Before diving into the technical details and the goals of this work, we will cover a number of topics that are important to understand in the context of this work.

0.1 Decentralized Exchanges

The first thing to understand is the recent rise of decentralized exchanges in recent months. The rise of tokenization on the Ethereum blockchain has also accelerated the developmemt of decentralized exchanges. Instead of relying on centralized server infrastructure, as traditional crypto asset exchanges do, decentralized exchanges execute the trade logic in a decentralized and trustless manner. The most popular decentralized exchange is Uniswap [**?**], which runs as a collection of smart-contract on the Ethereum blockchain [**?**]. There are a number of competing decentralized exchanges running on the ethereum blockchain, all generally functioning the same way. In this work we will be focus on Uniswap, as its currently the most popular, the system could easy be integrated most competing decentralized exchanges aswell.

0.1.1 Benefits of Decentralized Exchanges

By being decentralized, these exchanges can offer a number of features and have certain properties, that centralized exchanges can not have.

Custody of Funds When using a decentralized exchange, a user doesn't have to send its funds to another wallet. Deposits are not nececcary, as a user trades straight from its wallet. This means that a user always has custody over its funds. When making a trade, a user sends a transaction, attaching the funds looking to exchange. As Ethereum transaction are atomic, the user also keeps custody while trading. Either the trade transaction is succesful, in with case the traded funds will show up in the users wallet, or the transaction is reverted, in which case funds sent for trading stay in the wallet. This is a majoy benefit compared to centralized exchanges. Users looking to trade first need deposit funds into the exchanges wallet. While the funds will show up as balance in the users account, the user has lost custody of these funds complety and trusts the exchange to honor withdraw requests. The wallets of exchanges contain a large amount of funds at all time. At the same time the wallets balances are public knowledge, which results in centralized exchanges constantly being targeted by hacking attacks. These hacks are successful frequently, mostly resulting in affected users loosing their funds.

Permissionless These systems are also premissionless by design. No entity is able to 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. Its a very simple process, that anybody with broad knowledge of the Ethereum ecosystem can do. There is no mechanism to allow pair additions or to verify them. Anybody deciding to add a pair, can do that, with nobody beinfg able to prevent it. The situation is the same for trading. Anyone with an Ether wallet and enough balance, can trade at will. There are no restrictions in place. It can somewhat be seen as a utility service, that is available for anyone to use at will, living on the Ethereum blockchain forever. As a concept this is very new, shoule be explored further.

0.1.2 Pricing and Liquidity

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

0.1.3 Centralized Order Book

Centralized exchanges typically rely on the central order book to facilitate the trading of an asset pairs. 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, if its a buy or sell order, the amount to be bought/sold and a price that is 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 funds, thereby adding liquidity. It is important to note, that adding the order doen't 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 infeasble on the Ethereum blockchain. Memory and processing are very limited and expensive to use in smart-contracts, so a more efficient model must be found.

0.1.4 Automatic Market Maker Protocol

Uniswap solves this, 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

[1] For example, adding a large Bitcoin buy order at a price of 1$ is not considered a liquid asset.

fact the a liquidity pool always contains an equal value of both assets, enable 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 formular:

$$k = x \cdot y$$

where:

x is the amount of asset X

y if 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, that is 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 price impact of a trade is dependant on the size of the trade and the amount of funds in the pool. The asset price of a pool containing assets worth millions of dollars will be impacted less by a trade then 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 seperate class of users for liquidity, the so called liquidity providers. A liquidity provider can deposit funds into liquidity pool, having to provide 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 are open source and can be verified by anyone. Funds can only be withdrawn from a pool by burning the liquidity provider token, which reside in the liquidity providers 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 formular 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. This mirrors 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. On top of that, the amount of liquidity available any amount away from the current price is symetric 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. [**?**] have shown that this simple mechanism results in very accurate pricing on Uniswap.

0.1.5 Placeholdeer

By using these simple mechanisms, Uniswap sucessfully built a fully decentralized exchange. However, because of rising usage of the Ethereum blockchain, the cost per trade has increased signifcantly over the past couple of months. A number of techniques have been proposed of the last couple of years, how transaction aggregation could be used to reduce transaction costs.

0.2 Ethereum Scaling Solutions

Over the years a number of scaling solutions have been proposed that aim at increasing the transactional thoughput, 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.

0.2.1 Layer-1 Scaling

A blockchains throughput can be described best by looking at the rate data that can be processed. New blocks are typically sealed at a roughtly 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 throughput. Layer-1 scaling can be roughly seperated into two differnt approaches, that can be compared to horizontal and vertical scaling in a typical server contrext.

Vertical Scaling In IT systems the simplest solution often is to switch to a server with more capable hardware. This can be compared to scaling the data throughput of a blockchain by increasing the block size limit. By allowing more transactions to fit in a block, the throughput increases. While this can work for a while, its not a sustainable approach over the long term and will eventually result in centralization of the networks nodes or an unstable network. Just like with traditional software systems, the speed up of upgrading the hardware become unviable at 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 to a certain extent, resulting in much higher data throughput. In blockchain systems this can be achived by sharding the network. Instead of every node processing every transaction, we seperate the network into shards, each processing a different set of transactions. Increasing the number of shards now increases the data throughput, as we're able to paralellize 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]. For this reason, there is a limit of how many shards the network can be seperated in. 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.

