Lissy: Revisiting fully on-chain orderbooks

No Institute Given

Abstract. Order-driven exchanges on Ethereum tend to operate in a fully centralized fashion, or with some functionality performed on-chain (*e.g.,* loading accounts, order cancelation) and some off-chain functionality (*e.g.,* matching orders). Fully on-chain solutions have a superior threat model and were experimented with in the early days of Ethereum, but they have been largely abandoned for performance reasons. As blockchain performance continues to improve (both in research and in practice) and on-chain dealer-based exchanges like Uniswap (an alternative to order-based exchanges) rise in popularity, it seems to be an ideal time to revisit the idea of a fully on-chain order-driven exchange and benchmark what is possible performance-wise. We design and optimize an Ethereum-based call market exchange, Lissy, and conduct a variety of experiments to show the current benchmark is around one hundred trade executions per block. We also explore numerous extensions for further improvements.

1 Introductory Remarks

For better or worst, blockchain technologies like Ethereum have dramatically lowered the barrier to entry for developing and deploying financial technology. New tokens have been launched with a few clicks of a user interface, and large investment infrastructures have been developed and deployed with little regulatory oversight. Blockchain exchange services allow order-based trading of digital currencies, tokens, and other digital assets. Such exchanges are a key component to blockchain-based economic activity.

For years, exchanges have operated either entirely off-chain by trusted operators or partially on-chain by semi-trusted operators. A fully on-chain exchange is feasible but considered too slow, at least for a public blockchain like Ethereum. We believe it is a very good time to do a deep dive into understanding precisely *how slow* for the following reasons: (1) public blockchains are becoming faster (both in theory and slowly in practice) and the efficiency of an on-chain exchange will increase in time, (2) demand for on-chain trading is exemplified by the recent popularity of dealer-based (or quote-based) trading like Uniswap and Curve Finance, and (3) stablecoins have become a popular and allow on-chain trading with pricing in USD (or other government currency).

Contributions. We design and implement Lissy, a fully on-chain call market in Solidity—a high-level programming language for Ethereum. We choose Ethereum as a 'hostile' environment for an orderbook: Ethereum is public

(strongest adversary) and slow (lower-bound benchmark) relative to more scalable blockchains, including the anticipated Ethereum 2.0. Lissy is intended as a simple module that developers can modify as they choose. By adding Lissy to an ERC20 (or similar) token contract, everything that is needed for buying/selling tokens is immediately available on chain without involving any third party exchanges. Lissy is non-custodial, transparent, and mitigates front-running attacks that are generally possible on-chain.

While Solidity and EVM are like many common programming languages, they also have quirks that require experimentation to best optimize performance (*e.g.,* factoring in the gas costs of operations, gas refunds, limits to Solidity's object oriented design, clearing mappings). We test five priority queues—the core data structure of the call market—and various options for cleaning up our data once finished with it. The bottom line is that the current benchmark for a Lissy-esque design is in the low hundreds of trade executions per block on Ethereum today. This positions Lissy as a feasible design for certain types of markets today (low liquidity, small number of traders) but also as an early study on technology that is likely to improve vastly in the coming years.

2 Preliminaries

2.1 Market Structure

Execution systems. There are three main approaches to arranging a trade [10]. In a *quote-driven* market, a dealer uses their own inventory to offer a price for buying or selling an asset. In a *brokered exchange*, a broker finds a buyer and seller. In an *order-driven* market, offers to buy (*bids*) and sell (*offers*/*asks*) from many traders are placed as orders in an orderbook. Order-driven markets can be *continuous*, with buyers/sellers at any time adding orders to the orderbook (*makers*) or executing against an existing order (*takers*); or they can be *called*, where all traders submit orders within a window of time and orders are matched in a batch like an auction.

2.2 Decentralized Order Matching

Central exchanges (CEX). Traditional financial markets (*e.g.,* NYSE and NASDAQ) use order-matching systems to arrange trades. An exchange will list one or more assets (stocks, bonds, derivatives, or more exotic securities) to be traded with each given its own orderbook priced in a currency (*e.g.,* USD). Exchanges for blockchain-based assets (also called cryptoassets by enthusiasts) can operate the same way, using a centralized exchange (CEX) design where a firm operates the platform as a trusted third party in every aspect: custodianship over assets/currency being traded, exchanging assets fairly, and offering the best possible price execution. Security breaches and fraud (*e.g.,* MtGox [13], QuadrigaCX [14], and many others) in centralized exchanges have become a common source of lost funds for users, while accusations of unfair trade execution have been levelled but are difficult to prove.

