

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 entities 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 withdraws, 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, aggregating 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.

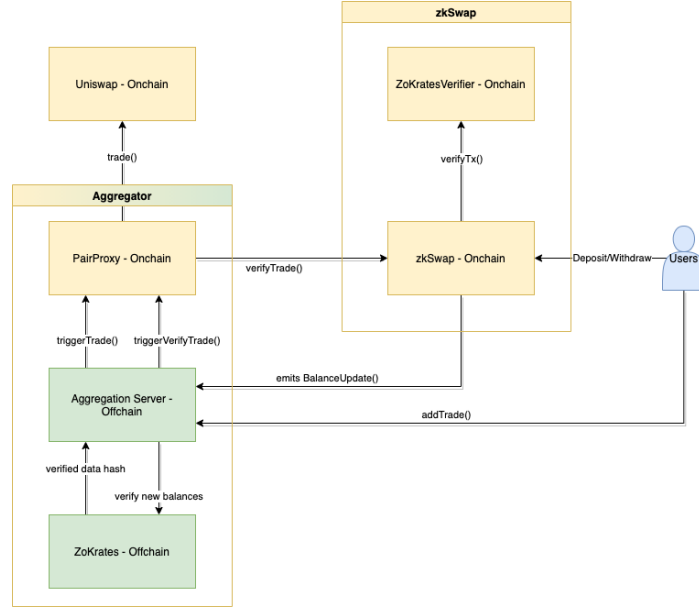


Figure 1: High level architecture of the system

0.1 Conceptual Design (Ignore)

The focus of this work is applying zk-rollup, explained in S. XX, in combination with Uniswap trades. To use the system, a user first needs to deposit funds. This is a on-chain transaction, moving the funds into the smart-contract. Instead of storing a users balance as a mapping the smart-contract, the balances are stored in an merkle tree in layer-2, and the state of that merkle tree is committed in the smart-contract as a merkle root. To update its balance, a user needs to provide a merkle path to its balance. The smart

contract can then prove the validity of the balance by recreating its merkle root. This ensures a user can always withdraw its balance. In combination with zkSNARKs, this also allows us to batch-update balances, significantly reducing the cost per balance update.

This enables the user to withdraw funds at any time via the smaer

The merkle tree is built and kept in sync by subscribing to 'BalanceUpdate' events, emitted by the smart-contract on any balance change. This ensures that all data is on-chain, while reducing the costs of storage significantly. To trade a user can send a trade order to an off-chain entity, which is not trusted in any way, which will aggregate a number of trades and execute them. After the trade execution, the new balances are calculated and the correctness of these balances is ensured by utilizing a zkSNARK proof, which can be verified on-chain. Since the majority of computations has been done off-chain, this results in the transaction costs to be reduced quite significantly. No trust is beinf

Balances can be updated in an on-chain transaction, providing the correct merkle path, which is done when depositing or withdrawing.

0.2 zkSwap Contract

The zkSwap contract, is the core entity the user interacts with. Its a smart-contract, that from the users perspective, is mainly used for depositing and withdrawing funds in out system.

0.2.1 Storing and Updating Balances

Two main factors are dictating the way balances are stored in the system. It is important to understand the core technique used to store and update balances before we look at the different functions that trigger them. We want to make balance updates as cheap as possible, while not relying on any external data availability. Essentially, this means that we need to store the balances on-chain. Storing data on-chain is typically very expensive. It is important to make a distinction between storing data in a smart-contracts runtime and storing data in the event log. Both are on-chain, the event log is significantly cheaper though. While the event log is cheap to use and can be queried by any client, it is not accesible to a smart-contracts runtime. This solves the external data availablity problem. We can store balances cheaply, without relying on other systems to stay online. A client can query the event log, gather the required data and pass it as parameters to the transaction. However, we now need a mechansism to ensure, the parameters passed are equal to the parameters found in the event log.

