Lissy: Revisiting fully on-chain order books

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 cancellation) 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 (or batch auction) 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, including a variant using off-chain execution ('rollups').

1 Introductory Remarks

For better or worse, 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 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 order book: 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.

Exchange Platform	Examples	Non-Custodial	No Trusted Third Party	Fully Autonomous	Resilient to Front-running	Resilient to Server Downtime	Resilient to Ethereum Network Congestion	High Trade Performance	Certain Fulfillment	Low Price Slippage	No Impermanent Loss	T+0 Settlement
Centralized Exchange	Binance Coinbase Bitfinex	○	○	○	○	○	●	●	○	●	●	○
DEX w/ Off-chain Order Book	0x EtherDelta IDEX	●	●	●	○	○	◐	●	○	●	●	●
DEX w/ Roll-up	Loopring	●	●	○	○	○	◐	◐	○	●	●	●
On-chain Continuous Market	OasisDEX	●	●	●	○	●	○	○	○	●	●	●
On-chain Automated Market Making	UniSwap Bancor Curve	●	●	●	○	●	○	○	●	○	○	●
On-chain Call Market	Lissy	●	●	●	●	●	○	○	○	●	●	●
On-chain Call Market w/ Roll-up	Lissy variant	●	●	○	●	○	◐	◐	○	●	●	●

Table 1: Comparative evaluation of different types of exchange platforms: ● indicates the properties (columns) are fulfilled by the corresponding mechanism (rows) within reason, ○ means the property is not fulfilled, and ◐ means the property is partially fulfilled.

2 Preliminaries

2.1 Market Structure

Execution Systems. There are three main approaches to arranging a trade [**?**]. In a *quote-driven* market, a dealer uses its 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 order book. Order-driven markets can be *continuous*, with buyers/sellers at any time adding orders to the order book (*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 other, given its own order book priced in a currency (*e.g.,* USD). Exchanges for blockchain-based assets (also called crypto assets 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 [**?**], QuadrigaCX [**?**], 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 Order Books. 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 order book 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 order book 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 a 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 order book, plus they are very efficient relative to an on-chain order book. 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 order book 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/ask 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, order books 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 avoid taking custodianship of assets off-chain, and virtually all (for order-driven markets) operate the order book 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 as well.

2.3 Related Works

Blockchain Limitations and Solution. While an order book is a ledger and blockchains provide a distributed ledger, it is not straightforward to drop a continuous-time order book onto a blockchain. An older 2014 paper [**?**] on the 'Princeton prediction market' [**?**] motivates our work. The authors observe the following limitations of on-chain continuous order books: 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/cancellations when broadcast to network but not in a block and have their orders appear first (as examined in later work on front-running: [**?**, **?**]).

Call Markets. The researchers propose using a call market instead of a continuous-time market [**?**]. Orders are collected and placed into the order book 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 [**?**]. 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 Academic Literature. There are numerous industry projects on blockchain-based exchanges and order books but most of the academic literature is on topics that are related but tangential to the mechanics of trade execution. Early (and some recent) literature consider trade execution under encryption (*i.e.,* dark markets) for securities [**?**, **?**, **?**, **?**] and futures [**?**]. A number of projects consider structuring derivatives in smart contracts—Velocity [**?**], Findel [**?**]—but once issued, the derivative can be traded using exchanges. Atomic swaps (*i.e.,* payment vs delivery) are necessary for settling trades and some general approaches include Arwen [**?**] and Tesseract [**?**].

[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 order book.

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 2: Operations for a generic priority queue.

3 Priority Queues

In designing Lissy within Ethereum's gas model, performance is the main bottleneck. For a call market, closing the market and processing all the orders is the 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 create 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. In 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 cancelled 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 2) 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 the tree are in a specific order and the root always represents the highest priority item of the data structure (the largest and smallest values in the Max-Heap and Min-Heap respectively). We implement 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 entire order is stored (as a struct) in the heap. In this variant, we store the order struct in a Solidity mapping and store the mapping keys in the heap.
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) and last node (tail) in the list.

3.1 Priority Queue Evaluation

3.1.1 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 variant) 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.

3.1.2 Dequeue Performance.

For each PQ variant storing 50 random integers, the `Dequeue()` function is iterated until the data structure is empty. The total gas cost for fully dequeuing the priority queue variants is outlined in Table 3. These tests are performed using the following Ethereum gas metrics: block gas

[2]Github: Link removed for anonymity.