On-chain orderbooks. For trades between two blockchain-based assets (*e.g.,* a digital asset priced with a cryptocurrency, stablecoin, or second digital asset), order matching can be performed 'on-chain' by deploying the order-matching system either on a dedicated blockchain or inside a decentralized application (DApp; *a.k.a.* smart contract). In this model, traders entrust their assets to an autonomously operating DApp with known source code instead of a third party custodian that can abscond with or lose the funds. The trading rules will operate as coded, clearing and settling can be guaranteed, and order submission is handled by the blockchain—a reasonably fair and transparent system (but see front-running below). Finally, anyone can create an on-chain orderbook for any asset (on the same chain) at any time. While they sound ideal, performance is a substantial issue and the main subject of this paper.

In this paper, we focus on benchmarking an orderbook for a public blockchain (*e.g.,* Ethereum). Ethereum is widely-used and we stand to learn the most from working in a performance-hostile environment (*i.e.,* Ethereum is good lower-bound). Exchanges could be given their own dedicated blockchain, where trade execution logic can be moved at the consensus level. Permissioned blockchains (*e.g.,* NASDAQ Linq, tZero) can also increase execution time and throughput, possibly with some reduction in transparency and trust.

On-chain dealers. Another on-chain advantage is that other smart contracts, not just human users, can initiate trades, enabling broader decentralized finance (DeFi) applications. This has fuelled a resurgence in on-chain exchange but through a quote-driven design rather than an order-driven one. Automated market makers (*e.g.,* Uniswap) have all the trust advantages of an on-chain orderbook, plus they are very efficient relative to an on-chain orderbook. The trade-off is that they operate as a dealer—the DApp exchanges assets from its own inventory. This inventory is loaded into the DApp by an investor who will not make money on the trades themselves, but hope for long term profit through trading fees. By contrast, an orderbook requires no upfront inventory and does not have to charge trading fees (but can). Finally, there is a complicated difference in their price dynamics (*e.g.,* market impact of a trade, slippage between the best bid/offer and actual average execution price, *etc.*)—deserving of an entire research paper to precisely define. We leave it as an assertion that with equal liquidity, orderbooks have more favourable price dynamics for traders.

Hybrid designs. Before on-chain dealers became prominent in the late 2010s, the most popular design was hybrid order-driven exchanges with some trusted off-chain components and some on-chain functionality. Some (*e.g.,* EtherDelta) were envisioned as operating fully on-chain, but performance limitations drove developers to move key components, such as the order matching system, off-chain to a centralized database. A landscape of DEX designs exist: many avoiding taking custodianship of assets off-chain, and virtually all (for order-driven markets) operate the orderbook itself off-chain. A non-custodial DEX solves the big issue of a CEX—the operator stealing the funds—however trade execution is still not provably fair. A mitigation is to issue a proof of correct execution to the DApp

(*e.g.,* Loopring) but these proofs have blindspots (discussed in section 5.6). Lissy offers key advantages in this model.

2.3 Related Work

Blockchain limitations and solution. While an orderbook is a ledger and blockchains provide a distributed ledger, it is not straightforward to drop a continuous-time orderbook onto a blockchain. An older 2014 paper [7] on the 'Princeton prediction market' [4] motivates our work. The authors observe the following limitations of on-chain continous orderbooks: block intervals are slow and not continuous, there is no support for accurate time-stamping, transactions can be dropped or reordered by miners, fast traders can react to submitted orders/cancelations when broadcast to network but not in a block and have their orders appear first (as examined in later work on front-running: [9,8]).

Call markets. The researchers propose using a call market instead of a continuous-time market [7]. Orders are collected and placed into the orderbook over a window of time (*e.g.,* 1 or more blocks), then the market is closed and the orders are processed in batch: the best bids are matched to the best asks in order. If the prices overlap, the miner keeps the difference (which they could extract anyways through front-running). Call markets largely side-step front-running attacks from other traders because reordering trades has no impact (discussed more in section 4.2). The paper does not include an implementation and was envisioned as an alt-coin (Ethereum was still in development in 2014) with market closing being done by the miners themselves as part of the blockchain logic.

Large exchanges, like the NYSE and NASDAQ, use a two minute call market every day at opening and closing time (in between, the exchange runs as a continuous time market). Other exchanges, called crossing networks, also operate as a call market at various times throughout the trading day.[1] Call market are studied widely in finance [10]. Time-sensitive traders submit orders early, especially in crossing networks that close at a randomly determined time (traders risk missing the call if they wait too long). A blockchain happens to provide this function naturally, as blocks are published unpredictably. Price-sensitive traders wait to base their pricing off the already submitted orders and do not mind missing a call if it obtains them a better price.

Other Literature.

3 Priority Queues

In designing Lissy within Ethereum's gas model, performance is the main bottleneck. For a call market, processing all the trades and closing the market is the

[1] A crossing network uses a secondary market for determining the closing price. Many prominent crossing networks are operated internally within a brokerage for its clients, and often as a ' dark pool' with an unpublished orderbook.

