zkSwap - Scaling Decentralized Exchanges through Transaction Aggregation

Paul Etscheit

March 2021

Contents

1 Introduction

When being launched in 2015, Ethereum [22] set out to change the way we compute. A trustless, permissionless, and decentralized world computer was envisioned, set to open a new class of applications. The importance of running verifiable, Turing-complete code in a permissionless and trustless manner

cannot be overstated and enables products and services not thought to be possible. However, the technical limitations have also become apparent quickly. Computations are expensive, theoretical transactions per second are low, and the overall throughput has been stagnant. While Eth2 gives a path towards scaling the network, it is expected to take years to complete.

The first major use case for Ethereum was tokenization. With the development of the ERC-20 standard, launching a token on the Ethereum blockchain was trivial. As tokens run as smart-contracts on the Ethereum blockchain, they are secured by its proof of work consensus, which, given Ethereums PoW hash rate, makes consensus attacks infeasible. Running on Ethereum blockchain is a significant benefit when looking to tokenize things, as network security can be assumed. While tokenizations are a step in the right direction, they do not come close to the initial vision. While the standardization enables simple integrations into exchanges and wallets, most tokens are isolated in their functionality and ecosystem and lack productive usage.

token all isolated, not working together... Not a lot of gas needed blabla

With all of these developments over the past couple of years, it seems we have now entered a new phase of smart-contract use-cases, namely Decentralized Finance (DeFi). While DeFi has many different products and functionalities, at its core aims to utilize tokenized assets in some productive form.

Lending and collateralized borrowing is possible with Aave [18], assets can be deposited into liquidity pools[7] to generate yields, flash-loans [18][7] enabled uncollateralized borrowing as long as the loan is repaid in the same transaction and assets can be traded in a non-custodial way with Uniswap [7] . It can be questioned how useful or necessary these protocols really are, but the core idea behind them is impressive. Rebuilding traditional financial products, running as non-custodial and permission-less smart-contracts, all based on the same standardizations, has the potential to reshape the way finance works. With these developments not looking to slow down, they are quickly overwhelming the Ethereum blockchain, pushing transaction costs [14] higher and higher.

Mention other swap protocols?

One of these new DeFi applications is Uniswap [7]. Uniswap is a crypto-asset exchange running as a collection of smart-contracts on the Ethereum blockchain, enabling non-custodial, trust-less, and permission-less trading of ERC-20 assets. Since its running on the Ethereum Blockchain, reducing the computational complexity of trade execution is essential for making it a viable product. In typical crypto-asset exchanges, trading is built around a central order book. Users can add buy or sell orders for a given trading pair, and a matching engine checks if these orders can be matched, executing the trade once they do. While running this on modern server infrastructure is feasible, running it on the blockchain is not. The demand for memory and processing power is too large, so a different approach must be taken. Uniswap solves this by applying the automated market maker (AMM) model, which will be explained in detail in sec. XX. By applying this model, Uniswap reduces the computational complexity to make this a viable business model. At least it

was, when Uniswap launched.

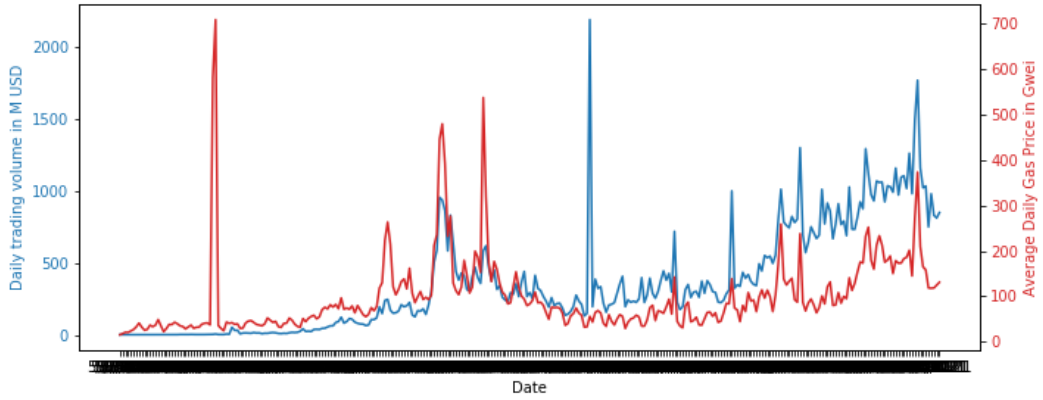


Figure 1: Combined daily Uniswap trading volume in million USD and average daily Ethereum gas price

The recent rise of Ethereums gas price [14] can also be attributed to the growing popularity of Uniswap. Currently, it is one of the most used smart-contract on the Ethereum blockchain, making up on average around 15% of gas [12] usage of a block at the time of writing. To date, it accrued over $280 million in transaction fees [15] and has settled over $100 billion in trading volume [2]. With the gas price having reached 500 gwei on a couple of occasions, a single Uniswap trade can cost upwards of $130. While it would be assumed that high gas prices cause a reduction in trading volume, the opposite is the case. As shown in F.1 there seems to be a strong correlation between daily trading volume on Uniswap and the average daily Ethereum gas price, so reducing gas consumption by Uniswap transactions should result in a reduced gas price for the entire network.

With longer-term scaling solutions in development but still years away, a shorter-term solution is needed. A couple of short-term scaling approaches have been proposed. While these do differ, they all aim to move transactional data to a layer-2[1] system, while ensuring correctness of that data in some way. One of the approaches is called zk-rollup, the focus of this work. Moving data to a layer-2 system can increase the number of transactions that fit into a block while also reducing transaction costs for the user, with is beneficial for Uniswap users and all other participants of the Ethereum network.

Current zk-rollup enabled applications running on the Ethereum mainnet, one of them being ZK Sync, are focused on reducing cost of Ether and ERC-20 transfers. Users deposit funds into its smart-contract, which results in the user's deposit being represented as balance in layer-2. When a user makes a transfer to another user, the involved balances get updated in layer-2, while the correctness of these updates is ensured via zkSNARK. It is important to

not a great sentence

[1] A layer-2 system is a data storage that does not reside on the blockchain but has its state committed to it in some way

note, that ZK Sync acts as a closed system, transfers only change balances in layer-2, while deposited funds in the smart-contract do not move. While this approach has significantly reduced costs of transfers, it only marks the first generation of potential(?) zk-rollup enabled apps.

Aggregating Uniswap trades is an interesting application to explore the potential of zk-rollup technology. It combines the layer-2 storing and updating of balances already done by ZK Sync while opening the system to interact with other smart-contracts. When aggregating trades, we need to interact with the Uniswap contracts to execute the aggregated trade, then update the layer-2 balances according to the trade and verify everything via zkSNARK. It is the next step in exploring the potential of zk-rollups as a generalizable scaling solution, applicable to any kind of smart-contract.

2 Background

Before diving into the technical details and the goals of this work, we will cover a number of topics that are important to understand in the context of this work.

2.1 Decentralized Exchanges

The first thing to understand is the recent rise of decentralized exchanges. Instead of relying on server infrastructure, as traditional crypto asset exchanges do, decentralized exchanges execute the trade logic in a decentralized and trustless manner. The most popular decentralized exchange is Uniswap [7], which runs as a collection of smart-contract on the Ethereum blockchain [22]. There are a number of

A common security consideration to make in cryptographic protocols are replay attacks. A replay attack occures when a valid cryptografic proof is maliciously resubmitted, and no security checks are in place to invalidate the resubmission. When making a transaction in a blockchain network, the transaction is signed with the users private key. Miners that receive transactions check the signatures validity, which authorizes the transaction as the user has access to the addresses private key. However, a malicious actor could store these signed transactions and resubmit them to the network. Since the signature is still valid, a mechanism is needed to invalidate transaction signatures, once they have been included in a block and thereby executed. In most blockchain systems, like Bitcoin or Ethereum, this is solved by tracking a nonce for each address. Miners extract an addresses current nonce by indexing the entire blockchain, simply counting the number of confirmed transactions. Among other things, the nonce is part of the signed transaction. When a miner checks the signature, it is also checked if the nonce set in the signed transaction has been incremented by one, when comparing with the nonce that was extracted. This invalidates a resubmission of the signed transaction and prevents replay attacks.

2.7.1 Transaction Replay Attacks in zkSwap

The way transactions are executed in this system is quite different, compared to the blockchain systems described above. As we will explorer, most verifications and computations happen off-chain and are then verified on-chain with a single proof, that

When making a transaction, which can be a trade, a deposit or a withdraw, a user signs the transaction details and sends them to our off-chain entity, the aggregator. The signature is then checked in a zkSNARK circuit, along with the nonce. The resulting proof can then be used to verify the correct execution of the zkSNARK circuit, as an effect verifying the correct. While the signature check happens off-chain, the mechanism of preventing replay attacks is comparable to most blockchain systems. By checking if the nonce signed in the transaction details has been incremented, we ensure the signed transaction is invalidated when resubmitted.

2.7.2 Proof Verification Replay Attacks

Another consideration to make, is the submission and verification of the zk-SNARK proof. It also could be resubmitted, thereby breaking the protocol if no measures are in place to prevent this. When verifying a zkSNARK proof, we're essentially proving the execution of the circuit was done correctly, and that the resulting outputs where computed by running the circuit. In this system, the zkSNARK circuit is used to ensure correct state transitions tbc

3 Approach

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. First, we'll look at the systems design to understand how the different entities work together.

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, verifying deposit and withdraw batches, processing instant withdraws and verifying trade batches. It holds users funds and exposes the on-chain functionality, to the user and emits balance updates.

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 deposits, withdraws or 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.

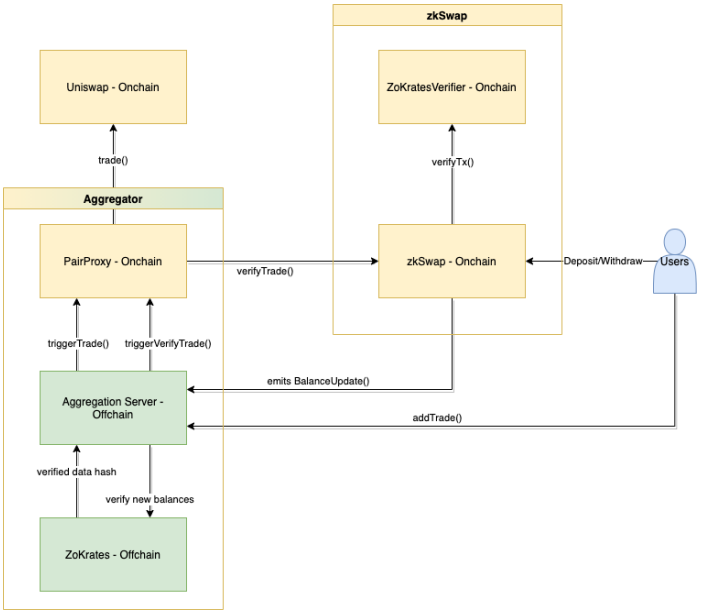


Figure 2: High level architecture of the system

3.1 Design

In this section we will explore the design of the system, looking at the different functionalities and dependencies to understand the core mechanics of how the different entities interact with each other.