Merkle Trees We can achive this, be using a merkle tree [?]. Merkle trees are a suitable data structure, as its root represents the entire tree state in a highly compressed form, while proving a leafs inclusion in the tree can be

is runtime the correct word?

done with $O(\log n)$. This is ideal for our use-case. Every balance is stored as a leaf in a merkle tree, running in layer-2. The merkle tree is built and kept in sync by subscribing to the 'BalanceUpdate' event emitted by our smart-contract. A client can query balances from this tree, receiving the valid merkle path along with the balance object. Since the smart-contract stores the root of the balance merkle tree in its runtime, it can verify the correctness of the passed balance by hashing it along with the provided merkle path. If the resulting hash is equal to the stored root, the correctness of the balance is proven. Any changes in the balance object by the client will result in the hashes to mismatch, thereby invalidating the data. While the hashing of the balances and merkle path does create some fixed, overhead cost, the savings by using the event log far outstrip it. The hashing costs will be analysed in S. Results.

Should I explain on what security assumptions this is based

This fulfills the balance storage requirements, reducing costs, not relying on external data availability and ensuring the correctness of data. It must be noted, that this also causes the smart-contract to be the single source of 'truth'. Since all balances changes are committed by a new event being emitted and the root being updated, any balance updates must be done through the smart-contract.

0.2.2 Deposits and Withdraws

When using the system, a user first has to deposit funds. Since the entire idea of zk-rollup is to move funds to layer-2, the deposit function can be seen as a bridge that connects the mainnet and layer-2. When a user makes a deposit, the funds are represented as a balance object in layer-2, which in turn give custody to these funds. When moving funds in layer-2, we don't actually move the funds residing in the smart-contract, but update the balance objects to represent the movement and verify that movement for correctness with a zkSNARK proof. Since a balance object gives a user custody of represented funds, it can always be redeemed, moving from layer-2 back to mainnet.

Maybe remove this?

Merkle Inclusion Proofs As described in S. 0.2.1 balance updates are secured by merkle inclusion proofs in the smart-contract. When depositing the user needs to provide a valid merkle path to its balance object, and the balance object itself. The balance object consists of four fields that are needed to represent the balance: ethAmount, tokenAmount, nonce and userAddress. As a first step, the balance object is hashed with the sha256 algorithm, the result being the leaf in the merkle tree. The leaf is now hashed with the merkle path, according to the standart merkle root hashing algorithm. If the resulting hash equals the stored balance root in the contract, we have proven, that the passed balance object is correct. We now add the value of the transaction object, which is the deposit amount sent with the transaction, to the ethAmount, increment the nonce and hash the new balance. Since the balance object has the same position in the tree, the same merkle path is

also valid for the new balance. The new leaf is hashed with the merkle path, and the resulting hash is set as the new balance root. As a last step we emit the ‘BalanceUpdate’ event, containing the new balance object, matching the balances with the newly set root.

Withdraws largely follow the same logic, there are small differences though. As we’re requesting funds, instead of sending them, we pass a `withdrawAmount` as an additional parameter to the function call. We then check if the `withdrawAmount` \leq `ethAmount` to make sure the withdraw is covered by the users balance. The inclusion proofs are the same, we then subtract the `withdrawAmount` from the balance, and update the balance root. As a last step we emit the ‘BalanceUpdate’ event with the new balance object, and send requested funds to the user as an on-chain transaction.

Authorizing Transactions This however, is an incomplete explanation, as we’re not checking if a user is permitted to deposit or withdraw funds. As balance objects are emitted as an event, anyone can access them and compute valid merkle paths for any balance. This would allow any user to withdraw any balance. To ensure a user is permitted to update a balance object, we need ensure the user controls the private key belonging to the balance objects user address. Fortunately, we can ensure this by accessing the sender in transaction object. The Ethereum blockchain ensures a user is allowed to make a transaction by requiring the transaction to be signed with the private key of the senders address. If that signature is valid, it is proven that the user has access to the addresses private key and the transaction can be executed. Because of this, the transactions object sender can be trusted to be in control of the corresponding private key. Instead of passing the users address as part of the balance object, the smart-contract uses the sender of the transaction object. This suffices as a security check.

It must also be noted, that ERC-20 deposits and withdraws behave a bit differently. As every ERC-20 token has its own smart-contract, representing a users balance as a mapping, we need to transfer the assets in that contract. This requires different functions to handle ERC-20 deposits and withdraws, however implementing this is trivial and not worth explaining in the context of this work.