Operation	Description
Enqueue()	inserts an element into the priority queue
Dequeue()	removes and returns the highest priority element
isEmpty()	checks if the priority queue is empty

Table 1: Operations for a generic Priority Queue.

most time consuming step. The most critical design decision is the data structure for holding orders. While data structures are well studied for many languages, Solidity/EVM has its own unique aspects (*e.g.,* gas refunds, a relatively cheap mapping data structure, only partial support for object oriented programming) that creates difficulties in assessing which will perform best without actually deploying and evaluating each variant.

When closing a call market, the orders are examined in order: highest to lowest price for bids, and lowest to highest price for asks. It most circumstances, the market closing algorithm does not have to consider any deeper bids/asks from the list when choosing whether the current best bid and ask can be fulfilled. The only exceptions are in the case of a tie on price or a canceled order, both of which we return to later. For this reason, the ideal data structure for storing bids/asks is a *priority queue* (see Table 1) where each order's priority is its price. Specifically, we use two PQs—one for bids where the highest price is the highest priority, and one for asks where the lowest price is the highest priority.

There are numerous ways of implementing a PQ. A PQ has an underlying list—common options include a static array, dynamic array, and linked list. The most expensive operation is keeping the data sorted—common options include (i) sorting during each enqueue, (ii) sorting for each dequeue, or (iii) splitting the difference by using a heap as the underlying data structure. Respectively, the time complexities are (i) linear enqueue and constant dequeue, (ii) constant enqueue and linear dequeue, and (iii) logarithmic enqueue and logarithmic dequeue. As closing the market is very expensive with any PQ, we rule out using (ii) as fully sorting while dequeuing would be prohibitive. We experiment with the following 5 options for (i) and (iii):

1. **Heap with Dynamic Array.** A heap is a type of binary tree data structure that comes in two forms of a (i) Max-Heap and (ii) Min-Heap. All the nodes of tree are in a specific order and the root of the tree always represents the highest priority item of the data structure (the largest and smallest values in the Max-Heap and Min-Heap respectively). We implemented a priority queue with a heap that stores its data in a dynamically-sized array.
2. **Heap with Static Array.** A heap can be also represented by a Solidity storage array in which the storage is statically allocated. To do this, we pass the required size of the array as a constructor parameter to the priority queue smart contract.
3. **Heap with Mapping.** In the above implementations, the heap stores the entire order (as a struct) in the heap. In this variant, we store the order struct in a Solidity mapping and store the mapping's key in the heap.

.	Gas Costs (`gasUsed`)	Refund (Manual)	Full Refund?
Heap with Dynamic Array	2,518,131	750,000	•
Heap with Static Array	1,385,307	750,000	•
Heap with Mapping	2,781,684	1,500,000	•
Linked List	557,085	1,200,000	•
Linked List with Mapping	731,514	3,765,000	•

Table 2: The gas metrics associated with dequeuing 50 items from five priority queues variants. For the refund, (•) indicates the refund was capped at the maximum amount and (○) means a greater refund would be possible.

4. **Linked List.** In this variant, we insert a new element into its correct position (based on its price) when running enqueue. The PQ itself stores elements in a linked list (enabling us to efficiently insert a new element between two existing elements). Solidity is described as object-oriented but the equivalent of an object is an entire smart contract. Therefore an object-oriented linked list must either (i) create each node in the list as a struct—but this is not possible as Solidity does not support recursive structs—or (ii) make every node in the list its own contract. The latter option seems wasteful and unusual, but we try it out anyways. Thus each node is its own contract and contains the order data and a pointer to the address of the next contract in the list.
5. **Linked List with Mapping.** Finally, we try a variant of a linked list using a Solidity mapping. The value of the mapping is a struct with order data and the key of the next (and previous) node in the list. The contract stores the key of the first node (head) in the list.

3.1 Priority Queue Evaluation

Enqueue performance. We implemented, deployed, and tested each priority queue using Truffle and Ganache.[2] We tried a variety of tests (including testing the full call market with each varient) with consistent results in performance. A simple test to showcase the performance profile is shown in Figure 1. We simply enqueue 50 integers chosen at random from a fixed interval in each PQ variant. The bigger the PQ gets, the longer enqueue takes—a linear increase for the linked list variants, and logarithmic for the heap variants.

Dequeue performance. For each PQ variant storing 50 random integers, the `Dequeue()` function is iterated until the data structure is empty. The total gas

[2] Github: Link removed for anonymity.

