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, 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.

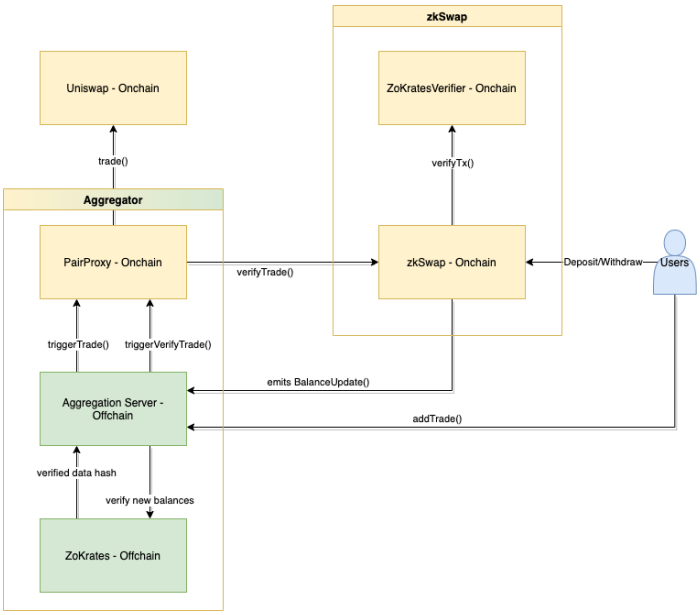

Figure 1: High level architecture of the system

0.1 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.

0.1.1 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 runtimeand 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.

is runtime the correct word?

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 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 availabilty 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.

0.1.2 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

Maybe remove this?

funds, it can always be redeemed, moving from layer-2 back to mainnet.

As described in S. 0.1.1 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 ¡= 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.

This however, is an incomplete explaination, 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 controlls the private key belonging to the balance objects user address. Fortunetly we can ensure this by accessing the sender in transaction object. The Ethereum blockchain ensures a user is allowed to make a transaction by requiering 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 controll 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.

Figure 2: Pseudocode of the deposit and withdraw steps

0.1.3 Batch Verification of Trades

When trading on Uniswap, every user send a seperate trade transaction, and each trade results in an on-chain movement of funds. This uses quite a lot of gas. zkSwap aims to change this, by aggregating a number of trades into one, executing it and sending the traded funds to th zkSwap contract. The balances of all involved users are then updated accordinly, giving custody to these funds. Since the aggregation happens off-chain, we need to verify its correctness by utilizing a zkSNARK proof. The content of the zkSNARK program, and performed checks and computations will be described in S. XX, we will now focus on the verification step on-chain.

The zkSwap contract has access to the ZoKrates verifier, which is used for checking if a submitted proof is valid. A valid proof implies, that inputs of the proof have been computed by using the zkSNARK program used for generating the verifier, and with the public inputs defined.

At its core, a trade on Uniswap is only a update to a users balance. Certain assets are subtracted, other added to the account. The main difference to zkSwap is that traded assets are moves as an on-chain transaction, which uses a large amount of gas. At the same time, every users wishing to trade will do so in a seperate transaction.

When trading on Uniswap, a user exchanges funds, swapping one for the other. This happens as an on-chain transacation and uses quite a lot of gas.

When trading with zkSwap, the balance of a user changes representing the trade in layer-2. However, we want to

At the same time, the funds residing in the smart contract must be able to cover the withdrawl of all balances at all times.

a users funds is changed in layer-2, representing the funds swapped. A user always has the option to withdraw these, but is free to leave them in layer-2.

0.2 Aggregator Entity

As the name implies, the aggregtor 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.

0.2.1 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.

explain that leafs can be updated in this implementation

0.2.2 Aggregation Server

It is important to remember, that trade orders are completly off-chain and are sent as an HTTP request. The orders are received and processed by the aggregation server, which at a later stage will run the aggregation. A number of checks are performed on the aggragtion server when a trade is received. These checks are techniqually not needed to ensure the correctness of the aggregation, as the ZoKrates programm 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 aggrgation. A trade is invalid, if it fails any of the checks described in this section.