0.2.3 Batch Verification of Trades

It is recommended to read S. 0.3 first, as it covers the previous steps of a trade aggregation life-cycle. When trading on Uniswap, the user is changing the assets in its wallet. One asset is swapped for another, resulting in the funds moving on-chain according to the trade taken. When trading with zkSwap however, the funds a user has traded are not moved as an on-chain transaction. Instead, the trade is represented by updating a users balance in layer-2, which gives the user access to these funds represented as balance. Instead of relying on merkle inclusion proofs to check the validity of balance

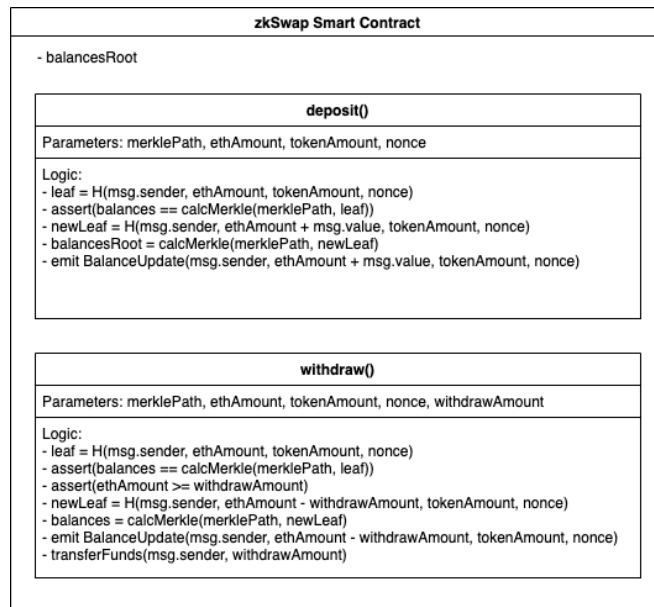


Figure 2: Pseudocode of the deposit and withdraw steps

updates, a correct balance update can also be ensured by utilizing zkSNARK proofs. The aggregated trades can be verified and applied by calling the ‘verifyTrades’ function in the zkSwap smart-contract. A number of checks have to be performed, which we will cover now.

Verifying the ZoKrates Proof As a first check, the ZoKrates proof is verified. The verifier smart-contract is generated along with the the ZoKrates program, and can be used to verify the correct execution of that program using the proof object it generates. The verifier is called, along with the proof object passed as parameter, which returns true if the proof object can be verified. We are now assured, that output of the ZoKrates program has been computed by executing the ZoKrates program, which in turn ensures the balance updates are correct.

Recreating the DataHash and Ensuring Correct Price As explained in S. 0.3.4, the ZoKrates program only returns a hash of the checked data, in order to reduce verification costs of the zkSNARK proof. To ensure the data passed is correct, we need to recreate the ZoKrates programs output, the dataHash, which is part of the proof object and has been verified in the previous check. Recreating the dataHash proves that the data we’ve passed along with the ZoKrates proof is the same data that was used to generate the proof. Just like the merkle root, this hash commits a certain state, which we can verify at a later stage. By using the properties of zkSNARK, we’re able to create this commitment off-chain, saving gas. We also check if the effectivePrice is in bounds of the worst-case price range.

Receiving Fund and Refunding Aggregator While the balances of users are updated in layer-2, funds between the aggregator and zkSwap smart-contract must still flow as an on-chain transaction. Since the aggregator has executed the ‘net trade’ and updated the balances accordingly, these funds need to be exchanged in order for the zkSwap contract to stay solvent¹ and for the aggregator to be refunded for the executed Uniswap trade. Since the net trade has been passed as a parameter and is verified by the dataHash, we check if the funds passed as part of this transaction match the amount of the net trade. If the amounts match, aggregator is refunded the amount spent in the Uniswap trade.