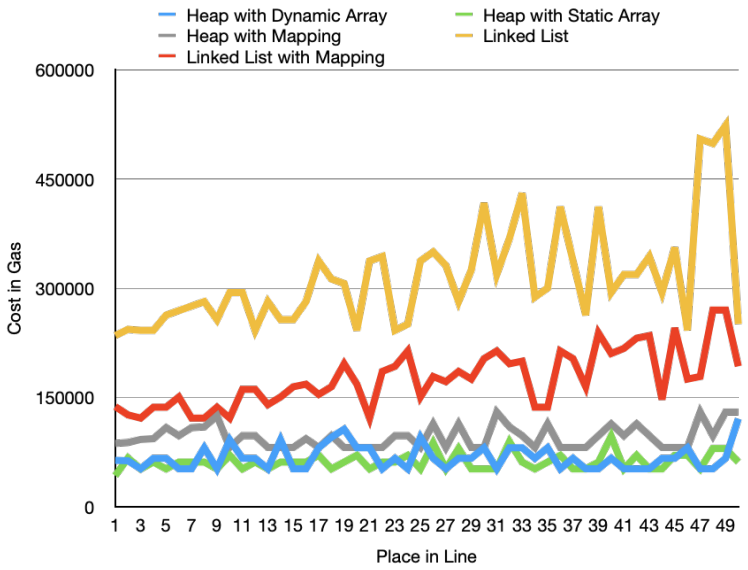

Fig. 1: Gas costs for enqueuing random items into five different priority queue variants. For the x-axis, a value of 10 indicates it is the 10th number entered in the priority queue. The y-axis is the cost of enqueue in gas.

cost for fully dequeuing the priority queue variants is outlined in Table 2. These tests are performed using the following Ethereum gas metrics: block gas limit =11,741,495 and 1 gas = 56 gwei.[3] Dequeuing removes data from the contract's storage. Recall this results in a gas refund. Based on our manual estimates (EVM does not expose the refund counter), every variant receives the maximum gas refund possible (*i.e.,* half the total cost of the transaction). In other words, each of them actually consumes twice the `gasUsed` amount in gas before the refund, however none of them are better or worst based on how much of a refund they generate.

Discussion. Based on enqueuing, the heap variants are the cheapest in terms of gas, while based on dequeuing, the link list variants are cheapest. This is in accordance with the theoretical worst-case time complexity for each. However, (i) the linked list variants are materially cheaper than the heap variants at dequeuing, and (ii) dequeuing in a call market must be done as a batch, whereas enqueuing is paid for one at a time by the trader submitting the order, and (iii) Ethereum will not permit more than hundreds of orders so the asymptotic behaviour is not a significant factor. For these reasons, we suggest using a linked list variant for this specific application. As it can be seen in Figure 1, the associated cost for inserting elements into a linked list PQ is significantly greater than the linked list with mapping, as each insertion cause the creation of a new

[3] EthStats (July 2020): `https://ethstats.net/`

.	Gas Costs (`gasUsed`)	Refund (Manual)	Full Refund?
Linked List w/o `SELFDESTRUCT`	721,370	0	×
Linked List with `SELFDESTRUCT`	557,085	1,200,000	•
Linked List with Mapping and w/o `DELETE`	334,689	765,000	•
Linked List with Mapping and `DELETE`	731,514	3,765,000	•

Table 3: The gas metrics associated with dequeuing 50 items from five priority queues variants. For the refund, (•) indicates the refund was capped at the maximum amount and (○) means a greater refund would be possible.

contracts. Accordingly, we choose to implement the call market with the linked list with mapping. Overall this PQ balances a moderate gas cost for insertion (*i.e.,* order submission) with one for removal (*i.e.,* matching the orders).

3.2 Cost/Benefit of cleaning up after yourself

One consequence of a linked list is that a new contract is created for every node in the list. Beyond being expensive for adding new nodes (a cost that will be bared by the trader in a call market), it also leaves a large footprint on the Ethereum blockchain, especially if we leave the nodes on the blockchain in perpetuity (*i.e.,* we just update the head node of the list and leave the previous head 'dangling.'). However in a PQ, nodes are only removed from the head of the list; thus the node contracts could be 'destroyed' one by one using an extra operation, `SELFDESTRUCT`, in the `dequeue()` function. As shown in Table 3, the refund from doing this outweighs to the cost of the extra computation: gas costs are reduced from 721K to 557K. This suggests a general principle: cleaning up after yourself will pay for itself in gas refunds. Unfortunately, this is not universally true as shown by applying the same principle to the Linked List with Mapping.

Dequeuing in a linked list with mapping can be implemented in two ways. The simplest approach is process a node, update the head pointer, and leave the 'removed' node's data behind in the mapping untouched (where it will never be referenced again). Alternatively we can call `DELETE` on each mapping entry once we are done with a node in the PQ. Deleting a storage variable is identical to setting a non-zero variable to zero (`SSTORE` 0) that costs 20,000 gas but with 15,000 refunded—a net positive gas cost [16]. As it can be seen in the last two rows of Table 3, leaving the data on chain is cheaper then cleaning it up.

