Decrypting ERC4337: Technical Architecture and Transactions

Albert Lin

Who am I

- Albert Lin
- Furucombo Solidity Engineer
- Started participating in the cryptocurrency realm since 2018.

Agenda

- Introduce ERC4337
- ERC4337 Architecture and Flow
- Decode EntryPoint and UserOperation
- Aggregate Signatures
- Takeaways

Introduce ERC4337

What is Account Abstraction

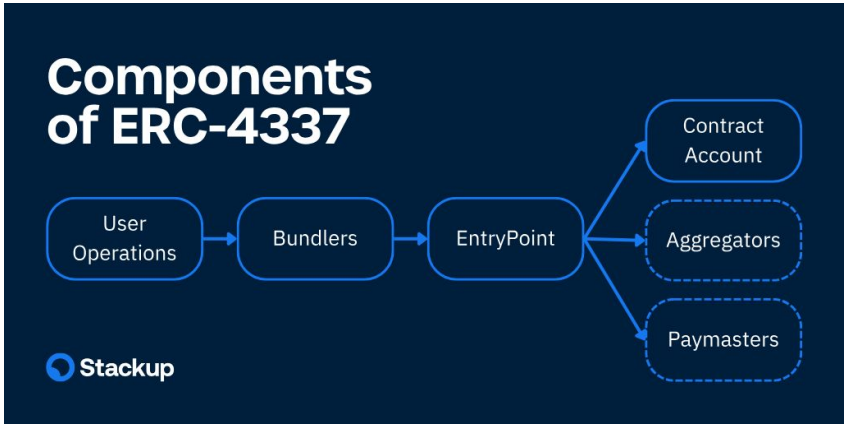
- EOA (Externally Owned Account) is the default wallet for interacting with Ethereum, but it has several limitations, making it **difficult to onboard more users**. For example
  - Losing private keys or getting hacked means losing all assets.
  - External owned accounts lack flexibility in defining functions and logic.
  - Only Ether can be used for transaction fees.
  - Native multi-signature wallets require smart contract implementation.
- Another solution to these issues is the **Smart Contract Account**, which can pre-define logic within a contract
- Challenges for Smart Contract Accounts: **Non-standardized norms** and **lack of anti-censorship** features.

What is Account Abstraction

- **Abstraction** simplifies complex details for easier understanding and problem-solving.
- **Account Abstraction** means abstracting certain consensus layer behaviors for handling at the application layer. For example,
  - Signature verification abstraction
  - Fee charging abstraction
- There are various ways to implement Account Abstraction. For example
  - EIP-2938: Introduce EVM instructions `PAYGAS (0x49)`
  - EIP-3074: A consensus layer proposal and user still need EOA with funds. Introduce EVM instructions `AUTH (0xf6)` and `AUTHCALL(0xf7)`

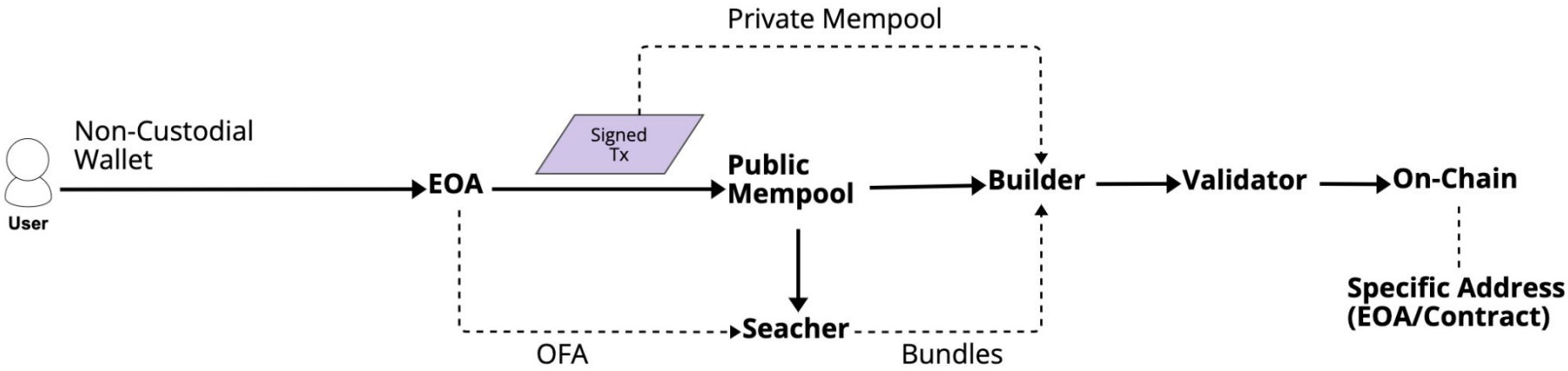
What is ERC4337

- ERC-4337: Account Abstraction Using Alt Mempool, proposed by Vitalik on 2021/9/29
- An account abstraction proposal which completely **avoids consensus-layer protocol changes**, instead relying on higher-layer infrastructure.
- **Account Abstraction** via **Entry Point Contract** specification



