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.

0.1 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.

0.1.1 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.

0.1.2 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.

0.1.3 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.

0.1.4 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

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

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:

$$k = x \cdot y$$

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. [**?**] 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.

0.2 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.

0.2.1 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.

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 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 [**?**] 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 withdrawal 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

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

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

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 [**?**] 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 signifi-

cant 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.

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

ZK-Rollup: ZK-Rollups, described 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 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 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.

0.3 Zero Knowledge Proofs

Zero knowlege proofs where first described by Goldwasser et al. [**?**] 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 [**?**]:

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

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

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. [**?**], who proved that all NP problems have zero-knowledge proofs. All NP problems having a zero-knowledge proof relies on two assumptions: 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.

0.3.1 Development of zkSNARK

The real breakthrough in zero-knowledge proof protocols can be attributed to Gennaro et al. [**?**]. Their work made the proof non-interactive and the proof size constant, bringing the technology into the realms of practical applications. Parno et al. [**?**], and J. Groth [**?**] 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.

0.3.2 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 the 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. **??**.

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.

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 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.

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

0.3.4 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. [**?**], 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 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. [**?**], 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.