zkSwap - Scaling Decentralized Exchanges through Transaction Aggregation

Paul Etscheit

April 2021

Abstract

The rising popularity of decentralized applications is quickly pushing the Ethereum blockchain to its capacity. With longer-term scaling solutions expected to take years to complete, and the popularity of decentralized applications not looking to slow down, a shorter-term solution is needed. Zk-rollup is one of these scaling solutions, developing quickly and promising to scale the blockchain. Zk-rollup utilizes zkSNARK proofs to enable trustless, off-chain transaction aggregation that is verified on-chain as on transaction. In this work, we build a system capable of aggregating numerous decentralized exchange trade transactions to be executed as one while remaining trustless and permissionless. Aggregating numerous trade transactions benefit the Ethereum blockchain, as less transaction needs to be processed. Users also benefit as the transaction fees are reduced. We will explore the problems that arise when integrating with third-party decentralized applications, the benefits of applying this technology, and the promising outlook of the technology.

Zusammenfassung

Die Popularität von dezentralen Anwendungen bringt die Ethereum Blockchain schnell an ihre Kapazitätsgrenzen. Da längerfristige Skalierungslösungen voraussichtlich noch Jahre in der Entwicklung sind, werden kurzfristige Lösungen benötigt. Zk-rollup ist einer dieser Skalierungstechnologien, die mit fortschreitender Geschwindigkeit entwickelt wird, und das Potential hat maßgeblich bei der Skalierung eingesetzt zu werden. Zk-rollup basieren auf zkSNARK-Beweisen, die eine Off-Chain Aggregation der Transaktionen ermöglicht, ohne der aggregierenden Partei zu vertrauen. In dieser Arbeit bauen wir ein System, das in der Lage ist, Handelstransaktionen von Dezentralen Krypto-börsen zu aggregieren. Das resultiert in einer Verminderung der Transaktionen auf der Ethereum Blockchain, was das Netzwerk entlastet und zu niedrigeren transaktionsgebühren der Nutzer führt. In dieser Arbeit analysieren wir die Probleme, die durch eine Integration mit öffentlichen dezentralen Anwendungen auftreten, der potentielle Nutzen der Technology und die generellen Aussichten.

Keywords: Zk-Rollup , zkSNARK, Ethereum

Contents

List of Figures

1 Introduction

When launched in 2015, Ethereum [36] was envisioned as a trustless, permissionless, and decentralized world computer, accessible to anyone. What sets Ethereum apart from other blockchains like Bitcoin [28] is the concept of smart-contracts. Smart-contracts are custom, Turing complete programs that are deployed and executed on the Ethereum blockchain in a trustless and permissionless manner. Anybody can interact with these programs by sending transactions to the smart-contract, which results in execution. Smart-contracts can also interact with each other, enabling permissionless composability between them. The technical limitations of this technology have become apparent as well. Computations are expensive, theoretical transactions per second are low, and the overall transaction throughput has been stagnant. While sharding approaches give a direction to scale the network, they are expected to take years to complete.

Tokenization The first major use case for Ethereum was tokenization. With the development of the ERC-20 standard [3], launching a token on the Ethereum blockchain is trivial. As tokens run as smart-contracts on the Ethereum blockchain, they are secured by its proof of work consensus. Given Ethereum's proof of work hash rate, consensus attacks are infeasible. Running on Ethereum blockchain is a significant benefit, as network security can be assumed. Tokenization, however, is also a rather unexciting use case. Most tokens are isolated in their functionality and ecosystem and lack productive usage.

Decentralized Finance A new use case of Ethereum has become apparent in the last months, namely decentralized finance (DeFi). At its core, DeFi protocols aim to productively utilize tokenized assets, somewhat mirroring the traditional financial system in a decentralized fashion. Aave [27] enables lending and collateralized borrowing of tokenized assets. Assets can also be deposited into liquidity pools on Uniswap [8], generating yield for the user and enabling decentralized and non-custodial trading. Novel concepts like flash loans have been developed that enable uncollateralized and interest-free loans for the duration of one transaction. It can be questioned how useful or necessary these protocols are. The idea behind them is impressive non the less. As a result of their popularity, these protocols are overwhelming the Ethereum blockchain, increasing the costs of transactions significantly.

One of these new DeFi applications is Uniswap [8]. Uniswap is a crypto-asset exchange running as a collection of smart-contracts on the Ethereum blockchain, enabling non-custodial, trustless, and permissionless trading of ERC-20 assets. Uniswap has grown in popularity in recent months, regularly processing over $1 billion in daily trading volume. The popularity has its toll on the Ethereum blockchain, using around 15% of Ethereum block space throughout the last months. 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.

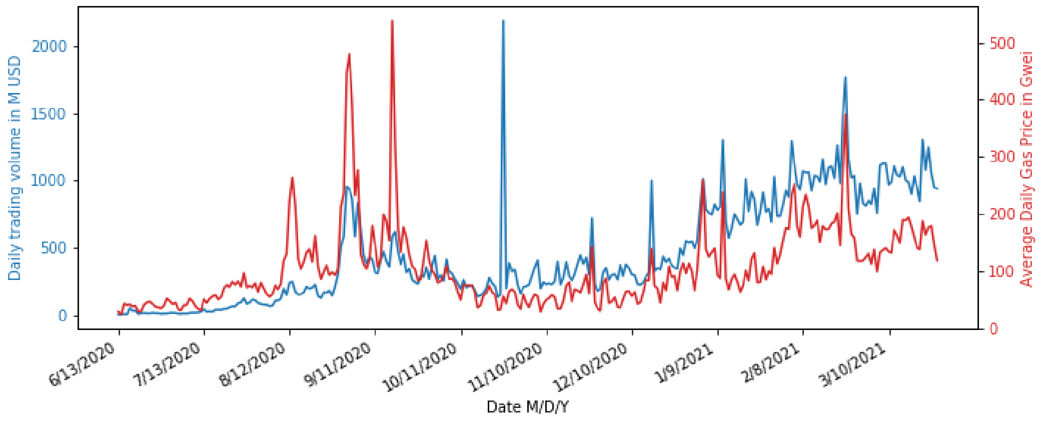


Figure 1: Combined daily Uniswap trading volume average daily Ethereum gas price

With longer-term scaling solutions in development but still years away, a shorter-term solution is needed. In this work, we will explore zk-rollup as a potential scaling solution for Uniswap trade transactions. The goal is to collect Uniswap trade orders in batches, offset them with each other, and then execute them as one on-chain Uniswap trade. This system aims to remain trustless and permissionless while reducing the transaction cost per trade. Several zk-rollup based applications are running in production, however, in their current state, they are isolated applications that do not rely on other smart-contract interactions to functions. In this work, we will attempt to integrate Uniswap into a custom-built zk-rollup application. The potential issues that arise from being dependant on another application indicate the problems of this approach in general.

2 Background

In this section, we will cover several topics relevant to the context of this work. We start by looking at the current state of decentralized exchanges, what sets them apart from centralized exchanges, and the problems they are currently facing. Next, we will look at different scaling solutions, classify the general approaches and explain what sets them apart. The last topic of this section will cover the fundamentals of zero-knowledge proofs and how they are relevant to this work.

2.1 Decentralized Exchanges

In recent months, decentralized exchanges have sharply grown in popularity. Instead of relying on centralized server infrastructure, as traditional crypto-asset exchanges do, decentralized exchanges execute the trade logic in a decentralized, permissionless, and trustless manner. The most popular decentralized exchange is Uniswap, which runs as a collection of smart-contract on the Ethereum blockchain. There are several competing decentralized exchanges running on the Ethereum blockchain, all generally functioning the same way. In this work, we will focus on Uniswap, competing platforms, however, could be integrated without much work.

2.1.1 Benefits of Decentralized Exchanges

By being decentralized, Uniswap offers a number of benefits to the user compared to centralized crypto-asset exchanges. We will cover the main ones here, namely custody of funds and the permissionless nature of the application.

Custody of Funds When using a decentralized exchange, a user does not have to send its funds to another wallet. Deposits are unnecessary, as a user trades straight from its wallet, keeping custody over its funds. When making a trade, a user sends a transaction, attaching the funds looking to exchange. As Ethereum transactions are atomic, custody is not lost during trading. Either the trade transaction is successful, which results in the funds showing up in the wallet, or the transaction is reverted, in which case funds sent for trading stay in the wallet. Keeping custody of funds is a significant benefit compared to centralized exchanges. Users looking to trade on a centralized exchange first need to deposit funds into the exchange's wallet. While the funds will show up as a balance in the user's account, custody of funds is lost. A user needs to trust the exchange to honor withdrawal requests. The wallets of exchanges contain a large amount of funds at all times. Simultaneously, the wallet balances are public knowledge, resulting in centralized exchanges being targeted by hacking attacks. These hacks are successful frequently, mainly resulting in affected users losing their funds.

Permissionless These systems are also permissionless by design. No entity can change the trading logic at will or even prevent people from using the platform. For example, anybody can add a new trading pair to Uniswap by using the user interface. It is a straightforward process that anybody with broad knowledge of the Ethereum ecosystem can do. Pair additions can not prevented by anyone. The same applies to trading: it is a permissionless system, and nobody can be stopped from using it. Uniswap can somewhat be seen as a utility service, available for anyone to use at will, living on the Ethereum blockchain forever.

2.1.2 Pricing and Liquidity

In exchange platforms, centralized and decentralized alike, the liquidity of trading pairs is essential. A pair's liquidity is defined by the liquid assets that can be traded on the platform immediately. Assets of a trading pair are considered liquid if they are priced close to the reference price, realistically having a chance to be traded in expected market conditions [1]. Efficient pricing of traded assets is an essential aspect of successful exchanges, greatly influenced by available liquidity. While liquidity is a universal concept of different exchanges, an asset's price can be defined a number of different ways, two of which we will explore briefly.

2.1.3 Centralized Order Book

Centralized exchanges typically rely on the central order book to facilitate the trading of an asset pair. An order book consists of two sides, the buy and the sell side, where users can add orders. When adding an order, a user must decide on three things: a buy or sell order, the amount to be bought/sold, and a price acceptable to the user. Once the order is added, it can be found in the order book on the respective side. By residing in the order book, we have increased the amount of liquid assets, thereby adding liquidity. It is important to note that adding the order does not automatically execute the specified trade. A trade order can be executed by another user taking the order found in the book or by a matching engine looking for orders that can be matched and executed. It is easy to see that this approach is memory intensive and requires constant processing of incoming orders. While running this on modern server infrastructure is no problem, it is infeasible on the Ethereum blockchain. Memory and processing are limited and expensive to use in smart-contracts, so a more efficient model must be found.

2.1.4 Automatic Market Maker Protocol

Uniswap solves the computational restrictions by applying the Automated Market Maker (AMM) protocol. Instead of relying on a central order book to represent liquidity and price assets, AMMs utilizes liquidity pools. A liquidity pool always contains both assets of the trading pair, containing an equal value of both assets. The fact that a liquidity pool always contains an equal value of both assets enables us to provide liquidity and define prices in a very efficient manner.

Constant Product Formular The price of an asset pair on Uniswap is defined by the constant product formula:

$$k = x \cdot y$$

[1]For example, adding a large Bitcoin buy order at a price of 1$ does not add liquidity.

where:

x is the amount of asset X

y is the amount of asset Y

To determine the current price of asset X, we calculate the following:

$$P_x = \frac{y}{x}$$

This mechanism enables prices to be determined by a simple calculation viable to be executed in a smart-contract. When a trade is executed, the trader adds one asset to the pool and receives a corresponding amount of the other asset. The amounts and implied prices of the assets change, while the product of the liquidity pool remains unchanged.

