In this section we will take a look at

0.1 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 [?] and compare them to the numbers 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 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. [?] 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.

0.1.2 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 coeffecent 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.

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-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.

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 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 [**?**]. 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.

0.2.2 Poseidon Hashes

The Poseidon [**?**] hashing algorithm is 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 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. 0.2.1 can be applied here aswell. 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 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.

0.3 PLONK

PLONK [**?**] 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 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.

---

[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 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 [**?**], 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 [**?**] 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.

0.5 Cross zk-rollup Transactions

Another technology currently in development are cross zk-rollup transactions, as described by C. Whinfrey [**?**]. 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.

0.6 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.