0.2.2 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 data on the blockchain, the general goal is to only publish it in a compressed form. The remaining data is published in a layer-2 network. Layer-2 is a loosly defined term, that is used to describe a secondary protocol that is built on top of a blockchain. The state of layer-2 is committed to the underlying blockchain to ensure 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 conceptialized and built over the last couple of years.

State Channels State channels [?] are a scaling technique can be used to enabled practically gas free transactions. This technique relies on representing 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 represent a withdraw of funds according to the post transfer balances of the participants. The signed transactions

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

however, don't 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, fortunetly different channels can be connected through other participants. State channels can be a useful technique for handling frequent and repetetive payments. There are a number of disadvantages, that limit the potential. For one, participants need to open a state channel to receive payments. The mechanism in place to handle non-responsive participants also requires participants to be online to prevent fraudulently broadcasted withdraw requests.

not best wording

Plasma Plasma is a scaling technique proposed by Poon et al. [?] 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 funds onto a plasma chain, a user deposits them into the 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, where anyone can challenge the correctness by submitting a merkle fraud proof and invalidate the batch. Finality of transactions is reached after the challenge period. As with state channels, the correctness of state transistions is relient on every participant checking if their balance has been updated correctly. This is a powerful technology for scaling simple transfer of assets, but is difficult to apply in applications that depend on state without a logical owner [?].

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 propergate all included transaction 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 started. But what happens when the operator includes a malicious transaction to compute the state root but doesn't propergate that transaction? We're not able to create a fraud proof on data we don't have. Ensuring all data is published can be enforced to a seperate 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. This creates a game theoretical problem known as the fisherman problem [?]. For a fisherman, there are three potential ways it can be rewarded for raising the alarm, taking a loss, breaking even or making a profit, all of which resulting malicious behaviour being incentivised 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 very compressed explaination of the data availability problem, as its a complex topic. In the context of this work, its important to understand that storing data exclusivly in layer-2 adds significant complexities that result in a game theoretical problem, that has not been solved at the time of writing. In Plasma this is solved by having a logical owner for all states, that ensure correctness of their state.While this can work, it adds liveliness assumptions to the system and also 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, while increasing the transactional throughput. In general, data availability can only be solved by having a trustless mechanism in place that publishes the data to a system that is known to always be online. In a blockchain systems this means the data needs to be stored on-chain. A rollup application is controlled by a smart-contract that handles deposits and withdraws of funds in to layer-2 and stores the root of the layer-2 state. When updating the state, a batch is published, which includes the updated state, along with the old and new state root. If the old state root matches the state root stored in the smart-contract the root is updated, and the updated states are added to the event log[3]. The data published in the event log includes the least possible amount of data, replacing computation for data whenever possible. The further the data can be reduced, the better our scaling works. Because we tie the data publishing to a consensus mechanism, we have solved the data availability problem. One question however remains. How do we ensure the new state root has been computed correctly? This is where the two different flavors of rollup start to differenciate.

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 are able to 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're also able to represent state that doesn't have a logical owner. this enables us to execute smart-contract logic in layer-2, relying on fraud proofs to ensure correct execution. This is a major advantage over the previously discussed techniques, as its a generale purpose scaling solution. A disadvantage of

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

optimistic rollup is transaction finality, as each state update is subject to a challenge period. There are a number of optimistic rollup based scaling solutions set to launch soon.

ZK-Rollup ZK-Rollups, desribed by V. Buterin in 2018 [?], are another approach to rollups. Instead of relying on fraud proof to revert malicious state updates, zk-rollups rely on validity proofs that are submitted along with a batch update. The validity proof verifies that the result of our computation was done with a zkSNARK circuit that correspondes to the verifier used to check the proof. The size to the validity proof is also constant, which allows us to build complex off-chain logic, without increasing the gas cost. Another benefit lies in the finality of state updates. Since a batch is only published if the validity proof is successful, it is reaches finality instantly. Conversly, 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 used in systems that are used for asset transfers, these systems don't integrate with other smart-contracts, relying on them to run an aggragation. In this work, we 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.

0.3 Zero Knowledge proofs

Zero knowlege proofs where first described by Goldwasser et al. [?] in 1989. In their work, they describe a protocol that enables a prover to convice a verifier of knowlege of a specific mathematical statement, without revealing any information of that statement. This can be achived with a interactive proof, where on going communication between prover and verifier is required. Each round of communication further convices the statement to be true. Overly simplified, this can be seen as a game, where a computationally bounded verifier is convinced by a computationally unbounded verifier that a certain statement is true. Since the entire proof is probobalistic, amount of communication rounds influences how likely the proof is to be valid. To work, the protocol needs to fulfill a number of properties [?]:

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 convice the verfier[4].

3. **Zero-Knowledgeness:** The verifier learns nothing about the proven statement, except that it is true.