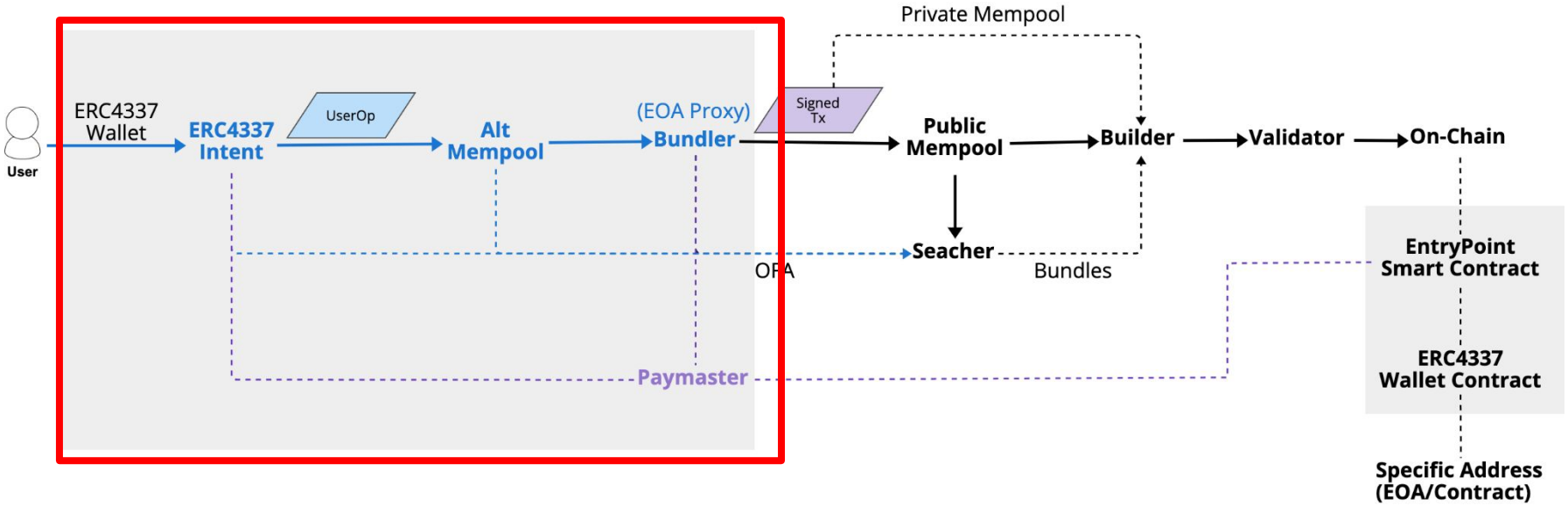
ERC4337 Architecture and Flow

Ethereum Tx Flow



Reference source: https://youtu.be/B6sN8EXszP8

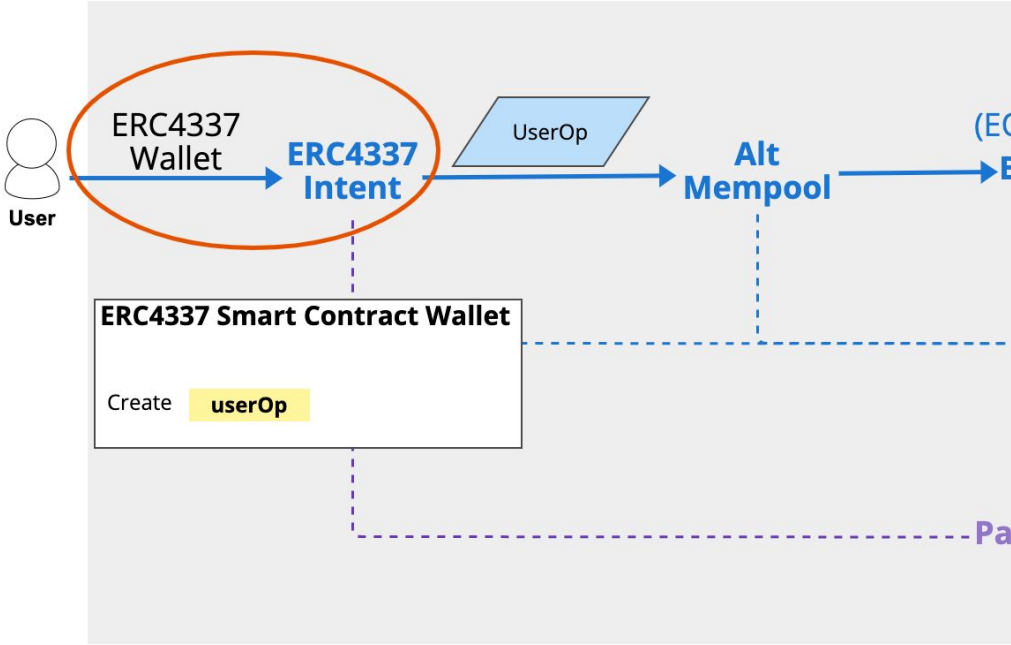
ERC4337 Tx Flow



Reference source: https://youtu.be/B6sN8EXszP8

ERC4337 Tx Flow - Create userOp

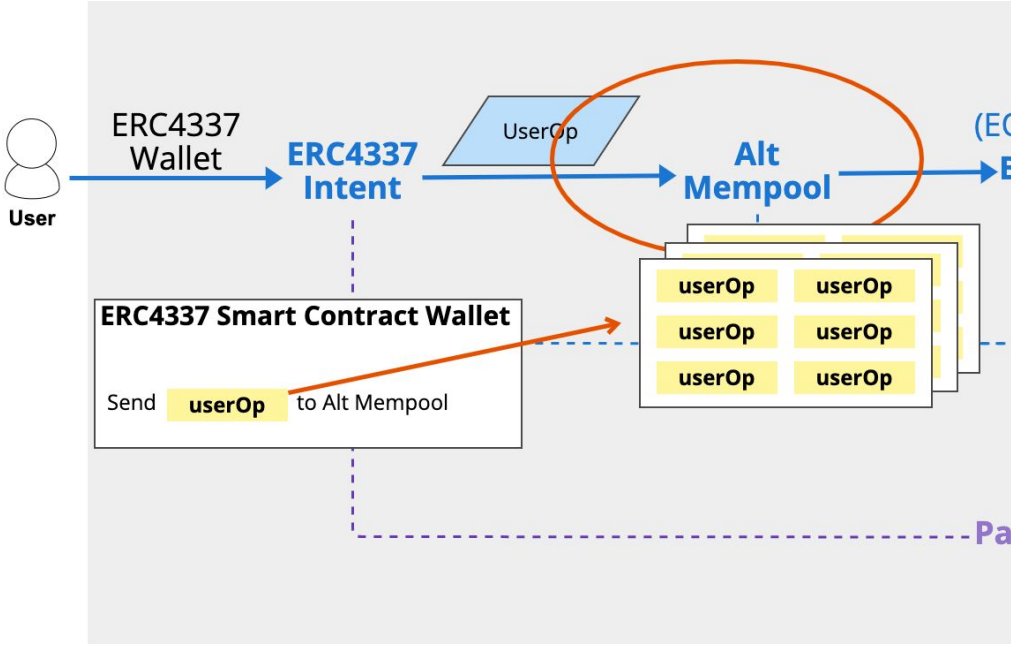
- Transform user-requested transaction data into the userOp data structure.
- userOp comprises interaction address and required calldata, akin to EOA-generated transaction content.
- Besides transaction data, userOp contains extra information for execution verification.



Reference source: https://youtu.be/B6sN8EXszP8

ERC4337 Tx Flow - Alt Mempool

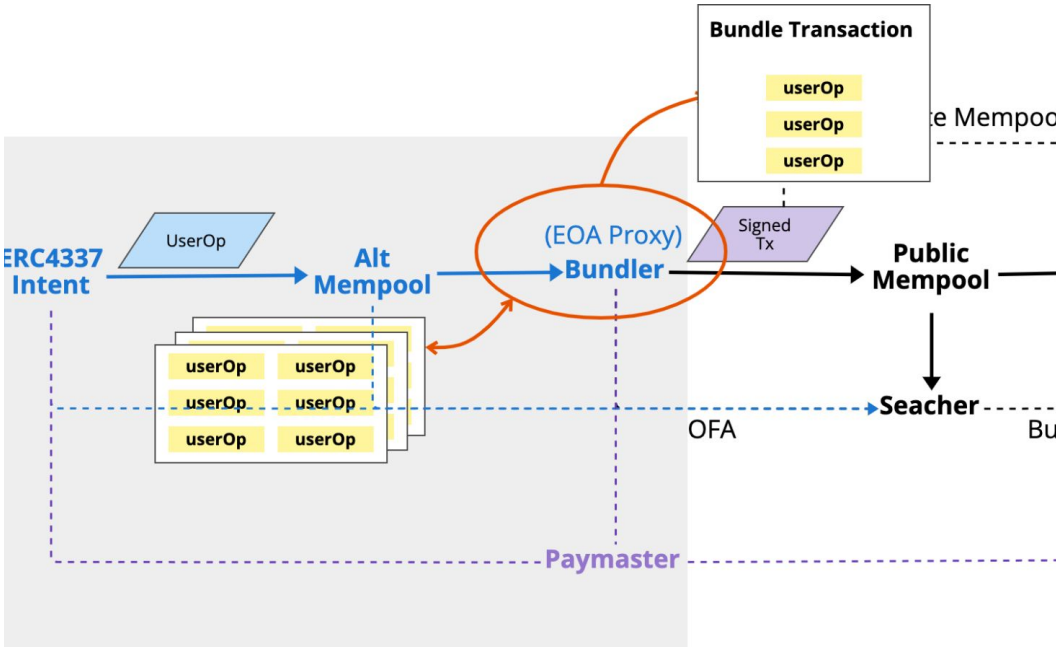
- Instead of utilizing the public mempool, userOps are directed to a specialized "higher-level" mempool designed exclusively for UserOperations.
- This enhancement is currently in the process of being developed for P2P networking.