The two main entities of this system are the zkSwap smart contract and the aggragator.

3.1.1 zkSwap Smart-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 and keeping track of user balances.

Deposits To use the system a user first has to deposit funds into the smart-contract. The funds need to be sent as a normal on-chain transaction to the smart-contract, where they will be stored, while the balance is then represented in layer-2. To deposit funds, a user calls the deposit function in the zkSwap smart-contract and adds the funds to be deposited to the transaction. When the funds have been received, an event is emited informing the aggregator about the new deposit. At the same time, the user signs the deposit data, and sends it to the aggregator as a message. The aggregator aggregates a number of deposits, verifies the correctness in the zkSwap smart-contract, at which point they will show up as balance for the user.

Withdraws When withdrawing funds, the user needs to decide between an aggregated withdraw, similar to the deposit, or an instant on-chain withdraw. The aggregated withdraw works just like the deposit, only difference being the a withdraw amount is specified instead of sending funds. After the aggregation is complete, the user will receive the funds as an on-chain transaction. A user can also withdraw funds by using the instant withdraw feature. While it costs significantly more gas to withdraw funds with this function, it can be used without the aggregator being online. The protects the user from not being able to withdraw its funds in case the aggregator is offline or has turned malicious.

Verification of Aggregated Trades The last main functionality of the zkSwap contract is the verification of aggregated trades. Once the aggregator has completed the aggregation batch, it send the new balances along with a zkSNARK proof and the traded funds to the zkSwap contract. If the proof is valid and the correct amount of funds have been sent to the zkSwap contract, the new balances are emitted, finanlizing the state. The aggregator is now refunded, receiving the funds it spent when executing the aggregated trade on Uniswap.

3.1.2 Aggregator

As the name implies, the aggregators job is aggregating deposits, withdraws and trades. It facilitates the aggregation of these operations and is built in a way that ensures correct execution, while no trust assumptions are made. The aggregator relies on a number of different systems to function, in the section we look at it from a functional perspective, explaining the core functionalities and what systems is relied on. This will be explained in the implementation section in more detail.

Explain role better

Deposits and Withdraws When a user deposits or withdraws funds, choosing the aggregated type, the user sends a signature of the deposit/withdraw operation to the aggregator. The aggregator receives theses messages, collecting them as the next aggregation batch. Once a number of messages have been received, the new balances are caluculated and passed to the corresponding ZoKrates program, where the correctness of the balances and signature is checked. If this is successful, a zkSNARK proof object is created. The aggregator now send the proof, along with the new balances to the zkSwap smart-contract. If the proof is valid, the new balances are emitted by the zkSwap contract. Each withdraw will now be credited by sending the requested funds to the users.

Aggregating Trades A user can make a trade by sending a message to the aggregator. Similarly to deposits and withdraws, the aggregator collects messages as the next aggregation batch. All received trade messages are

now aggregated and offset internally, resulting in the 'net-trade' that must be executed to honor all trades of the batch. The 'net-trade' is then sent to the 'PairProxy' smart-contract as a on-chain transaction, which in turn will execute it on Uniswap. The details and reasoning of this contract will be explained in the implementation section. Once the trade is executed, the new balances are calculated and the correctness ensured by using the corresponding ZoKrates program. The resulting proof and the new balances are sent to the PairProxy smart-contract. The previously purchased funds are now added to the transaction, forwarding it to the zkSwap smart-contract. The proof is verified, the new balances emitted, and the aggregator is refunded the amount paid in the 'net-trade'.

Storing Balances in a Merkle Tree The aggregator keeps track of the balances by listening to events emitted by the zkSwap smart-contract. When an event is received, the balance is either updated or added to the merkle tree if its a new user. Since the events stay on-chain, the merkle tree can always be rebuilt by querying these events, and updating the merkle tree sequentually. The aggregator provides endpoints where balances can be queried, along with the corresponding merkle paths, which are used throughout the system. It is important to remember, that the balances are public and can be queried by anyone.

PairProxy Smart-contract This smart-contract is controlled by the aggregator and is used to execute trades on Uniswap and forward transactions to the zkSwap smart-contract. The aggregator can use it to hold funds, which makes the transactions cheaper.

3.2 Implementation

In this section we will look at the way these fuctionalities are implemented and how they function.

3.2.1 Storing and Updating Balances

Balances are represented as a balance object in our system. This object consists of four fields that are neccecary to represent balances correctly, the ethAmount, the tokenAmount, the users address and a nonce. It is important to understand the core technique used to store and update balance objects before we look at the different functions that trigger them. We want to make balance object 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 methods of storage are on-chain, using the event log is significantly cheaper though. A disadvantage

of using the event log however, is the fact that it can't be accessed from a smart-contracts runtime, and must be queried by a client. This solves the external data availablity problem. We can store balances cheaply by emitting the 'BalanceUpdate' event, 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 user is passing correct data.

Merkle Trees We can achive this, be using a merkle tree [19]. Merkle trees are a suitable data structure, as the merkle 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. The correctness of that data can be proven by recreating the merkle root, which is stored and updated in the zkSwap smart-contract. Since all changes in balance are committed by emitting the 'BalanceUpdate' event, it must be ensured that merkle root is changed according to the updated balance. It is important to understand, that the only way balances can be updated is with the 'BalanceUpdate' event.

3.2.2 Aggregating Balance Updates

Updating a users balance is at the core of this system. Deposits results in a balance update, as do trades and withdraws. Before looking at these in detail, it is important to understand how balance updates can be aggregated, reducing the transaction costs for these operations. Since balances are stored in a merkle tree, we can ensure the correctness of a balance by running an inclusion proof. However, running this in a smart-contract is expensive, as a lot of hashing is required. To reduce this cost, we can ensure the correctness by running this in a ZoKrates progam. If the ZoKrates program exits successfully, a zkSNARK proof is generated, which can be used to verify our execution on-chain.

Merkle 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 balance provides its merkle path which it can be hashed with. If the resulting hash matches the current merkle root stored in the zkSwap smart-contract, we can be assured the provided balance is included in the tree. At the same time, this enables us to reuse the merkle path for updating the balance. We can simply change the balances values after passing the inclusion proof, rehash with the merkle path, and the result is the correct root for the updated balance leaf. Since the majority of hashing is done in ZoKrates programs, looking for a hashing

function that can efficiently be executed in a zkSNARK circuit is important. At the same time, we also want the hashing function to be somewhat affordable when executing in the smart-contract. In this implementation we will utilize the MiMC [8] hashing function, as it can used in zkSNARK circuits efficiently. The MiMC function is used with the feistel structure and setup with 220 rounds, which is deemed secure. The merkle tree is hashed with the MiMC hashing algorithm.

Chaining Inclusion Proofs When dealing with multiple balances, the inclusion proof can be done the same way. Every balance provides its merkle path, the resulting hashes should be the same for each balance. Things become more difficult when updating the balances. Updating the first balance in the batch now invalidates the merkle path of all following balances. In order for this to work, the merkle paths for each balance must be created sequetually. This can be solved by sorting the balances before hand, and generating each merkle path based on the changes of the previous balance. The new root of the first balance is the old root of the next balance. This can be chained to an infite length and results in a constant number of hashes required for each balance update. The last hash to be computed is the new merkle root, representing all balance updates.

Algorithm 1: Chained merkle inclusion proofs for verifying and updating balances

```
1 function verifyAndUpdateMerkle;
  Input  : oldBalances[], newBalances[], merklePaths[], root
  Output: newMerkleRoot
2 foreach oldBalances do
3 |   assert(computeRoot(oldBalances[i], merklePaths[i]) == root)
4 |   root ← computeRoot(newBalances[i], merklePaths[i])
5 end
6 return root
```

The merkle path has a binary as the first element of each hash value. That binary is used to represent if the hash is on the left or the right side of the pair. A nice solution is to sort the pairs before hashing, and decide the position based on the larger value, however limitiaions of the number range usable in ZoKrates make this infeasible.

Authorizing Balance Updates We still need to ensure the user has authorized this balance update though. As balance updates are emitted as an event, anyone can access them and compute valid merkle paths for any balance in our system. The data is public. This would, for example, allow any user to withdraw any balance. To ensure a user is authorized to update a balance, we need ensure the user controlls the private key belonging to the

Algorithm 2: Computes merkle root of given parameters

```
function computeRoot;
Input  : balance, merklePaths[]
Output: root
computedHash ← MiMC(balance)
foreach merklePath do
    if merklePath[i][0] = 0 then
        computedHash ← MiMC(computedHash, merklePath[i][1]);
    else
        computedHash ← MiMC(merklePath[i][1], computedHash);
    end
end
return computedHash
```

balances user address. This can be achived by requesting a signature from the user. 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 [11]. For that reason the Baby JubJub curve is used in combination with the EdDSA signature scheme, which can be run more efficiently in a ZoKrates program. The user submits a signature containing the current merkle root and the update message. This ensures three things. It proves the user controlls the private key belonging the the balance objects address, thereby authorizing the order. We also make sure, the balance update corresponds to the amount signed by the user, ensuring the transition is done correctly. By signing the current merkle 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 users consent if this was omitted. By checking if the current root is equal to the signed root when verifying the zkSNARK proof, we prevent replay attacks. It is important to note, that different programs are used for deposits/withdraws and trade aggregation. As already mentioned, these programs have a number of checks that ensure the changes in balance correspond to the values specified in the signed update message. These will be explained in the respective sections.

Creating a EdDSA Signature At the time of writing, metamask does not support signing with the EdDSA signature scheme on the Baby JubJub curve. Fortunetly, we can derive a Baby JubJub private key from an EcDSA signature, and then use the derived key to sign with the EdDSA signature scheme over the Baby JubJub curve. This signature can then be verified cheaply in a zkSNARK circuit. It must also be mentioned, that we utilize the MiMC hashing function to hash the message, as its efficient to run in a zkSNARK circuit.

explain situation with hermez

Executing and Reducing On-chain Verification Costs All of these checks are performed as a ZoKrates program. If no checks fail, the proof can now be generated. 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 the updates, the aggregator can pass that data as part of the verify transaction, but excluded from the ZoKrates proof object. By hashing the data with the SHA256 hashing algorithm 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, which we call the data hash. While the SHA256 hashing algorithm is inefficient to run in zkSNARK, and its very cheap use in a smart-contract. Since every balance update in an aggregation batch needs to be hashed on-chain, reducing the on-chain cost is more favorable, compared to reducing the complexity of the zkSNARK program.

