This section will analyze what implications the presented results have in practice. We will start by interpreting the results based on the different metrics previously introduced. These metrics will give a good understanding of the current bottlenecks in zk-rollup based applications. At the end of this section, we will cover the limitations and weaknesses of the proposed design and explain 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 analyze the results, considering the gas savings of the systems and several performance metrics of the zkSNARK circuits. As this system's primary goal is to reduce gas per trade on Uniswap, we will begin with these results. After, we will look at the performance results of the zkSNARK circuits, which gives a good understanding of the current limitations of the technology.

0.1.1 Gas results Trades

The most important metric of this system is the gas amount per trade that can be achieved. After all, reducing gas is the primary goal of this system. Using the biggest possible batch size, a trade would cost 6886 gas, which is a significant reduction compared to an unaggregated trade. 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. Simultaneously, a batch size of 200 already reduces the gas amount per trade to 9036. Looking at the results presented in F. **??**, we see that the batch size does not reduce the cost per trade linearly. Non-linearity is good news as it results in small batch sizes already having substantial gas savings. The break-even point per aggregation batch is reached once four trades are included in a batch. The low break-even point results in overall cost per trade to be reduced once reached, as shown by the curve in F. **??**. Overall these are promising results. We reach our break-even point quickly, while the cost per trade drops sharply, even with small batch sizes. 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 required.

0.1.2 Gas results Deposits and Withdraws

Another aspect to consider is the cost of deposits and withdraws. A user needs to deposit funds in order to use the system. The cost per deposit and withdrawal is different depending on the asset that is being moved. When moving Ether, each additional deposit adds 44k gas, and each withdrawal 18k gas to the overall gas cost of a batch. The cost of Ether deposits and withdrawals is acceptable, given that the fixed fee of an Ethereum transaction is 21k gas.

Moving ERC20 tokens is significantly more expensive. Since ERC20 token balances are represented as a mapping in the tokens smart-contract, moving them requires the state to be updated in the respective smart-contract. Updating a smart-contracts state is expensive on the Ethereum blockchain. Depositing ERC20 funds also requires two separate transactions. The first is to set an allowance for the zkSwap smart-contract, the 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 withdrawal 65k gas. While there is no clear path for reduction, a potential solution will be presented in S. **??**.

0.1.3 Circuit results

The performance of the circuits has to be measured differently than the trades. It must be remembered that the gas usage for verifying a zkSNARK proof is fixed and does not increase with the complexity of the circuit. However, the computational complexity of the proving steps increases with the size of the circuits. The larger a circuit is, the more computationally demanding the proving steps are. These results can be quantified by measuring the execution time and the memory requirements.

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 that is a long time, it does not have to hinder a practical application. In 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 price changes, given the right incentives for the aggregator, cannot 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 processed constraints per second or reduce the constraints of the circuit. Several promising approaches are currently in development and would enable the processing of far larger batch sizes. A number of these approaches will be presented in S. **??**.

Memory Requirements: Another consideration to make when analyzing the circuit's 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 additional element in 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 significant bottleneck for using large circuits in practice.

0.1.4 Results overall

Looking at the system overall, we can measure promising results. In the system's current form, we can reduce the cost per trade to around 20k gas. That is a reduction between 78% and 88%, compared to an unaggregated Uniswap trade while 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 many changes. The variable gas amount per trade can still be reduced with more optimizations, though the results measured in this implementation are overall satisfying already. Increasing the batch size would reduce the gas cost per trade even further. Increasing the batch size would require improvements of the zkSNARK circuits performance. We will explore several approaches in S. **??**.

0.1.5 Usability of the System

The results overall look promising on paper. Another consideration to make is the usability as a product from a users perspective. The obvious consideration to make is the delay of trade execution the aggregation is causing. An aggregation batch can be seperated into two stages: trade collection and aggregation exection. Both of these stages add a delay to the execution of a trade. On the other hand, network congestion can cause similar delays. A user might not choose to set the gas price high enough to ensure a quick execution. Since this system reaches its break-even point quickly, the aggregator could set the gas price high enough for a quick execution. This could result in instances, where the aggregation is quicker then an unaggregated trade.

0.2 Limitations

Having covered the overall results, we also have to consider a number of limitations that the proposed design has.