Reference source: https://youtu.be/B6sN8EXszP8

ERC4337 Tx Flow - Bundler

- Bundlers choose userOps from the alt mempool for their bundle transaction to Ethereum's **EntryPoint** contract.
- Bundlers handle **gas costs** for all UserOperations.
- The inclusion, exclusion, and ordering process leads to **MEV** emergence.

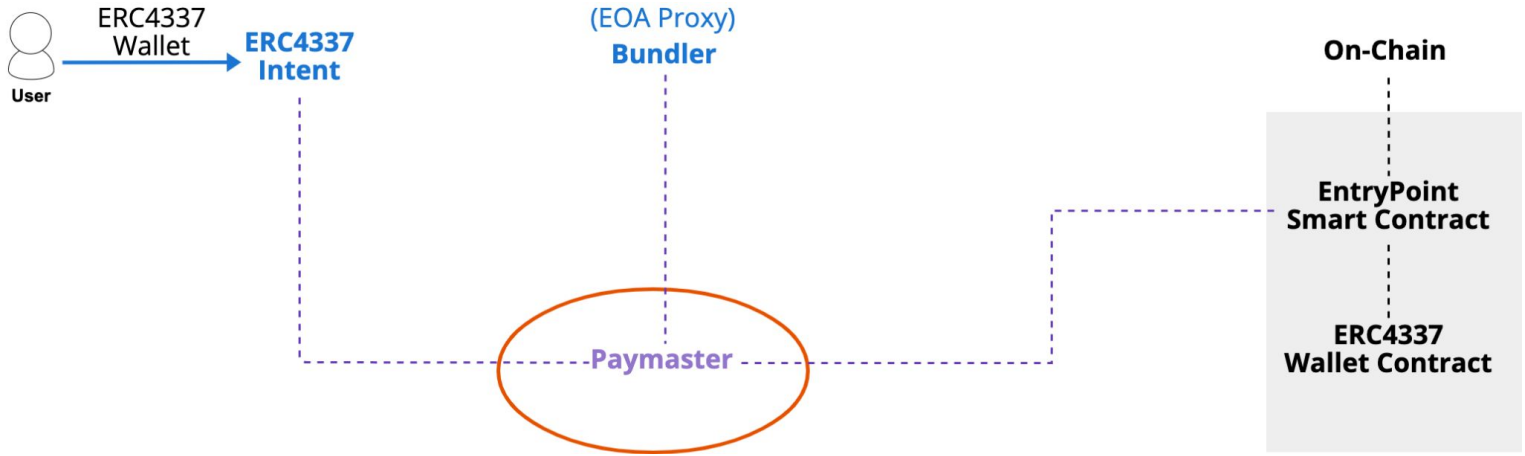


Reference source: https://youtu.be/B6sN8EXszP8

ERC4337 Tx Flow - Paymaster

Reference source: https://youtu.be/B6sN8EXszP8

- An **optional** third-party **contract** account responsible for paying gas fees.
- Enable programmable subsidization of **userOp** gas fees, such as
  - Utilizing ERC20 tokens
  - Applying special conditions like the first 100 intents or
  - Specific NFTs in the wallet, and more.

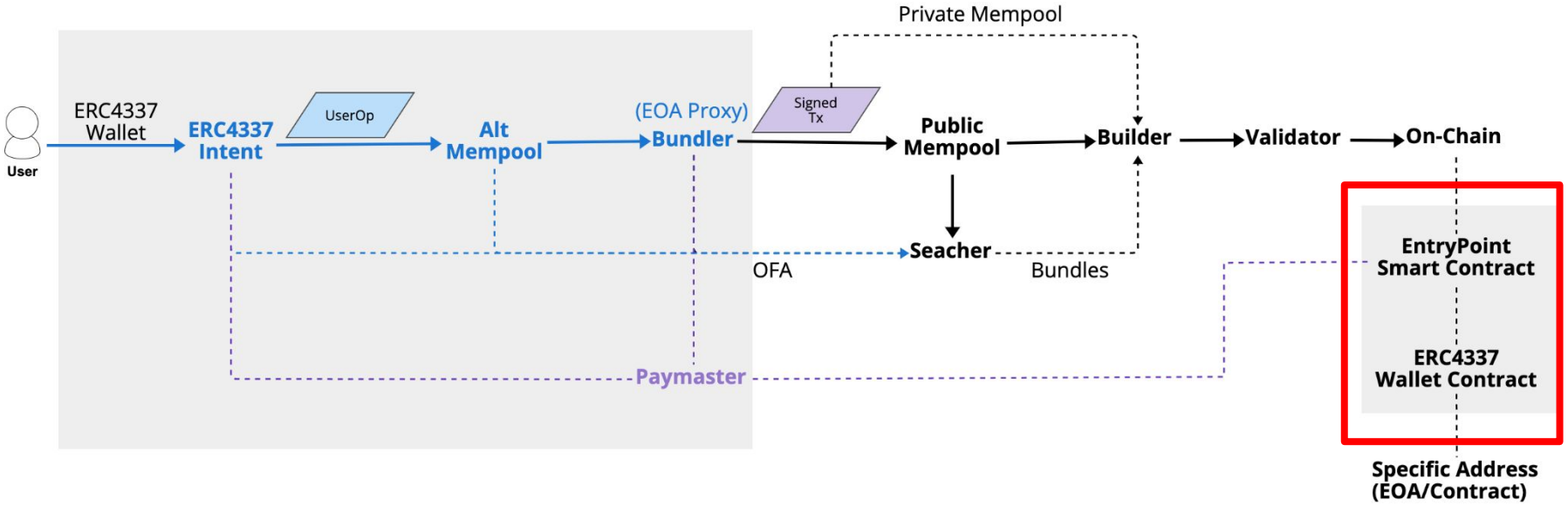


ERC4337 Flow Components Recap

- **UserOp**: A new off-chain transaction format initiated by users, distinct from traditional transactions.
- **Alt Mempool**: A dedicated mempool for accumulating pending userOps to be executed, separate from the transaction mempool.
- **Bundler**: Responsible for bundling user operations and delivering them to the EntryPoint Contract.
- **Paymaster**: An optional third-party contract account responsible for paying gas fees