3.2.3 Deposits

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 gives 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?

Movement of Funds To move funds to layer-2, a user first needs to send the funds to the zkSwap smart-contract. This is done by calling the deposit function in the zkSwap smart-contract and attaching the funds to the transaction. The zkSwap smart-contract stores a deposits hash value. When a deposit is received, the current hash value is hashes along with the deposited amount, type and user address. This is done sequentually for every incoming deposit and ensures the user has actually has sent the funds at a later stage. When a deposit batch is is verified, the hash value is set to '0x0' again. This is the first step of the deposit.

Aggregating Deposits After the funds have been sent to the smart-contract, the user creates a signature containing the type of aggregation (deposit/with),

the changes in Ether and token balance, the users address and the current merkle root. This signature is sent to the aggregator as an HTTP request. The aggregator checks the signature of each incoming request and makes sure it can recreate the deposit hash stored in the zkSwap smart-contract. This ensures the user has actually sent the correct amount funds to the smart-contract. After a number of deposits have been received, the aggregation is started. The aggregator now generates a 'BalanceMovementObject' for each deposit, containing the old balance, the new balance, the merkle path and the signature. This is passed to the 'ProcessBalanceMovement' ZoKrates program, along with the current merkle root. As explained in S. 3.2.2, the inclusion proof is now performed on the old balance. If the old balance is included in the merkle tree, the signature is checked, along with the change of balance. The signed amount should equal the added amount in the new balance, the nonce must be incremented and the address the same. If these checks pass, we calculate the new new merkle root, by rehashing the new balance with the merkle path. While iterating through the 'BalanceMovementObjects' we also compute the deposit hash, hashing amount, type of funds and depositors address. As a last step, we hash the new balances, along with the old merkle root, the new merkle root and the deposit hash as explained in S. 10, reducing the gas cost for the on-chain verification.

Verifying Deposits On-chain Once the ZoKrates program has run successfully, the proof can be generated, which is used for verification in the zkSwap smart-contract. The aggregator calls the 'verfiyBalanceMovement' function in the zkSwap smart-contract and attatched the proof, along with the new balances and the new merkle root to the transaction. As a first step, the balances, old and new merkle roots and deposit hash are hashed, and the result of the hash compared to the output field in the ZoKrates proof object. As the old root and deposit hash are stored in the smart-contract, we don't need to pass them in the transaction. If these hashes match, we can be assured, the the aggregator has attatched the data that has been checked by the ZoKrates program. Next, we check if the old root, matches the current root stored in the zkSwap smart-contract. This ensures that a generated proof can't be reused in replay attacks. As a last check, the ZoKrates proof object is checked with the verifier smart-contract deployed for this purpose. If this is also successful, we have proven, that the aggregator has proccessed the deposits correctly. We now iterativly emit the new balance objects with the 'BalanceUpdate' event, and update the merkle root in the smart-contract. The deposits have now been processed and show up as balance for the users.

3.2.4 Withdraws

The aggregated withdraws largly follow the logic of the deposits, so only the core differences will be explained here. Instead of sending funds to the zkSwap smart-contract, we are requesting them. Since the user already has

a balance in the system, we don't need to notify the aggregator about the amount a user wants to withdraw, which in turn means we don't need an on-chain transaction to trigger the withdraw. A user creates a signature of its address, the current merkle root, type of funds and the amount to be withdrawn, and sends it to the aggregator as an HTTP request. Once the aggregation starts, the aggregator checks the users signature, and ensures that the users balance covers the withdraw. Just like with deposits, a 'BalanceMovementObject' is created, the type being set to withdraw. The withdraws are verified along with the deposits in the same ZoKrates program. This results in only one on-chain verification, which reduces verification costs. The 'ProcessBalanceMovement' checks signature, the balances and if the the balance update corresponds to the amount signed by the user. On top of hashing the new balances, a withdraw object is also hashed, containing the type of funds and the amount. When verifying withdraws, along with the deposits, the withdraw objects will be used to send the funds from the smart-contract to the users. The new balances are emitted and the root updated.

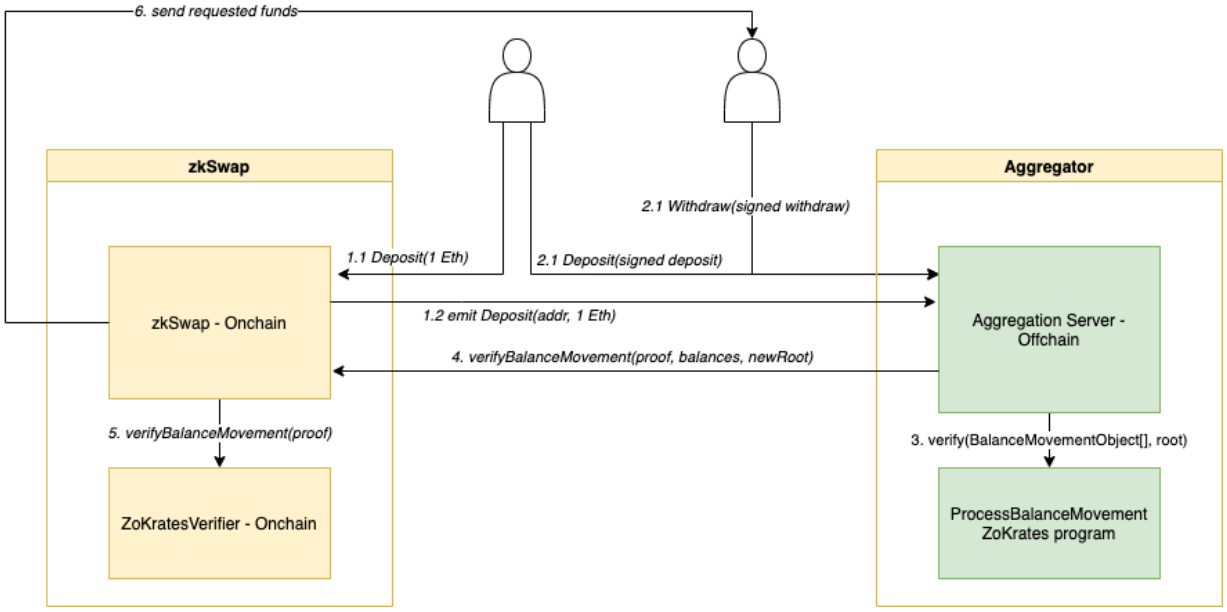


Figure 3: Interaction diagram of aggregated deposits and withdraws

3.2.5 Instant Withdraws

A user also has the option to withdraw instantly, without being dependant on the aggregator. This ensures a user can always withdraw funds, even when the aggregator is failing or offline. Instead of sending the withdraw request to the aggregator, the withdraw is processed completly on-chain. The user attaches its balance object, along with the corresponding merkle path and the withdraw amount and fund type (Ether/ERC20) to the transaction. As

a first step, the merkle inclusion proof is performed. The balance object is hashed, sequeutually with the merkle path. The resulting hash now equals the the merkle root stored in the zkSwap smart-contract if the correct balance object and merkle path have been submitted. It is checked if the balance can cover the withdraw, if thats the case the nonce is incremented, the new balance is calculated, and the new balance object hashed again. The new root is now computed by hashing with the merkle path, and updated in the smart-contract. The funds are now sent to the user and the new balance is emitted. It must be reiterated, that this is done completly on-chain and doesn't require the aggregator to be online. However, the gas costs of this transaction are significant, as hashing with the MiMC hashing algorithm is expensive in a smart-contract.

Authorizing Instant Withdraws This however, is an incomplete explaination, as we're not checking if a user is permitted to 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.

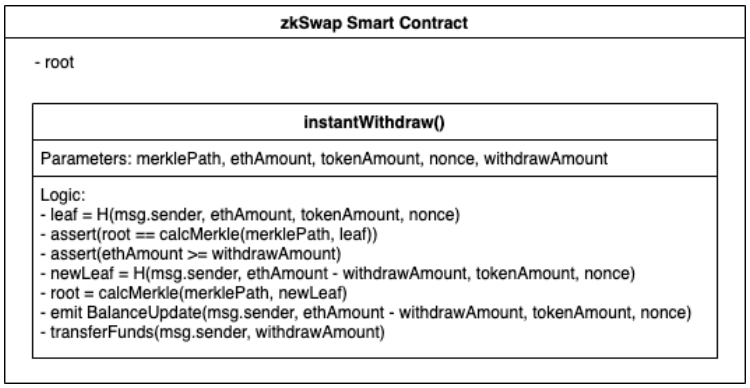


Figure 4: Pseudocode of instant withdraw steps

3.2.6 Aggregating Trades

Before explaining the life-cycle of a trade aggregation batch, it makes sense to understand the mechanism that ensures the correct price of trades in an aggregation batch. After, we will go through the entire life-cycle of an aggregation batch, starting with a user adding a trade.

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 of an asset can change. On top of that, network congestion on the Ethereum blockchain can cause fruther 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. Intuitivly, this makes sense, the spread should at least equal the cost of converting from one asset to the other. There are 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'. Since we don't know what orders will be received in an aggregation batch, we're unable to predict if the 'net-trade' will be a buy or sell order. 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

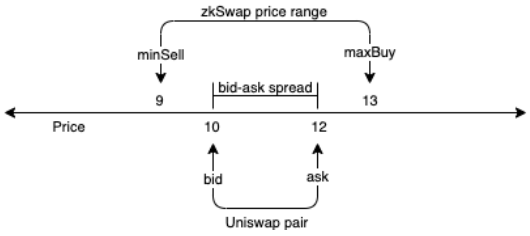


Figure 5: Bid-ask spread and zkSwap price range

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.

Adding an Trade Order To make a trade, a user must create an trade object and sign it with its private key. The trade object consists of five fields that are needed to define the trade, the tradeDirection, deltaEth, deltaToken and the users address and current merkle root. Since ZoKrates only uses unsigned integers, we need the tradeDirection to calculate the new balance. Once signed, the trade object is sent to the aggregator as an HTTP request. As a first check, the aggregator checks if the signature is valid. This ensures, that the trade object was created by the someone in control of the addresses private key, thereby authorizing the trade order. The aggregator now queries the users balance from its merkle tree, and checks if the merkle root in the trade object is equal to the merkle root set in the zkSwap smart-contract. This ensures the signed trade object can't be reused in replay attacks. The aggregator also checks if the users balance can cover the trade, making sure enough balance has been deposited. 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. It must be noted, that these checks are techniqually not needed to ensure the correctness of the aggregation, as the ZoKrates program performes the same checks. They are however needed, to prevent the aggregator 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 paragraph.

