zkSwap - Scaling Decentralized Exchanges through Transaction Aggregation

Paul Etscheit

March 2021

1 Introduction

When being launched in 2015, Ethereum [10] set out to change the way we compute. A trustless, permissionless, and decentralized world computer was envisioned, set to open a new class of applications. The importance of running verifiable, Turing-complete code in a permissionless and trustless manner cannot be overstated and enables products and services not thought to be possible. However, the technical limitations have also become apparent quickly. Computations are expensive, theoretical transactions per second are low, and the overall throughput has been stagnant. While Eth2 gives a path towards scaling the network, it is expected to take years to complete.

The first major use case for Ethereum was tokenization. With the development of the ERC-20 standard, launching a token on the Ethereum blockchain was trivial. As tokens run as smart-contracts on the Ethereum blockchain, they are secured by its proof of work consensus, which, given Ethereums PoW hash rate, makes consensus attacks infeasible. Running on Ethereum blockchain is a significant benefit when looking to tokenize things, as network security can be assumed. While tokenizations are a step in the right direction, they do not come close to the initial vision. While the standardization enables simple integrations into exchanges and wallets, most tokens are isolated in their functionality and ecosystem and lack productive usage.

token all isolated, not working together... Not a lot of gas needed blabla

With all of these developments over the past couple of years, it seems we have now entered a new phase of smart-contract use-cases, namely Decentralized Finance (DeFi). While DeFi has many different products and functionalities, at its core aims to utilize tokenized assets in some productive form.

Lending and collateralized borrowing is possible with Aave [7], assets can be deposited into liquidity pools[2] to generate yields, flash-loans [7][2] enabled uncollateralized borrowing as long as the loan is repaid in the same transaction and assets can be traded in a non-custodial way with Uniswap [2] . It can be questioned how useful or necessary these protocols really are, but the core idea behind them is impressive. Rebuilding traditional financial products, running as non-custodial and permission-less smart-contracts, all based on the same standardizations, has the potential to reshape the way finance works. With these developments not looking to slow down, they are quickly overwhelming the Ethereum blockchain, pushing transaction costs [5] higher and higher.

Mention other swap protocols?

One of these new DeFi applications is Uniswap [2]. Uniswap is a crypto-asset exchange running as a collection of smart-contracts on the Ethereum blockchain, enabling non-custodial, trust-less, and permission-less trading of ERC-20 assets. Since its running on the Ethereum Blockchain, reducing the computational complexity of trade execution is essential for making it a viable product. In typical crypto-asset exchanges, trading is built around a central order book. Users can add buy or sell orders for a given trading pair, and a matching engine checks if these orders can be matched, executing the trade once they do. While running this on modern server infrastructure is feasible, running it on the blockchain is not. The demand for memory and processing power is too large, so a different approach must be taken. Uniswap solves this by applying the automated market maker (AMM) model, which will be explained in detail in sec. XX. By applying this model, Uniswap reduces the computational complexity to make this a viable business model. At least it was, when Uniswap launched.