Liquidity in Uniswap pools When a trade is executed, the asset prices of the pool change in a predictable manner. The impact a trade has on a pools price is dependant on the size of the trade and the amount of assets in the pool. The asset price of a pool containing assets worth millions of dollars will be impacted less by a trade than a pool containing a couple of thousands of dollars. The way liquidity is represented is very different compared to a central order book. The effect liquidity has on efficient pricing, however, remains the same. Higher liquidity of assets results in more efficient pricing. Uniswap relies on a separate class of users for liquidity, the so-called liquidity providers. A liquidity provider can deposit funds into the liquidity pool, providing an equal value of both assets. In return, the liquidity providers receive liquidity provider tokens. These tokens represent the share of funds in the pool owned by that user. Liquidity providers are incentivized to deposit funds by the 0.3% liquidity provider fee Uniswap charges for each trade, receiving a proportional share based on the amount of liquidity provider tokens held. Depositing funds into a liquidity pool is also considered non-custodial. The smart-contracts code is open source and can be verified by anyone. Funds can only be withdrawn from a pool by burning the liquidity provider token, which resides in the liquidity provider's wallet. It must be considered that the smart-contract can contain bugs, potentially resulting in the funds being stolen.

Price Convergence to Reference Rates The last concept to explain is how the prices defined by the constant product formula stay in sync with the reference rates on other exchanges. In a central order book, prices move in different directions based on the available liquidity on each side of the order book. If the buy-side contains little liquidity close to the current pricing, odds are the price is going to fall, mirroring the law of supply and demand. The way Uniswap represents liquidity prevents this. By definition, we always have an equal amount of liquidity on both sides. The amount of liquidity available is always symmetric for both sides. The liquidity available 100 dollars above and below the current price is the same. For this reason, Uniswap relies on arbitrage to converge with reference rates. An arbitrageur constantly monitors the price differences between reference markets and Uniswap. If, for example, Ether trades at a $100 premium on Uniswap, the arbitrageur can buy Ether at a reference market and sell it on Uniswap. The arbitrageur makes a profit, while the Uniswap price converges with the reference rates. Angeris et al. [10] have shown that this simple mechanism results in very accurate pricing on Uniswap. Uniswap successfully built a fully decentralized exchange that has processed billions of dollars in trading volume by using these simple mechanisms. However, the rising usage of the Ethereum blockchain has resulted in trading transaction cost rising significantly in recent months. Several aggregation techniques have been proposed, aiming to reduce the cost per trade, one of which we will be exploring in detail.

2.2 Ethereum Scaling Solutions

Over the years, several scaling solutions have been proposed to increase the transactional throughput while reducing the cost of transactions. Blockchains can be scaled in two different ways, layer-1, and layer-2 scaling. We will begin by looking at the different layer-1 approaches briefly and then dive into the layer-2 approaches in more detail.

2.2.1 Layer-1 Scaling

A blockchain's throughput can be described best by looking at the rate at which data can be processed. New blocks are typically sealed at a roughly fixed interval and have a maximum size limitation. These numbers can be used to calculate the maximum data throughput of the blockchain network. The goal of layer-1 scaling is to increase that data throughput. Layer-1 scaling can be roughly separated into two different approaches: horizontal and vertical scaling.

Vertical Scaling In IT systems, a simple scaling solution is upgrading the hardware of a server. In blockchain systems, this can be compared to increasing the block size limit. By allowing more transactions to fit in a block, the throughput increases. While this approach is easy to implement, it is not a sustainable approach over the long term. Eventually, large block sizes will result in centralization or an unstable network. Like traditional software systems, the speed up expected by this approach is limited to a certain point.

Horizontal Scaling Another scaling approach that is applied in traditional software systems is horizontal scaling. In horizontal scaling, we aim to parallelize our system, resulting in much higher data throughput. In blockchain systems, this can be achieved by sharding the network. Instead of every node processing every transaction, we separate the network into shards, processing a different set of transactions. Increasing the number of shards now increases the data throughput as we are parallelizing transaction processing. However, the shards still need to operate under the same consensus mechanism. The overhead of forming the consensus does not increase linearly with the number of shards[2]. This quadratic overhead results in a non-linear relationship between the number of shards and the increases in throughput. A new, sharded version of Ethereum is currently in development. While the details of the are not finalized yet, it is said to contain 1024 shards. Until the sharding has reached maturity and its full capability is said to take years.

2.2.2 Layer-2 Scaling

Another approach to increasing the transactional throughput is to reduce the data required to represent and execute a transaction on the blockchain. Instead of publishing all transactional data on the blockchain, the general goal is to publish a compressed form. The remaining data is published in a layer-2 network. Layer-2 is a loosely defined term, used to describe a secondary protocol built on top of a blockchain. The state of layer-2 is committed to the underlying blockchain, ensuring the correctness of the state can always be verified. A variety of mechanisms can be applied to ensure correct state updates. We will explore the different types of layer-2 systems that have been conceptualized and built over the last couple of years.

State Channels A state channel [16] is a scaling technique that enables practically gas-free transactions. This technique represents the transfer of funds between two participants by exchanging signed transactions of a mutually controlled multi-wallet containing their funds. Transfers are done by signing a transaction that represents a withdraw of funds according to the post-transfer balances of the participants. The signed transactions, however, do not have to be broadcasted to the network. Participants can exchange an unlimited amount of them, resulting in gas-free transfers. Opening and closing a state channel results in on-chain transactions. Fortunately, different channels can be connected through other participants. State channels can be a useful technique for handling frequent and repetitive payments. Several disadvantages limit the potential. For one, participants need to open a state channel connected to the sender to receive payments. The mechanism to handle non-responsive participants also requires participants to be online to prevent fraudulently broadcasted withdrawal requests.

Plasma Plasma is a scaling technique proposed by Poon et al. [30] in 2017 that enables the creation of plasma chains. Every plasma chain is managed by a smart-contract that is deployed on the Ethereum blockchain. To move

[2]This is a common problem in proof of stake systems and described by Silvia et al. [31]

funds onto a plasma chain, a user deposits them into the corresponding smart-contract, which credits the funds on the plasma chain by storing the state in a Merkle tree. Transfers in the plasma chain are batched by an operator, who creates a Merkle tree containing the transactions and stores the Merkle root in the managing smart-contract at a regular interval. The correctness of plasma transactions is ensured by a challenge period. Anyone can challenge the correctness by submitting a Merkle fraud-proof and invalidate the batch. The finality of transactions is reached after the challenge period. As with state channels, the correctness of state transitions is reliant on every participant checking if their balance has been updated correctly. This is a powerful technology for scaling simple transfer of assets but is challenging to apply in applications that depend on state without a logical owner [15].

The Data Availablity Problem At the core of all layer-2 scaling problems lies the data availability problem. Plasma, for example, relies on the operator to update the state root and propagate all included transactions to the plasma users. Users receive their transactions and can locally check if the published root is correct. If the roots mismatch, a fraud-proof can be submitted. Nevertheless, what happens when the operator includes a malicious transaction to compute the state root but does not propagate that transaction? We are not able to create a fraud-proof on data we do not have. Ensuring all data is published can be enforced to a separate entity, the fisherman, that checks each block for missing data. If the fisherman raises the alarm on missing data, the operator can quickly publish the missing data. A fisherman could, however, also raise the alarm maliciously. For other nodes, it is impossible to determine if the operator withheld data maliciously or the fisherman raised the alarm maliciously once the data has been published. A game-theoretical problem known as the fisherman problem [18] arises from this. There are three potential ways for a fisherman to be rewarded for raising the alarm: taking a loss, breaking even, or making a profit. All outcomes result in malicious behavior being incentivized in some way. If the fisherman takes a loss every time it reports missing data, an attacker can economically outlast the fisherman. In a break-even situation, missing data could be reported for every batch, resulting in clients needing to download the entire state from the blockchain. If the fisherman makes a profit, it would report missing data for each batch to profit. This is a compressed explanation of the data availability problem, as it is a complex topic. In this work, it is important to understand that storing data exclusively in layer-2 adds significant complexities that result in a game-theoretical problem that has not been solved at the time of writing. Plasma solves this by having a logical owner for all states, ensuring the correctness of their state. While this can work, it adds liveliness assumptions to the system and limits the scaling approach to applications where each state has a logical owner.

Rollup Rollup is a concept that was designed to solve the data availability problem of scaling techniques. In general, data availability can only be solved by having a trustless mechanism in place that publishes the data to a system that is always known to be online. In a blockchain system, this means the data needs to be stored on-chain. A rollup application is controlled by a smart-contract, handling deposits and withdraws of funds. The state of rollup applications is stored in three different places: the event log, layer-2, and the smart-contracts runtime. To store state, we emit the data, in highly compressed form, as an event[3]. The general idea is to replace data with computation wherever possible, compressing the data as much as possible. These events are synced in layer-2, constructing a Merkle tree. The Merkle tree root is stored in the smart-contracts runtime. The goal is to update serveral states at once, batching operations into one. The state can be updated by publishing a batch, emitting the new state as an event, and updating the Merkle root. As we store the state in the event log, we have now solved the data availability problem. One question, however, remains. How do we ensure balances are updated correctly? This is where the two different flavors of rollup start to differentiate.

Optimistic Rollup Optimistic rollup relies on fraud proofs to ensure the correctness of a new state root. The concept is very similar to plasma, the main difference being that all state updates are published on-chain. When a new batch is added, a challenge period starts. Clients now have the chance to submit fraud proofs. Since we always expect to have access to the entire state of the rollup application, we can detect missing data in a fraud-proof, without running into the game-theoretical problems that lie at the core of the data availability problem. As data is published on-chain, we can also represent a state that does not have a logical owner. This approach enables us to execute smart-contract logic in layer-2, relying on fraud proofs to ensure correct execution. These properties are a major advantage over the previously discussed techniques, as it is a general-purpose scaling solution. A disadvantage of optimistic rollup is transaction finality, as each state update is subject to a challenge period. There are several optimistic rollup-based scaling solutions set to launch soon.

ZK-Rollup ZK-Rollups, described by V. Buterin in 2018 [34], are another approach to rollups. Instead of relying on fraud-proof to revert malicious state updates, zk-rollups rely on validity proofs submitted along with a batch update. The validity proof verifies that the proof output was computed by using a specific zkSNARK circuit. The zkSNARK proof verifier is generated along with the circuit, having the sole purpose of verifying proofs generated by it. The proof object has a constant size, which allows us to build complex

[3]The event log can be used to store data cheaply. It will be explained in more detail in S. 3.2.2

off-chain logic without increasing the gas required for verification. Another benefit lies in the finality of state updates. Since a batch is only published if the proof object is verified successfully, it reaches finality instantly. Conversely, zk-rollup relies on very new cryptographic protocols that are still in development. In this work, we will focus on a zk-rollup based scaling solution for Uniswap trades. While zk-rollups are already applied in systems used for asset transfers, these systems do not integrate with other smart-contracts, relying on them to run an aggregation. This work will attempt to integrate a use-case-specific zk-rollup app with a running third-party smart-contract and explore the challenges posed by this integration.