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

Their system was able to prove quadratic residuosity to a verifier, without revelaing the factorization. This is very similar to proving knowledge of a factorization in the discrete logarithm problem, without revealing it. Given that dicrete logarithm is one of the foundations of modern cryptography, this marks a breakthrough. This discovery was still very limited however. 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. [?], who proved that all NP problems have zero knowledge proofs. This however is dependant on the existance of one-way functions, which is an open problem in computer science until this day, and requires an unbound number of communcation rounds between the prover and the verifier.

0.3.1 Development of zkSNARK

The real breakthrough in zero knowledge proof protocols can be attributed to Gennaro et al. [?] making the proving non-interactive and the proof size succinct and Parno et al. [?] who further reduced the proof size and verification time of a proof. These developments result in what is known as 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. In 2016 J. Groth [?] presented a new proving scheme, that further increases the efficiency of zkSNARK. These properties now enables the practical application of the technology.

0.3.2 Foundations of zkSNARK

With zkSNARKs, arbitrary statements can be converted into a zero knowledge proof, and proven to a verifier without revealing any information on what is proven. Especially important is the constant proof size, and the quick verification of the proof, even on restricted hardware. This enables verifiable computations of arbitrary sizes to be verified on pretty much any 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 flattend, converted into a Rank-1 Constraint System (R1CS) and converted once again into a Quadratic Arithmetic Program (QAP).

Trusted Setup When evaluating a proof, the verifier doesn't evaluate the entire statement, but only random points of it. The idea is, that given that a prover isn't aware which points are being evaluated, cheating by guessing is unlikely to happen. Since zkSNARKs are non-interactive, the verifier can't request the result of the circuit at different random points. For this reason, the points that are being evaluated must be decided upon before hand, which happens during the setup. The setup can be seen as a function that receives the compile 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 its called 'toxic waste'. It is the achilles heel of every non-interactive zero-knowledge proof protocol and deleting it once the setup is complete is essential. There are approaches being developed to make this trustless that are mentioned in S. **??**.

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

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 are used with it. If the verify program returns true, the verfier known the public inputs where computed by using the circuit specified in the setup step.

0.3.3 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 can't be deleted, as it would corrupt the entire blockchain, making it impossible to verify. This limits the potential applications, as certain data needs to stay private. 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 that was used. The same properties can also be used to offchain complex computations that are unviable to run on-chain. Given that our proof is succinct, we can create arbitrary sized circuits and 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 big implications for scaling blockchain systems.

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

0.3.4 Other Zero Knowledge Constructs

There are a number of 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. [**?**], bulletproofs are a non-interactive construct that dont require a trusted setup. A downside compared to zk-SNARKs 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 complexity increases with O(N). This results in an expensive on-chain verification of the proofs, which is not ideal for the usecases outlined above.

STARKs Developed by Ben-Sasson et al. [**?**], STARKs are also non-interactive and don't 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.

0.4 Replay Attacks

A common security consideration to make in cryptographic protocols are replay attacks. A replay attack occures when a valid cryptografic proof is maliciously resubmitted, and no security checks are in place to invalidate the resubmission. When making a transaction in a blockchain network, the transaction is signed with the users private key. Miners that receive transactions check the signatures validity, which authorizes the transaction as the user has access to the addresses private key. However, a malicious actor could store these signed transactions and resubmit them to the network. Since the signature is still valid, a mechanism is needed to invalidate transaction signatures, once they have been included in a block and thereby executed. In most blockchain systems, like Bitcoin or Ethereum, this is solved by tracking a nonce for each address. Miners extract an addresses current nonce by indexing the entire blockchain, simply counting the number of confirmed transactions. Among other things, the nonce is part of the signed transaction. When a miner checks the signature, it is also checked if the nonce set in the signed transaction has been incremented by one, when comparing with the nonce that was extracted. This invalidates a resubmission of the signed transaction and prevents replay attacks.

0.4.1 Transaction Replay Attacks in zkSwap

The way transactions are executed in this system is quite different, compared to the blockchain systems described above. As we will explorer, most verifications and computations happen off-chain and are then verified on-chain with a single proof, that

When making a transaction, which can be a trade, a deposit or a withdraw, a user signs the transaction details and sends them to our off-chain entity, the aggregator. The signature is then checked in a zkSNARK circuit, along with the nonce. The resulting proof can then be used to verify the correct execution of the zkSNARK circuit, as an effect verifying the correct. While the signature check happens off-chain, the mechanism of preventing replay attacks is comparable to most blockchain systems. By checking if the nonce signed in the transaction details has been incremented, we ensure the signed transaction is invalidated when resubmitted.

0.4.2 Proof Verification Replay Attacks

Another consideration to make, is the submission and verification of the zkSNARK proof. It also could be resubmitted, thereby breaking the protocol if no measures are in place to prevent this. When verifying a zkSNARK proof, we're essentially proving the execution of the circuit was done correctly, and that the resulting outputs where computed by running the circuit. In this system, the zkSNARK circuit is used to ensure correct state transitions tbc