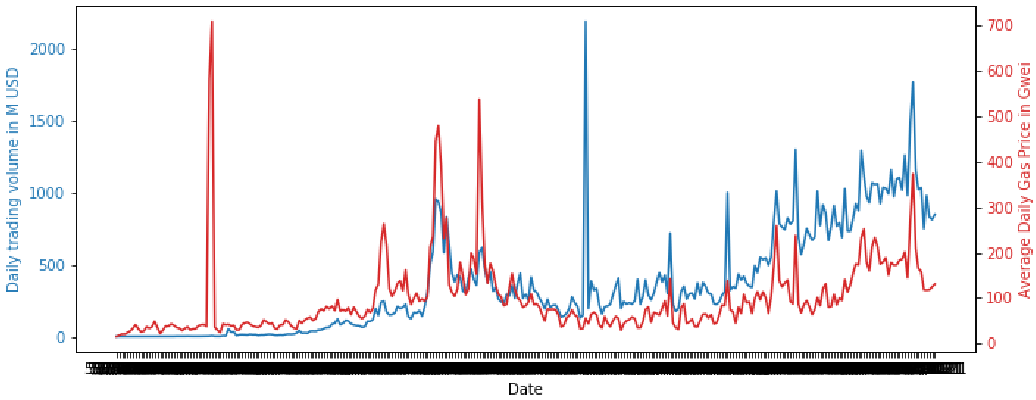

Figure 1: Combined daily Uniswap trading volume in million USD and average daily Ethereum gas price

The recent rise of Ethereums gas price [5] can also be attributed to the growing popularity of Uniswap. Currently, it is one of the most used smart-contract on the Ethereum blockchain, making up on average around 15% of gas [4] usage of a block at the time of writing. To date, it accrued over $280 million in transaction fees [6] and has settled over $100 billion in trading volume [1]. 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.

With longer-term scaling solutions in development but still years away, a shorter-term solution is needed. A couple of short-term scaling approaches have been proposed. While these do differ, they all aim to move transactional data to a layer-2[1] system, while ensuring correctness of that data in some way. One of the approaches is called zk-rollup, the focus of this work. Moving data to a layer-2 system can increase the number of transactions that fit into a block while also reducing transaction costs for the user, with is beneficial for Uniswap users and all other participants of the Ethereum network.

[1] A layer-2 system is a data storage that does not reside on the blockchain but has its state committed to it in some way

Current zk-rollup enabled applications running on the Ethereum mainnet, one of them being ZK Sync, are focused on reducing cost of Ether and ERC-20 transfers. Users deposit funds into its smart-contract, which results in the user's deposit being represented as balance in layer-2. When a user makes a transfer to another user, the involved balances get updated in layer-2, while the correctness of these updates is ensured via zkSNARK. It is important to note, that ZK Sync acts as a closed system, transfers only change balances in layer-2, while deposited funds in the smart-contract do not move. While this approach has significantly reduced costs of transfers, it only marks the first generation of potential(?) zk-rollup enabled apps.

not a great sentence

Aggregating Uniswap trades is an interesting application to explore the potential of zk-rollup technology. It combines the layer-2 storing and updating of balances already done by ZK Sync while opening the system to interact with other smart-contracts. When aggregating trades, we need to interact with the Uniswap contracts to execute the aggregated trade, then update the layer-2 balances according to the trade and verify everything via zkSNARK. It is the next step in exploring the potential of zk-rollups as a generalizable scaling solution, applicable to any kind of smart-contract.

2 Background

2.1 zk-rollup

Zk-rollup [9] is a layer-2 scaling approach first introduced to mass validate transfer of assets on the Ethereum blockchain in 2018. A user can deposit funds into a smart contract by providing a merkle path to its balance and adding funds to be deposited to the transaction. The smart-contract checks if its root can be recreated with the provided merkle path, updates the balance according to the funds in the transaction, rehashes the entire tree and updates its root to the resulting hash. An event is then emitted, containing the new balance, putting the data on-chain is a cheap way. The merkle tree, which is required for generating the merkle paths, containing the balances can be kept in sync by listing to these events.

To make a transfer, a user sends the receivers address, tranfer amount and its signature to the relayer as an http request. Once enough transfer requests have been received, the relayer checks if this transfer is covered by a users balance and if the signature is valid, and updates the balance of involved users accordingly. All of this is done in a zkSNARK program, which will return a proof object, the new balances and the new merkle root. These are then sent to the smart-contract. If the zkSNARK proof can be verified, we have proven that the new balances and the root are correct. We now emit the new balances as an event, thereby moving custody of transfered funds to the receiving users, who are now able to transfer or withdraw them. Withdrawing of funds follows the same logic as depositing, however instead of sending funds, a parameter is added containing the requested amount.