2.3 Zero Knowledge Proofs

Zero knowlege proofs where first described by Goldwasser et al. [24] in 1989. Their work describes a protocol that enables a prover to convince a verifier of knowledge of a specific mathematical statement without revealing any information of that statement. This can be achieved with an interactive proof, where ongoing communication between prover and verifier is required. Each round of communication further convinces the verifier that the statement is true. Overly simplified, this can be seen as a game, where a computationally unbounded prover convinces a computationally bounded verifier that a certain statement is true. Since the entire proof is probabilistic, the amount of communication rounds influences how likely the proof is valid. This protocol relies on several properties to function [23]:

1. **Completness:** If the statement is true, the honest verifier will be convinced by an honest prover.
2. **Soundness:** If the statement is false, a cheating prover can't convince the verfier[4].
3. **Zero-Knowledgeness:** The verifier learns nothing about the proven statement, except that it is true.

Their system was able to prove quadratic residuosity to a verifier without revealing the factorization. Proving quadratic residuosity is very similar to proving knowledge of a factorization in the discrete logarithm problem, without revealing it. Given that discrete logarithm is one of the foundations of modern cryptography, this marks a breakthrough. This discovery was still very limited. For one, it was a solution that worked in theory but only under the assumption that the prover has unlimited computing resources. It was also domain-specific, only working for this specific problem.

The development of zero-knowledge proofs was later extended by Goldreich et al. [22], who proved that all NP problems have zero-knowledge proofs. All NP problems having a zero-knowledge proof relies on two assumptions:

[4]Except for a small probability, that can be reduced by increasing communication rounds.

the existence of one-way functions, which is an open problem in computer science until this day, and an unbound number of communication rounds between the prover and the verifier.

2.3.1 Development of zkSNARK

The real breakthrough in zero-knowledge proof protocols can be attributed to Gennaro et al. [21]. Their work made the proof non-interactive and the proof size constant, bringing the technology into the realms of practical applications. Parno et al. [29], and J. Groth [26] further improved the efficiency, reducing the proof size and verification complexity. These developments result in what is known as a zkSNARK proofs, which stands for "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge". zkSNARK enables the proof creation of an arbitrary statement, is non-interactive, has a fixed proof size of 288 byte and can be verified quickly with undemanding hardware. These properties now enable the practical application of the technology.

2.3.2 Foundations of zkSNARK

With zkSNARKs, arbitrary statements can be converted into zero-knowledge proofs and proven to a verifier without revealing any information on the statement. Essential is the constant proof size and the quick verification of the proof, even on restricted hardware. We will now go through the different steps of a proofs lifecycle, explaining the steps and what they are needed for. We will only cover the different steps briefly, as this is a field of research in itself.

Compilation of Circuit In this step, the statement is flattened, converted into a Rank-1 Constraint System (R1CS), and converted once again into a Quadratic Arithmetic Program (QAP). In this form, the initial statement is represented in degree-3 polynomials.

Trusted Setup When evaluating a proof, the verifier does not evaluate the entire statement but only random points. The idea is, that given that a prover is not aware which points are being evaluated cheating by guessing is unlikely to happen. Since zkSNARKs are non-interactive, the verifier cannot request the result of the circuit at different random points. For this reason, the evaluation point must be picked in advance, which happens during the setup. The setup can be seen as a function that receives the compiled circuit and a secret parameter and returns the proving key and verification key. Knowing the secret parameter enables the prover to generate fake proofs, which is why it is called 'toxic waste'. Toxic waste is the Achilles heel of every non-interactive zero-knowledge-proof protocol, and deleting it once the setup is complete is essential. Approaches are being developed to make this trustless that are mentioned in S. 5.2.3.

Witness Computation The compiled circuit is executed with given inputs. As a result, the witness is computed, assigning a value to each variable in the circuit. These values result in all constraints of the circuit being fulfilled.

Generate Proof The proof is generated by receiving the proving key, the public inputs, and the witness.

Verify The proof can be verified if the public inputs and proving key is used with it. If the verify program returns true, the verifier is assured the public inputs were computed using the circuit specified in the setup step.

2.3.3 zkSNARK and Blockchain

zkSNARKs offer potential solutions to the two main problems blockchains face today: dealing with private data and off-chaining computationally expensive processes in a trustless manner. In current blockchain systems, all state-changing transactions must be stored forever and are public by definition. Data also cannot be deleted, as it would corrupt the entire blockchain, making it impossible to verify. Being unable to keep data private limits the potential applications. The properties of zkSNARK offer a potential solution to this problem. Programs that work with private data can be implemented as a zkSNARK circuit and executed by the prover. The resulting proof can be verified on-chain, giving the same assurances an on-chain execution would provide while keeping sensitive data private[5]. The blockchain is not corrupted, as the proof will be stored on-chain forever while leaking no details of the private data used. The same properties can also be used to off-chain complex computations that are unviable to run on-chain. Given that our proof is succinct, we can create arbitrary-sized circuits, execute them on a central server, and then verify the proof on-chain. As the proof verifies a correct execution, the prover can be untrusted. This potentially has significant implications for scaling blockchain systems.

2.3.4 Other Zero Knowledge Constructs

There are several other zero-knowledge proof constructs that need to be mentioned. We will briefly look at the differences compared to zkSNARK but focus on the properties that make them inferior in a blockchain context for the applications stated above.

Bulletproofs Developed by Bunz et al. [14], Bulletproofs are non-interactive constructs that do not require a trusted setup. A downside compared to zkSNARKs is the non-succinct nature of the proofs and the complexity of the verification. The proof size increases with O(log(N)), while the verification

[5]Assuming the prover is trusted with the private data

complexity increases with O(N). This results in an expensive on-chain verification of the proofs, which is not ideal for the use cases outlined above.

STARKs Developed by Ben-Sasson et al. [12], STARKs are also non-interactive and do not require a trusted setup. Furthermore, the only cryptographic assumption made are collision-resistant hashing functions, which makes them post-quantum secure. Like Bulletproofs, the proof size is succinct, the verification complexity growing with the proof size. This results in expensive on-chain verifications.

3 zkSwap

This work aims to reduce the gas amount required for a Uniswap trade by utilizing the properties of zk-rollups. The prototype's goal is to aggregate trade orders of a single trading pair, Ether, and an ERC-20 token of choice. The system is also supposed to remain trustless and not rely on any external data availability to function. By aggregating and offsetting buy and sell orders of a specific pair, the trades will be executed as one, reducing the on-chain gas costs to one trade execution. Correctness of an aggregation batch is ensured by the properties of zkSNARK, allowing for the execution to be verified on-chain. In the proposed design, a Merkle tree with a depth of 16 is used, allowing a maximum of 65536 users in the system. We will now look at the proposed system design and explore the technical implementation after.

3.1 Design

In this section, we will explore the proposed design of the system, looking at the different entities, the functionalities they provide, and the interactions between them. After the high-level design is clear, we will look at the implementation, explaining the system in detail. This system is made up of two entites[6] that are required for the system to function: the zkSwap smart-contract and the aggregator.

3.1.1 zkSwap Smart-Contract

The zkSwap smart-contract lies at the center of the system. Any state-changing operation is verified by the smart-contract and then finalized by emitting the new state. Assets moving in and out of the system are also processed and stored by the zkSwap smart-contract. We will now look at the different functionalities the zkSwap smart-contract is involved with.

Deposits To use the system, a user first has to deposit funds into the smart-contract. The funds need to be sent as a on-chain transaction to the smart-contract, where they will be stored, while the balance is then represented in

[6]The frontend is omitted here, as it is technically not required for the system to function.

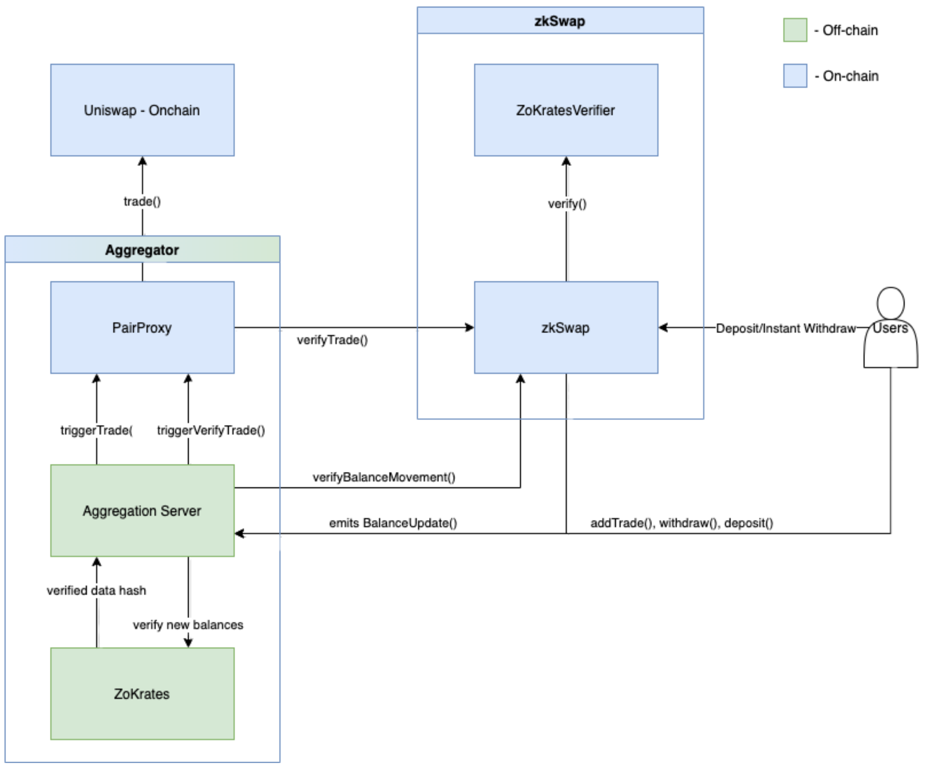


Figure 2: High level architecture of zkSwap

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. The deposit details are hashed and stored as a variable, enabling verification of the deposit later. Simultaneously, 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. This approach of storing funds is also considered non-custodial. As we will explore in the following sections, the funds are secured exclusively by the user's private key. Losing access to that private key results in the funds remaining locked in the smart-contract forever.

Withdraws When withdrawing funds, the user needs to decide between an aggregated withdrawal, similar to the deposit, or an instant on-chain withdrawal. In an aggregated withdraw, a user defines the withdrawal details, signs them and sends them to the aggregator. These are then aggregated, verified in the zkSwap smart-contract, and sent to the users as an on-chain transaction. A user can also withdraw funds by using the instant withdraw feature. While instant withdraws cost significantly more gas, they can be used without the aggregator being online. This protects the user from not being able to withdraw its funds if the aggregator is offline or has turned malicious.

Trades When adding a trade order, a user sends the order details to the aggregator. These are then aggregated and executed by the aggregator. Once complete, the funds are sent to the zkSwap smart-contract, were a couple of checks are performed, after which the batch is verified on-chain. This updates the user's balance according to the executed trade.

Verification of Aggregations The deposit, withdrawal, and trade features rely on the on-chain verification of a batch, so it makes sense to mention this separately. When aggregating a batch, the aggregator utilizes a zkSNARK circuit that will output a proof object. Each circuit has a corresponding verifier smart-contract that can be used to verify the correctness of the execution by submitting the proof object. When verifying a batch, the zkSwap smart-contract calls the corresponding verifier, checking the proof and thereby the correct aggregation. The new balances are now emitted, finalizing the aggregation.

3.1.2 Aggregator

As the name implies, the aggregator's job is aggregating deposits, withdraws, and trades. It facilitates the aggregation of these operations and is built to ensures correct execution while no trust assumptions are made. The aggregator relies on several different systems to function. In this section we will look at the aggregator from a functional perspective, explaining the core functionalities and what systems are relied on.

