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.

0.1 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

0.1.1 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. **??**, 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. **??**, 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.

0.1.2 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. **??**.

0.1.3 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. **??**, 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. **??**, 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. **??**, 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.

0.1.4 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.

0.1.5 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.

0.2 Limitations

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

0.2.1 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.

0.2.2 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.

0.2.3 Trusted Setup Phase

The non-interactive nature of zkSNARK requires a common reference string to be shared among prover and verifier [?]. 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 [?]. This requires only one party of the computation to be honest, which is deemed secure with enough participants.

0.3 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.

0.3.1 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.

0.3.2 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. 0.3.1 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.

0.3.3 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

0.3.4 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.

0.3.5 Sandwich Attacks

A sandwich attack, as described by Zhou et al. [?] 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' [?] 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.