Since this operation is running off-chain, we first need to verify the user is authorized to make the trade order. This can be achived by requesting a signature from the user, verifying it is in controll of the addresses private key. However, it must be remembered, that this signature must also be verifiable in our ZoKrates programm, which is unable to utilize the secp256k1 curve, used for signing Ethereum transaction, 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 addresses private key, which authorizes the trade order. By signing the balance root, we make sure, that the signature can't by reused in a replay attack. For instance, the aggregator could decide to store these signatures secretly, and reuse them without the users consent if this was omitted.

When the trade order is received, the aggregator first checks if the signature is valid and contains the current balances root, by querying it from the zkSync smart contract. As a next step, it is ensured, the users balance is able to cover the trade. The aggregator queries the merkle tree for the users 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 price between two assets is constantly changing. At the same time, we're collecting trades in order to aggregate them. This results in a delay between an user sending a trade order and the actual trade execution, during which the price can change drastically. For this reason the zkSwap smart-contact stores a 'worst-case' price for buy and sell orders, which is used to calculate the trade order. The worst-case price is used to invalidate an order at a later stage, if the price of an asset pair has crossed it. At this stage, it is ensured that the trade orders price 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.

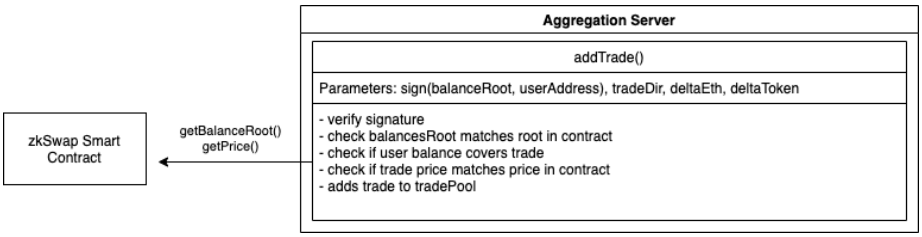

Figure 3: Pseudocode of addTrade()

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 aggragation 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. 0.2.3.

The aggregator waits for the PairProxy smart-contract to emit the 'Trade-

Complete' event, containing the amount of assets aquired 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 then 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 an parameter, along with the old and new balances and the effective prices, paid in the trade step. This is described in detail in S. 0.2.4. 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.

0.2.3 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 slippage[1] for the trade. Since network congestion and the current gas price influence when a transaction is executed, it's a neccecary mechansism 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 functions return value. Since this amount might be larger then the amount passed as minAmountReceived, we need it to calculate the post-trade balances[2].

However, a quirk in Ethereums way of handling return values makes this more difficult. A smart-contracts functions 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 recipe, which doesn't contain the

[1] Slippage is the difference of the expected and executed price of a trade

[2] The trade also throw an error, when the minAmoutReceived amount can't be fullfilled. In this case the aggregator cancels the aggregation

return value. For this reason, we need the PairProxy smart-contract, which receives transanactions, 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 Uniswaps 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 securly.

When verifying the aggregated trades in the zkSwap smart-contract, the transactions is 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.

0.2.4 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 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 aggragator, 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 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 involed 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 seperate 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 used for checking inclusion and updating the entire tree. Instead of generating a seperate 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.

```
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 4: Multi-leaf Inclusion verification. TODO: Transform to algo notation

The first check performed in the ZoKrates program, is the multi-leaf inclusion proof, shown in F. 4. 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 correct[3].

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

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

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 aggregator[4]. 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 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.

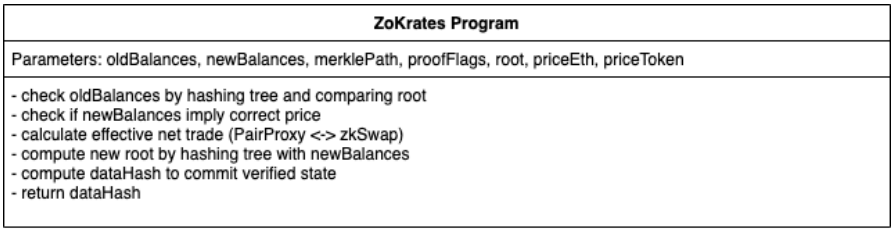

Figure 5: ZoKrates program checks

[4]The correctness of this is checked in th zkSwap contract