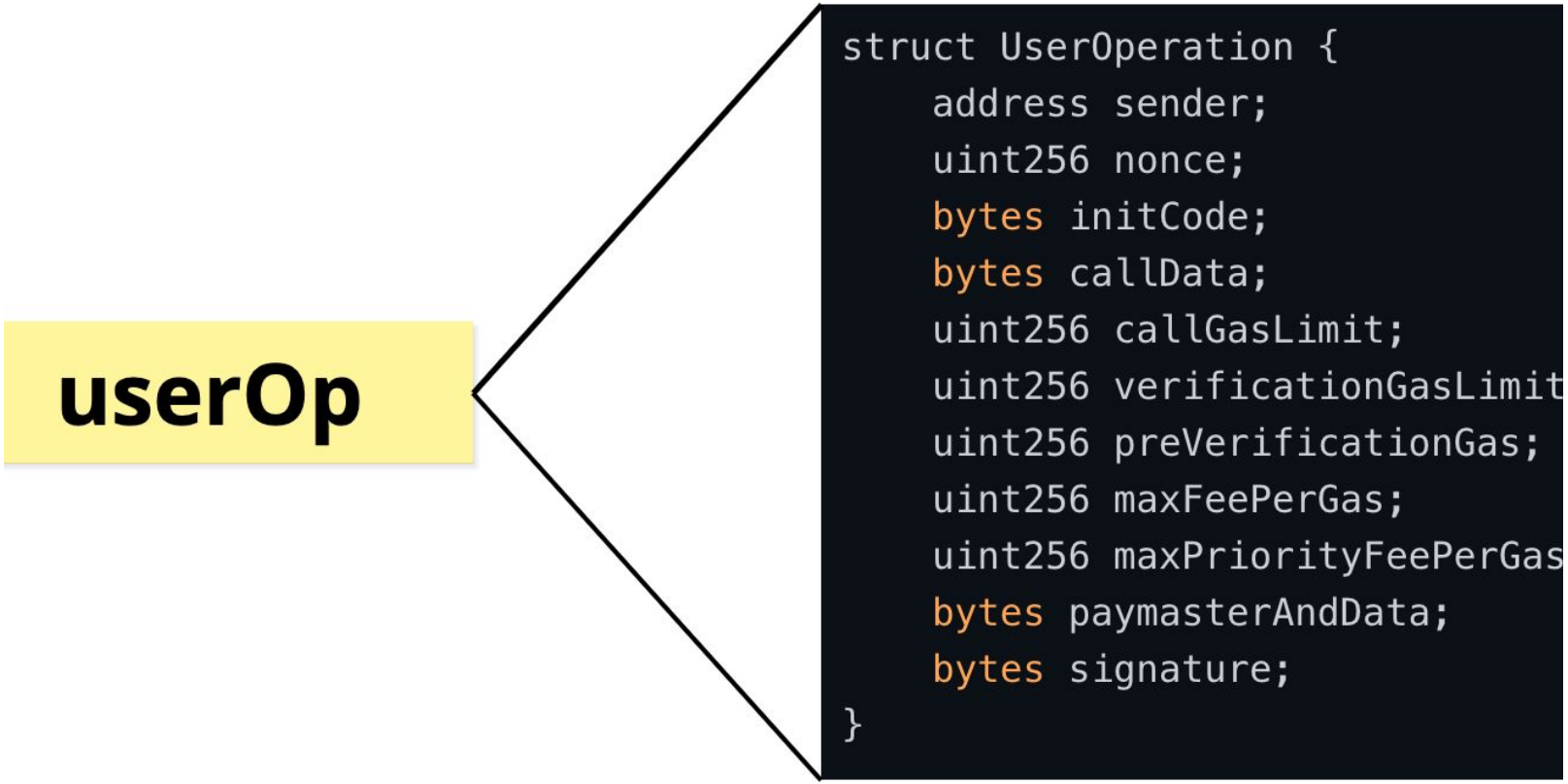
Decode EntryPoint and UserOperation

ERC4337 Tx Flow



Reference source: https://youtu.be/B6sN8EXszP8

UserOperation Data Structure



User Operation Column Explanation

sender	Smart Contract Wallet address
nonce	Nonce value verified in EntryPoint to avoid replay attacks. SCW is not expected to implement this replay prevention mechanism.
initCode	Bytes containing calldata for SCW Factory contract. First 20 bytes are Factory contract address and rest is calldata of function to be called on Factory Contract.
callData	Calldata of function to be executed on SCW. It can be any function on SCW (e.g., execute or executeBatch) which usually then further calls a dApp smart contract. It can even call other methods of SCW internally as well. It's up to you how you implement this method in SCW.
callGasLimit	Gas limit used while calling the SCW method from EntryPoint contract using callData above.

Source:Decoding EntryPoint and UserOperation with ERC-4337 Part 1

User Operation Column Explanation

verificationGasLimit	This value is used for multiple purposes. 1. It is used as gas limit while calling SCW Factory contract, 2. calling verification methods on SCW and Paymaster and 3. calling postOp method on Paymaster. In short, this is gas limit used in calling verification methods on SCW and Paymaster along with postOp method on Paymaster. To be more precise, on top of it, there are other lines in EP whose gas used is accounted in verificationGasLimit.
preVerificationGas	This field is also critical to understand properly. In short, bundler can make profit using this field, if used properly. This is the gas counted on EP as a part of transaction execution which can't be tracked on chain using gasleft() opcode.
maxFeePerGas	This is the max fee per unit of gas that UserOp is willing to pay. It is similar to how maxFeePerGas is defined in EIP-1559 for gas calculation of Ethereum transaction.

Source:Decoding EntryPoint and UserOperation with ERC-4337 Part 1

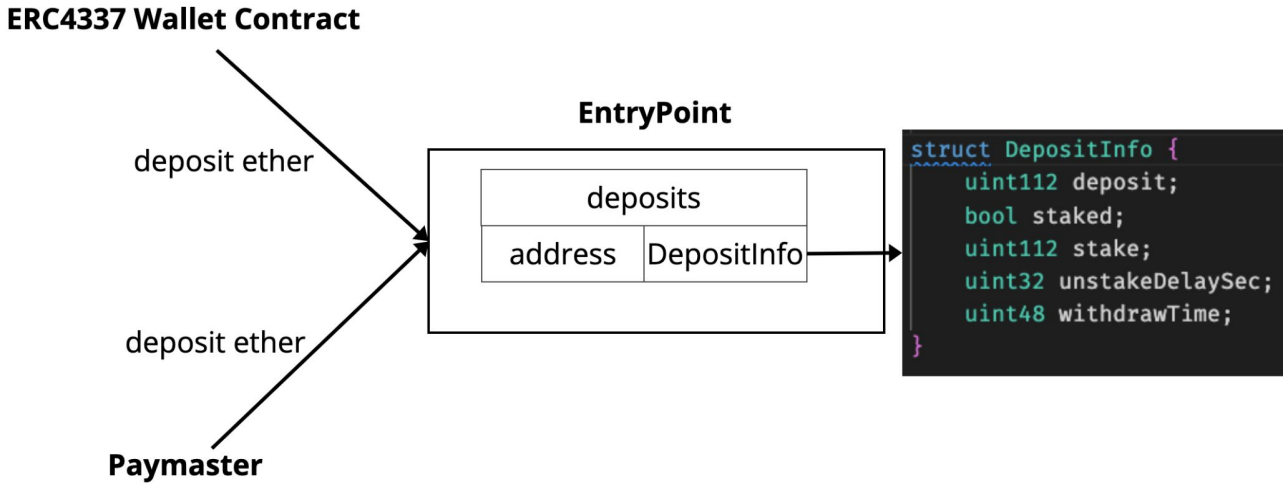
User Operation Column Explanation

maxPriorityFeePerGas	This is max priority fee per gas that UserOp is willing to pay. It is similar to how maxPriorityFeePerGas is defined in EIP-1559 for gas calculation of Ethereum transaction.
paymasterAndData	It contains bytes representing paymaster related information given by UserOp for verification. First 20 bytes is paymaster address and rest represents data to be used by Paymaster for verification. It is empty if paymaster is not used to sponsor the transaction for given UserOp.
signature	It represents the data to be passed to SCW for verification purpose. Usually it's the signature of userOpHash signed by the owner of SCW but it can be utilised in other ways also.