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 these messages, collecting them as the next aggregation batch. Once a number of messages have been received, the new balances are calculated and passed to the corresponding zkSNARK circuit, where the correctness of the balances and signature is checked. If this is successful, a zkSNARK proof object is created. The aggregator now sends 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 withdrawal 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. Similar 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 batched trades. The net-trade is then sent to the 'PairProxy' smart-contract as an on-chain transaction, which in turn will execute the trade 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 is ensured by using the corresponding zkSNARK circuit. 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 a new event is emitted, the aggregator either updates or adds the balance to the Merkle tree. Since the events stay on-chain, the Merkle tree can always be rebuilt by querying these events and updating the Merkle tree sequentially. The aggregator provides endpoints where balances and corresponding Merkle paths can be queried. 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.

3.1.3 Client Frontend

The front-end allows the user to interact with the system, calling the functions while providing necessary data in the background. The front-end also can sync the Merkle tree, containing the systems state, locally. While this puts computational strain on the client, it enables instant withdraws without the aggregator being online. In its current design, the front-end could be hosted as a static file in IPFS [13], not relying on any server to function, which closely follows Ethereums unstoppable applications ethos. Deposits, withdrawals, and trades are defined and the necessary messages are sent by the front-end. A user's balance and address are also displayed.

3.2 Implementation

In this section, we will look at the different features in detail, explaining how they function. We will begin by mentioning the technologies used and explain the mechanism used for updating balances, as it is generally the same throughout the system. After, we will explain the different transaction types, explaining them in detail.

3.2.1 Technologies Used

In this implementation, a variety of technologies are applied. The smart-contracts are built with Solidity[7], a programming language used for building Ethereum smart-contracts. The functionalities of the aggregator are written entirely in Node[8]. The zkSNARK side of the system was built with ZoKrates, a toolchain that includes a domain-specific language (DSL) for creating ZoKrates programs. These programs can be compiled into a zk-SNARK circuits, along with the corresponding verifier as a Solidity smart-contract. Throughout this work, ZoKrates program and zkSNARK circuit are used somewhat interchangeably. ZoKrates also includes numerous libraries for standardized operations like hashing. The frontend is built in React[9], a popular javascript-based frontend framework. The prototype that was built for this project can be found on online[10].

3.2.2 Storing and Updating Balances

Balances are represented as a balance object in our system. This object consists of four fields necessary to represent balances correctly: *ethAmount, tokenAmount, userAddress, nonce*. 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 crucial to distinguish between storing data in a smart-contracts runtime and storing data in the event log. Both methods of storage are on-chain, while the event log is significantly cheaper to use. However, a disadvantage of using the event log is that it cannot be accessed from a smart-contracts runtime and must be queried by a client. Using the event log solves the external data availability 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 mechanism to ensure the passed data is correct.

Merkle Trees We can achive this by using a Merkle Tree [32]. Merkle trees are a suitable data structure, as the Merkle root represents the entire tree's state in a highly compressed form while proving a leaf's inclusion in the tree can be done in *O(log(N))*. These properties are 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 balance object can be queried 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 stored in

[7]https://docs.soliditylang.org/en/v0.8.3/
[8]https://nodejs.org/en/
[9]https://reactjs.org/
[10]https://github.com/petscheit/bachelor

the zkSwap smart-contract. By storing the Merkle root in the zkSwap smart-contract, we're committing the entire trees state on the blockchain, preventing unauthorized balance updates. It is essential to understand that balances are only updated by emitting the 'BalanceUpdate' event and changing the root accordingly.

3.2.3 Aggregating Balance Updates

Updating a user's balance is at the core of this system. Deposits result in a balance update, as do trades and withdrawals. 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 much hashing is required. The properties of zkSNARK enable us to run the inclusion proofs off-chain in a ZoKrates program. If the ZoKrates program exits successfully, the zkSNARK proof can be used to verify the correct execution of the inclusion proofs on-chain. This verification gives us equal assurances that the execution was done correctly as executing this on-chain directly. We can run inclusion proof for multiple leaves at the same time while the on-chain verification costs largely stay constant[11], resulting in gas savings.

Merkle Inclusion Proofs To ensure the correctness of balance updates, we first need to verify that the balance is 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 that 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 change the balance values after a successful inclusion proof and rehash with the Merkle path. The result is the new Merkle root of the entire tree. Since most hashing is done in ZoKrates programs, looking for a hashing function that can efficiently be executed in a zkSNARK circuit is essential. In this implementation, we will utilize the MiMC [9] hashing function, as it can be used in zkSNARK circuits efficiently. The MiMC function is used with the Feistel structure and set up 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 Merkle roots should be the same for each balance. Things become more difficult when also updating the balances. Updating the first

[11]There is a certain amount of overhead per leaf, the details of which will be explained in S.4

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 sequentially and relative to each other. We can chain inclusion proofs by sorting the balances beforehand and generating each Merkle path based on the previous balance changes. The new root of the first balance is the old root of the following balance. This can be chained to an infinite 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 $root$
```

Algorithm 2: Computes merkle root of given parameters