The lesson here is that gas refunds incentivize developers to clean up storage variables they will not use again, but it is highly contextual as to whether it will pay for itself. Further the cap on the maximum refund means that refunds

Operation	Description
DepositToken()	Deposits ERC20 standard compliant tokens in the call market contract
DepositEther()	Deposits ETH in the call market contract
OpenMarket()	Opens the market
CloseMarket()	Closes the market
SubmitBid()	Inserts the upcoming bid order messages inside the priority queue
SubmitAsk()	Inserts the upcoming ask order messages inside the priority queue
MatchOrders()	Matches the orders against each other
ClaimTokens()	Transfers collateralized tokens back to the trader
ClaimEther()	Transfers collateralized ETH back to the trader

Table 4: Primary operations of the call market smart contract.

are not fully received for large cleanup operations (however removing the cap impacts the miners' incentives to include the transaction). This is a complicated and under-explored area of Ethereum in the research literature. For our own work, we strive to be good citizens of Ethereum and cleanup to the extent that we can—thus all PQs in Table 2 implement some cleanup and we select Linked List with Mapping and `DELETE` for Lissy.

4 Call Market Design

Lissy is intended as a module that developers can modify as they choose—thus we tried to simplify the design at every step to make it highly extensible but still functional without any extensions. A call market will open for a specified period of time during which it will accept orders. Orders are added to a PQ. Our vision (discussed below) is the market would be open for a very short period of time, close, and then reopen immediately (*e.g.,* every other block). We keep the design simple by not allowing cancellations. Because markets are relatively short-lived, cancelations require a second transaction, and front-running attacks apply to cancellation orders [9], orders simply expire when the market call period ends.

Another simplifying assumption was to implement a *collateralized* call market. We assume all trades are between ETH and an ERC20 token, all orders are pre-funded in the contract with ETH (for bids) and tokens (for asks), and once ETH or tokens are committed to an order, they cannot be withdrawn until the market closes. We revisit alternatives to this design below. This ensures all executed orders clear and settle (*i.e.,* no defaults on payment or delivery).

Lissy is written in 225 lines (SLOC) of Solidity (including 3 external libraries),[4] a high level programming language that is syntactically similar to Java [1]. We tested it with the Mocha testing framework (tests included in the codebase), as deployed using Truffle on Ganache-CLI to obtain our performance metrics. Table 4 represents the call market's primary operations.

[4] Github: linked removed for anonymity

.	Max trades (w.c.)	Gas for max trades	Gas for 1000 trades	Gas for order (avg)
Heap with Dynamic Array	38	5,372,679	457,326,935	207,932
Heap with Static Array	42	5,247,636	333,656,805	197,710
Heap with Mapping	46	5,285,275	226,499,722	215,040
Linked List	152	5,495,265	35,823,601	735,243
Linked List with Mapping	86	5,433,259	62,774,170	547,466

Table 5: Performance of Lissy for each PQ variant.

4.1 Measurements

The main research question is how many orders can be processed in a single Ethereum transaction when the closing the call market, using Ethereum today. As our previous experiments indicated, the choice of priority queue (PQ) implementation is the main influence on performance (see Table 5). We implemented a generic call market that interfaces with a generic PQ (at its own contract address) and ran experiments for each PQ implementation. We looked at the *worst-case* for performance which is when every submitted bid and ask is marketable (*i.e.,* will require fulfillment). In the first two columns, we determined the highest number of orders that can be processed in a single call to `CloseMarket()` and not exceed the the current Ethereum block gas limit of 11,741,495. Since Ethereum will become more efficient over time, we also were interested in how much gas it would cost to execute 1000 pairs of orders which is given in the third column. The fourth column indicates the cost to the trader of submitting a bid or ask — since this costs will vary depending on how many orders are already submitted (recall Figure 1), we average the cost of 200 order submissions.

As expected, the numbers closely track the performance of the PQ itself suggesting the PQ is indeed the main influence on performance. In Ethereum today, call markets appear to be limited to processing about a hundred orders per transaction. If markets open on every other block and the call market could monopolize an entire block to close, a few hundred orders per minute (worst-case) can be processed. The main takeaway is that the transparency, front-running resistance, and low barrier to entry of Ethereum comes with an enormous cost (*i.e.,* an institutional exchange like NASDAQ can process 100K trades per second). That said, many exchanges trade the same assets under different trading rules (*i.e.,* market fragmentation) because traders have different preferences. Lissy can work today in some circumstances like very low liquidity tokens, or markets with high volumes and a small number of traders (*e.g.,* liquidation auctions).

4.2 Security Analysis

Code Security. We used two security audit tools, *Slither*[5] and *SmartCheck*,[6] on Lissy to find vulnerabilities and bad practices across 67 tests. Lissy passes all test except some 'informational' warnings (*e.g.,* a costly loop) that are intentional design choices.

Front-running. The primary motivation for using a call market, as opposed to a continuous time orderbook, is to mitigate front-running attacks [7,9,8]. Consider a sequence of 100 orders submitted within a window of time to market A, and the same sequence randomly shuffled and submitted to market B. If A and B are continuous, the orders that are executed in A could be quite different from B. In a call market, the outcome of A and B will be exactly equivalent. For this reason, there is no threat from miners reordering transactions or users offering higher gas rates to have their orders executed before other orders already broadcasted.