0.2.1 Fixed Batch Size in Circuit

The main limitation of the proposed design is the statically typed nature of zkSNARK circuits. When creating a circuit with ZoKrates, it feels like programming with a Turing complete programming language. However, this is not the case. When compiling a ZoKrates program, the circuit will represent every path through the program. When computing the witness and generating the proof, every constraint, and thereby every path, will be evaluated, adding to the complexity of the circuits. For example, when hashing a Merkle pair, we use an If/Else construct to decide which hash is on the left and right

sides. Under the hood, the circuit is built for both paths, which results in the constraints being doubled. The statically types nature also requires us 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 different batch sizes. Compiling multiple circuits results in a separate verifier smart-contract needing to be deployed for each circuit, which costs gas for deployment and complicates the verification of a batch.

0.2.2 State Updates During Aggregations

A running aggregation batch can be seen as a blocking process in our system. When aggregating a batch, we are essentially ensuring the state updates are done correctly. To ensure the states are updated correctly, the circuit receives a pre- and post aggregation state and checks if the state transitions follow the rules defined 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. A mechanisms to block instant withdraws during an ongoing aggregation needs to be developed.

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 that 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, where the toxic waste could be stored secretly. Approaches for trustless setups are being developed, for example, by multi-party computations (MPC) as described by the ZCash team [**?**]. The described MPC approach only requires one party of the computation to be honest, which is deemed secure with enough participants.

0.3 Open Problems

In this section, we will look at the open problems that have become apparent while building this system. None of these problems seems impossible to solve but were not focused on in this research and remain open.

0.3.1 Canceled Aggregations

When a price of an asset changes, exceeding the defined price range, the aggregation is canceled. The trade can not be executed for the defined worst-case price, which means the user's 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 no one can update the price range 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. A proof construct of some kind must be used to verify a batch has been canceled, preventing malicious worst-case price changes. 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 for dealing with empty batches must be created. The two main problems explained in S. 0.3.1 apply here as well. 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 choose to ignore deposits of a specific user. When a user makes a deposit, the funds are sent as an on-chain transaction 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 ignores a user's deposit. The funds are held in the zkSwap smart-contract, but the user has no custody until the deposit is added to a batch. This is no problem when dealing with trades or withdraws, as a user can not lose custody of its funds.

0.3.4 Ensuring Correct Effective Price Reporting by Aggregator

After the net-trade is executed on Uniswap, the new balances should be updated according to the effective price. In the current design, this can not be enforced, however. The aggregator could theoretically always use the worst-case price and keep the remaining funds to itself. Failing to enforce this is a problem, as a user would expect to receive the worst possible price when trading using the system. This problem could be solved by executing the Uniswap trade from the zkSwap smart-contract instead of the PairProxy smart-contract. The effective price can be stored and used to ensure the correct price was used in the aggregation batch. Executing the trade from the zkSwap smart-contract, however, introduces several new problems. When ex-

ecuting the trade from. the zkSwap smart-contract, there are three realistic[1] combinations of who is paying for the trade and where the funds are stored until the aggregation batch is verified:

1. **Aggregator pays, Aggregator stores** - The trade execution function must be guarded, as anybody could change the effective price by triggering a trade during an ongoing aggregation, invalidating the batch.

2. **Aggregator pays, zkSwap stores** - An indirect guard is achieved, as a malicious actor would lose its funds when triggering a trade. However, any change in the effective price would result in the invalidation of a batch, so large amounts of capital are not required.

3. **zkSwap pays, zkSwap stores** - This approach could result in the insolvency of the zkSwap smart-contract. The aggregator could decide to stop aggregation, which would result in the balances not being covered.

Looking at the problems that arise, finding a solution is not trivial. It must be considered that the role of the aggregator influences the problems stated above. For example, the aggregator could be required to register to be able to aggregate batches. This aspect was omitted for the context of this work, as it warrants research of its own.

0.3.5 Role of the Aggregator

The role of the aggregator also has to be defined more clearly. For one, it must be decided how many aggregators are part of the system. The system could be envisioned with a pool of aggregators, all competing with each other on speed and their fees. Another approach would be a single aggregator, that is operated by the developers of the system. The fee structure is also an important aspect, especially for incentivizing empty or canceled batches to be reported. The potential approaches and their implications on the system require research of their own, which is why they where omited for the context of this work.

0.3.6 Sandwich Attacks

A sandwich attack, as described by Zhou et al. [**?**] is an attack that targets decentralized exchange transactions. The basic idea of a sandwich attack is to influence a trade transaction by having one transaction executed before, one after the actual trade. Sandwich attacks can be used either for profit or to prevent the successful execution of a specific transaction. When the net-trade of an aggregation batch is executed, an attacker can analyze the transaction in the mempool and estimate the price impact of that trade. By setting a higher gas price, the attacker can front-run [**?**] the original trade transaction,

[1] zkSwap pays, Aggregator stores is omitted, as there are obvious problems

having it executed before. The attacker 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. Sandwich attacks can also be used 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.