Executing Trade and Calculating New Balances 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 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, buy or sell order, which results one side to equal zero. This trade is now sent as an on-chain transaction to the 'PairProxy' contract, where the trade is now executed with Uniswap. The PairProxy contract is explained in detail in S. 3.2.7.

The aggregator waits for the PairProxy smart-contract to emit the 'TradeComplete' event, which will fire once the trade has been executed, containing the amount of assets aquired in the Uniswap trade. The amount received 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. For each trade, a 'BalanceUpdateObject' is created, containing the old balance, the new balance, the merkle path and the signed trade object. The merkle paths for each 'BalanceUpdateObject' are created as explained in S. 3.2.2. Now the 'ProcessTrades' ZoKrates program is called, along with the 'BalanceUpdateObjects', the current merkle root and the worst-case price.

Checking Pricing in ZoKrates We want to make sure that each trade has the same price, no matter if its a buy or sell order. It is also important that this price is no worse then the worst-case price, defined in the zkSwap smart-contract. We iterate through the 'BalanceUpdateObjects', checking if each trade has the same price, and making sure it's greater or equal to the worst-case price. While doing this, we also calculate the 'net-trade', which will represent the flow of funds between the zkSwap smart-contract and the aggregator. After this has completed, we are assured that each user is receiving a equal rate, at least matchig the worst-case price[2], and we have calculated the 'net-trade', which will be important when finalizing the aggregation.

Verifying Balances and Authorization in ZoKrates To ensure the correct aggregation of these trades, we still need to ensure the submitted old balances are part of the merkle tree and that the user has authorized the trade. While the aggregator has checked this already, it must be remembered, that the aggregator is an unstrusted party. We need to be able to verify the correct execution of these checks on-chain, which can be achived with zkSNARK. To ensure this, we largly folloe the steps described in S. 3.2.2. We iterate through

[2]The worst-case price will be compared to the one stored in the zkSwap smart-contract at a later stage, enforcing it for the entire aggregation.

the 'BalanceUpdateObjects' sequentially, checking if the merkle root can be recreated. If thats the case, we check if the signature is valid, and if the balance change matches the amounts specified in the trade object. Since the price hase been checked already, we're only checking the amounts here. Now we compute the new merkle root, based on the new balance. This root is then used to verify the balance of the next 'BalanceUpdateObject', details of this are explained in S. 3.2.2. The last merkle root computed is the new merkle root, representing the entire merkle tree with the updated balances.

Reducing On-chain Verification Costs in ZoKrates 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. As explained in S. 10 we can still reduce the gas needed for verifying the aggregation batch on-chain, by hashing the results, thereby removing them from the zkSNARK proof. As the last step of the ZoKrates program, we hash all new balances, the old and new merkle root, the net trade and the worst-case price. The resulting hash, called the data hash, is the only output of the ZoKrates program.

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

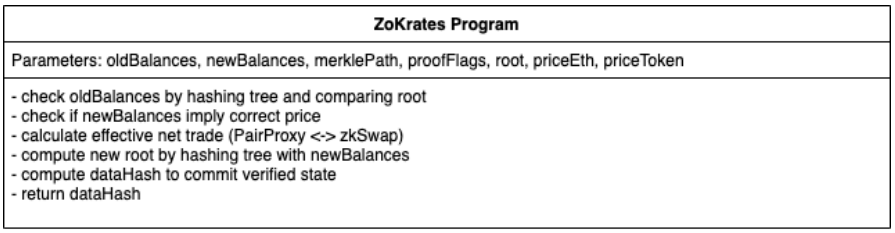


Figure 6: ZoKrates program checks

Generating Proof and Verifying At this point the aggregator can start the proof generation of the ZoKrates program. This proof object is needed to verify the correct execution of our ZoKrates program as an on-chain transaction in the zkSwap smart-contract and includes, among other things, our data hash. To verify everything on-chain, and thereby updating balances of all balances involved in the aggregation, the proof object is passed, along with the new balances, the new merkle root and the net trade and sent to the PairProxy smart-contract. The 'PairProxy' smart-contract receives the transaction, adds the funds previously traded with Uniswap to the transaction and forwards it to the 'verifyTrades' function in the zkSwap smart-contract.

Verifying the ZoKrates Proof The first thing checked in the zkSwap contract when receiving a trade aggregation batch, is the ZoKrates proof object. 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. If the verifier return true, we have proven that our data hash was computed by running the 'ProcessTrades' ZoKrates program, which in turn was used to generate the verifier. The remaining steps of the trade aggregation life-cycle happen in the zkSwap smart-contract, purely on-chain.

Recreating the DataHash and Ensuring Correct Price As a next check, we need to ensure that the aggregator has attatched the same data verified by the ZoKrates proof generation. This can be done by recreating the data hash. 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 hash the balances, along with the worst-case price, the new and old merkle root and the net trade with the SHA256 hashing algorithm. The old merkle root and worst-case price, however are not parameters attached to the transaction. Since these values are stored in the zkSwap smart-contract, we use these values. This ensured the correct values where used throughout the entire life-cycle of the aggregation. The aggregator could use an incorrect worst-case price and old merkle root, along with valid merkle paths. By having these values part of the data hash and querying them from the smart-contract, we can ensure the aggregator stays truthful and provides correct data. Failing to recreate the data hash will result in the aggregation being canceled.

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 aggragator has executed the 'net trade' and updated the balances accordingly, these funds need to be exchanged in order for the zkSwap contract to stay solvent[3] 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 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 starts.

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

3.2.7 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[4] 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[5].

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 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.

3.2.8 Client Frontend

The frontend allows the user to interact with the system, calling the functions, while providing neccecary 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

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

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

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 [10], not relying on any server to facilitate withdraws of user funds, which closely follows Ethereums unstopable 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 users 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 neccecary 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. **????**. 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.

3.3 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. **??**, the authorization of an order or deposit 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. Hermez [**?**], a zk-rollup based asset transfer system, has solved this by creating a Baby

JubJub[9] private key from a signed EcDSA message, which can be requested by Metamask. The generated private key can then be used to sign in EdDSA on the BN128 curve. I don't have a written source for this, but talked to there team members, who explained how they do this. Since their system is running on mainnet now, it can be assumed to work. In the systems current form, signatures are not checked at all.

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.

Hashing Function In its current form, the system doesn't utilize the MiMC hashing yet. All hashing is currently done with SHA256, which limits the number of trades or deposits/withdraws that can realistically be part of a batch. The ZoKrates programs have been updated to utilize MiMC, however I was unable to create matching hashes in the smart-contract.

Aggregating Deposits and Withdraws Since the MiMC hashes are not implemented yet, the aggregated withdraws arn't ether. Using SHA256 in a smart-contract is cheap, so executing the inclusion proofs on-chain is viable to a certain extent. The ZoKrates programs have been finalized, and the changes in the aggregator are also rather small, so this could be implemented without to much work, once the smart-contract MiMC implementation produces matching hashes. So in this form, deposits and withdraws are done on-chain, pretty much the way the instant withdraw works, but using SHA256 to hash the merkle tree.

4 Results

In this section we will look at a number of mertics that are useful for judging the systems performance. In zkSNARK enabled systems, the results must always be considered out of two different point of views. For one, the changes in gas usage are important. After all, this system is built to reduce gas consumption per trade. Another thing to consider, is the performance of the zkSNARK steps that are neccecary to execute and verifya circuit. As zkSNARKs rely on complex cryptographic primitives, the generation of proofs can take a long time, even on powerful hardware. We will start by looking at the gas usage results first, and then look at a number of metrics for the zkSNARK circuits.

Notarize execution maybe the correct word?

4.1 Gas Cost

In this system, we have four different operations that require gas to be executed. We will now look at these operations, presenting the measured results. The following three charts are scaled logarithmically on the x-axis. The logarithmical scale was chosen, as these operations have a large overhead (fixed cost) and a porportionally small amount of variable cost. This results in the cost per trade to reduce significalty in the beginning, which can be visualized best on a logarithmic x-axis scale.

4.1.1 Trade Aggragation

To break down the gas cost of a trade aggregation batch, we must first differentiate between fixed and variable gas cost that need to be paid per batch and trade. For one, the gas for the net trade, executed on Uniswap, must be paid. This amount varyies, depending on the direction of the net trade, 142k when trading from Ether and 167k gas when trading from an ERC20 token.

Another fixed cost to consider is the gas for verifying the zkSNARK proof, along with handling the refund payment of the aggregator and the checks defined in S. 3.2.6. Executing these costs 342k gas per aggregation batch. The combined fixed amount of gas per aggregation batch is 484k/ 509k gas, depending on the net trade direction. For each trade in a batch, we must pay 6619 gas, which is used for recreating the data hash, as well as emitting the BalanceUpdate event. When using these numbers, we get the following cost per trade, depending on the batch size. In this diagram, we're using the more expensive net trade (ERC20 to Ether) to calculate the results, which results in the maximum cost per trade for the corresponding batch size. As the difference is small, the difference in gas per trade converges quickly. The theoretical maximum batch size is 1811, which is where Ethereums block gas limit is reached.

To compare the results to a normal Uniswap trade, we define a break even price, which is equal to the cost of a Uniswap trade. Our system makes sense economically once our gas per trade cost is lower then the break even point. We will use 142k gas as our break even point, which is the best case for a Uniswap trade. The numbers found in this diagram do not contain any gas cost required for making a deposit or withdraw.