Source:Decoding EntryPoint and UserOperation with ERC-4337 Part 1

Deposit Ether to EntryPoint

- Deposit Ether into EntryPoint before executing userOp as gas fee for the bundler.
- EntryPoint maintains a 'deposits' map to record relevant information related to these deposits.

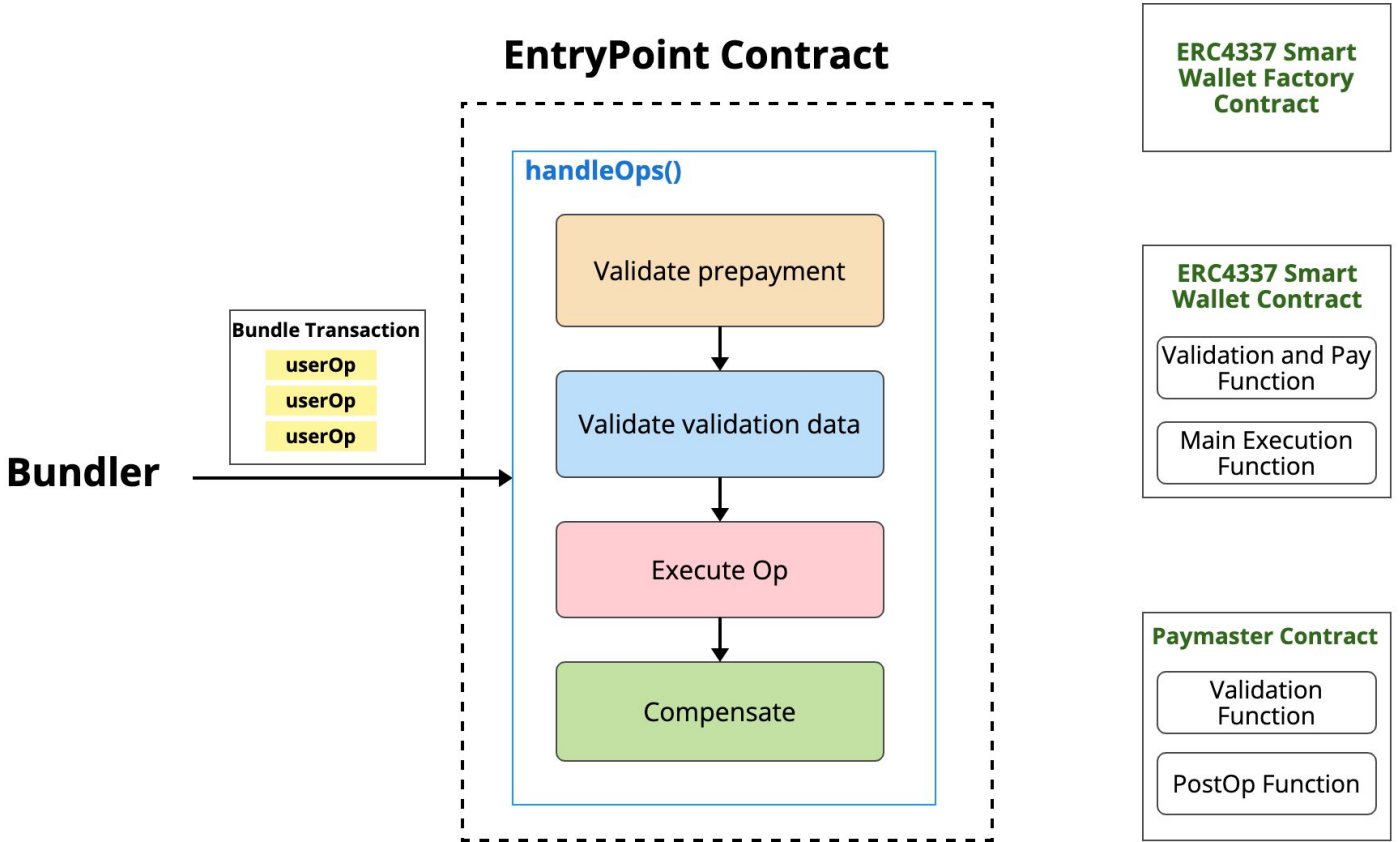


Decode handleOps()

- GitHub: [eth-infinitism/account-abstraction]
- EntryPoint allows two ways to execute userOps:
  - **handleOps()**
  - **handleAggregatedOps()**.
- We'll explain userOps in EntryPoint using **handleOps()** for simplicity.
- handleOps() breaks down into four steps:
  - Validate prepayment
  - Validate validation data
  - Execute Op
  - Compensate

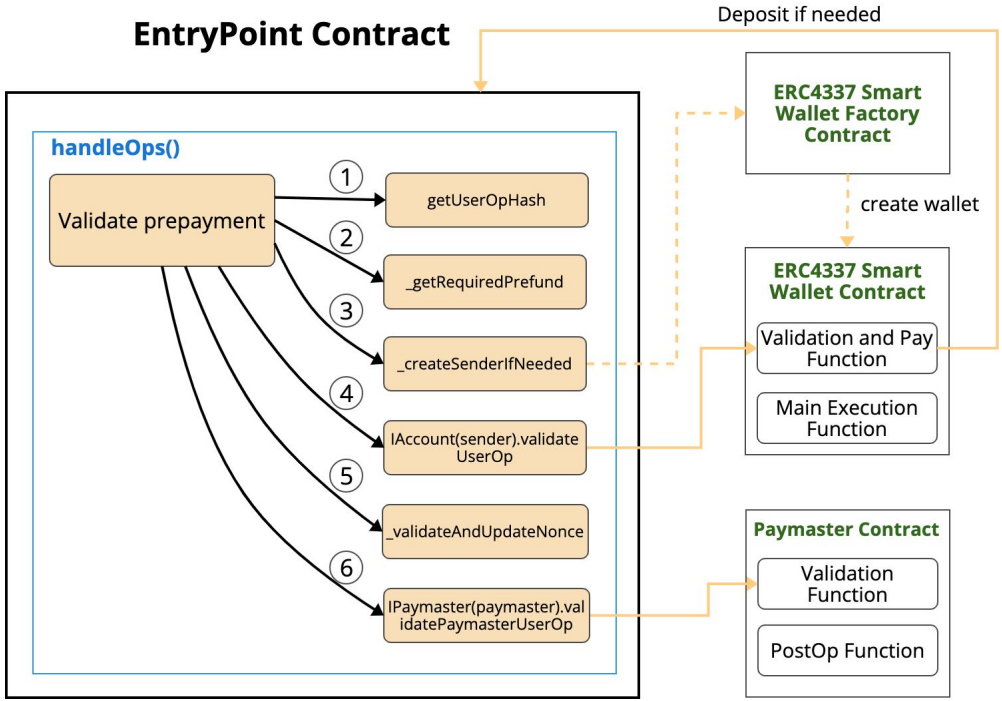
```
/// @inheritdoc IEntryPoint
function handleOps(
    UserOperation[] calldata ops,
    address payable beneficiary
) public nonReentrant {
    uint256 opslen = ops.length;
    UserOpInfo[] memory opInfos = new UserOpInfo[](opslen);

    unchecked {
        for (uint256 i = 0; i < opslen; i++) {
            UserOpInfo memory opInfo = opInfos[i];
            (
                uint256 validationData,
                uint256 pmValidationData
            ) = _validatePrepayment(i, ops[i], opInfo);
            _validateAccountAndPaymasterValidationData(
                i,
                validationData,
                pmValidationData,
                address(0)
            );
        }

        uint256 collected = 0;
        emit BeforeExecution();

        for (uint256 i = 0; i < opslen; i++) {
            collected += _executeUserOp(i, ops[i], opInfos[i]);
        }

        _compensate(beneficiary, collected);
    }
}
```