Updating Root and Emitting Balances As a last check, it is ensured that the oldRoot passed as a parameter, matches the current root stored on the zkSwap smart-contract. This ensures a proof can’t be reused, preventing replay attacks. The root is updated in the smart-contract, the worst-case prices are updated by querying Uniswap and the new balances are emitted via the ‘BalanceUpdate’ event, updating the state for all involved users. The lifecycle of a trade aggregation is now complete, and the next batch of trades can be updated.

0.3 Aggregator Entity

As the name implies, the aggregator is tasked aggregating the incoming trades. In order to function, a couple of services are needed, with run as smart-contracts on-chain or on a classical server.

We will now explain each service.

0.3.1 Merkle Tree

The first thing to look at is the merkle tree, the aggregator is running. As previously discussed, all balance updates will be committed by the zkSwap smart-contract by emitting the ‘BalanceUpdate’ event. By subscribing to these events, the merkle tree can be built and kept in sync, always providing the complete merkle tree belonging to the balance root stored in the contract. When a ‘BalanceUpdate’ event is received, the balance object is extracted, the corresponding leaf is found in the tree and then replaced with the new data. Rehashing the tree should now result in the balance root set in the contract. The state of the merkle tree can only change by incoming ‘BalanceUpdate’ events.

0.3.2 Aggregation Server

It is important to remember, that trade orders are completely off-chain and are sent as an HTTP request. The orders are received and processed by

¹The zkSwap contract is solvent if its always able to cover the withdraw of all balances. The zkSwap contract should always be solvent.

explain that leafs
can be updated in
this implementation

the aggregation server, which at a later stage will run the aggregation. A number of checks are performed on the aggregation server when a trade is received. These checks are technically not needed to ensure the correctness of the aggregation, as the ZoKrates program performs the same checks at a later stage. They are however needed, to prevent the server from processing invalid trades, which would cause the ZoKrates program to exit in an error state, preventing the entire aggregation. A trade is invalid, if it fails any of the checks described in this section.

Ensuring Correct Pricing The price between assets is constantly changing. At the same time, trades are being collected for aggregation. This results in a delay between an user sending a trade order and the actual trade execution, during which the price can change. On top of that, network congestion on the Ethereum blockchain can cause further delays in the execution. A mechanism is needed to define a 'worst-case' price, that is defined before users add orders to the aggregation. Once the aggregation is complete, a user can be sure to having paid no worse then the 'worst-case' price.

Another thing to consider is the bid-ask spread that exists in a trading pair. A spread is the difference between the current bid and ask price for an asset, where the bid always has to be a lower price. Intuitively, this makes sense, the spread should at least equal the cost of converting from one asset to the other. There a number of other factors that influence the bid-ask spread for a Uniswap trading pair. In this context however, it is sufficient to know that a spread is expected in any trading pair. This complicates the mechanism to ensure a worst-case price.

Buy and sell orders are off-set internally once the aggregation starts, which results in the 'net trade'. The net trade can be a buy or a sell order, depending on the orders part of the aggregation batch, and it is not possible to predict the direction before the aggregation has started. Since we have a bid-ask spread, and we can not predict which direction our net trade will be, we need to define a price range that at least equals the current bid-ask spread. This would suffice to ensure a worst-case price for a net trade in either direction if executed immediatly. As the aggregation is also adding a delay between defining the price range and executing the trade, the price range should be larger then the spread. For this reason the zkSwap smart-contract defines a minSell and maxBuy price, defining that range. If the price on Uniswap has moved out of that range, while the aggregation was in progress, the trade won't be executed and the aggregation canceled. This can be formalized in the following way:

$$\forall A_o \in A : x_s \leq x_e \leq x_b$$

where:

A is the current aggregation batch

A_o a trade order

x_e is the effective price

x_s is the minSell price

x_b is the maxBuy price

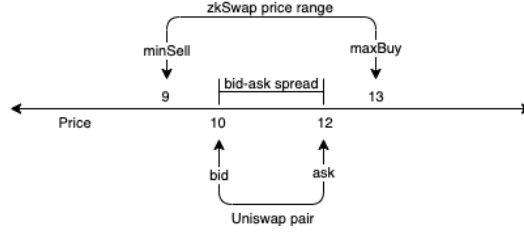


Figure 3: Bid-ask spread and zkSwap price range

While this method ensures a 'worst-case' price for a trade, at the same time a maximum price is defined with it. Since we're matching buy and sell orders in one aggregation, there is no way around this. Once the aggregation is completed and verified on-chain, the new minSell and maxBuy prices are set, based on queried Uniswap prices, which are valid for the next aggregation batch.

