be significantly restrictive to avoid undesirable behaviour?

- Side channels -

this pattern currently relies on privacy of execution for user guarantees. This is not attainable due to side channels. Even if one bundle implements mitigations like ORAM, bundles executed afterwards may not. One important detail to remember is that we only need privacy to last for a few seconds * An even stronger concern is the second bundle directly leaking information about blockState.

- An even stronger concern is the second bundle directly leaking information about blockState.

- Blind Misbehaviour -

even if we can perfectly implement privacy, there may still be possible attacks. For example, a sequencing algorithm could always try to probabilistically sandwich the block and rely on the relay contract to select the highest value sandwich. It may be that such attacks are not valuable in expectations, but further analyses is required.

- Statement of properties

- the design outlined above doesn't provide any statement of exact properties its trying to enforce. What makes a building process "secure"? If we had some specific properties in mind, we could have the long term goal of having the block construction contract check proofs that sequencing algorithms satisfied certain properties.

Efficiency & Econ

- Latency
- how to make go brrr without crazy sidechannel vulns?
- Information
- while individual bundles can control how much privacy influences their execution by choosing how much information is revealed, there is still the issue of interim information. For example, in the given block building example, information about blockstate halfway through the block cannot be used to intelligently adapt the sequencing rule for security reasons. How do we balance security and efficiency when it comes to information about block execution?
- “gas” -

not all bundles are the same. The MOSS pattern allows bundles to define arbitrary logic. Thus some bundles may consume more gas than others. This slows down block computation time which is to the detriment of other agents. There must be some disincentive for consuming too much gas. One way to do this is to pass information about simulation load (MEVM gas basically) to the sequencing algorithm, but this leakage must be balanced with security concerns. Another approach is to charge per unit gas consumed, but its unclear how to do this given bundles are likely to be executed multiple times per export block construction.

Functionality and Market Structure

- does this design disallow certain desirable functionality?
- is this design likely to lead to undesirable market dynamics?
- does this design introduce unnecessary complexity?

There’s more to say on how searchers can use contracts to collaborate, how relay contracts can implement PEPC and how you actually get to the point where you have enough flow for this to be viable, but that might bloat the post so we’ll save it for later.