4.1.2 Deposits and Withdraws

Before a user is able to trade, funds must be deposited into the system. Presenting these results is a bit more diffecult compared to the trade aggregation results for two reasons. For one, the deposits and withdraws are aggregated as one aggregation batch. Since the gas costs for a deposit and withdraw are different, and the number of deposits and withdraws included in a batch is not predicatable, the cost per deposit/withdraw depends on the proportion of deposits and withdraws in a batch. On top of that, the gas cost of an

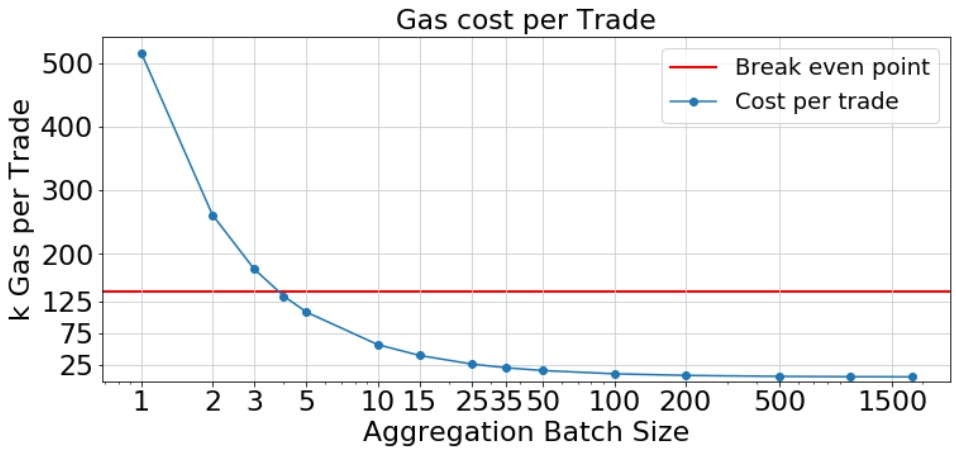


Figure 7: Gas cost per trade and breakeven point

deposit/withdraw also depends on the asset. Ether deposits/withdraws are cheaper than ERC20, which is a result of how tokens are represented on the Ethereum blockchain. For this reason we will present three quartiles for deposits and withdraws, that represent the share of an asset in the batch. For example, as seen in F. 8, '25% Eth' represents the gas cost per deposit if 25% of deposits in that batch are Ether deposits. While the numbers are a bit inaccurate for smaller batch sizes, it seems like the best approach to present the results overall.

Deposit Aggregation As with the trade aggregation, there is a fixed and a variable gas cost required per batch. For each deposit aggregation we have 312k gas as a fixed cost. This involves all checks and verifications explained in S. 3.2.3. The variable gas cost depends if an Ether or an ERC20 token is deposit. Depositing an ERC20 token requires significantly more gas then sending Ether, as multiple smart-contract interactions are neccecary. An Ether deposit adds 23k gas, an ERC20 deposit 116k gas. The maximum batch size is chosen based on the worst-case propotions, in this case only ERC20 deposits and the number of deposits we can include before Ethereum block gas limit is reached. For deposits thats a maximum batch size of 105.

Withdraw Aggregation The fixed costs of the withdraws is equal to the amount specified in the deposit aggregation, as they are being verified in the same batch. As with the deposits, the variable withdraw cost is different, depending if Ether or an ERC20 token is withdrawn. The maximum batch size is chosen based on the worst-case propotions, in this case only ERC20 withdraws and the number of deposits we can include before Ethereum block gas limit is reached. For deposits thats a maximum batch size of 186.

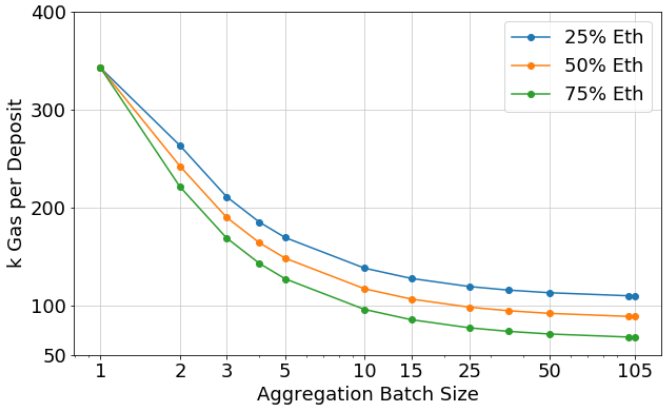


Figure 8: Gas cost per deposit assuming different share of Ether deposits

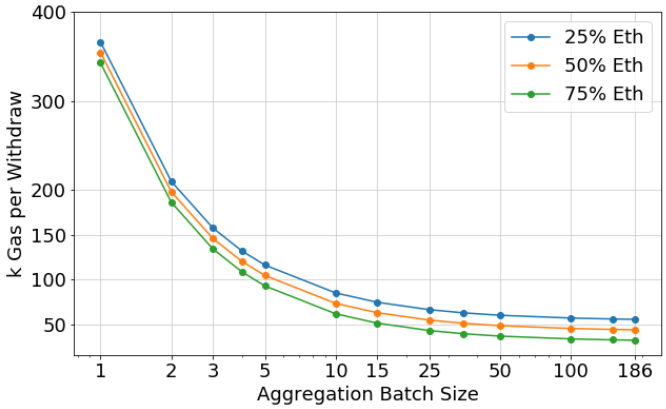


Figure 9: Gas cost per withdraws assuming different share of Ether withdraws in batch

Combining Deposit and Withdraw Gas Cost Modeling the combined gas costs of deposits and withdraws is omitted in the work, as there are to many assumptions that need to be made. As mentioned before, the proportion of Ether and ERC20 operations, as well as the porportion of deposits and withdraws influence the results. For the deposits and withdraws a valid approach was taken to model the potential gas cost, depending on the proportional share of funds. Combining these values, defining the proportional share of deposits and withdraws in an aggregation batch wouldn't yield representetive results, as there are a number of valid combinations that would need to be considered. The diagrams presented above give a clear indication of the potential cost, which suffices for this work.

Instant Withdraws The cost of instant withdraws is also worth considering. Since we're not batching instant withdraws, there is only a fixed cost per withdraw that needs to be paid. Since this hasn't been implemented at the time of writing, a rough estimate of the gas costs can only be provided. To update a balance in a merkle tree with a depth of 16, we need to hash a total of 34 times. 32 times for the merkle inclusion proof and update, twice for hashing the old and new balance. Computing as MiMC hash currently costs 94k gas, which would bring the cost to around 3.2m gas per instant withdraw.

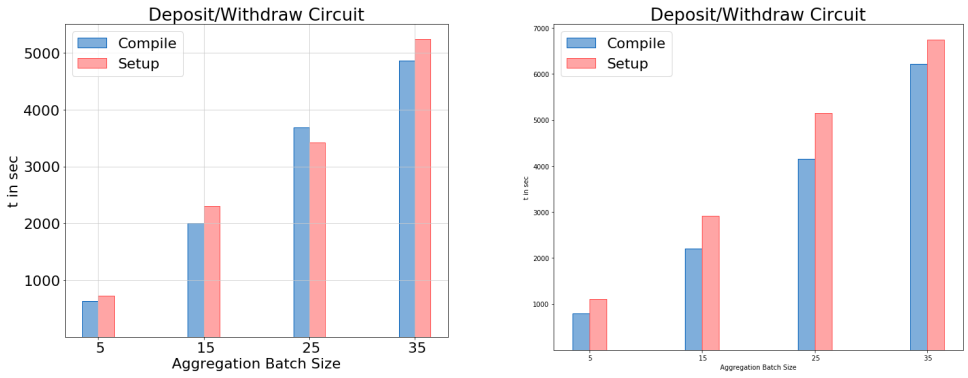
4.2 zkSNARK Circuit Metrics

Another aspect to consider is the performance of the zkSNARK circuits. The benchmarks for the zkSNARK circuits where performed on a Google Cloud Plattform C2 instance, with 60 vCores (3.1GHz base and 3.8GHz turbo) and 240Gb of memory. This instance was chosen because of the large amount of memory and the fast clock speed. The number of cores doesn't impact the benchmarking results, as the zkSNARK steps can't be parallelised at the time of writing.

4.2.1 Execution Time

The first obvious metric to consider is execution times of the different steps required to utilize our circuits.

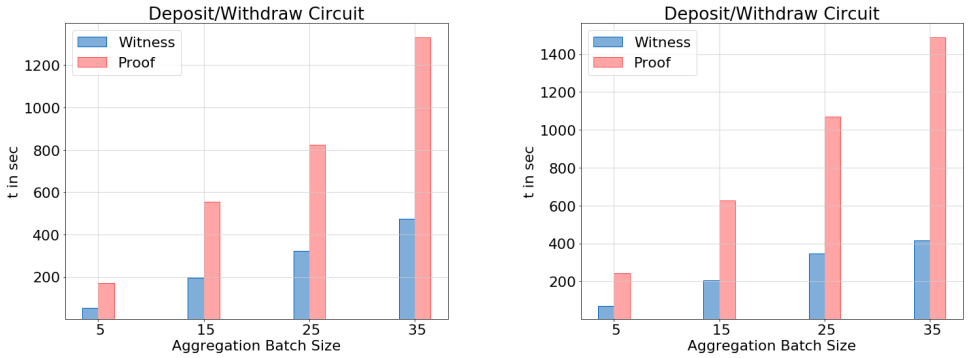
Compilation and Setup Before we can generate any zkSNARK proofs, we have to compile our circuits and run the setup. These two steps only need to be run once per circuit, so they're not significant for the viablilty of this system. However I have the numbers, and it would feel incomplete to not present them. These are the results, for the trade and deposit/withdraw circuit.



(a) (b)

Figure 10: Compilation and setup execution times

Witness Computations and Proof Generation For each aggregation batch we need to first run the witness computation, after which the proof generation can be run. These two steps need to be run for every aggregation batch, so the performance is a indicator for the viability of this system. It decides how long the aggregation of a batch takes, which impacts the pratical application of this system.



(a) (b)

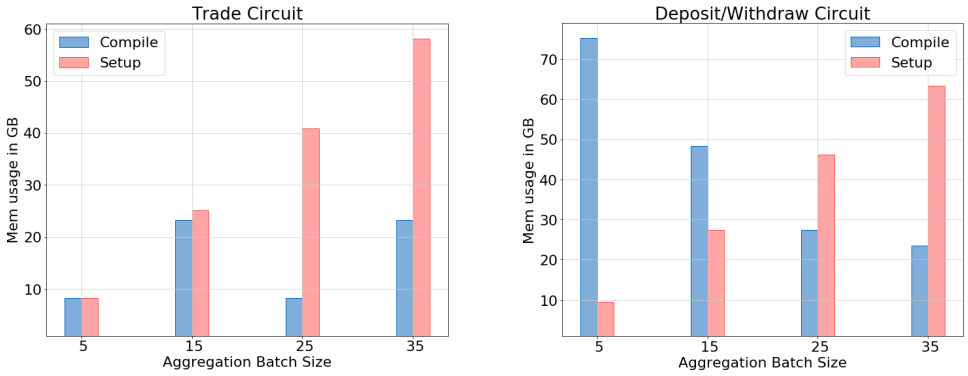
Figure 11: Witness computation and proof generation

4.2.2 Memory Usage

Only looking at the execution times gives us an idea how many operation can be batched. It tells us nothing about the hardware requirements needed for working with circuits of this size. One thing to look at, is the memory used in the different steps. In general, the memory consumption of these processes is high, which is why a server instance with such a large amount of memory was chosen.