Authorizing an Order Since this operation is running off-chain, we first need to verify the user is authorized to make the trade order. This can be achieved by requesting a signature from the user, verifying it is in control of the address's private key. However, it must be remembered, that this signature must also be verifiable in our ZoKrates program, which is unable to utilize the secp256k1 curve, used for signing Ethereum transactions, efficiently [?]. For that reason the BN128 curve is used in combination with the EdDSA signature scheme, which can be more efficiently run in a ZoKrates program. The user signs the trade order and current balance root, ensuring two things. It proves that the user has access to the address's private key, which authorizes the trade order. By signing the balance root, we make sure, that the signature can't be reused in a replay attack. For instance, the aggregator could decide to store these signatures secretly, and reuse them without the user's consent if this was omitted.

When the trade order is received, the aggregator first checks if the signature is valid and contains the current balance root, by querying it from the zkSync smart contract. As a next step, it is ensured, the user's balance is able to cover the trade. The aggregator queries the merkle tree for the user's balance, and checks if the trade can be covered by deposited funds. A last thing to consider is ensuring the correct price of a trade. The aggregator checks if the implied price of the trade matches the 'worst-case' price stored

in the zkSwap smart-contract. If all of these checks pass, the order is added to the trade pool, where it resides until the aggregation starts.

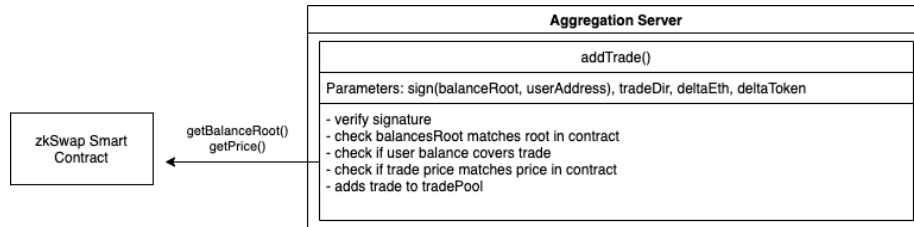


Figure 4: Pseudocode of addTrade()

Running the Aggregation At some point the trade aggregation is started. This could be triggered by a set blocknumber, the number of trade orders that have been received or any other useful condition defined by the aggregator. When aggregation is started, the first step is to calculate the ‘net trade’. Since our system aggregates buy and sell orders, we can first offset those internally. By doing this, we’re able to reduce the entire aggregation to one Uniswap trade, which saves gas. At the same time, we’re saving on the 0.3% liquidity provider fee, which is charged based on a trades volume. The net trade is the result of off-setting all trades in aggregation, which results one side to equal zero. This trade is now sent as an on-chain transaction to the ‘PairProxy’ contract. The PairProxy contract is explained in detail in S. 0.3.3.

The aggregator waits for the PairProxy smart-contract to emit the ‘Trade-Complete’ event, containing the amount of assets acquired in the Uniswap trade. The amount must at least imply the worth-case price, defined by the zkSwap smart-contract. In most situations, the implied price (effective price from here) will be better than the worst-case price. Based on the effective price, the users post-trade balances are calculated. Calculating the balances, poses a problem which arises by aggregating buy and sell orders in the same batch, which will be discussed in the limitations/open problems section.

When the new balances have been created, the aggregator triggers the witness generation of the ZoKrates program. To execute the merkle inclusion proofs in the ZoKrates program the multi-leaf merkle path is generated and passed as a parameter, along with the old and new balances and the effective prices, paid in the trade step. This is described in detail in S. 0.3.4. Once the witness has been generated, the proof generation is started, which results in the proof objects needed for the on-chain verification of the entire aggregation. To verify everything on-chain, and thereby updating balances of all all balances involved in the aggregation, the proof object is passed, along with the new balances, the old merkle root, the new merkle root, the effective net trade and the effective price and sent to the PairProxy smart-contract. At this point, the net-trade represents the funds that need to be exchanged by the PairProxy and the zkSwap smart-contract, to reimburse the aggregator

for the funds spent in the Uniswap trade.