Potential for front-running still exists around ties on price and cancelations (see below). Finally, *displacement attacks* [9] are possible where competitive orders are delayed long enough for the market to close without them. In a traditional call market, a market clearing price is chosen and all trades are executed at this price. All bids made at a higher price will receive the assets for the lower clearing price (and conversely for lower ask prices): this is called a price improvement. A miner about to mine on a set of transactions, including its own orders, could drop other traders' orders to maximize its own price improvement. For this reason, in Lissy, all price improvements are given to the miner (using `block.coinbase.transfer()`). This does not actively hurt traders—they always receive the same price from their order—and it removes any incentive to front-run these profits. For all these reasons, we say Lissy reduces front-running attacks but stop short of saying it is completely solved by Lissy.

5 Design alternatives and extensions

As a module, we try to keep Lissy as simple as possible, and allow customization and extensions as appropriate. We discuss potential modifications here.

5.1 Token Divisibility and Volumes

We implement a very simple market where orders all have the same volume: 1 token. Traders submit multiple orders to increase volume. To extend orders with a volume field, it must first be determined if the token are non-divisible or divisible (and to what precision). In this case, `MarketClose()` needs look at multiple bids to fulfill a single ask (and vice-versa) and declare its trading rules on partial fills. We also simplify the market rules around ties on price: we execut them FIFO (breaking front-running resistance for ties on prices). A common

[5] https://github.com/crytic/slither
[6] https://tool.smartdec.net

trading rule (that does resist front-running) is to fill ties in proportion to their volume (*i.e., pro rata* allocation)[7] however this approach is also contingent on the divisibility of tokens. Consider the following corner-case: 3 equally priced bids of 1 non-divisible token and 1 ask at the same price. There is not good option: (1) the bids could all be dropped (fair but not market efficient), (2) bids could be prioritized based on time as in Lissy (front-running is viable, but in this corner case only), or (3) the bid could be randomly chosen (*cf.* [12]; blockchain 'randomness' is generally deterministic and manipulatable by miners [3,5] and counter-measures could take a few blocks to select [2]).

5.2 Cancelations

5.3 Market clearing price -> informational only

5.4 Closing and Reopening the Market

At the end of the trading period, the market needs to be closed and reopened. However, there is no automatic process to called an Ethereum function and contracts can only run when a function is called.

5.5 Who Pays the Cost for Closing and Reopening the Market?

We think it is useful to explore the landscape of possible designs for closing the market.

Miners Close and Reopen the Market. The difference between the best bid and ask prices is called the *bid-ask spread*. In our design, when the trade occurs between the the highest bid (the highest amount a buyer is willing to pay for an asset) and lowest ask (the lowest amount a seller is willing to accept for an asset) orders, the bid-ask spread is paid to the miner. There are possibilities that (i) no trade occurs or (ii) the bid-ask spread is zero (*i.e.,* the best bid and best ask prices are identical). So there is enough economic incentive for the the miner to execute the `CloseMarket()` function and get refunded as the refund amount could be potentially higher than the bid-ask spread.

Processing Orders in Groups. Another solution pattern is to process certain number of bid and ask orders upon every execution of the `CloseMarket()` function rather than treating them as one substantial transaction. A market participant P_i would process n orders from the previous market (`CloseMarket(n)`) when sending new orders to the current market. This process continues until all the orders in the previous markets are processed.

[7] If Alice and Bob bid the same price for 100 tokens and 20 tokens respectively, and there are only 60 tokens left in marketable asks, Alice receives 50 and Bob 10.

Using the Meta Transactions. Meta transactions enable users to execute Ethereum functions without paying the gas. Rather than spending gas, users sign their intended action using their private keys and broadcast it to the network with no cost. A third party process (*a relayer*) then crafts the actual transaction on user's behalf, sends the transaction to the Ethereum blockchain, and charges the base contract with the associated fees (see Figure **??**). The required gas to pay for the `Match()` function could be collected as fees. So market participants are charged with certain amounts of fees every time they submit an order, these fees are accumulated in the CallMarket contract and will be used to pay for executing the `CloseMarket()` function.

Using the "Contract Pays" Model. An alternative solution is to design the market such that the last person to submit an order calls the `CloseMarket()` function, but in contrast to a normal transaction (where the person initiating the transaction must pay the fee), here the CallMarket contract pays the cost for closing the market and matching the orders respectively. To enforce this design we can use Solidity function modifiers; every time a new order is submitted, a function modifier checks whether (i) the auction period has to end and/or (ii) the maximum number of total orders has reached. If any of these two conditions are met, the `CloseMarket()` will be called. Again, market participants are charged with certain amounts of fees every time they submit an order, these fees are accumulated in the CallMarket. Once the `CloseMarket()` is successfully executed and orders are matched, the contract transfers its funds to that person. Note that here the person must still have enough gas to cover the execution of the transaction as the funds will be only transferred after the transaction is fully executed. However, market participants are incentivized to do so as they may receive more ethers than they have spent.