```
1  function computeRoot;
   Input  : balance, merklePaths[]
   Output: root
2  computedHash ← $MiMC(balance)$
3  foreach merklePath do
4      if $merklePath[i][0] == 0$ then
5          computedHash ← MiMC($computedHash$, $merklePath[i][1]$);
6      else
7          computedHash ← MiMC($merklePath[i][1]$, $computedHash$);
8      end
9  end
10 return $computedHash$
```

The Merkle path has a binary as the first element of each hash value. That binary represents if the hash is on the left or the right side of the pair. A cleaner solution is to sort the pairs before hashing and decide the position based on the larger value. Limitations of the number range usable in ZoKrates make this infeasible.

Authorizing Balance Updates We still need to ensure the user has authorized a balance update. As balance updates are emitted as events, anyone can access them and compute valid Merkle paths for any balance in our system. The data is public, allowing any user to withdraw any balance. To ensure a user is authorized to update a balance, we need to ensure the user controls the private key belonging to the balance user address. We can ensure this by requesting a signature from the user. However, it must be remembered that this signature must also be verifiable in our ZoKrates program, which cannot utilize the secp256k1 curve used for signing Ethereum transactions efficiently [17]. For this reason, the Baby JubJub [11] 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, ensuring three things:

1. The user is in controls the balance objects addresses private key
2. The balance update corresponds to the amount in the update message
3. By signing the current Merkle root, we make sure that the signature cannot be reused in replay attacks

It is important to note that different programs are used for deposits/ withdraws and trade aggregation. As already mentioned, these programs have several checks that ensure the balance changes 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[12] does not support signing with the EdDSA signature scheme on the Baby JubJub curve. Fortunately, 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 efficiently in a zkSNARK circuit. It must also be mentioned that we utilize the MiMC hashing function to hash the message, as it is efficient to run in a zkSNARK circuit.

Reducing On-chain Verification Costs All of these checks are performed in a ZoKrates program. At this point, we could return the new balances that we have checked for correctness and verify the proof on-chain. Returning the new balances and other variables would result in high gas costs since every output of a ZoKrates program is part of the proof object, which will add an iteration to the verification logic. For this reason, we create a hash of the return values and only return the hash from the ZoKrates program. The aggregator can compute the same values and pass them to the verify transaction, excluded from the proof object. If the passed data is correct, the hash can be recreated in the smart-contract and should match the output

[12]https://metamask.io/

of the proof object. Since we need to recreate this hash for every batch we verify on-chain, we use a hashing function that is cheap on-chain. In our case, this is SHA256. While this optimization adds complexity to our circuits, it effectively reduces the gas required to verify a batch, which is the system's goal.

3.2.4 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. The balance object gives custody to these funds. When moving funds in layer-2, the funds residing in the smart-contract are not moved. Instead, the balance objects are updated, representing the movement of funds. Since a balance object gives a user custody of represented funds, it can always be redeemed, moving from layer-2 back to mainnet.

Movement of Funds To move funds to layer-2, a user first needs to send the funds to the zkSwap smart-contract. Funds can be sent by calling the deposit function in the zkSwap smart-contract and attaching the funds to the transaction. To verify the deposits in a later step, we create a recursive deposit hash by hashing the deposited amount, type, and user address with the existing deposit hash. The deposit hash is computed sequentially and ensures users deposited the funds they claimed.

Aggregating Deposits After the funds have been sent to the smart-contract, the user creates a signature containing the type of aggregation (deposit/withdraw), Ether and token balance changes, the user's address, and the current Merkle root. This signature is sent to the aggregator as an HTTP request. The aggregator checks the following:

1. Validity of signature
2. Merkle root in zkSwap smart-contract matches signed root
3. Funds were deposited into the zkSwap smart-contract by recreating the deposit hash

These checks are strictly speaking not necessary for a valid aggregation. However, they ensure invalid orders are not added, which would result in the entire aggregation failing. After several 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. The 'BalanceMovementObjects' are passed to the 'ProcessBalanceMovement' ZoKrates program, along with the current

Merkle root. As already explained in detail in S. 3.2.3, the following is performed for each 'BalanceMovementObject' in the ZoKrates program:

1. Check if the tree contains the old balance via inclusion proof
2. Check the validity of the signature
3. Check if balance change corresponds to the signed values
4. Check if new balance contains the same address and incremented nonce
5. Merkle root in zkSwap smart-contract matches signed root
6. If a withdraw, generate withdraw object[13]
7. If a deposit, compute deposit hash
8. Compute the new Merkle root

If these steps terminate without an error, we compute the data hash by hashing *newBalances, oldRoot, newRoot, depositHash, withdrawObjects* with the SHA256 algorithm. The data hash is the only output of our ZoKrates program, as it reduces on-chain verification costs as explained in S. 10.

Verifying Deposits On-chain Once the ZoKrates program has run successfully, the proof is generated. The aggregator calls the 'verfiyBalanceMovement' function in the zkSwap smart-contract and attaches the proof, along with the new balances, the new Merkle root, and the withdraw objects. The zkSwap smart-contract now performs the following steps:

1. Compute withdraw hash by hashing withdraw objects
2. Compute data hash by hashing balances, old root, new root, deposit hash, and withdraw hash
3. Ensure computed data hash is equal to zkSNARK proof output
4. Check zkSNARK proof object with verifier smart-contract
5. Update root stored in smart-contract and reset deposit hash
6. Emit new balances, and process withdraw objects by sending funds

The attentive reader might have realized that the old root and the deposit hash were not passed as a transaction parameter. Since these values are computed and stored in the smart-contract, we can use them to compute the data hash. While this is more efficient and reduces the amount of gas needed, it also ensures the correct values were used throughout the aggregation. By including the old Merkle root in the data hash, we ensure a proof object cant be used in replay attacks. Once the balances have been emitted, the balance updates are finalized.

[13]The details of this will be explained in S. 3.2.5

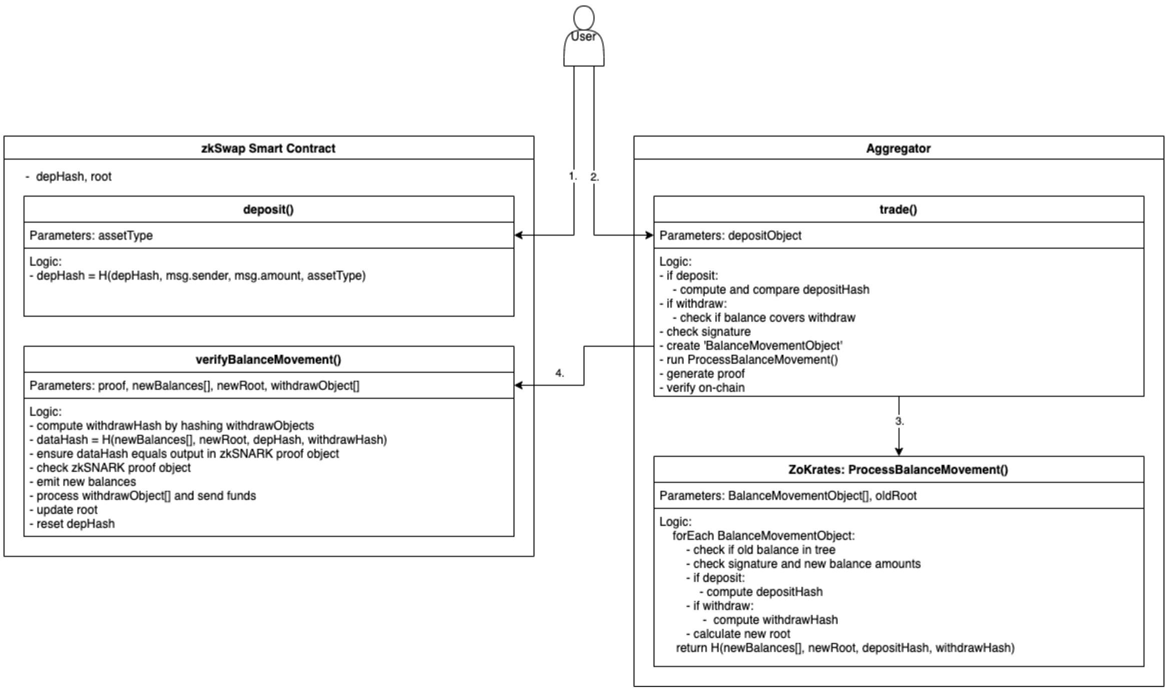

User
1.
2.
3.
4.
zkSwap Smart Contract
- depHash, root
deposit()
Parameters: assetType
Logic:
- depHash = H(depHash, msg.sender, msg.amount, assetType)
verifyBalanceMovement()
Parameters: proof, newBalances[], newRoot, withdrawObject[]
Logic:
- compute withdrawHash by hashing withdrawObjects
- dataHash = H(newBalances[], newRoot, depHash, withdrawHash)
- ensure dataHash equals output in zkSNARK proof object
- check zkSNARK proof object
- emit new balances
- process withdrawObject[] and send funds
- update root
- reset depHash
Aggregator
trade()
Parameters: depositObject
Logic:
- if deposit:
- compute and compare depositHash
- if withdraw:
- check if balance covers withdraw
- check signature
- create 'BalanceMovementObject'
- run ProcessBalanceMovement()
- generate proof
- verify on-chain
ZoKrates: ProcessBalanceMovement()
Parameters: BalanceMovementObject[], oldRoot
Logic:
forEach BalanceMovementObject:
- check if old balance in tree
- check signature and new balance amounts
- if deposit:
- compute depositHash
- if withdraw:
- compute withdrawHash
- calculate new root
return H(newBalances[], newRoot, depositHash, withdrawHash)

Figure 3: Interaction diagram of aggregated deposits and withdraws

3.2.5 Withdraws

As briefly mentioned before, withdrawals are batched together with deposits. For this reason, we will only briefly cover the process, as it is largely the same. Instead of sending funds to the zkSwap smart-contract, we are requesting them. As the user already has a balance in the system, an on-chain transaction is not required to trigger a withdrawal. 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' signatures and ensures that the balance covers the withdrawal. Just like with deposits, a 'BalanceMovementObject' is created, the type set to withdraw. 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 and 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.

3.2.6 Instant Withdraws

A user also has the option to withdraw instantly without being dependant on the aggregator. Instant withdrawals ensure a user can always withdraw funds, even when the aggregator is failing or offline. Instead of sending the

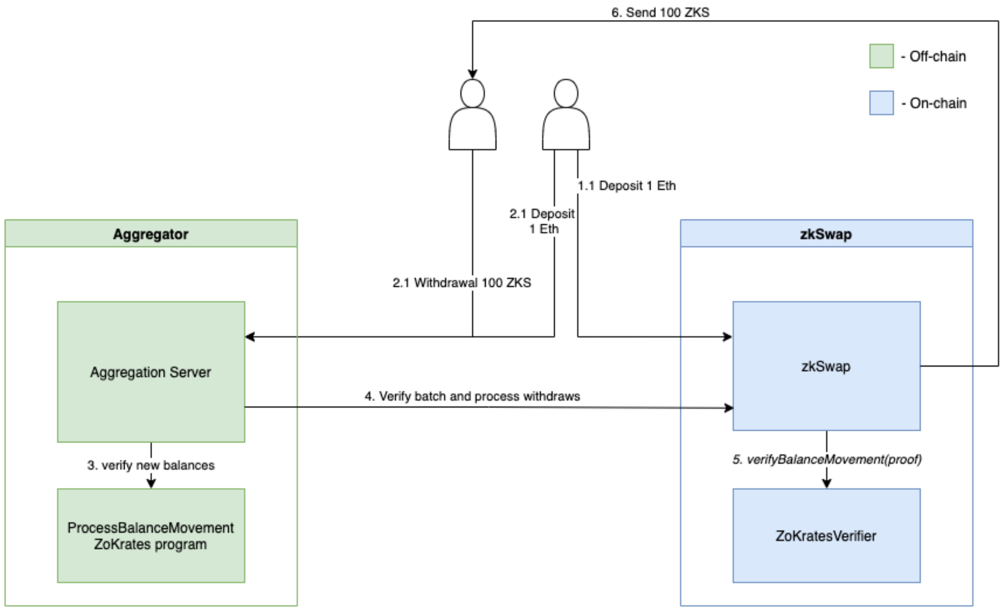
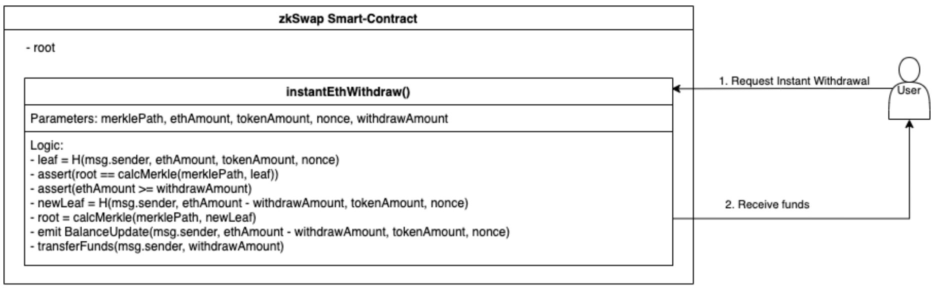


Figure 4: Aggregated deposits and withdrawals in detail

withdrawal request to the aggregator, the withdraw is processed entirely on-chain. The user attaches its balance object, along with the corresponding Merkle path and the withdrawal amount and fund type (Ether/ERC20), to the transaction. As a first step, the Merkle inclusion proof is performed. The balance object is hashed sequentially with the Merkle path. The resulting hash now equals 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 that is 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 completely on-chain and does not require the aggregator to be online. However, the gas costs of this transaction are high, as using the MiMC hashing algorithm is expensive in a smart-contract.

Authorizing Instant Withdraws This, however, is an incomplete explanation, as we are 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. Not adding an additional check would allow any user to withdraw any balance. To ensure a user is permitted to update a balance object, we need to ensure the user controls the private key belonging to the balance object's user address. Fortunately, we can ensure this by accessing the sender in the transaction object. The Ethereum blockchain ensures a user can make a transaction by requiring the transaction to be signed with the private key of the sender's address. If that signature is valid, it is proven that the user has access to the address's private key, and the transaction can be executed. Thus, the transactions object sender can be trusted to be in control of the corresponding private key. Instead of passing the user's address as part of the balance object, the smart-contract uses the sender of the transaction object. This mechanism suffices as a security check.



