The goal of the work is to explore if zk-rollups can be used to aggregate Uniswap trades in an effective manner. The prototype is able to aggregate trades for a single trading pair, Ether and an ERC-20 token of choice. The system consists of two main entites that are required for it to function. The first entity to look at, is the on-chain entity, we call zkSwap. zkSwap is a smart-contract deployed on the Ethereum blockchain and has three main jobs, processing deposits and withdraws, aswell as verifying batched trades. It holds users funds and exposes the on-chain functionality, namely deposits and withdrawls, to the user.

The second entity to look at is the aggregator. The aggregator consists numerous systems, both off-chain and on-chain, and is mainly tasked with receiving trade orders, batching and executing them, and then verifying them with the zkSwap contract. The aggregator stores a merkle-tree of users balances and keeps it in sync by listening for event emitted by the zkSwap contract. To understand how these entities interact with each other, the first thing to understand is how balances are stored and updated.

0.0.1 Storing and Updating Balances

Two main factors are dictating the ways balances are stored in the system. We want to make balance updates as cheap as possible, while not relying on any external data availability. To achive this, we store balances in a merkle tree in layer-2, each balance stored as a seperate leaf. Merkle trees are a suitable data structure, as the root represents the entire tree state in a highly compressed form, while proving a leafs inclusion in the tree can be done with O(log n). This is ideal for our use-case. We can cheaply commit the merkle trees state to our smart-contract by storing the root, while verifying inclusion of a leaf is very efficient. To deposit funds, a user need to provide a valid merkle path to its current balance and the current balance object. The user calls the deposit function, passing the merkle path and balance object as a parameter and adds the funds wishing to be deposited to the transaction.

First we check if the balance is included in the tree. To do this, we first hash the balance object with the sha256 algorithm. The balance object contains the following fields: ethBalance, tokenBalance, nonce and address. When hashing the balance object, we extract the users address from the transaction. This ensures that the user is in control of the addresses private key which suffices as security check. Since the contract stores the merkle root, we can now hash the path and balance, checking if we can recreate the root. If the roots match, the trader is permitted to update its balance in the tree. We now update the balance object by adding the deposited amount to the corresponding field, increment the nonce, hash the updated balance object, and rehash the entire tree with the the new balance object. The resulting merkle root is set in the contract, which commits the updated state.

However, we still need to store the balances somewhere. As we don't want to rely on external data availability, while keeping storage costs as low as possible, we emit the new balance as an event. Writing data to the event log is a lot cheaper, compared to storage as a variable in a smart-contract. While the event log can't be accessed from a smart-contracts runtime, a client can query the event log and pass the data as a parameter. As the merkle trees state is committed in the contract with the merkle root, correctness can always be proven.

Withdrawing funds follows the same logic as deposits. Instead adding funds to the withdraw function, the user passes the amount to withdraw as a parameter. The merkle tree is rehashed, checking if the root is correct, it is ensured the withdrawAmount ¡= balance, the balance and nonce are updated, creating the new root. We emit the new balance and send the funds to the user.

Deposits and withdraws are complete on-chain operations by design. All data needed to withdraw funds are public, except for the private key. This ensures funds can always be withdrawn, as long as the user controlls its private key.

0.0.2 zkSwap Contract

Since we're looking to aggregate Uniswap trades, the obvious functionality required is for users deposit and withdraw funds, which will then in turn be used for trading. Users can deposit or withdraw funds by sending the corresponding transactions to the zkSwap contract, which will then

As mentioned before, the zkSwap contract is the only state-changing entity in this system. All state-changing operations are checked and then committed by this contract. This might be counter intuitive at first, as the goal is to move as much data as possible off-chain. It is important to remember though, that one main goal is not to rely on any external data availibility. This can only be achived by storing state on-chain, as compressed as possible.

At the center of storing state is our system, is the root of the balance merkle tree, stored in the zkSwap contract.

Since we're looking to aggregate Uniswap trades, the obvious functionality required is for users deposit and withdraw funds, which will then in turn be used for trading. As we're looking to move as much data as possible to layer-2.