3 Design

The goal of the work is to explore if zk-rollups can be used to aggregate Uniswap trades in an effective manner. The prototype is able to aggregate trades for a single trading pair, Ether and an ERC-20 token of choice. The system consists of two main entites that are required for it to function. The first entity to look at, is the on-chain entity, we call zkSwap. zkSwap is a smart-contract deployed on the Ethereum blockchain and has three main jobs, processing deposits and withdraws, aswell as verifying batched trades. It holds users funds and exposes the on-chain functionality, namely deposits and withdrawls, to the user.

The second entity to look at is the aggregator. The aggregator consists numerous systems, both off-chain and on-chain, and is mainly tasked with receiving trade orders, aggregating and executing them, and then verifying them with the zkSwap contract. The aggregator stores a merkle-tree of users balances and keeps it in sync by listening for event emitted by the zkSwap contract.

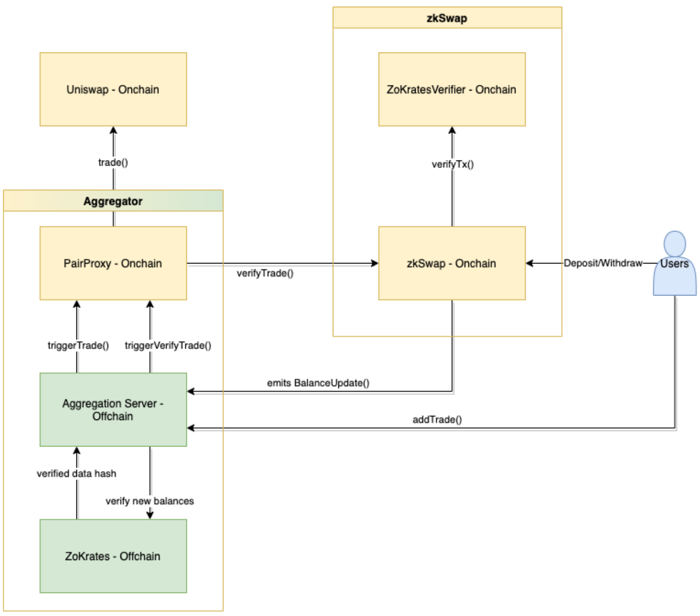

Figure 2: High level architecture of the system

3.1 zkSwap Contract

The zkSwap contract, is the core entity the user interacts with. Its a smart-contract, that from the users perspective, is mainly used for depositing and withdrawing funds in out system.

3.1.1 Storing and Updating Balances

Two main factors are dictating the way balances are stored in the system. It is important to understand the core technique used to store and update balances before we look at the different functions that trigger them. We want to make balance 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 important to make a distinction between storing data in a smart-contracts runtimeand storing data in the event log. Both are on-chain, the event log is significantly cheaper though. While the event log is cheap to use and can be queried by any client, it is not accesible to a smart-contracts runtime. This solves the external data availablity problem. We can store balances cheaply, 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 mechansism to ensure, the parameters passed are equal to the parameters found in the event log.

is runtime the correct word?

We can achive this, be using a merkle tree [8]. Merkle trees are a suitable data structure, as its root represents the entire tree state in a highly compressed form, while proving a leafs inclusion in the tree can be done with O(log n). This is 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 client can query balances from this tree, receiving the valid merkle path along with the balance object. Since the smart-contract stores the root of the balance merkle tree in its runtime, it can verify the correctness of the passed balance by hashing it along with the provided merkle path. If the resulting hash is equal to the stored root, the correctness of the balance is proven.Any changes in the balance object by the client will result in the hashes to mismatch, thereby invalidating the data. While the hashing of the balances and merkle path does create some fixed, overhead cost, the savings by using the event log far outstrip it. The hashing costs will be analysed in S. Results.