Compilation and Setup When measuring the compilation memory consumption, we get a confusing picture. The results don't really make sense, as smaller circuits sometimes require more memory as smaller ones. However, I repeated this measurement on different machines and operating systems, always receiving inconclusive results, similar to these. I watched the compilation on the server with htop, and observed the same amount of memory that the memory usage script was detecting. The script uses the command line tool 'ps' to take these measurements, which measures reserved memory by a process. Alternativly, a profiler could be used to measure the actual memory usage. This would impact the performance of the application severly though. As these steps only have to executed once, they are not a meaningful metric.

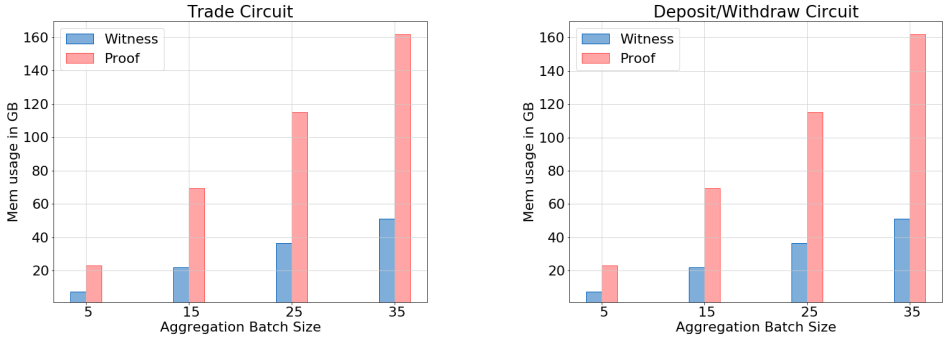
Witness Computation and Proof Generation The memory required for running the witness computation and proof generation is an important metric and dictates the hardware needed for the aggregation process. We



(a) (b)

Figure 12: Compilation and setup memory consumption

need to run these steps for every aggregation batch, so hardware capable of handling the memory requirements must be available.



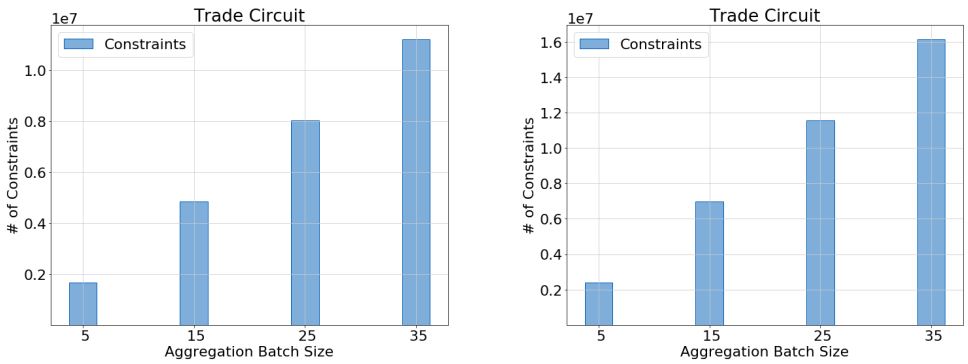
(a) (b)

Figure 13: Witness computation and proof generation memory consumption

4.2.3 Constraints

Looking at the results, we see that the execution times increase linearly with the batch size. The same pretty much goes for the memory consumption of our circuits. As a general rule, the complexity of a zkSNARK circuit is defined by amount of constraints the circuit is made of. Each additional element in the batch adds a certain number of constraints to the circuit. Looking at our circuits, we get the following constraint counts for different batch sizes.

Origin of Constraints Both circuits can be broken down into three different main segments, that add a certain number of constraints. 1) We need to run the inclusion proof and update the merkle tree. 2) We check the signature and if the balance update follows the signed amounts. 3) We need



(a) (b)

Figure 14: Constraint count for the circuits in different batch sizes

to compute the data hash. For each of these operations, we get constraint numbers that are added per additional batch element. These numbers where measured by compiling the segments seperatly and checking the constraint count. Comparing these to the total constraint values, we realize that they are higher. This can be explained by the ZoKrates optimizer, which runs a number of optimizations during compilation. As we can see, these values are very similar in both circuits, which is why they have comparable performance.

	Fixed Cost:	Variable Cost:
Trade Circuit:		
Merkle Tree	1	179781
Verify sig and balance update	1175	25675
Data hash	182254	184806
Deposit/Withdraw Circuit:		
Merkle Tree	1	179781
Verify sig and balance update	1	28154
Data hash	55970	184806

update constraints for depoist

Hashing and Constraints The MiMC hashing algorithm was used for hashing the merkle tree, as it's more efficient to use in zkSNARK circuits. The most common hashing operation used in the system is the pair hashing of the merkle tree. Every balance update requires 34 pair hashes in total, 1 for hashing the old balance, 16 for the merkle inclusion proof, 1 for hashing the new balance and 16 for updating the merkle tree. We must also remember, that the pair elements need to be sorted according to their position (left, right), which doubles the constraints in a zkSNARK circuit. On the other hand, we also need to hash the balances in the zkSwap smart-contract for

recreating the data hash. In Solidity the sha256 hashing algorithm is a lot cheaper than MiMC. Since we have to recreate the data hash for every batch we want to verify on-chain, the data hash is computed with the SHA256 hashing algorithm.

Hashing Op	Constraints	Gas Usage
MiMC pair	2642	38840
MiMC pair sorted	5285	38840
SHA256 pair	56227	2179
SHA256 pair sorted	112453	2179

Constraints Processed per Second As last metric we want to take a look at, is constraints processed per second in the witness computation and proof generation step. We can calculate these values by dividing the exectution time by the number of constraints. At the time of writing, these steps are not parallelizable and only run on one CPU core. The Loopring protocol claims to have parallelized the libsnark library, which we will look at in S. XX. For this reason we are introducing this metric, as we can compare the outcomes and the potential speed up of parrallelizing. On top of that, this metric can also be helpful to test the performance of the circuits on a CPU with higher clock speed.

5 Discussion

In this section we will analyse the results, take a look at the limitiation of the system and take a look at the open problems.

5.1 Interpretation of results

To understand how this system, and by extension zk-rollup as a whole, performs practically we will analyse the results and consider the

batch_size	Witness: constraints / sec	Proof: constraints / sec
5	27604	9685
15	29615	8417
25	29088	9612
35	27797	8382
Average	28526	9024

5.1.1 Gas results Trades

The most important metric of this system is the gas amount per trade that can be achived. After all, reducing gas is the main goal of this system. Using the biggest possible batch size, a trade would cost 6886 gas, which is a significant reduction when compared to a trade without aggregation. Using a batch of this size however, does not seem viable in practice. For one, gathering this many trades for a batch would require either a high usage of the system, or long trade gathering periods. At the same time, a batch size of 200, already reduces the gas amount of a trade to 9036. Looking at the results presented in F. 7, we see that the batch size does not reduce the cost per trade linearly, which is caused by the gas required for verifying the zkSNARK proof. This is good news, because it results in small batch sizes already having substantial gas savings. Adding to that, the break even point per aggregation batch is reached once 4 trades are part of a batch. This means the the overall cost per trade is reduced from that point onwards, as shown by the curve in F. 7, compared to a trade without aggregation. Overall these are promising results. We reach our break even point quickly, while the cost per trade drops sharply, even with a small batch size. It is also worth noting, that the on-chain verification of a batch is not a bottleneck for the system. Large batches can be verified with predictable gas costs if that is required.

5.1.2 Gas results Deposits and Withdraws

Another aspect to consider is the cost for deposits and withdraws. A user need to deposit funds in order to use the system. The cost per deposit and withdraw is different depending on the asset that is being moved. When moving Ether, each deposit adds 44k gas and each withdraw 18k gas to a batch. This is an acceptable amount, given that the fixed fee of an Ethereum transaction is 21k gas. Moving ERC20 tokens is significalty more expensive though. Since ERC20 token balances are represented as a mapping in the tokens smart-contract, moving them requires the state to be updated in respective smart-contract. Updating a smart-contracts state is expensive on the Ethereum blockchain. Depositing ERC20 funds also requires two seperate transactions, the first to set an allowance for the zkSwap smart-contract, a second to trigger the zkSwap contract to receive the funds previously set as allowance. For these reasons, moving ERC20 tokens requires a lot more gas, each deposit adds 115k gas, each withdraw 65k gas. While there is no clear path for reduction, a potential solution will be presented in S. 6.5.

5.1.3 Circuit results

Since both circuits are similar performance wise, we will not differentiate between them in this section. The performance our circuits has to be measured

differently then the trades. It must be remembered, that the gas usage for verifying a zkSNARK proof is fixed, and doesn't increase with the complexity of the circuit. While the on-chain gas cost is fixed, the computational complexity of the proving steps grow with the overall complexity of the circuits. The larger a circuit is, the more computationally demanding the proving steps are. This can be measured in the execution time of these steps, but also in the memory required for them.

Execution Times When looking at the result presented in F. 11, we see that the zkSNARK proving steps take some time to terminate. With a batch size of 35, it takes around 30 minutes to compute the witness and generate the proof. Although thats a long time, it doesn't have to be a hinderence in a practical application. I our system, the actual trade on Uniswap is executed before the proving steps are started. Having the trade executed before the aggregation starts means we have locked in the trading price already. Any changes in price, given the right incentrives for the aggregator, can't impact the aggregation anymore. However, there is a limit to the execution time of the circuits that users would be willing to accept. In general the amount of constraints a circuit is made of dictates the execution time of the proving steps. To speed up the execution times, we must increase the number of constraints that are processed per second. This can be achived with a higher clockspeed or by utilizing more cores. In our benchmarking server, which has 60 cores, only one was able to be utilized. As shown in T. 4.2.3, our system can process around 28000 constraints per second during the witness generation and 9000 constraints per second during the proof generation. The main bottleneck of the circuits performance is the unparallelizability of these steps, the potential speedup being significant.

Memory Requirements Another consideration to make when analysing the circuits performance is memory utilization. Looking at the memory consumption used in the proving steps in F. 13, we see that a batch size of 35 requires over 160GB memory during the proof generation. Running this on hardware with enough memory is crucial, as the execution times can be impacted by slow swap memory. Each aditional element is the batch adds around 4.5GB of memory. Ignoring the execution times for a moment, running these circuits with large batch sizes requires server instances with hard drive sized memory. For example, running an aggregation with a batch size of 200 would require roughly 900GB of memory. The memory requirements are a major bottleneck for using large circuits practically.