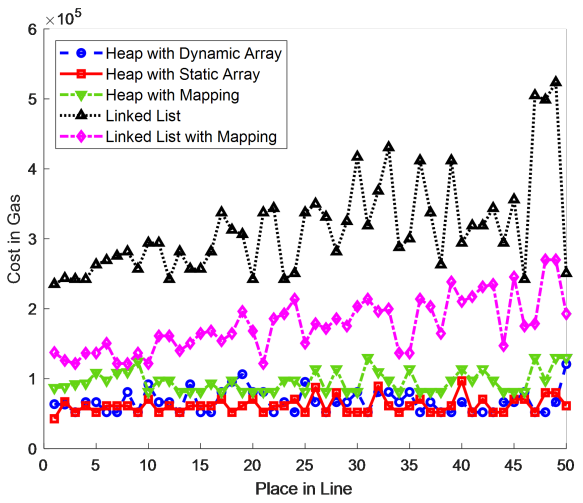

Figure 1: Gas costs for enqueuing 50 random integers into five priority queue variants. For the x-axis, a value of 9 indicates it is the 9th integer entered in the priority queue. The y-axis is the cost of enqueuing in gas.

	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 3: The gas metrics associated with dequeuing 50 integers from five priority queue variants. For the refund, (●) indicates the refund was capped at the maximum amount and (◐) means a greater refund would be possible.

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 worse based on how much of a refund they generate.

3.1.3 Discussion.

Based on enqueuing, the heap variants are the cheapest in terms of gas, while based on dequeuing, the link list variants are the 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 causes the creation of a new contract. 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] 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 4: The gas metrics associated with dequeuing 50 integers from four linked list variants. For the refund, (●) indicates the refund was capped at the maximum amount and (◐) means a greater refund would be possible.

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 in the active Ethereum state, 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 4, 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 to process a node, update the head pointer, and leave the 'removed' node's

Operation	Description
depositToken()	Deposits ERC20 standard compliant tokens in Lissy smart contract
depositEther()	Deposits ETH in Lissy smart contract
openMarket()	Opens the market
closeMarket()	Closes the market and processes the orders
submitBid()	Inserts the upcoming bid order messages inside the priority queue
submitAsk()	Inserts the upcoming ask order messages inside the priority queue
claimTokens()	Transfers tokens to the traders
claimEther()	Transfers ETH to the traders

Table 5: Primary operations of Lissy smart contract.

	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 6: Performance of Lissy for each PQ variant Reviewer's comment: Gas for max trades in table 5 is about 5M gas, why it can exceed 11M gas limit of Ethereum?.

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 processing 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 [**?**]. As it can be seen in the last two rows of Table 4, leaving the data on chain is cheaper than 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 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 clean up to the extent that we can—thus all PQs in Table 3 implement some cleanup and we select linked list with mapping and `DELETE` for Lissy.

4 Call Market Design

Reviewer's comment: DDOS against submitbid and submitask functions. Our answer: we put a cap on the number of orders submitted to the market so that we can always close. Also talk about bulk displacement attack (see reviewers comments)

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, and these 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 cancellations require a second transaction, and front-running attacks apply to cancellation orders [**?**].Also, markets are relatively short-lived and 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. This ensures all executed orders clear and settle (*i.e.,* no defaults on payment or delivery).

Lissy is deployed on the Rinkeby— Ethereum testnet (or test network). The contract address is '0x40F3089C4C404dA9E9655AD230aDD2E493BE8e5c', more details can be found on `https://rinkeby.etherscan.io`

Lissy is written in 324 lines (SLOC) of Solidity, [4] a high level programming language that is syntactically similar to Java [**?**]. 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 5 represents Lissy's primary operations. Mahsa: Lissy's size of bytecode in bytes: 17212, Lissy's size of deployed in bytes: 10812, initialization and constructor code in bytes: 6400

4.1 Measurements

The main research question is how many orders can be processed in a single Ethereum transaction when 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 6). 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 determine the highest number of orders that can be processed

[4] Github: linked removed for anonymity

in a single call to the `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 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 come 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 tests except some 'informational' warnings (*e.g.,* a costly loop) that are intentional design choices.