Should I explain on what security assumptions this is based

This fulfills the balance storage requirements, reducing costs, not relying on external data availabilty and ensuring the correctness of data. It must be noted, that this also causes the smart-contract to be the single source of 'truth'. Since all balances changes are committed by a new event being emitted and the root being updated, any balance updates must be done through the smart-contract.

3.1.2 Deposits and Withdraws

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, which in turn give custody to these funds. When moving funds in layer-2, we don't actually move the funds residing in the smart-contract, but update the balance objects to represent the movement and verify that movement for correctness with a zkSNARK proof. Since a balance object gives a user custody of represented funds, it can always be redeemed, moving from layer-2 back to mainnet. Maybe remove this?

As described in S. 3.1.1 balance updates are secured by merkle inclusion proofs in the smart-contract. When depositing the user needs to provide a valid merkle path to its balance object, and the balance object itself. The balance object consists of four fields that are needed to represent the balance: ethAmount, tokenAmount, nonce and userAddress. As a first step, the balance object is hashed with the sha256 algorithm, the result being the leaf in the merkle tree. The leaf is now hashed with the merkle path, according to the standart merkle root hashing algorithm. If the resulting hash equals the stored balance root in the contract, we have proven, that the passed balance object is correct. We now add the value of the transaction object, which is the deposit amount sent with the transaction, to the ethAmount, increment the nonce and hash the new balance. Since the balance object has the same position in the tree, the same merkle path is also valid for the new balance. The new leaf is hashed with the merkle path, and the resulting hash is set as the new balance root. As a last step we emit the 'BalanceUpdate' event, containing the new balance object, matching the balances with the newly set root.

Withdraws largely follow the same logic, there are small differences though. As we're requesting funds, instead of sending them, we pass a withdrawAmount as an additional parameter to the function call. We then check if the withdrawAmount ¡= ethAmount to make sure the withdraw is covered by the users balance. The inclusion proofs are the same, we then subtract the withdrawAmount from the balance, and update the balance root. As a last step we emit the 'BalanceUpdate' event with the new balance object, and send requested funds to the user as an on-chain transaction.

This however, is an incomplete explaination, as we're not checking if a user is permitted to deposit or withdraw funds. As balance objects are emitted as an event, anyone can access them and compute valid merkle paths for any balance. This would allow any user to withdraw any balance. To ensure a user is permitted to update a balance object, we need ensure the user controlls the private key belonging to the balance objects user address. Fortunetly we can ensure this by accessing the sender in transaction object. The Ethereum blockchain ensures a user is allowed to make a transaction by requiering the transaction to be signed with the private key of the senders address. If that signature is valid, it is proven that the user has access to the addresses private key and the transaction can be executed. Because of this, the transactions object sender can be trusted to be in controll of the corresponding private key. Instead of passing the users address as part of the balance object, the smart-contract uses the sender of the transaction object. This suffices as a security check.

It must also be noted, that ERC-20 deposits and withdraws behave a bit differently. As every ERC-20 token has its own smart-contract, representing a users balance as a mapping, we need to transfer the assets in that contract. This requires different functions to handle ERC-20 deposits and withdraws, however implementing this is trivial and not worth explaining in the context of this work.

