This section will cover the general outlook and related work of zk-rollup based applications. As we have discussed the results, we will begin by covering several approaches that would allow larger batch sizes to be processed in the system's current design. Once these have been presented, we will explorer several developments that show promise in improving zkSNARKs and by extension zk-rollup based applications.

0.1 Prover optimizations

The main bottleneck of this system's performance is the prover. Speeding up the prover would enable much larger batch sizes, which would reduce the cost per trade even further. Another significant improvement would be the reduction of memory usage of the prover. Loopring, a zk-rollup based decentralized exchange, claims to have achieved that. We will look at the improved performance the Loopring team has claimed to have achieved [**?**] and compare them to the performance metrics measured in this system.

0.1.1 Parallelizing the Prover

Loopring was able to parallelize the prover. As we have previously discussed, the main bottleneck of a circuit's performance is the CPU. Loopring uses the libsnark as a proving library, which can be used with ZoKrates as well. 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 noted, however, that other optimizations were already implemented before parallelizing. When running in parallel, Looprings claims 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 can 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. Achieving this processing speed would significantly improve the usability of zk-rollups, so research in this direction should be done. In this context, Wu et al. [**?**] 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, the claims made by the Loopring team seem achievable.

0.1.2 Reducing Memory Usage

Another aspect that needs to be improved is the memory requirements 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 cannot be compared directly, as a different hashing function is used in Looprings implementation. Hashing functions can impact

the memory requirements as they have different linear combination lengths. The techniques used should result in a positive effect on our system as well. For one, memory requirements can be achieved by not storing every coefficient independently. Loopring reduced the number of coefficients stored while generating the proof from around 1 billion down to just over 20k. The memory is further reduced by not storing each constraint independently. Since most of our constraints are caused by the hashing functions, we have many duplicate constraints that work on different variables. Reducing our memory requirements by a similar amount would greatly improve the batch size in practice.

0.2 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-invertible, 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, compressing the zkSNARK proof, and hashing deposit values in the zkSwap smart-contract. This system would not work without hashing algorithms. There is currently 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.

0.2.1 MiMC on Ethereum

By reducing the multiplicative complexity, the MiMC hashing algorithm can be efficiently used in a zkSNARK circuit. As shown in S. **??**, the constraints required per hash are significantly lower than SHA256, which speeds up the proving steps. Conversely, the MiMC hashing algorithm requires much gas per hash, while the SHA256 hashing algorithm does not. This is a limiting factor, as it requires us to make tradeoffs between the hashing functions. For example, in this project, we use MiMC hashes for the Merkle tree and a SHA256 hash for the data hash. While this does use a minimal amount of gas, creating the data hash with SHA256 doubles the constraint count of our circuits. Simultaneously, this also makes the instant withdraws, which must use MiMC hashes on-chain prohibitively 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 hard fork included EIP-1108 [**?**]. This improvement proposal reduced the addition and multiplication operations on the BN254 curve, which are often used during the verification of a zkSNARK proof. A similar approach seems likely. However, since the MiMC algorithm is still relatively new, the algorithm has to be studied closer.

0.2.2 Poseidon Hashes

The Poseidon [**?**] hashing algorithm is a novel hashing algorithm that aims to be efficient in zkSNARK circuits. A 128-bit hash with an arity of 2:1[1] only adds 276 constraints per hash. This is significantly less than 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 accomplished 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. A similar approach described in S. 0.2.1 can be applied here as well. Poseidon was not used in this work, as a collision was found by Udovenko [**?**]. 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 equal on-chain cost as SHA256, would reduce the amount of constraints for the data hash by 99.85%. Poseidon shows great promise for increasing the throughput of zk-rollup enabled applications.

0.3 PLONK

PLONK [**?**] is a universal proving scheme that could significantly improve the usability of zero-knowledge protocols. PLONK increases the usability of zero-knowledge protocols because it enables the common reference string generated during the setup to be used by any number of circuits. Using the same common reference string enables a single verifier to verify any number of different circuits. PLONK solves a big challenge that arises when working with zero-knowledge protocols. In this system, for example, a big limitation is the static nature of a circuit. We always need to pass the exact number of arguments specified in the circuit to execute it. The static nature is very unflexible and requires us to compile and set up many variations of our circuit to make sure we can work with a changing number of inputs. While this works in theory, it also requires us to deploy a separate 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 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 slightly, which will increase the gas needed for the on-chain verification step. Other changes in performance must be explored and tested.

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

0.4 zkSync and the Zinc Programing Language

zkSync [**?**] is an 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 transfers aggregated with zk-rollup. It is one of the few systems utilizing 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 [**?**], 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 functions known from Solidity, while making 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 programs are deployed to zkSync layer-2 network and are verified recursively with the zkSync circuit. Zinc programs can call each other, offer the same composability known from Ethereum main chain. When calling a Zinc program, a validator is picked to compute the witness and generate the proof. Several proofs can then be verified recursively, 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 [**?**] 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, it can be argued that it is important to mention this work. However, it must be taken with a grain of salt.

0.5 Cross zk-rollup Transactions

Another technology currently in development is cross zk-rollup transactions, as described by C. Whinfrey [**?**]. The main idea is to connect different zk-rollup applications and enable transactions between them without moving funds over the main chain. Cross rollup transactions can be achieved by having intermediaries that deposit funds in two zk-rollup applications. A user can then send funds to 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 a decentralized fashion. Cross rollup transactions are an important development, as it makes balances transferable between different zk-rollup enabled applications. Without this, zk-rollup would be a questionable scaling solution. It would break the composability of applications and require funds to move through the main chain when transferring to a different zk-rollup application. As mentioned in the results, deposits and withdrawals also require a large amount of gas, with no real path of improval.