Decode handleOps()



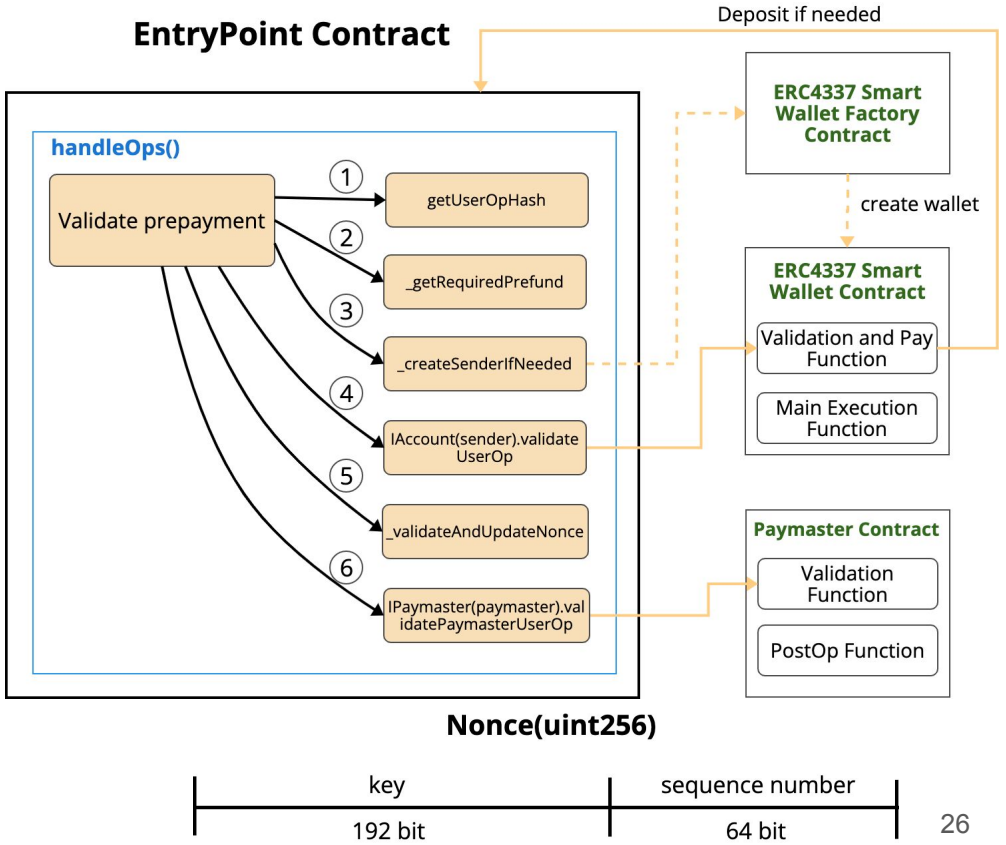
Decode EntryPoint - Validate Prepayment

1. getUserOpHash() contains information about the userOP (**excluding the signature**), the **EntryPoint address**, and the **chainId**.
2. Calculate a maximum gas fee based on specific formulas:
   a. For SWC: callGasLimit + verificationGasLimit + preVerificationGas
   b. For Paymaster: callGasLimit + verificationGasLimit * 3 + preVerificationGas
3. If **initCode** isn't empty, we create a wallet contract using factory address and calldata.



Decode EntryPoint - Validate Prepayment

4. Use **validateUserOp()** in the wallet to authorize the userOp and obtain **validationData**.
   a. Deposit ether into EntryPoint if deposited fund is insufficient
5. Update the wallet's **nonce** in the EntryPoint for security against **replay attacks**
   a. nonce consists of a 192-bit key(left) and a 64-bit sequence number(right).
6. If paymasterAndData contains data, verify paymaster's sufficient funds for the gas fee and request fee coverage using **validatePaymasterUserOp()**. Return **paymasterValidationData**.



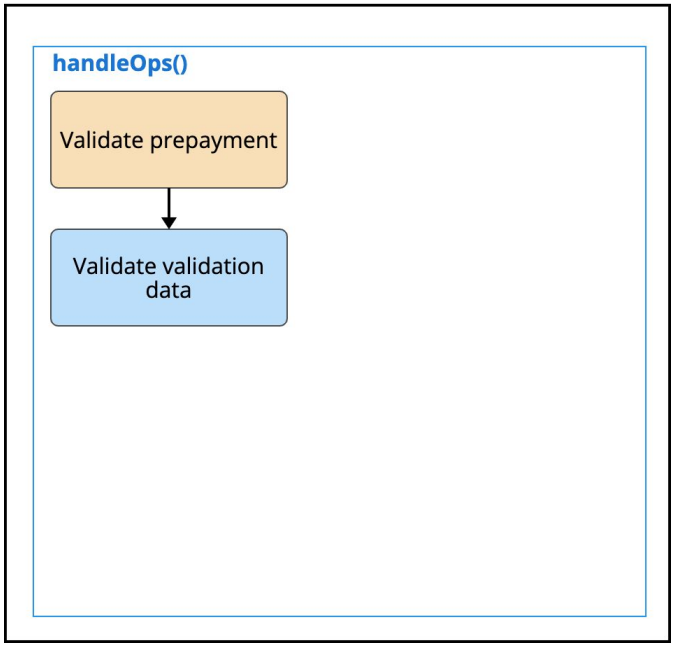
Decode EntryPoint - Validate validation data

- This step checks if there's a timeout in the data returned by the wallet's **validateUserOp()** and the paymaster's **validatePaymasterUserOp()**.
- Both **validationData** and **paymasterValidationData** are of the uint256 data type. The EntryPoint splits these uint256 values into three parts:
  - authorizer
    - 0 for valid signature
    - 1 to mark signature failure.
    - an address of an `authorizer` contract.
  - validUntil
  - validAfter
- The action can only be executed when **validAfter ≤ block.timestamp ≤ validUntil.**

Validation Data Format (uint256)