```
Parameters: merklePath, ethAmount, tokenAmount, nonce, withdrawAmount
```

```
Logic:
- leaf = H(msg.sender, ethAmount, tokenAmount, nonce)
- assert(root == calcMerkle(merklePath, leaf))
- assert(ethAmount >= withdrawAmount)
- newLeaf = H(msg.sender, ethAmount - withdrawAmount, tokenAmount, nonce)
- root = calcMerkle(merklePath, newLeaf)
- emit BalanceUpdate(msg.sender, ethAmount - withdrawAmount, tokenAmount, nonce)
- transferFunds(msg.sender, withdrawAmount)
```

Figure 5: Instant withdrawals in detail

3.2.7 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 a batch, starting with a user adding a trade order.

Ensuring Correct Pricing The price between assets is constantly changing. At the same time, trades are being collected for aggregation, which results in a delay between a user sending a trade order and the actual trade execution. During this delay, the price of an asset can change. On top of that, network congestion on the Ethereum blockchain can cause further delays in the execution of a trade. A mechanism is needed to define a worst-case price that is defined before users add orders to the aggregation batch. Once the aggregation is complete, a user can be sure to have paid no worse than 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. Intuitively, this makes sense. The spread should at least equal the cost of converting from one asset to the other. Several other factors influence the bid-ask spread for a Uniswap trading pair. In the context of this work, it is sufficient to know that a spread is expected in any trading pair. The bid-ask spread further 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 do not know what orders will be received in an aggregation batch, we cannot predict if the net-trade will be a buy or sell order. Since we have a bid-ask spread and can not predict which direction our net-trade will be, we need to define a price range that, at a minimum, equals the current bid-ask spread. An equal price range would suffice to ensure a worst-case price for a net-trade in either direction if executed immediately. As the aggregation also adds a delay between defining the price range and executing the trade, the price range should be larger than the bid-ask spread. For this reason, the zkSwap smart-contract defines a 'minSell' and 'maxBuy' price, defining that range. If the price of an asset moves out of the defined range, the trade is not 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

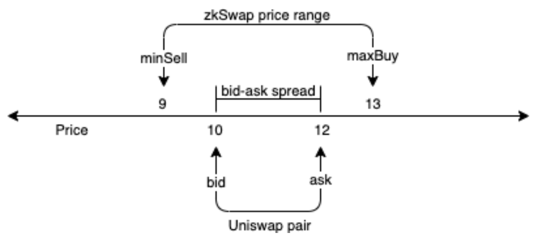


Figure 6: Bid-ask spread and zkSwap price range

While this method ensures a worst-case price for a trade, at the same time, a maximum price is defined with it. Since we are 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 a trade object and sign it with its private key. The trade object consists of five fields needed to define the trade: *tradeDirection, deltaEth, deltaToken, userAddress and merkleRoot*. 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. The aggregator performs the following steps for each incoming trade object:

1. Check the signatures validity
2. Check if users balance covers the trade
3. Ensures signed merkle root equals the current merkle root[14]
4. Checks if implied asset price matches worst-case price

As with deposit and withdrawal aggregation, these checks are only performed to prevent invalid trade orders from being aggregated, as they would result in the cancelation of the entire aggregation batch. Invalid trade are removed from the trade pool.

Executing Trade and Calculating New Balances At some point, the trade aggregation is started. A set block number could trigger this, the number of trade orders 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 can reduce the entire aggregation to one Uniswap trade, which saves gas. At the same time, we are saving on the 0.3% liquidity provider fee, which is charged based on the trades volume. This trade is now sent as an on-chain transaction to the 'PairProxy' contract, where it is executed with Uniswap. The PairProxy contract is explained in detail in S. 3.2.8.

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 acquired 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 effective price will be better than the worst-case price. Based on the effective price, the user's 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 '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 ensure each trade has the same price, whether it is a buy or sell order. It is also essential that this price is not worse than 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 is 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 been completed, we are assured that each user receives an equal price, at

[14]This prevents orders from being used in malicious replay attacks by the aggregator.

least matching the worst-case price[15], 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 untrusted party. We need to verify the correct execution of these checks on-chain, which can be achieved with zkSNARK. To ensure this, the following steps are performed for each 'BalanceUpdateObject':

1. Check if the tree contains old balance via inclusion proof
2. Check the validity of the signature
3. Check signed Merkle root equals the current Merkle root
4. Check balance change is equal or better than signed values[16]
5. Check if new balance contains the same address and incremented nonce
6. Compute the new Merkle root

These steps largely follow what is described in more detail in S. 3.2.3. If these checks pass, we are assured that each balance update was done correctly. The last Merkle root computed in this iteration is the new Merkle root, later updated in the zkSwap smart-contract, and commits the updated state that has been verified here.

Reducing On-chain Verification Costs in ZoKrates We have now successfully verified the new balances and 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 create the data hash by hashing *newBalances, oldRoot, newRoot, netTrade, worstCasePrice* with the SHA256 algorithm. The data hash is the only output of the ZoKrates program.

[15]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.

[16]The signed values will contain the worst-case price. The effective balance change could be better than that.

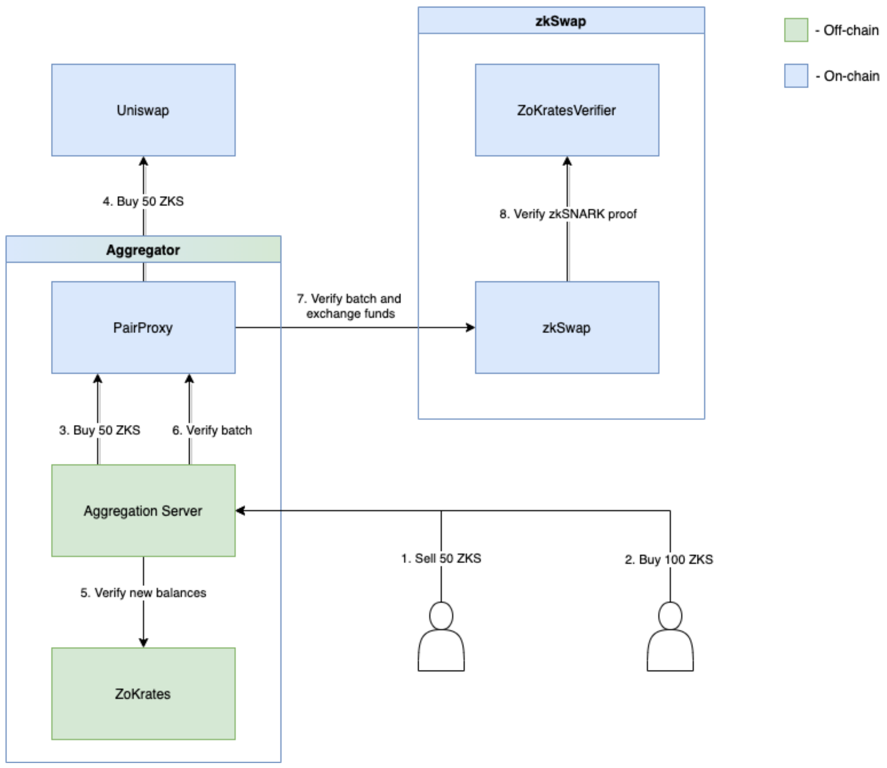


Figure 7: Interaction diagram of aggregated trades

Generating Proof and Verifying At this point, the aggregator can start the proof generation of the ZoKrates program. To verify the aggregation batch on-chain, 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, where all following steps are performed.

Recreating the Data Hash As a first step, we recreate the data hash previously computed in the ZoKrates program. By recreating this hash, we ensure the aggregator has performed several steps correctly:

1. The balances and new root passed by the aggregator were verified in the ZoKrates program
2. The net-trade corresponds to the net change of balances
3. The correct worst-case price was used throughout the aggregation
4. The current Merkle root equals the old Merkle root in the aggregation, preventing proofs from being used in replay attacks

The worst-case price and old Merkle root are not passed as parameters and are stored in the smart-contract. Including these values in the data hash ensures the aggregator used the correct values when the aggregation batch was started. By including the balances, new root, and net-trade, we ensure the aggregator passes the correct values after the aggregation. The method saves gas in the on-chain verification step and ensures the aggregator executed its tasks correctly. If the hashed value does not equal the output of the proof object, some values were incorrect, resulting in the entire aggregation being canceled.

Verifying the ZoKrates Proof The next thing to be checked is the validity of the zkSNARK proof object. The zkSwap smart-contract can access a verifier contract that was explicitly deployed for this purpose. The 'verifyTx()' function is called, and the proof object is passed as a parameter. If the verifier returns true, we have proven that our data hash was computed by running the 'ProcessTrades' ZoKrates program. By having proven knowledge of the preimage, the properties of the data hash are extended to the preimage[17]. At this point, we have proven that the balances were updated correctly.

Exchanging Traded Funds Now the funds traded on Uniswap must be exchanged between the aggregator and zkSwap smart-contract. We check if the funds attached to the transaction are equal to the amount specified in the net-trade. It is important to remember that we have checked the correctness of the net-trade by recreating the data hash already. The exchange of funds always needs to result in a solvent zkSwap smart-contract[18]. If the aggregator has attached the correct amount of funds, the zkSwap smart-contract now refunds the aggregator by sending the funds it spent for the Uniswap trade. If an incorrect amount was attached, the aggregation is canceled.

Updating Root and Emitting Balances In the last step, we update several variables that are needed in the next batch. The Merkle root stored in the smart-contract must be updated. By recreating the data hash, we have proven the new Merkle root to be correct, so we update it accordingly. The worst-case price also needs to be updated, which can be achieved by querying the current price from Uniswap. The new balances are emitted as 'BalanceUpdate' events to finalize the aggregation batch, resulting in the balances being updated for the users, representing the trade. The lifecycle of a trade aggregation is now complete, and the next batch starts.

[17]Given that SHA256 is assumed to be collision resistant

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

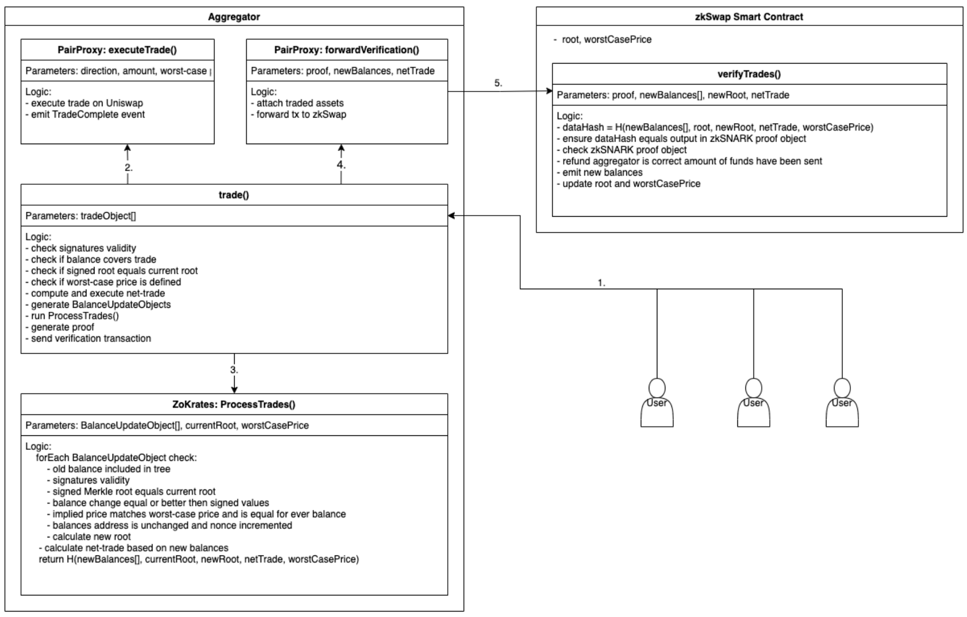

Aggregator
PairProxy: executeTrade()
Parameters: direction, amount, worst-case p
Logic:
- execute trade on Uniswap
- emit TradeComplete event
PairProxy: forwardVerification()
Parameters: proof, newBalances, netTrade
Logic:
- attach traded assets
- forward tx to zkSwap
2.
4.
trade()
Parameters: tradeObject[]
Logic:
- check signatures validity
- check if balance covers trade
- check if signed root equals current root
- check if worst-case price is defined
- compute and execute net-trade
- generate BalanceUpdateObjects
- run ProcessTrades()
- generate proof
- send verification transaction
3.
ZoKrates: ProcessTrades()
Parameters: BalanceUpdateObject[], currentRoot, worstCasePrice
Logic:
forEach BalanceUpdateObject check:
- old balance included in tree
- signatures validity
- signed Merkle root equals current root
- balance change equal or better then signed values
- implied price matches worst-case price and is equal for ever balance
- balances address is unchanged and nonce incremented
- calculate new root
- calculate net-trade based on new balances
return H(newBalances[], currentRoot, newRoot, netTrade, worstCasePrice)
5.
zkSwap Smart Contract
- root, worstCasePrice
verifyTrades()
Parameters: proof, newBalances[], newRoot, netTrade
Logic:
- dataHash = H(newBalances[], root, newRoot, netTrade, worstCasePrice)
- ensure dataHash equals output in zkSNARK proof object
- check zkSNARK proof object
- refund aggregator is correct amount of funds have been sent
- emit new balances
- update root and worstCasePrice
1.
User
User
User

Figure 8: Aggregated trades in detail

3.2.8 PairProxy Smart Contract

Before explaining the functionalities of this smart-contract, it is essential 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[19] for the trade. Since network congestion and the current gas price influence when a transaction is executed, it is necessary to ensure 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 function's return value. Since this amount might be larger than the amount passed as 'minAmountReceived', we need it to calculate the post-trade balances[20].