5.6 Off-Chain Closing

Using Rollups. Rollup is a scaling method that moves the storage and computation of the smart contracts off-chain while maintaining the transaction data on the main chain as call-data. In this technique, any Ethereum user can act as a validator; they can execute the `CloseMarket()`function and only post the new state of the contract (the updated balance of traders) in the form of *assertions* to the main chain. Rollups improve scalability, provide faster and cheaper execution of the contracts, and eliminate the gas limit as the contract is no longer executed on-chain. In the followings we briefly discuss different rollup proposals and techniques. Each approach uses a different method to ensure correction of assertions:

Non-interactive Rollups. In this rollup technique, assertions are posted together with a validity proof that would be later used by validators to check if the `CloseMarket()` function has been executed correctly. ZK-Rollup scheme is one of the solutions that uses ZK-SNARKs to prove the validity of the assertions in zero-knowledge. ZK proofs are quick and cheap to verify but they are expensive

and time consuming to generate. These proofs could be generated (i) for free or (ii) the CallMarket contract could collect proportional fees for every trade that is successfully executed.

Optimistic Rollups. In this scheme, assertions are assumed to be valid if there is no dispute posted about them with a certain window of time (a.k.a. "the challenge period'). Here, dispute resolution is a gas-costly method as the CallMarket contract would have to emulate the transaction on-chain to ensure the correctness the assertion. This scheme introduces a tradeoff between privacy and performance as all the assertions are publicly available and accessible. However, here the new state only reflects the updated balances of traders and no secret is involved.

Multi-round Interactive Rollups. In this design paradigm, *pending assertions* are posted on-chain and they are open to dispute. Once the challenge period is over and no challenge is submitted, the assertion is confirmed and the CallMarket contract transitions to the new state (*i.e.,* updates traders' balances). This scheme takes the overhead for the CallMarket contract to execute the `Close()` on-chain by using rounds to the dispute resolutions. The two parties (asserter and challenger) must run an interactive protocol and the CallMarket smart contract would have to act as a referee and decides which party's claim is true. Arbitrum is an example of multi-round interactive rollups that uses an efficient challenge-based protocol to penalize the dishonest parties [11].

Using Trusted Execution Environments. Another way of achieving execution of the `CloseMarket()` function is incorporating the Ethereum blockchain into the Trusted Execution Environments (TEEs) and decoupling the contract execution from consensus mechanism. TEEs enable secure execution of applications in an isolated processing environment called the *enclave*. Here, the enclave could execute the `CloseMarket()` function off-chain in TEEs and publish an on-chain attestation Quote to the CallMarket contract. The contract then verifies the correctness of the Quote and if validated correctly, it transitions to the new state. Ekiden is an example that uses Intel SGX to solve the scalability and confidentiality issues with the smart contract execution [6]. A drawback of this scheme is in order to achieve confidentiality-preserving smart contracts we have to trust a trusted party in the form of the hardware manufacturer (*e.g.,* Intel).

5.7 Transferring the Collateral back to Traders.

5.8 Clearing Mappings

To maintain the collateral balance of each market participant, we use two Solidity type one-to-one mappings that map Ethereum addresses to 256 bits unsigned integers; `TotalBalance` and `UnavailableBalance`. Once market closes and orders are matched, the `UnavailableBalance` needs to be cleared. However, since

it is not possible to delete the entire mapping without knowing the keys [8], clearing the `UnavailableBalance` mapping remains a challenging issue to solve. Here we provide a landscape of solutions for that.

- **Creating a New Mapping Every Time the Market Opens.** Instead of clearing the `UnavailableBalance` of traders at the end of a matching process, we could create a new mapping every time the market opens. Note that in this scheme traders can claim their funds back (using the `ClaimEther()` and/or `ClaimToken` methods) only when the market is closed (*i.e.,* the market is in the `Closed` state).
- **Creating Custom Keys for the Mapping.** We can create custom keys for the mapping by defining a counter as a global variable inside the call market smart contract and increment it only once at the end of the matching process. So instead of clearing the mapping, we only use another portion of it every time the market opens.
- **Storing the Mapping in a Data-Contract.** Another design proposal is to create a new smart contract every time the market opens, this data contract only stores the `UnavailableBalance` mapping and will be destroyed (using the `SELFDESTRUCT` operation) at the end of the matching process.
- **Storing the Mapping keys in an additional Array.** Another common pattern is to create an additional array on top of the mapping and iterate over that. This array (*e.g.,* address[]) stores the traders' addresses and enables us to iterate over the mapping and delete individual keys and what they map to at the end of the matching process. Note that this is a gas-costly design pattern as we would need to maintain a secondary data structure.