5.1.4 Results overall

Looking at the system overall, we can measure promising results. In the systems current form, we're able to reduce the cost per trade to around 20k gas,

which is a reduction between 78% and 88%, compared to an unaggregated Uniswap trade, all wile remaining trustless and not relying on external data availability. The zkSwap smart-contract is future proof, and can be used with larger batch sizes without a lot of changes. The variable gas amount per trade can still be reduced with more optimizations, though the results measured in this implementation are overall satifying already. On the other hand, the zkSNARK proving steps could speed up quite significalty with the proper optimizations. The easiest path for reducing the gas fees per trade is to increase the batch size for now. To enables that, multi-core witness computation and proof generatation will be nececcary and major improvements in memory consumption.

5.1.5 Usability of the System

The results overall look promising opn paper. Another consideration to make is the usability as a product. For one, aggregating a trade willl result in quite significant delays in execution. In the first period of an aggregation we are collecting trades, waiting for users to add requests to an aggregation batch. This can take some time. After a number of trades have been collected, we have to run the aggregation, which also takes some time. Overall the user can expect to wait quite a while for an aggregation batch to finalize.

5.2 Limitations

In this section we will look at current limitations of the system.

5.2.1 Fixed Batch Size in Circuit

A main limitation of the system currently, is a statically typed nature of zkSNARK circuits. When creating a circuit with ZoKrates, it feels like programming with a turing complete DSL. However, this is not the case. When compiling a ZoKrates programm, the circuit will be build represent every path through the programm. When computing the witness and generating the proof every constraint, and thereby every path, will be evaluated adding to the circuits complexity. For example, when hashing a merkle pair we use an If/Else construct to decide which hash is on the left side, and which on the right side. Under the hood, the circuit is built for both paths, which results in the constraints being doubled. This also means that we need to define the number of inputs to the circuit before compilation. Since we never know how many trades/deposits/withdraws will be received per batch, we need to compile multiple versions of the circuit, for a different batch sizes. This however also results in a seperate verifier smart-contract needing to be deployed for each circuit.

5.2.2 Running Aggregation Blocking State Updates

A running aggregation batch can be seen as a blocking process in our system. When aggregating a batch, we're essentially ensuring the state updates are done correctly. This mean that our circuit receives a pre- and post aggregation state and ensures the transistion of states was done according to the rules specified in the circuit. As a result, the entire state of the system can't be changed while an aggregation batch is running. For example, imagine an instant withdraw being executed while a batch is being aggregated. The instant withdraw would change the merkle root, which would in effect invalidate the verification of the aggregation. While deposits and withdraws are mostly being aggregated, it also means they can't run while the trade aggregation is ongoing.

5.2.3 Trusted Setup Phase

The non-interactive nature of zkSNARK requires a common reference string to be shared among prover and verifier [4]. To generate these parameters, we rely on secret randomness thats is created during the setup phase and should not be stored by any party. Having access to the secret parameter enables the creation of fake proofs. For this reason the secret is called toxic waste, as storing them breaks all cryptographic assurances of zkSNARK. Currently, the setup is performed on the aggregation server, which is where the toxic waste could be stored secretly. This can be solved, for example by multi-party computations as described by the ZCash team [4]. This requires only one party of the computation to be honest, which is deemed secure with enough participants.

5.3 Open Problems

In this section we will take a look at the open problems that have become apparent while building this system. None of these problems seems impossible to solve, but where not a focus of this research.

5.3.1 Canceled Aggregations

When a price of an asset changes, passing the defined price range the aggregation is canceled. The trade cant be executed for the defined worst case price, which means the users balances remain unchanged. A problem however, is the updating of the worst case prices. Since the price of an asset has changed far enough for the aggregation to be canceled, the pricing range must be updated in the zkSwap smart-contract. At the same time it must be ensured, that the price range can simply be updated by anyone at any time. It must be remembered, that correct pricing is checked in the zkSwap smart-contract when verifying an aggregation. Maliciously changing the worst case price can invalidate an entire aggregation batch. To prevent this, a proof of some kind

must be used to verify a batch has been canceled. On top of that, the aggregator must be incentivized to report a canceled aggregation batch, which will cost gas.

5.3.2 Empty Aggregation Batch

Similarly to a canceled aggregation, a mechanism dor dealing with empty batches must be created. The two main problemes explained in S. 5.3.1 apply here aswell, the aggregator must be incentivized and a proof for the empty batch must be submitted. The main goal is to update the worst case prices.

5.3.3 Ignoring Deposits

An aggregator could chose to ignore deposits of a certain user. When a user makes a deposit, an on-chain transaction, along with the funds is sent to the zkSwap smart-contract. At the same time, the details of the deposit are signed and sent to the aggregator. A problem arises when the aggregator simply ignores a user. The funds are held in the zkSwap smart-contract, but the user has no custody of them, until the deposit is added to a batch. This isnt really a problem when dealing with trades or withdraws, as the can't lose access to the funds. In the current implementation, the funds would be locked up

5.3.4 Ensuring Correct Effective Price Reporting by Aggregator

Once the aggregator has executed the net trade on Uniswap, the post-trade balances are supposed to be updated according to the actual price the trade has been executed for. Ensuring the correct price is used is not possible in the current implementation. The aggregator could theoretically always use the worst case price and keep the remaining funds to itself. This is a problem, as a user would always receive the worst possible price when trading using the system.

5.3.5 Sandwich Attacks

A sandwich attack, as described by Zhou et al. [24] is an attack, that targets decentralzied exchange transactions. The basic idea of a sandwich attack is influence a trade transaction by having one transaction executed before, one after the actual trade. This can be exploited, either for profit or resulting in the attacked transaction to fail. When the net trade of an aggregation batch is executed, an attacker can analysize the transaction in the mempool and estimate the price impact of that trade on the assets pricing. By setting a higher gas price, the attacker can 'front-run' [7] the original trade transaction, having it executed before. This can be exploited for profit. The attackes analysis the trade, then front runs the trade transaction, buying before the

original trade. The original trade will cause a predictable price change, which can be exploited by selling the previously bought funds after the original trade was executed. The profits made in this attack are paid indirectly by the original trade, receiving a worse exchange rate. This can also be exploited to cancel an aggregation batch. The price range is publically available, so an attacker could front run the aggregator, moving the price out of the defined range.

6 Related Work and Outlook

In this section we will take a look at

6.1 Prover optimizations

The main bottleneck of this systems potential performance is the prover. Speeding up the prover would enabled much larger batch sizes, that would reduce the cost per trade even further. Another important improvement would be the reduction of memory usage of the prover. Loopring, a zk-rollup based decentralized exchange claims to have achived that. We will look at the improved performance the Loopring team have claimed to have achived [6] and compare them to the numbers measured in this system.

6.1.1 Parallelizing the Prover

Loopring was able to parallelize the prover. As we have previously discussed, the main bottleneck of a circuits performance is the CPU. Loopring uses the libsnark as a proving library, which can be used with ZoKrates aswell. Before parallelizing the prover, Looprings prover was able to process 40000 constraints per second in the proof generation step. Our system performs much worse, with around 9000 constraints per second on a comparable CPU. It must be notes however, that other optimizations where already implemented before parallelizing. The x4 speed compared to an unoptimized bellman backend lines up with the numbers measured in the results section. When running in parallel, Looprings claims to be able to scale the constraint per second throughput linearly with up to 16 cores, processing over 620k constraints per second. The biggest circuit they use has 64M constraint, which they're able to generate a proof for in 106 seconds. For comparison, the largest circuit tested in our system is around 12m constraints large, and it takes 21 minutes to generate the proof. Being able to achive this kind of processing speed would greatly improve the usability of zk-rollups, so research in this direction should be done. In this context, Wu et al. [23] must also be mentioned, as they have shown that running zkSNARK circuit compilation and proving steps is viable in a distributed manner. For this reason I believe the claims made by the Loopring team to be achivable, as they also focused on optimizing fast fourier transforms and multi-exponentiations for distributed computation.

6.1.2 Reducing Memory Usage

Another aspect that need to be improved increasing the batch size is the memory required in the proving steps. Looking at our results, we require around 13GB of memory per million constraints when generating the proof. The Loopring team has claimed to reduce their memory requirements from 5GB to 1GB per million constraints. These values can't be compared directly, as a different hashing function is used in Looprings implementation, which can have a big impact on the memory requirements by having different linear combination lengths. The techniques used should result in a positive effect in our system aswell. For one, a lot of memory can be saved by not storing every coefficent independently. Loopring was able to reduce the number of coefficients stored while generating the proof from around 1 billion down to just over 20k. The memory further reduced by not storing each constraint independently. Since most of our constraints are caused by the hashing functions, we have a lot of duplicate constraints that just work on different variables. Reducing our memory requirements by a similar amount, would greatly improve the batch size that can be aggregated in practice.

6.2 Hashing Algorithms

The entirety of this system can only function by utilizing hashing algorithms. The properties of hashing functions, namely being deterministic, collision resistant, non-invertable and quick to execute, enable us to verify the correctness of data in a cheap and compressed form. We apply this by storing the balances in the merkle tree, by compressing the zkSNARK proof and and by hashing deposit values in the zkSwap smart-contract. This system would not work without hashing algorithms. Currently however, there is no hashing function available that is efficient to use in a zkSNARK circuit while also being cheap on the Ethereum blockchain, which is not ideal.

6.2.1 MiMC on Ethereum

By reducing the multiplicative complexity the MiMC hashing algorithm can be efficiently used in a zkSNARK circuit. As shown in S. 4.2.3, the constraints required per hash are significantly lower then SHA256, which speeds up the proving steps. Conversely, the MiMC hashing algorithm requires a lot of gas per hash, while the SHA256 hashing algorithm doesn't. This is a limiting factor, as it requires us to make tradeoffs between the hashing functions. In this project for example, we use MiMC hashes for the merkle tree and a SHA256 hash for the data hash. While this does use the minimal amout of gas, it doubles the constraint count of our circuits. At the same time, this also makes the instant withdraws, which must use MiMC hashes on-chain prohibitivly expensive.

A solution would be a precompiled MiMC hashing smart-contract, and reducing the operation costs most relevant for computing the hashes in the

Ethereum Virtual Machine. Similarly, Ethereums Istanbul hardfork included EIP-1108 [13]. This improvement proposal reduced the addition and multiplication operations on the BN254 curve, which are operations used often during the verification of a zkSNARK proof. A similar approach seems likely, however since the MiMC algorithm is still quite new, the algorithm has to be studied closer.

6.2.2 Poseidon Hashes

The Poseidon [17] hashing algorithm is novel hashing algorithm that aims to be efficient in zkSNARK circuits. A 128-bit hash with an arity of 2:1[6] only adds 276 constraints per hash. This is significantly less then the 2642 constraints a MiMC hash requires, or even the 56227 constraints a SHA256 hash requires. The merkle inclusion proof and update in this system could be done with under 28k constraints, which is a significant reduction.