0.3.3 PairProxy Smart Contract

Before explaining the functionalities of this smart-contract, it is important to understand why it is required for the system to function. There are two reasons, a quirk in the way Ethereum handles return values, and the result of dealing with changing price data. When performing a trade on Uniswap, a user is asked to define a slippage² for the trade. Since network congestion and the current gas price influence when a transaction is executed, it's a necessary mechanism for ensuring users can set a 'worst-case' price. For this reason, when sending a transaction to the Uniswap trade function, the `minAmountReceived` parameter must be passed, which we provide by using our 'worst-case' price, explained in a previous section. When calling the trade function, the actual amount received is returned as the function's return value. Since this amount might be larger than the amount passed as `minAmountReceived`, we need it to calculate the post-trade balances³.

However, a quirk in Ethereum's way of handling return values makes this more difficult. A smart-contract's function return value can only be accessed, when called by another smart-contract function. If calling a function as a normal transaction, as the aggregator does, instead of receiving the return value of the function, we receive the transaction receipt, which doesn't contain the return value. For this reason, we need the PairProxy smart-contract, which receives transactions, forwards them to the respective smart-contract, emitting the return value as an event, which can be consumed by the transactor.

The PairProxy smart-contract is used for forwarding transactions to the Uniswap or the zkSwap contracts. After the aggregator has calculated the 'net trade', it calls the trade function in the PairProxy contract, passing the calculated trade parameters. The PairProxy contract now calls Uniswap's trade function, receiving funds and the amount as a return value. As it has access to the return value, it emits the 'TradeComplete' event, containing the amount received in the trade. As it would be inefficient to send the funds back to the aggregator, they reside in the smart-contract. Since the aggregator is set as the owner of the contract, the funds are stored securely.

When verifying the aggregated trades in the zkSwap smart-contract, the transactions are forwarded by the PairProxy again. Since the funds previously traded still reside in the smart-contract, they are attached to the transaction when forwarded to the zkSwap smart-contract.

0.3.4 ZoKrates Program

The tasks and checks performed by the aggregator ensure that the trade aggregation is done correctly. It was verified a every trade order is authorized

²Slippage is the difference of the expected and executed price of a trade

³The trade also throw an error, when the `minAmountReceived` amount can't be fulfilled. In this case the aggregator cancels the aggregation

by the user, a users balance can cover the trade and that each order contained the correct worst-case price stored in the zkSwap smart-contract. However, this would require us to trust the aggregator, which is not the goal of this implementation and zk-rollups in general. For this reason, we rely on the properties of zkSNARK to make the correctness of the aggregations verifiable on the blockchain in a compressed form. It is important to note, that the aggregator has computed all values needed for the trade aggregation. The ZoKrates program is only used to verify the correctness of the computed values. For efficiency we pass those computed values to the ZoKrates program, which will check them for correctness, defined in the program.

Then general properties will be described in background i guess

Merkle Multi-leaf Inclusion Proofs In order to ensure the correctness of balance updates, we first need to verify the inclusion of the balances involved in the merkle tree. Doing this one by one is simple. Every leaf provides its merkle path which can be hashed with the leaf. If the merkle root matches the current merkle root, we can be assured the provided balance leaf is correct. At the same time, this enables us to reuse the merkle path for updating the balance leaf. We can simply change the balance leafs values after passing the inclusion proof, rehash with the merkle path, and the result is the correct root for the updated balance leaf.

When dealing with multiple balance leafs, the inclusion proof can be done the same way. Every balance leaf provides its merkle path, the resulting hash should be the same for each leaf. Things become more difficult when updating the balance leafs. Updating the first leaf in the batch now invalidates the merkle path of all following leafs. This could be solved by sorting the leafs before hand, and generating each merkle path based on the changes of the previous leaf. At the same time, this results in the merkle path to be invalidated when proving the inclusion, so a separate path for would be needed, which in turn undermines the validity of the proof. Another solution is needed.