6 Concluding Remarks

Imagine you have just launched an ERC20 token on Ethereum. Now want your holders to be able to trade it. While the barrier to entry for exchange services is low, it still exists. For a centralized or decentralized exchange, you have to convince the operators to list your token and you will be delayed while they process your request. For an automated market maker, you will have to lock up a large amount of ETH into the DApp, along with your tokens. For rollups, you will have to list your token, host your own servers, and/or enlist arbitrators. By contrast to all of these, with Lissy you just deploy the code alongside your token and trading is immediately supported. Even if Lissy is a fallback solution today, we believe it is helpful for every token to be accompanied by on-chain trade execution. And with future improvements to blockchain scalability, it could become the defacto trading method.

[8] https://solidity.readthedocs.io/en/v0.5.12/security-considerations.html

References

1. Ethereum development tutorial · ethereum/wiki wiki. https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial. (Accessed on 24/08/2020).
2. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO*, 2018.
3. J. Bonneau, J. Clark, and S. Goldfeder. On bitcoin as a public randomness source. https://eprint.iacr.org/2015/1015.pdf, 2015. Accessed: 2015-10-25.
4. R. Brandom. This princeton professor is building a bitcoin-inspired prediction market, Nov 2013.
5. B. Bünz, S. Goldfeder, and J. Bonneau. Proofs-of-delay and randomness beacons in ethereum. 2017.
6. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
7. J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, and A. Narayanan. On decentralizing prediction markets and order books. In *Workshop on the Economics of Information Security, State College, Pennsylvania*, 2014.
8. P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. In *IEEE Symposium on Security and Privacy*, 2020.
9. S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. In *WTSC*. FC, 2019.
10. L. Harris. *Trading and exchanges: market microstructure for practitioners*. Oxford, 2003.
11. H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.
12. V. Mavroudis and H. Melton. Libra: Fair order-matching for electronic financial exchanges. In *ACM AFT*, 2019.
13. A. Norry. The history of the mt gox hack: Bitcoin's biggest heist. https://blockonomi.com/mt-gox-hack/, June 2019. (Accessed on 12/31/2019).
14. Securities and E. B. of India. Sebi | order in the matter of nse colocation. https://www.sebi.gov.in/enforcement/orders/apr-2019/order-in-the-matter-of-nse-colocation_42880.html, 2019. (Accessed on 11/11/2019).
15. C. Signer. Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science, 2018.
16. G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. Technical Report 2014, Ethereum, 2014.

A Ethereum's Gas Model

Gas is the Ethereum's pseudo-currency and a key variable for the execution of smart contracts. DApps written in high-level programming languages are compiled and translated into a compact representation (called 'bytecode') to be further executed on the Ethereum virtual machine or EVM. Each of these opcodes has a fixed amount of gas assigned and is a measure of computational effort. Separating ETH from gas prevents transactions cost from being too expensive as they are less impacted by ETH price fluctuations. Gas unit does not a have a monetary value and as mentioned, it only measures the computational work undertaken by miners. To pay miners, Ethereum introduces a `gas price` — small denomination of ether called **Gwei** attached to each gas unit. Essentially, gas price indicates how much users are willing to pay per unit of gas, clearly the higher gas price leads to the faster execution of a transaction. Miners also need to know the total amount of computational work a user is requesting, called the `gas limit`. `Gas limit` is a parameter that limits the amount of gas users would spend in a transaction. It protects users from spending unlimited ETH on their transactions and must be set carefully. If the `gas limit` is too low, the transaction will exceed the limit, all operations will be reverted while the user must pay for the computational work performed by the miner. By default the gas limit for an Ethereum transaction is 21,000 gas. An Ethereum block also has a `block gas limit` field which is set by the Ethereum miners and indicates the maximum amount of gas all the transactions in that block are allowed to consume, the Ethereum `block gas limit` is currently 11,741,495. [9]

Gas Refunds.

There are two particular EVM operations with negative gas — certain amount of gas is refunded to the sender at the end of the transaction. These operation include:

`SELFDESTRUCT`. This operation destroys and deletes the originating contract and refunds its balance (if any) to a designated receiver address. Note that the Ethereum storage is implemented in the form of a has map and EVM is not statistically aware of which storage slots are held by the contract. Because of this the `SELFDESTRUCT` operation does not remove the initial byte code of the contract from the chain, but it frees up the state storage and has a refund of 24,000 gas.

`SSTORE`. This operation clears the Ethereum storage and has15,000 gas refunds.

Note that in order to urge miners to process smart contracts with refunds, the accumulated gas refund can never exceed half the gas used up during computation [16]. So at the end of a successful transaction, the amount of gas in the refund counter (capped at half the net gas used) is returned to the caller.

[9] July 2020: https://ethstats.net/

At the time of this writing, Ethereum transaction receipts only account for the `gasUsed`, which is the total amount of gas units spent during a transaction, and we are not able to obtain the value of the EVM's refund counter from inside the EVM [15]. So in order to account for refunds, we decide to calculate them manually; first we figure out exactly how much storage is being cleared or how many smart contracts are being destroyed, then we multiply these numbers by 24,000 and 15,000 respectively.