Using Poseidon on the Ethereum blockchain is still quite expensive, a hash with a 2:1 arity costing 28858 gas currently. A similar approach described in S. 6.2.1 can be applied here aswell. Poseidon was not used in this work, as a collision was found by Udovenko [20]. The problems seem to have by fixed by now and the security analysis of this hashing function is ongoing. Utilizing Poseidon for the merkle tree described in this work, would reduce the constraints used for the merkle tree by 80%. Pretending Poseidon has the same on-chain cost as SHA256, would reduce amount of constraints for the data hash by 99.85%. Poseidon shows great promise for increasing the throughput of zk-rollup enabled applications.

6.3 PLONK

PLONK [16] is a universal proving scheme that has the potential to greatly improve the usability of zero knowledge protocols. PLONK increases the usability of zero knowlege protocols, because it enables the common reference string that is generated during the setup to be used by any number of circuits. This means that a single verifier can be used to verify any number of circuits. This solves a big challenge that arises when working with zero knowlege protocols. In this system for example, a big limitiation is the static nature of a circuit. We always need to pass the exact number of arguments specified in the circuit to execute it. This is very unflexible and requires us to compile and setup a large number of variations of our circuit to make sure we're able to work with a changing number of inputs. While this works in theory, it also requires us to deploy a seperate verifier for each circuit that needs to be deployed on the Ethereum blockchain. The deployment of these contracts costs gas, as does the on-chain logic to pick the correct verification contract to verify a batch. PLONK solves this. We can create any number of

[6]The merkle tree used in the implementation also uses a 2:1 arity for the merkle tree pair hashing

circuits, compile them, and then use the same common reference string to set them up. We can now verify all circuits with the same verifier. Using PLONK with zkSNARK will increase the proof size a bit, which will increase the gas needed for the on-chain verification step. Other changes in performance must be explored and tested.

6.4 zkSync and the Zinc Programing Language

zkSync [5] is a application that uses zk-rollup to enable cheap Ether and ERC20 transfers. Users can move their funds into layer-2 and send them to other users in the layer-2 system cheaply, by having the transfered aggregated with zk-rollup. It is one of the few systems that is utilzing zk-rollup in a production environment, showcasing that viable systems can be built with the technology. Instead of using zkSNARK, zkSync relies on the PLONK proving scheme, which results in greater circuit flexibility.

Matterlabs, the company behind zkSync, is also developing Zinc [3], a DSL that can be used to create Ethereum-type smart-contracts that are compiled as a zero knowledge program. Zinc is built to mirror the concepts and fuctions known from Solidity, to make porting of smart-contracts as easy as possible. What makes this interesting, is Matterlabs claim of having developed a recursive PLONK proof construct. The idea is, that Zinc program are deployed to zkSync layer-2 network and can be verified recursivly with zkSync circuit. Zinc programs can call each other, offering the same composability known from Ethereum main chain and can transfer funds in the zkSync network. When calling a Zinc programm, a validator is picked to compute the witness and generate the proof. A number of proofs can then be verified recursivly, resulting in one verification transaction on the Ethereum blockchain. This can potentially bring entire smart-contract ecosystems into layer-2.

A testnet of this technology is online at the time of writing, and a decentralized exchange [1] has been built to showcase the technology. This is all in very early stages currently, and the literature is not good. Matterlabs is a known entity in this space and given the technological potential, I would argue that it is important to mention this work. However, it must be taken with a grain of salt.

6.5 Cross zk-rollup Transactions

Another technology currently in development are cross zk-rollup transactions, as described by C. Whinfrey [21]. The main idea is to connect different zk-rollup applications with each other and enable transactions between them without having to move funds over the main chain. This can be achived by having intermediarys that deposit funds in two zk-rollup application. A user can then send funds the intermediary on zk-rollup application A, the intermediary will then send these funds to the user in zk-rollup application B. This can be done in decentralized fashion.

This is an important development, as it makes balances transferable between different zk-rollup enabled applications. Without this, zk-rollup would be a questionable scaling solution as it would break composability of applications and require funds always to move through the main chain when transfering to a different zk-rollup application. This would put strain on the Ethereum blockchain and make moving funds between zk-rollup applications expensive.

6.6 Aggregator Fee Structure and Jobs

The role of the aggregator also has to be defined more clearly. In this work, its tasks and jobs where only outlined from a fuctional perspective, somewhat ignoring the economical aspects. A fee structure must be developed, along with a system to incentivize reporting empty or canceled batches, as described in S. **??**. Fortunelty, the break even point of a trade is quickly reached, so setting economical incentives for the aggregator shouldn't be to difficult.

7 Conclusion

We will now conclude on the findings of this work, looking at the viablility of trade aggregation, but also of the scaling implications for other dapps or usecases.

7.1 zkSwap

In general, the approach explored in this work seems to be viable. We were able to build a trustless system, that successfully reduces the gas required for trading. The entire system is also non-custodial, mirroring the properties of decentralized exchanges. The data availability problem has also been solved, reaching the goals set for this work. The gas savings also look encouraging. We're able to reduce the cost per trade in this implementation, increasing the batch size can reduce the gas cost per trade even further. While the smart-contract logic used for verifying aggregation batches can still be optimized for gas efficiency, it could process much larger batch sizes, without many changes. It can not be considered a bottleneck in this system, and is only limited by the Ethereums block gas limit. The bottleneck of the system is the aggregator, and the complexity of computing the proofs for each batch. The approaches to scale the aggregation batches can broadly be separated into two groups.

Reducing Constraint Count Zk-rollup enabled applications largely rely on the hashing functions to function. For one, its thought to be the most efficient approach to store state in a merkle tree. A trees state can be commited on-chain cheaply by storing the root, and an inclusion proofs complexity correlates logarithmically to the trees depth. This however, requires a large amount of hashing in the circuits. New hashing functions like Poseidon look

promising in reducing the constraint count per hashing operation, but the security of these functions is still being researched. Some hashes need to be computed in a circuit, but also on-chain. When computing a hash in a circuit and on-chain, the decision to opt for a hashing function that can be computed efficiently on-chain is obvious, after all we're optimizing for gas consumption. However, this increases our circuit size substantially, thereby reducing the batch sizes that can be aggregated in a reasonable amount of time. Having access to a hashing function that can efficiently be used in a circuit and on-chain would increase the throughput of zk-rollup enabled applications significantly.

maybe mention percent of constraints from hashing?

Increasing Constraint Throughput The throughput of an aggregation batch can also be increased by speeding up the proving steps. A large speed up could be achived by parallelizing the proving steps, which would make larger batch sizes viable. As shown by Loopring, there seem to be a number of areas where large efficency gains can be achived.

ZK-rollup as a Whole Considering that there is a clear path for improvement, zk-rollup is a strong contender to scale the Ethereum blockchain to process orders of magnitude more transactions. This work has also shown, that it is viable to aggregate user transactions for existing smart-contracts. Having to interact with other smart-contracts to execute an aggregation batch does add complexities and requires more gas to verify on-chain. The gas savings that can be achived still make it a viable approach. Another thing to consider when working with other smart-contracts in a zk-rollup application are the return values from those smart-contracts. In our case for example, the amount received when executing a Uniswap trade can't be predicted. While solving these issues add even more complexities and inefficiencies to the protocol, they can be solved with current technologies. However, it remains to be seen how much the technology continues to develop. The potential of recursive PLONK proofs cant be understated here. Being able to recursivly prove arbitrary circuits, resulting in only one on-chain verification transaction could enable scaling solutions, far superior to the approach presented here. Entire smart-contract ecosystems can potentially be brought to layer-2, Ethereums blockchain only being used to commit and emit state. Different zk-rollup systems can also interact with each other in a trustless and decentralized manner, ensuring composability among smart-contracts, even if deployed on different zk-rollup systems. The literatur on Zinc and the utilization of recursive PLONK is far from ideal, so it must be taken with a grain of salt. Given that Matterlabs, the company behind zkSync and Zinc, is experienced in this field and the potential implications of this technology, its worth mentioning at this point. A testnet utilizing early versions are also online, with an example app being deployed.

Bibliography

[1] Curve on zinc, https://medium.com/matter-labs/curve-zksync-l2-ethereums-first-user-

[2] Uniswap cummulative volume, https://duneanalytics.com/projects/uniswap

[3] zinc, https://zinc.zksync.io/

[4] What are zk-snarks? (Sep 2019), https://z.cash/technology/zksnarks/

[5] zksync (Dec 2019), https://medium.com/matter-labs/introducing-zk-sync-the-missing-l

[6] Loopring's zksnark prover optimizations (Mar 2020), https://blogs.loopring.org/looprings-zksnark-prover-optimizations/

[7] Adams, H., Zinsmeister, N., Robinson, D.: Uniswap v2 core. URl: https://uniswap.org/whitepaper.pdf (2020)

[8] Albrecht, M., Grassi, L., Rechberger, C., Roy, A., Tiessen, T.: Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 191–219. Springer (2016)

[9] Baylina, J., Bellés, M.: Eddsa for baby jubjub elliptic curve with mimc-7 hash

[10] Benet, J.: Ipfs-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561 (2014)

[11] Deml, S.: Efficient ecc in zksnarks using zokrates (Aug 2019), https://medium.com/zokrates/efficient-ecc-in-zksnarks-using-zokrates-bd9ae37b818

[12] Ethereum gas guzzlers, https://ethgasstation.info/gasguzzlers.php

[13] Ethereum: eip-1108 (Sep 2020), https://github.com/ethereum/EIPs/blob/master/EIPS/eip-

[14] Ethereum gas price, https://etherscan.io/chart/gasprice

[15] Uniswap total fees used, https://etherscan.io/address/0x7a250d5630b4cf539739df2c5dacb

[16] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. IACR Cryptol. ePrint Arch. **2019**, 953 (2019)

[17] Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: Proceedings of the 30th USENIX Security Symposium. USENIX Association (2020)

[18] Kulechov, S.: The aave protocol v2 (Dec 2020), `https://medium.com/aave/the-aave-protocol-v2-f06f299cee04`

[19] Szydlo, M.: Merkle tree traversal in log space and time. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 541–554. Springer (2004)

[20] Udovenko, A.: Optimized collision search for stark-friendly hash challenge candidates (2020)

[21] Whinfrey, C.: Hop: Send tokens across rollups (2021)

[22] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)

[23] Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: {DIZK}: A distributed zero knowledge proof system. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 675–692 (2018)

[24] Zhou, L., Qin, K., Torres, C.F., Le, D.V., Gervais, A.: High-frequency trading on decentralized on-chain exchanges. arXiv preprint arXiv:2009.14021 (2020)