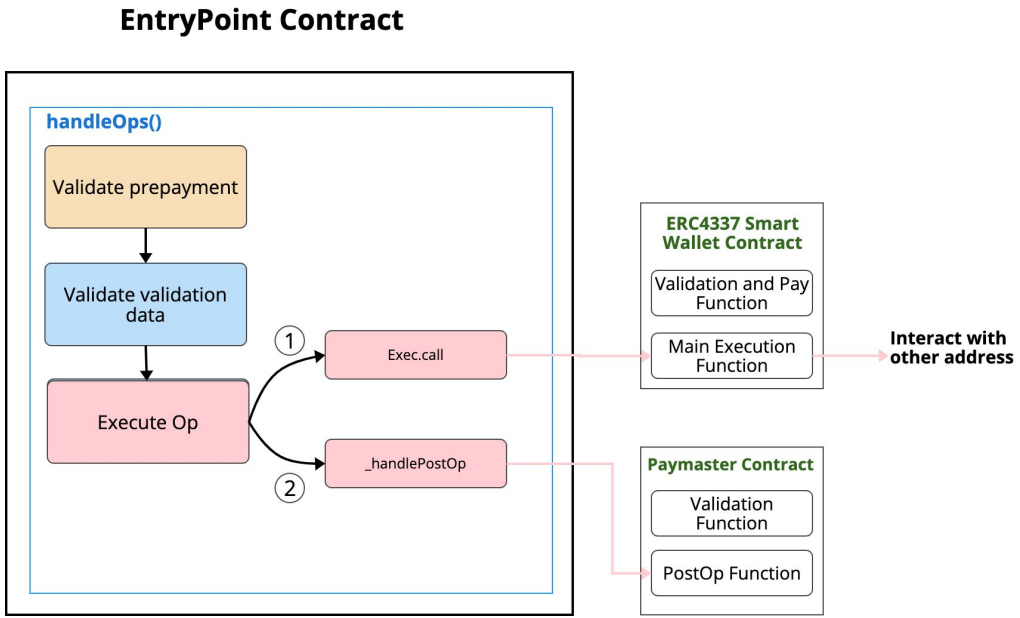
validAfter	validUntil	0 or 1 or aggreator address
6 bytes	6 bytes	20 bytes

EntryPoint Contract



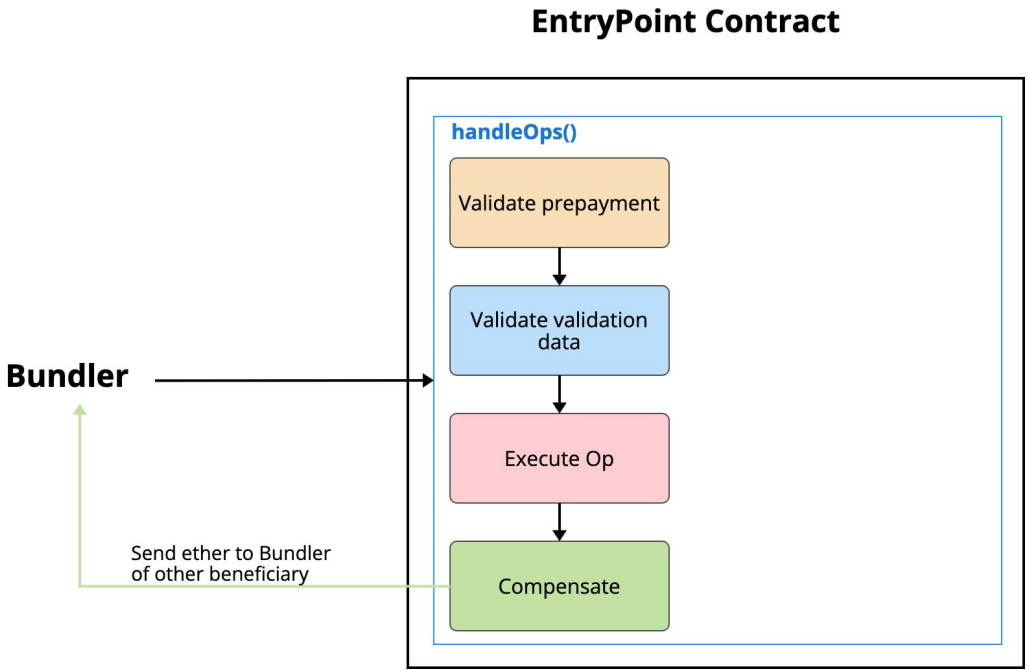
Decode EntryPoint - Execute Op

1. We perform predefined wallet operations (e.g., token swaps) using **Exec.call**.
2. _handlePostOp
   - If there's a paymaster, after the Exec.call, inform the paymaster that the operation is done by calling paymaster.**postOp()**. This notification includes whether the
     - execution succeeded,
     - the actualGasCost, and
     - data specified by the paymaster.
   - Also give back any unused gas to the wallet or paymaster's deposit using **_incrementDeposit.**