However, a quirk in Ethereums way of handling return values makes this more difficult. A smart-contracts function's 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, we receive the transaction recipe instead of receiving the function's return value, which does not contain the return value. For this reason, we need the PairProxy smart-contract, which receives transactions, forwards them to the respective smart-contract,

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

[20]The trade also throw an error when the 'minAmountReceived' amount cannot be fulfilled. In this case, the aggregator cancels the aggregation

emitting the return value as an event, which the transactor can consume.

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 securely.

When verifying the aggregated trades in the zkSwap smart-contract, the transactions are 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.

3.3 Limitiations of Current Implementation

The working implementation of this system is different from the proposed design in certain areas. The differences are mainly the result of two factors: MetaMask not supporting EdDSA natively and being unable to create matching MiMC hashes in ZoKrates and Solidity. The necessary ZoKrates programs have been built and used in the results section but are not integrated into the prototype for the reasons stated above.

Signatures As described in S. 10, 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, limiting us to use the EdDSA scheme with the BN128 curve. Hermez [1], a zk-rollup based asset transfer system, has solved this by creating a Baby JubJub[11] 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 do not have a written source for this but talked to their team members, who explained how they do this. Since their system is running on the mainnet now, it can be assumed to work. In the system’s 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 ensure 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 does not utilize the MiMC hashing yet. All hashing is currently done with SHA256, limiting the number of trades or deposits/withdraws that can realistically be included in a batch due to the circuit size. The benchmarking results where generated with circuits that utilize MiMC hashes, however, the inability to create matching hashes in Solidity prevents the utilization of the ZoKrates program in the prototype.

Aggregating Deposits and Withdraws Since the MiMC hashes are not implemented yet, the aggregated withdraws are not ether. The necessary ZoKrates programs have been developed, however, due to the inablility to create matching MiMC hashes where never used. In its current 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

This section will present the results of the proposed implementation, introducing several metrics that quantify the system's capabilities and limitations. In zkSNARK enabled systems, the results must always be considered out of two different points of view. For one, gas savings are essential. After all, this system is built to reduce gas consumption per trade. Another thing to consider is the performance of the zkSNARK steps necessary to execute and verify an aggregation batch. As zkSNARKs rely on complex cryptographic primitives, the generation of proofs is computationally demanding, resulting in demanding hardware requirements and long execution times. As this system's primary goal is to reduce the required 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.

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 significant overhead (fixed cost) and a proportionally small amount of variable cost. The way fixed and variable costs are distributed results in the cost per trade to reduce significantly initially, 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 the fixed and variable gas costs that need to be paid per batch and trade. For one, the gas for the net-trade, executed on Uniswap, must be paid. The gas required for executing the net-trade varies, 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, handling the refund payment of the aggregator and the on-chain checks, costing 342k gas per aggregation batch. The combined fixed amount of gas per aggregation batch is 484k/509k gas, depending on the net-trades direction. For each trade in a batch, we must pay 6619 gas, which is used for recreating the data hash and emitting the BalanceUpdate event. When using these numbers, we get the following cost per trade, depending on the batch size. This diagram uses the more expensive net-trade (ERC20 to Ether) to calculate the results, which is the upper bound 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 of an unaggregated Uniswap trade, we define a break-even price equal to the cost of a Uniswap trade. Our system makes economic sense once our gas per trade cost is lower than 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 withdrawal.

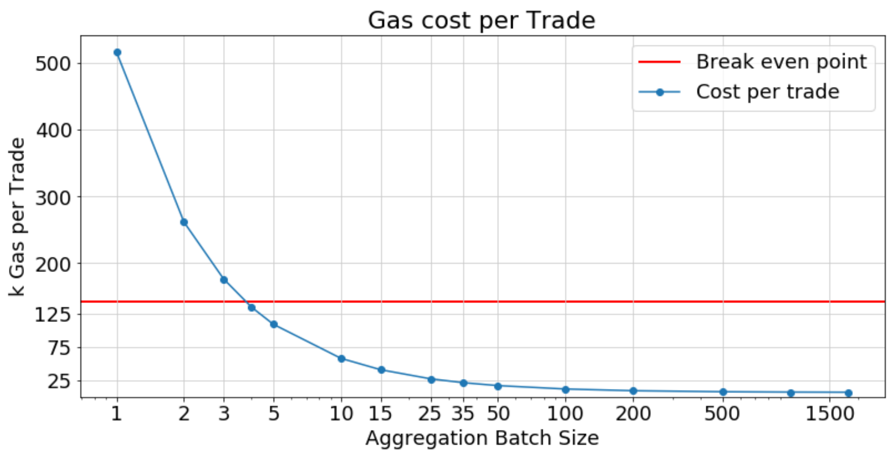


Figure 9: Gas cost: per trade and breakeven point

4.1.2 Deposits and Withdrawals

Before a user can trade, funds must be deposited into the system. Presenting these results is a bit more complex compared to the trade aggregation results for two reasons. For one, the deposits and withdrawals are aggregated as one aggregation batch. Since the gas costs for a deposit and withdrawal are different, and the number of deposits and withdraws included in a batch is not predictable, the cost per deposit/withdrawal depends on the proportion of these operations. On top of that, the gas cost of a deposit/withdrawal also depends on the asset. Ether deposits/withdrawals are cheaper than ERC20, which is a result of how tokens are represented on the Ethereum blockchain. Thus, we will present three quartiles for deposits and withdraws, representing the share of an asset in the batch. For example, as seen in F. 10, '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 variable gas cost required per batch. For each deposit aggregation, we have 312k gas as a fixed cost. This amount covers all checks and verifications explained in S. 3.2.4. The variable gas cost depends on the type of asset being deposited. Depositing an ERC20 token requires significantly more gas than depositing Ether, as multiple smart-contract interactions are necessary. An Ether deposit adds 44k gas, an ERC20 deposit 128k gas. The maximum batch size is calculated by checking how many deposits can fix in a block if only the most expensive operation is included. In our case, a batch only containing ERC20 deposits is the most expensive scenario, resulting in a maximum batch size of 94.

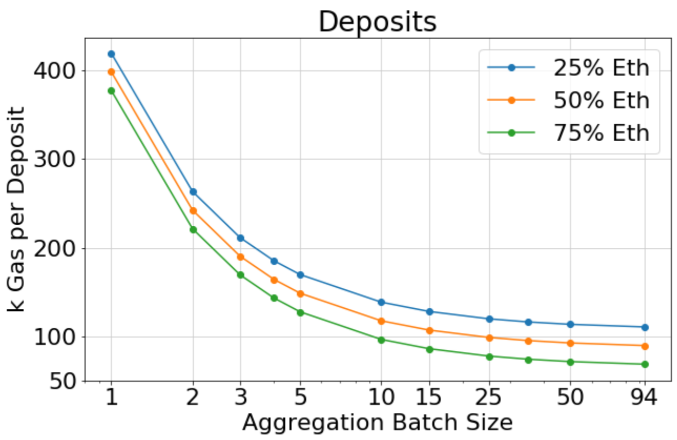


Figure 10: Gas cost: per deposit assuming different share of Ether deposits

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. An Ether withdrawal adds 19k gas, an ERC20 withdrawal 65k gas. 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.

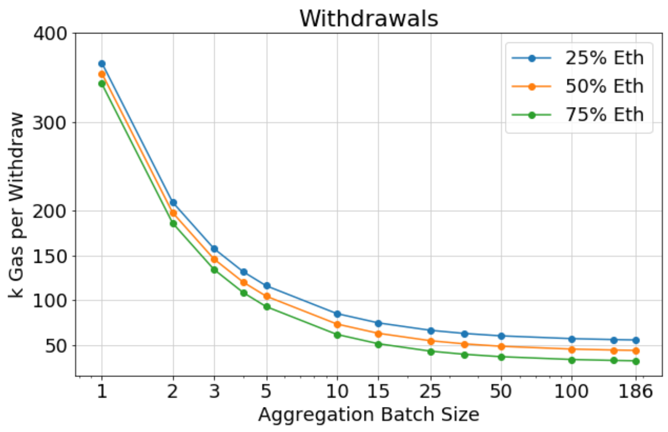


Figure 11: Gas cost: per withdraws assuming different share of Ether withdrawals

Combining Deposit and Withdraw Gas Cost Modeling the combined gas costs of deposits and withdraws is omitted in this work, as many assumptions are required to calculate this. As mentioned before, the proportion of Ether and ERC20 operations and the proportion of deposits and withdrawals influence the results. For the deposits and withdrawals, 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 withdrawals in an aggregation batch would not yield representative results, as several valid combinations would need to be considered. The diagrams presented above give a clear indication of the potential cost, which suffices for this work.

Instant Withdrawals The cost of instant withdraws is also worth considering. Since we are not batching instant withdraws, there is only a fixed cost per withdraw that needs to be paid. Since instant withdrawals are not 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 a MiMC hash currently costs 38k gas, bringing the cost to around 1.3m gas per instant withdrawal.

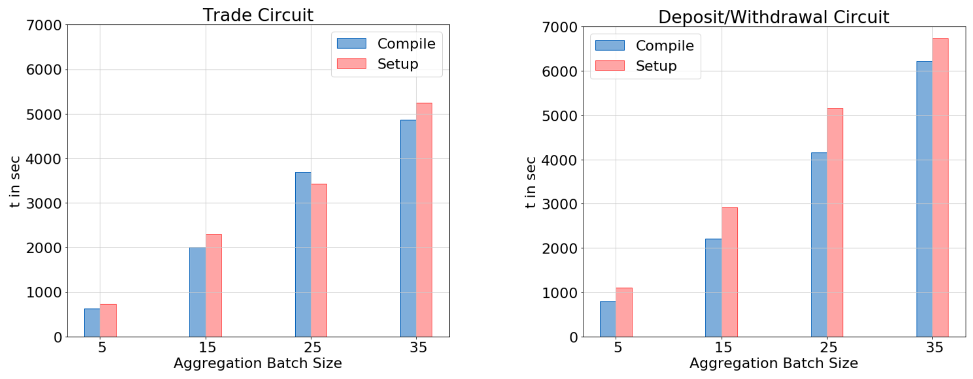
4.2 zkSNARK Circuit Metrics

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 clock speed) and 240Gb of memory. This instance was chosen because of the large amount of memory and the fast clock speed. The number of cores does not impact the benchmarking results, as the zkSNARK steps cannot be parallelized at the time of writing.

4.2.1 Execution Time

The first obvious metric to consider is the execution time of the different steps required when using zkSNARK circuits. As we will see in the following sections, using zkSNARK circuits requires powerful hardware. We will begin by looking at the performance metrics based on execution times first and then look at more hardware-specific metrics.

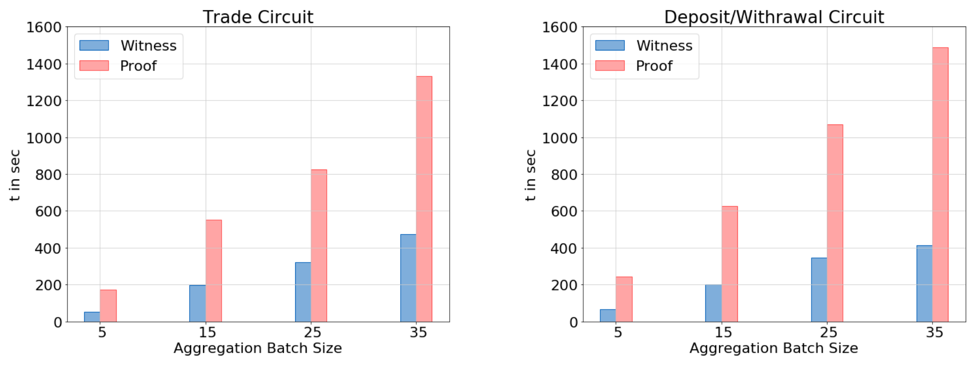
Compilation and Setup Before we can generate any zkSNARK proofs, we have to compile our circuits and run the setup. These two steps only need to be run once per circuit, so they are not significant for the viability of this system. However, the results were measured during the benchmarking, so it would feel incomplete not to present them.



(a) (b)

Figure 12: Execution time: Compilation and setup

Witness Computations and Proof Generation For each aggregation batch, we need to run the witness computation first. After the witness computation is complete, the proof can be generated. As these two steps are required for every aggregation batch, the execution times impact the practical viability of the system.



(a) (b)

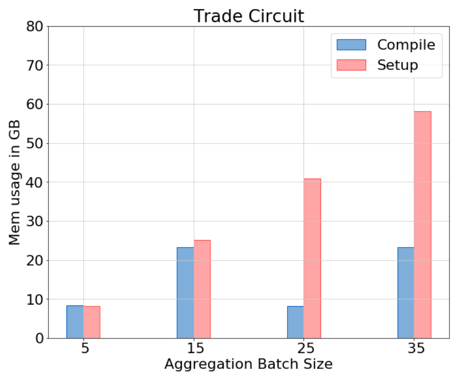
Figure 13: Execution time: Witness computation and proof generation

4.2.2 Memory Usage

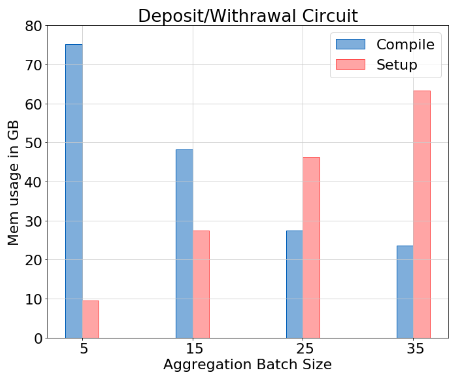
Only looking at the execution times gives us an idea of how many operations can be batched. It tells us nothing about the hardware requirements needed for working with circuits of this size. One thing to look at is the memory used in the different steps. In general, the memory consumption of these processes 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 do not really make sense, as smaller circuits sometimes require more memory than smaller ones. However, the measurements were repeated on different machines and operating systems, always resulting in inconclusive results. The memory measurement script uses the command line tool ‘ps’ to take these measurements, which measures reserved memory by a process. Alternatively, a profiler could be used to measure the actual memory usage. Using a profiler would severely impact the application’s performance, which is why it was not attempted. As these steps only have to be 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 requirements for the aggregation process. We need to run these steps for every aggregation batch, so hardware capable of handling the memory requirements must be available.

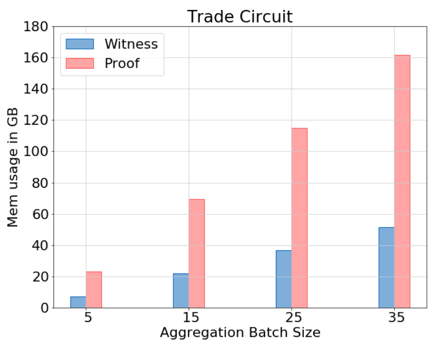


(a)

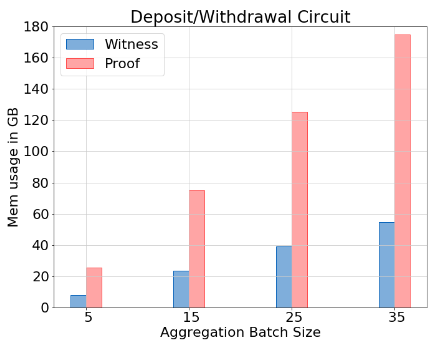


(b)

Figure 14: Memory consumption: compilation and setup



(a)



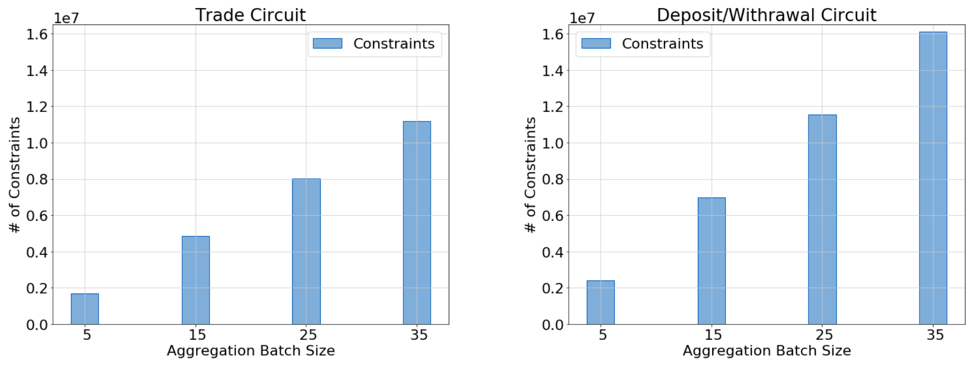
(b)

Figure 15: Memory consumption: witness computation and proof generation

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. In general, the complexity of a zkSNARK circuit is defined by the number 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 to compute the data hash. For each of these operations, we get constraint numbers that are added per additional batch element. These numbers were measured by compiling the segments separately and checking the constraint count. Comparing these to the total constraint values, we realize that they are higher