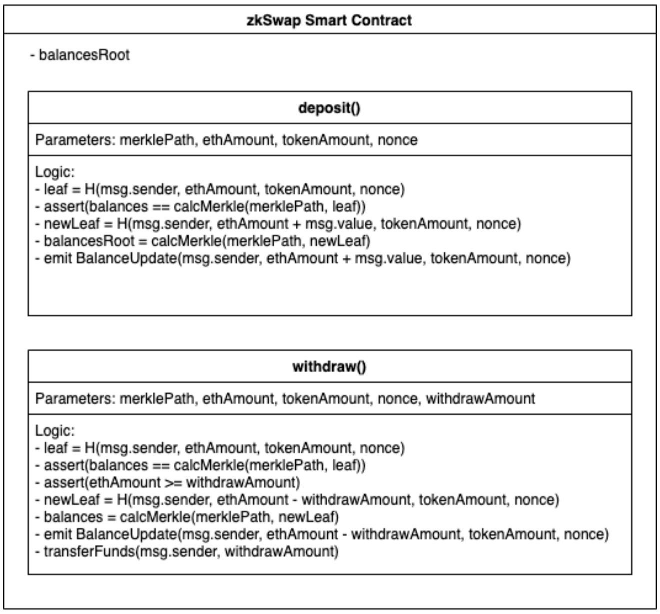

Figure 3: Pseudocode of the deposit and withdraw steps

3.1.3 Batch Verification of Trades

When trading on Uniswap, every user send a seperate trade transaction, and each trade results in an on-chain movement of funds. This uses quite a lot of gas. zkSwap aims to change this, by aggregating a number of trades into one, executing it and sending the traded funds to th zkSwap contract. The balances of all involved users are then updated accordinly, giving custody to these funds. Since the aggregation happens off-chain, we need to verify its correctness by utilizing a zkSNARK proof. The content of the zkSNARK program, and performed checks and computations will be described in S. XX, we will now focus on the verification step on-chain.

The zkSwap contract has access to the ZoKrates verifier, which is used for checking if a submitted proof is valid. A valid proof implies, that inputs of the proof have been computed by using the zkSNARK program used for generating the verifier, and with the public inputs defined.

At its core, a trade on Uniswap is only a update to a users balance. Certain assets are subtracted, other added to the account. The main difference to zkSwap is that traded assets are moves as an on-chain transaction, which uses a large amount of gas. At the same time, every users wishing to trade will do so in a seperate transaction.

When trading on Uniswap, a user exchanges funds, swapping one for the other. This happens as an on-chain transacation and uses quite a lot of gas. When trading with zkSwap, the balance of a user changes representing the trade in layer-2. However, we want to

At the same time, the funds residing in the smart contract must be able to cover the withdrawl of all balances at all times.

a users funds is changed in layer-2, representing the funds swapped. A user always has the option to withdraw these, but is free to leave them in layer-2.

3.2 Aggregator Entity

As the name implies, the aggregtor is tasked aggregating the incoming trades. In order to function, a couple of services are needed, with run as smart-contracts on-chain or on a classical server.

We will now explain each service.

3.2.1 Merkle Tree

The first thing to look at is the merkle tree, the aggregator is running. As previously discussed, all balance updates will be committed by the zkSwap smart-contract by emitting the 'BalanceUpdate' event. By subscribing to these events, the merkle tree can be built and kept in sync, always providing the complete merkle tree belonging to the balance root stored in the contract. When a 'BalanceUpdate' event is received, the balance object is extracted, the corresponding leaf is found in the tree and then replaced with the new data. Rehashing the tree should now result in the balance root set in the contract. The state of the merkle tree can only change by incoming 'BalanceUpdate' events.

3.2.2 Aggregation Server

It is important to remember, that trade orders are completly off-chain and are sent as an HTTP request. The orders are received and processed by the aggregation server, which at a later stage will run the aggregation. A number of checks are performed on the aggragtion server when a trade is received. These checks are techniqually not needed to ensure the correctness of the aggregation, as the ZoKrates programm performs the same checks at a later stage. They are however needed, to prevent the server from processing invalid trades, which would cause the ZoKrates program to exit in an error state, preventing the entire aggrgation. A trade is invalid, if it fails any of the checks described in this section.

Since this operation is running off-chain, we first need to verify the user is authorized to make the trade order. This can be achived by requesting a signature from the user, verifying it is in controll of the addresses private key.