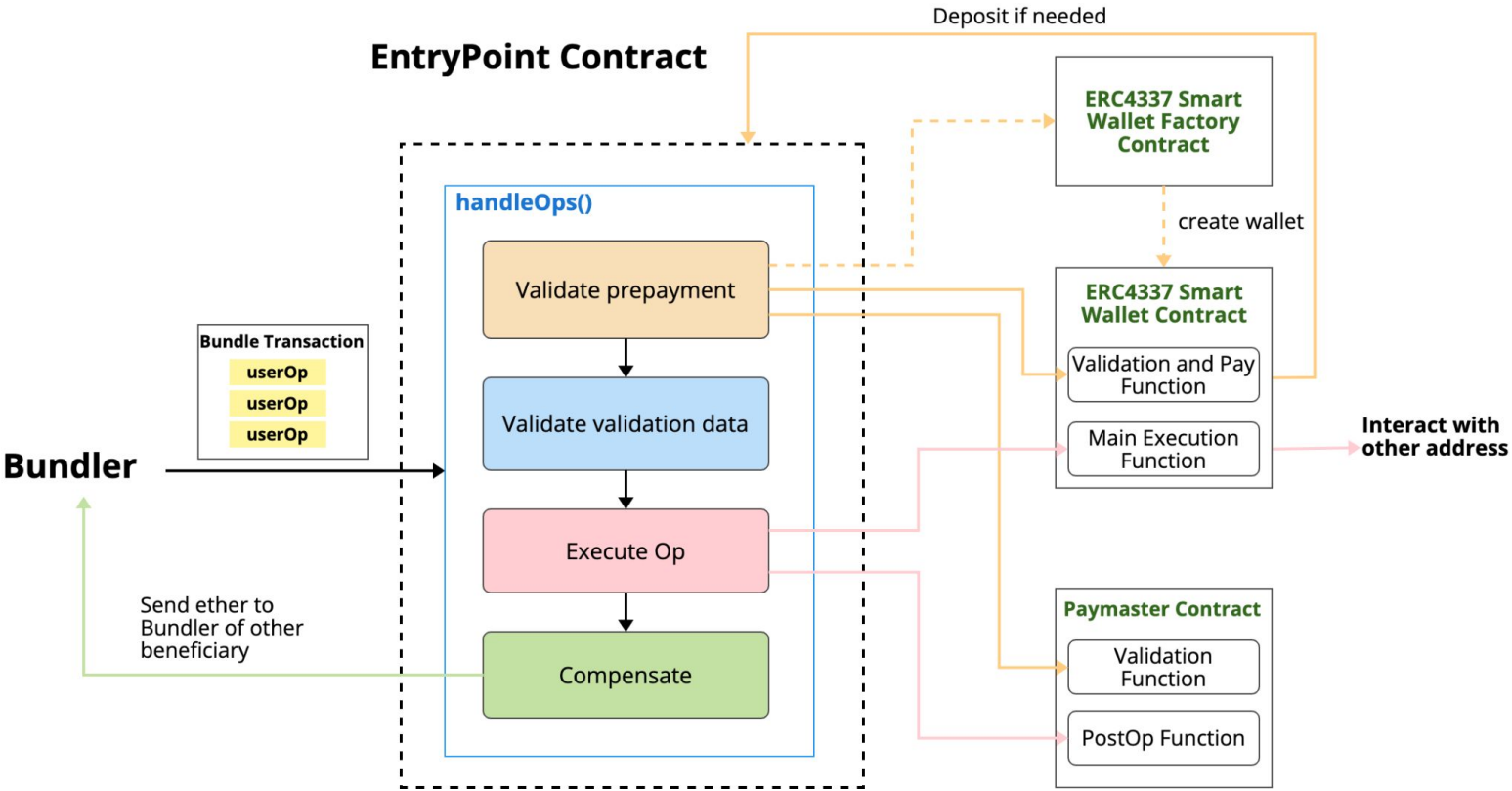


Decode EntryPoint - Compensate

- Send ether to the Bundler as compensation.
- Not only Bundler, but you can also specify any address to receive the ether.



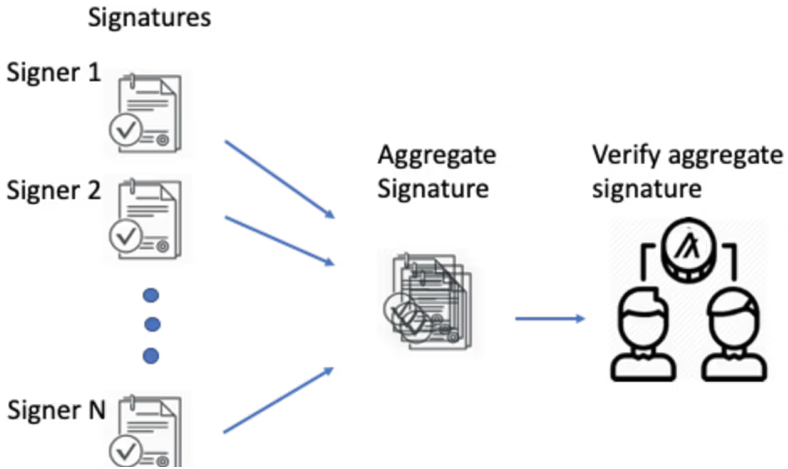
Decode EntryPoint - handleOps



Aggregate Signatures

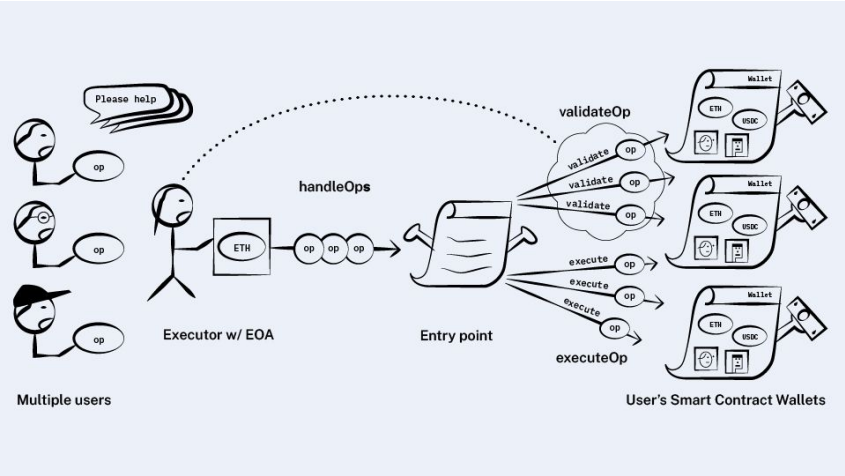
Aggregate Signatures

- Aggregated signatures enhance transaction processing efficiency and scalability.
- Aggregator is a helper contract trusted by accounts to validate an aggregated signature
- handleAggregatedOps() is called to execute userOps involving aggregated signatures.
- The ETH Infinitism team has implemented signature aggregation using BLS (Boneh–Lynn–Shacham) signatures in the example.

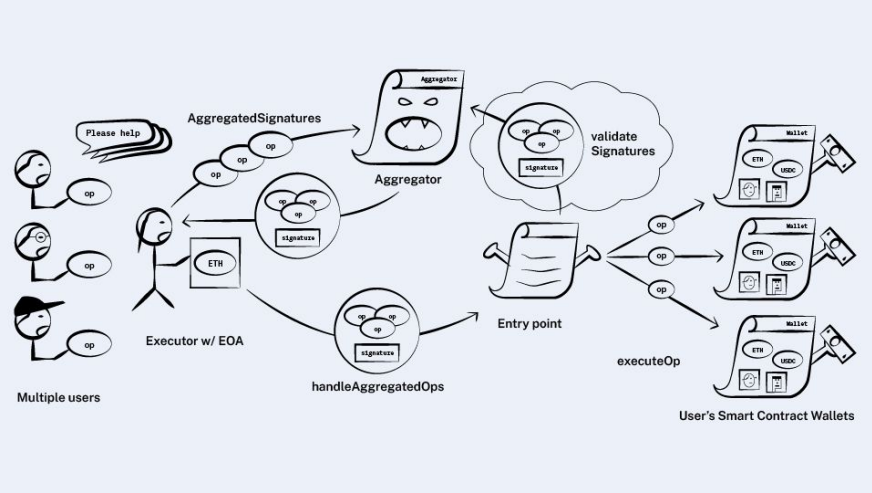


Aggregate Signatures

Wallet Signature



Aggregate Signature



Takeaways

- ERC4337 provides a method for achieving Account Abstraction without requiring modifications to the consensus layer. This is achieved through the utilization of the userOp mempool and EntryPoint mechanisms.
- Users have the ability to substitute Ethereum transactions with userOp to convey their desired actions for execution.
- The userOp mempool also introduces fresh possibilities for MEV
- The Bundler is responsible for the aggregation of user operations and their delivery to the Entry Point Contract.
- The Paymaster serves as an optional third-party contract account tasked with covering gas fees.
- The utilization of Aggregate Signatures enhances transaction processing efficiency and scalability.

Reference

- Account Abstraction is NOT coming
- 從抽象帳戶到 ERC4337
- Account Abstraction 介紹(一) : 以太坊的帳戶現況
- ERC-4337: Exploring the Technical Components of Account Abstraction — Part 2
- Decoding EntryPoint and UserOperation with ERC-4337 Part 1
- Decoding EntryPoint and UserOperations with ERC-4337 Part 2
- Account Abstraction Part 4: Aggregate Signatures
- Account Abstraction Part 2: Sponsoring Transactions Using Paymasters
- Efficient and Secure Digital Signatures for Proof-of-Stake Blockchains
- ERC 4337 | Account Abstraction 中文詳解
- 從抽象帳戶到 ERC4337
- https://eips.ethereum.org/EIPS/eip-4337