A multi-leaf inclusion proof ensures that the same merkle path can be used for checking inclusion and updating the entire tree. Instead of generating a separate merkle path for each leaf, we can construct the path in a way, that can be used with any number of leafs. We can now verify the inclusion of a balance and update these balances by hashing the tree twice in total, using the same merkle path for every leaf. This solves all problems described above. Since we now have a list of leafs, we need a way to decide if the next hash is computed with the next element of the merkle path or the next balance. For this reason we introduce proof flags. The proof flags are a boolean list, containing an element for each hashing operation needed to recreate the tree, ensuring we hash the correct values with each other.

The first check performed in the ZoKrates program, is the multi-leaf inclusion proof, shown in F. 5. We pass the old balances, merkle path and proof flags, receiving the computed merkle root as a result. If the merkle root equals the root that has been passed, we can be assured these balances are

```

function calcMerkle(leafs, proofs, proofFlag){
  const leafsLen = leafs.length;
  const totalHashes = proofFlag.length;
  let hashes = [];
  let leafPos = 0;
  let hashPos = 0;
  let proofPos = 0;
  for(let i = 0; i < totalHashes; i++){
    let a = proofFlag[i] ? (leafPos < leafsLen ? leafs[leafPos++] : hashes[hashPos++]) : proofs[proofPos++]
    let b = leafPos < leafsLen ? leafs[leafPos++] : hashes[hashPos++]
    hashes[i] = hashPair(a, b)
  }
  return hashes[hashes.length - 1]
}

function hashPair(a, b){
  return a < b ? solidityPairHash(a, b) : solidityPairHash(b, a);
}

```

Figure 5: Multi-leaf Inclusion verification. TODO: Transform to algo notation

correct⁴.

Checking Balance Updates and Signatures As the old balances of users have now been verified, the next step is to check if the state transistions, resulting in the new balances, are correct. The first thing to be check is the trade order and the corresponding signature. Depending on the trades direction, the amount paid for the trade is checked with the new balance. This ensures, the trade size has not been changed by the aggregator. If the signature can be verified, we are assured the order is authorized and the size correct. The price implied by the balance transistions also needs to be checked, making sure it matches the effective price reported by the aggregator⁵. It is also checked if the nonce is incremented correctly. While checking balances, the net trade is calculated in the ZoKrates program, which is needed to check the on-chain flow of funds between the PairProxy and zkSwap smart-contracts at a later stage.

Computing new Merkle Root As the correctness of balances and the state transitions have been proven, the next step is to compute the new merkle root. As mentioned above, we can do this by reusing the merkle path and proof flags, but passing the new balances as leafs this time. The resulting hash is the new merkle root, that will be stored in the zkSwap contract when the aggregation is verified on-chain.

Reducing On-chain Verification Costs We have now successfully verified the new balances, and we could use these values to generate the proof, which will then be used to verify everything on-chain. When verifying the ZoKrates program on-chain, each output of the program is part of the proof object, adding an iteration to the proving logic. The amount of outputs the

⁴The zkSwap contract ensures the root that was passed to the ZoKrates programm corresponds to the one stored in the contract

⁵The correctness of this is checked in th zkSwap contract

ZoKrates program has, influences the verifications costs. We can reduce this cost by returning a hash of the resulting data, thereby reducing the amount of outputs. Since the aggregator computed the balances in the first place, and the ZoKrates program only verified it, it can pass that data as part of the verify transaction, but excluded from the ZoKrates proof object. By hashing the data in the zkSwap smart contract, we can ensure that data correctness by comparing it to the hash that is part of the proof object. As a result, the ZoKrates program only returns this hash as an output value.