However, it must be remembered, that this signature must also be verifiable in our ZoKrates programm, which is unable to utilize the secp256k1 curve, used for signing Ethereum transaction, efficiently [3]. For that reason the BN128 curve is used in combination with the EdDSA signature scheme, which can be more efficiently run in a ZoKrates program. The user signs the current balance root, ensuring two things. It proves that the user has access to the addresses private key, which authorizes the trade order. By signing the balance root, we make sure, that the signature can't by reused in a replay attack. For instance, the aggregator could decide to store these signatures secretly, and reuse them without the users consent if this was omitted.

When the trade order is received, the aggregator first checks if the signature is valid and contains the current balances root, by querying it from the zkSync smart contract. As a next step, it is ensured, the users balance is able to cover the trade. The aggregator queries the merkle tree for the users balance, and checks if the trade can be covered by deposited funds. A last thing to consider is ensuring the correct price of a trade. The price between two assets is constantly changing. At the same time, we're collecting trades in order to aggregate them. This results in a delay between an user sending a trade order and the actual trade execution, during which the price can change drastically. For this reason the zkSwap smart-contact stores a 'worst-case' price for buy and sell orders, which is used to calculate the trade order. The worst-case price is used to invalidate an order at a later stage, if the price of an asset pair has crossed it. At this stage, it is ensured that the trade orders price matches the worst-case price stored in the zkSwap smart-contract. If all of these checks pass, the order is added to the trade pool, where it resides until the aggregation starts.

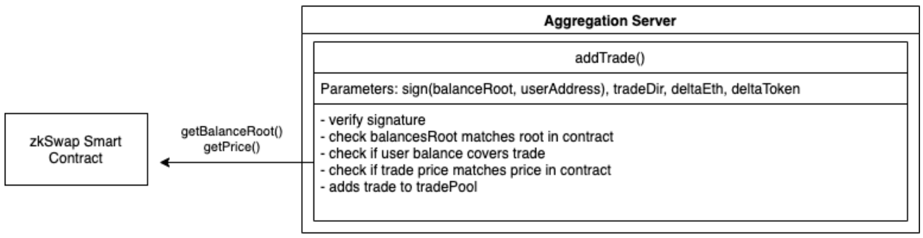

Figure 4: Pseudocode of addTrade()

At some point the trade aggregation is started. This could be triggered by a set blocknumber, the number of trade orders that have been 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 first offset those internally. By doing this, we're able to reduce the entire aggragation to one Uniswap trade, which saves gas. At the same time, we're saving on the 0.3% liquidity provider fee, which is charged based on a trades volume. The net trade is the result of off-setting all trades in aggregation, which results one side to equal zero. This trade is now sent as an on-chain transaction to the ‘PairProxy‘ contract.

3.2.3 PairProxy Smart Contract

Before explaining the functionalities of this smart-contract, it is important 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[2] for the trade. Since network congestion and the current gas price influence when a transaction is executed, it's a neccecary mechansism for ensuring 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 functions return value. Since this amount might be larger then the amount passed as minAmountReceived, we need it to calculate the post-trade balances.

However, a quirk in Ethereums way of handling return values makes this more difficult. A smart-contracts functions 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, instead of receiving the return value of the function, we receive the transaction recipe, which doesn't contain the return value. For this reason, we need the PairProxy smart-contract, which receives transanactions, forwards them to the respective smart-contract, emitting the return value as an event, which can be consumed by the transactor.

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 securly.

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

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

Bibliography

[1] Uniswap cummulative volume, https://duneanalytics.com/projects/uniswap

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

[3] Deml, S.: Efficient ecc in zksnarks using zokrates (Aug 2019), https://medium.com/zokrates/efficient-ecc-in-zksnarks-using-zokrates-bd9ae37b818

[4] Ethereum gas guzzlers, https://ethgasstation.info/gasguzzlers.php

[5] Ethereum gas price, https://etherscan.io/chart/gasprice

[6] Uniswap total fees used, https://etherscan.io/address/0x7a250d5630b4cf539739df2c5dacb

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

[8] 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)

[9] 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-t

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