(a) (b)

Figure 16: Constraint count of circuits for different batch sizes

than expected. The increased number of constraints can be explained by the ZoKrates optimizer, which runs several optimizations during compilation. As we can see, the final hash produces significantly more constraints, which is caused by creating the deposit and withdraw hash.

	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	111172	395076

Hashing and Constraints The MiMC hashing algorithm was used for hashing the Merkle tree, as it is 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, one for hashing the old balance, 16 for the Merkle inclusion proof, one 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

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

Constraints Processed per Second The last metric we want to introduce is constraints processed per second in the witness computation and proof generation step. We can calculate these values by dividing the execution 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[21], which we will look at in S. 6.1.1. For this reason, we are introducing this metric, as we can compare the outcomes and the potential speedup of parallelizing. This metric can also be helpful to test the performance of the circuits on a CPU with a higher clock speed.

5 Discussion

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.

5.1 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

[21]https://github.com/scipr-lab/libsnark

zkSNARK circuits, which gives a good understanding of the current limitations of the technology.

5.1.1 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. 9, 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. 9. 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.

5.1.2 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. 6.5.

5.1.3 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. 13, 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. 6.

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. 15, 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.

5.1.4 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. 6.

5.1.5 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.

5.2 Limitations

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

5.2.1 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.

5.2.2 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.

5.2.3 Trusted Setup Phase

The non-interactive nature of zkSNARK requires a common reference string to be shared among prover and verifier [5]. 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 [5]. The described MPC approach only requires one party of the computation to be honest, which is deemed secure with enough participants.

5.3 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.

5.3.1 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.

5.3.2 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. 5.3.1 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.

5.3.3 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.

5.3.4 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 executing the trade from. the zkSwap smart-contract, there are three realistic[22] 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,

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

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.

5.3.5 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.

5.3.6 Sandwich Attacks

A sandwich attack, as described by Zhou et al. [38] 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 [8] the original trade transaction, 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.

6 Related Work and Outlook

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.

6.1 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 [7] and compare them to the performance metrics 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 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. [37] 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.

6.1.2 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.

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-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.

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 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 [19]. 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.

6.2.2 Poseidon Hashes

The Poseidon [25] 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[23] 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. 6.2.1 can be applied here as well. Poseidon was not used in this work, as a collision was found by Udovenko [33]. 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.

6.3 PLONK

PLONK [20] 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.

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

6.4 zkSync and the Zinc Programing Language

zkSync [6] 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 [4], 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 [2] 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.

6.5 Cross zk-rollup Transactions

Another technology currently in development is cross zk-rollup transactions, as described by C. Whinfrey [35]. 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.

7 Conclusion

This work suggests that Uniswap trades can be aggregated with zk-rollup, reducing the gas amount per trade. The proposed design is non-custodial, trustless and the data availability problem has been solved by storing state in the event log. Increasing the batch size will further reduce the gas cost per trade while reducing strain on the Ethereum network. The overall efficiency of the proving steps has to be improved to allow bigger batch sizes. The batch size can be increased by several improvements, like utilizing more efficient hashing functions and parallelizing the proving steps.

This work was also aimed at exploring the viability of integrating a zk-rollup based application with third-party smart-contracts. As described in S. 5.3.4, one core problem that has been identified is ensuring the correct return values of the third-party smart-contract interaction are used in the aggregation. In our case, that results in custodial issues of assets. Other problems in other scenarios can be expected as well. Solving this problem would make integrating third-party smart-contracts into a zk-roll application a viable solution.

Projects like zkSync have shown the technological potential of zk-rollup. However, one core difference to our system is that aggregation batches are not dependant on external smart-contract interactions to be executed. As described above, being dependant on other smart-contracts adds complexities to the system that are difficult to overcome. While zkSync is an isolated application that interacts with no third-party smart-contract, rollup to rollup transactions preserve composability between rollup enabled applications. Another development worth mentioning is the prospect of recursive PLONK proofs. Recursive PLONK proofs could enable different smart-contract-like applications to be deployed into a zk-rollup based applications layer-2, secured by its on-chain verification. The idea, proposed by Matterlabs, envisions a zkSNARK based sidechain essentially, securing correct execution with a recursive proof construction, verified on Ethereums mainnet.

Zk-Rollup shows great promise to reduce strain on the Ethereum blockchain. Trustless and permissionless applications can be built with zk-rollup that do not rely on external data availability. Transaction costs can also be reduced significantly, saving users money and reducing strain on the Ethereum blockchain. Some issues remain and need to be resolved, but in general, zk-rollup is a promising scaling solution. Failing to resolve these problems would lead to applications being rebuilt as a zk-rollup native application. Projects like zkSync have shown this to be viable. The ongoing development of zero-knowledge protocols is expected to make the technology a foundation for scaling the Ethereuem blockchain in the near future.

References

[1] `https://docs.hermez.io/#/`

[2] Curve on zinc, https://medium.com/matter-labs/curve-zksync-l2-ethereums-first-user-defined-zk-rollup-smart-contract-5a72c496b

[3] Erc-20 token standard, https://ethereum.org/en/developers/docs/standards/tokens/erc-20/#:~:text=What%20is%20ERC%2D20%3F,to%20all%20the%20other%20Tokens.

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

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

[6] zksync (Dec 2019), https://medium.com/matter-labs/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58

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

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

[9] 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)

[10] Angeris, G., Kao, H.T., Chiang, R., Noyes, C., Chitra, T.: An analysis of uniswap markets. arXiv preprint arXiv:1911.03380 (2019)

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

[12] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptol. ePrint Arch. **2018**, 46 (2018)

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

[14] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 315–334. IEEE (2018)

[15] Buterin, V.: An incomplete guide to rollups, https://vitalik.ca/general/2021/01/05/rollup.html

[16] Coleman, J.: State channels - an explanation (Nov 2015), https://www.jeffcoleman.ca/state-channels/

[17] Deml, S.: Efficient ecc in zksnarks using zokrates (Aug 2019), https://medium.com/zokrates/efficient-ecc-in-zksnarks-using-zokrates-bd9ae37b8186

[18] Ethereum: ethereum/research, https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding

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

[20] 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)

[21] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 626–645. Springer (2013)

[22] Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. Journal of the ACM (JACM) **38**(3), 690–728 (1991)

[23] Goldreich, O., Oren, Y.: Definitions and properties of zero-knowledge proof systems. Journal of Cryptology **7**(1), 1–32 (1994)

[24] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on computing **18**(1), 186–208 (1989)

[25] 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)

[26] Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.S. (eds.) Advances in Cryptology – EUROCRYPT 2016. pp. 305–326. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

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

[28] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system bitcoin: A peer-to-peer electronic cash system. Bitcoin. org. Disponible en https://bitcoin. org/en/bitcoin-paper (2009)

[29] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy. pp. 238–252 (2013). https://doi.org/10.1109/SP.2013.47

[30] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper pp. 1–47 (2017)

[31] Silva, F., Alonso, A., Pereira, J., Oliveira, R.: A comparison of message exchange patterns in bft protocols. In: IFIP International Conference on Distributed Applications and Interoperable Systems. pp. 104–120. Springer (2020)

[32] 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)

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

[34] Vbuterin: On-chain scaling to potentially 500 tx/sec through mass tx validation (Sep 2018), `https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477`

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

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

[37] 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)

[38] 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)