Explain assumptions that can be made from proof in background, hashing as algo blaaa

ZoKrates Program
Parameters: oldBalances, newBalances, merklePath, proofFlags, root, priceEth, priceToken
<ul style="list-style-type: none"> - check oldBalances by hashing tree and comparing root - check if newBalances imply correct price - calculate effective net trade (PairProxy <-> zkSwap) - compute new root by hashing tree with newBalances - compute dataHash to commit verified state - return dataHash

Figure 6: ZoKrates program checks

0.3.5 Client Frontend

The frontend allows the user to interact with the system, calling the functions, while providing necessary data is the background. The frontend also listens for ‘BalanceUpdate’ events and keeps the merkle tree in sync locally. This allows the client to access the merkle tree in order to provide the merkle path for a transaction for example, without relying on an external system to provide this data. In its current design, the frontend could be hosted as a static file in IPFS [?], not relying on any server to facilitate withdraws of user funds, which closely follows Ethereum’s unstoppable applications ethos. Running this with a large merkle tree becomes unfeasible quickly though, so a hybrid approach can be envisioned. A server is used to provide the user with requested data from the merkle tree. If the server is offline or provides incorrect data, the client can sync merkle tree itself. While this would put computational strain on the client, it ensures that a user is always able to withdraw funds, no matter what entities are offline or have turned malicious.

Deposits and Withdraws In order to deposit and withdraw funds, the user needs to specify a number of parameters, that are needed to execute the transaction. First of all the merkle path is needed, which the client can generate by querying its local instance of the merkle tree. The balance object can also be provided by the merkle tree. A form in the frontend is used to define the amounts wishing to be deposited/withdrawn, which will gather the necessary parameters and add them to a Ethereum transaction. Metamask or any other browser compatible wallet will open, summarizing the transaction, which a user can now sign, bringing it on chain. Once the ‘BalanceUpdate’ event is fired, the frontend will update the balance in its merkle tree.

Adding a Trade When adding a trade a user defines the direction of the trade and the amount wishing to be traded. The trade form will display a minimal amount received, which is calculated based on the worst-case price for either trade direction. Since this is an off-chain transaction, when sending the order a user is asked to sign the order and the current balance root with its addresses private key, explained in detail in S. 0.30.3.4. The order is now sent to the aggregator as an HTTP request.

Displaying Account Data The frontend also displays basic account data, which makes usage of the system easier. This includes the accounts current balances, as well as the address. Redux is used in the background to keep the data in sync.

0.4 Limitiations of Current Implementation

The final implementation of this work does not include all attributes mentioned in the previous sections.

Signatures As described in S. 0.3.2, the authorization of an order entirely relies on a user signing the trade order and merkle root. This signature needs to be checked in a ZoKrates program, which limits us to use the EdDSA scheme, in combination with the BN128 curve. However Metamask, a browser compatible Ethereum wallet, does not support this. Since the private key is stored in Metamask, there are no theoretical limitiations in adding the curve in signature scheme, it simply hasn't been done yet. The outlook section will also look at alternative solutions. For this reason, the signature checks are omitted in the prototype, their computational cost is included in the results section though.

Updating Balances According to Effective Price After the aggregator has executed the net trade on Uniswap, the new balances are calculated based on the worst-case price instead of the effective price the aggregator has paid. Currently, there is no mechanism in place that can ensures this. The aggregator could always claim the worst possible price has been paid (depending on net trade direction), while keeping the difference for itself. This will be addressed in the open problems section.

Arbitrary Combinations of Merkle Tree Leafs In its current form, each aggregation must contain three trades, and the involved balances must be neighbors in the merkle tree. When creating the Merkle multi-leaf inclusion proofs, the positions of the leafs in the tree influences the number of hashes in the merkle path. Each hash value in the merkle path needs to be hashed once as part of the inclusion proof, so a longer merkle path results in a higher number of hashes being computed. Two leafs that are a pair in the tree,

BLS aggregate signatures

contain the same number of hashes in the merkle path as a single leaf does⁶. If the leafs are at the opposite half of the tree, the number of hashes in the merkle path almost doubles. This makes implementation in ZoKrates more complicated, as it can not deal with arbitrary length data. This can be solved by compiling a number of different versions of the program, all accepting different merkle path lengths, which will be covered in the limitations section.

⁶Actually, the merkle path is one element smaller when generated for two pair leafs, as the neighboring leafs hash has moved from the merkle path, to the leafs list.