Front-running. we have to revisit this section: talk about our front-running paper: there are various ways to prevent frontrunning, one is through confidentiality..., then cite this paper: 'TEX – A Securely Scalable Trustless Exchange' [**?**] The primary motivation for using a call market, as opposed to a continuous-time order book, is to mitigate front-running attacks [**?**, **?**, **?**]. Consider a sequence of 100 orders submitted within a window of time to market A, and the same sequence is 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. There is no threat from miners reordering transactions or traders 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 order cancellations (see below). Finally, *displacement attacks* [**?**] 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 orders—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 this issue is completely solved by Lissy.

5 Design Alternatives and Extensions

We need to find a home for this: another solution to reopen the market immediately after the closing call is to have many similar contracts for the market and a factory contract to guide users to the correct address (from the reviewer's comment)

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

5.1 Token Divisibility and Volumes

Mahsa: Partial Filling: we ran the worst case matching test with random volumes between 1 and 10 on linked list with mapping (which handles maximum pairs of 80 orders with single volumes). With partial filling, it can clear the book that contains 70 pairs at max (which on average ends up being 118 trades). SO it is moderately more expensive

We implemented 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 is non-divisible or divisible (and to what precision). In this case, `closeMarket()` needs to 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 execute them in a FIFO manner (breaking front-running resistance for ties on prices). A common 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 no good option: (1) the bids could all be dropped (fair but not market efficient), (2) bids

[5] https://github.com/crytic/slither

[6] https://tool.smartdec.net

[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.

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.* Libra [**?**]); blockchain 'randomness' is generally deterministic and manipulatable by miners [**?**, **?**] and counter-measures could take a few blocks to select [**?**]).

5.2 Order Cancellations

Lissy does not support order cancellations. We intend to open and close markets quickly (on the order of blocks), so orders are relatively short-lived. Support for cancellation also opens the market to new front-running issues where other traders (or miners) can displace cancellations until after the market closes (however one benefit of a call market is that beating a cancellation with a new order has no effect, assuming the cancellation is run any time before the market closes). Finally, cancellations have a performance impact. Cancelled orders can be removed from the underlying data structure or accumulated in a list that is cross-checked when closing the market. Removing orders requires a more verbose structure than a priority queue (*e.g.,* a binary search tree instead of a heap; or methods to traverse a linked list rather than only pulling from the head). While client software could help point out where the order is in the data structure, the order book can change between submitting the cancellation request and running the method. A linked list with mapping that returns the key for each submitted order seems to be the most tenable data structure.

5.3 Market Clearing Price

Call markets are heralded for fair price discovery. This is why many exchanges use a call market at the end of the day to determine the closing price of an asset, which is an important price both optically (it is well-published) and operationally (many derivates settle based on the closing price). We purposely do not compute a 'market clearing price' with Lissy because miners can easily manipulate the price (*i.e.,* include a single wash trade at the price they want fixed), although they forgo profit for doing so. This is not merely hypothetical—Uniswap (the prominent quote-drive, on-chain exchange) prices have been manipulated to exploit other DeFi applications relying on them. Countermeasures to protect Uniswap price integrity could also apply to Lissy: (1) taking a rolling mediant of prices over time, and (2) using it alongside other sources for the same price and forming a consensus. While Lissy does not emit a market clearing price, it can be computed by a web application examining the order book at market close.

5.4 Scheduling Events

As a simple module, Lissy is a one-shot market. However it can be extended to re-open with a clean order book after `closeMarket()` is run. Modifiers can enforce when the market operates openly (collecting orders) and when close can be run. In the Princeton paper [**?**], the call market is envisioned to run as an alt-coin, where orders accumulate within a block and a miner closes the market as part of the logic of producing a new block (*i.e.,* within the same portion of code as computing their coinbase transaction in Bitcoin or gasUsed in Ethereum).

If the call market runs as a DApp on Ethereum, it seems difficult to open and close the market every block. Someone needs to call the `closeMarket()` for every block (we return to who this is next) but the market will only work as intended if miners execute this function after every `submitBid()` and `submitAsk()` invocation. Since price improvements are paid to the miners, the miner is incentivized to run `closeMarket()` last to make the most profit. Perhaps in a world where these markets became immensely popular, miners would actually do this, but as it stands, miners do not generally search for the transaction ordering that nets them the most profit.

A close alternative is to allow markets to open and close on different blocks. Specifically, the `closeMarket()` function calls `openMarket()` as a subroutine and sets two modifiers: orders are only accepted in the block immediately after the current block (*i.e.,* the block that executes the `closeMarket()`) and `closeMarket()` cannot be run again until two blocks after the current block.

The final issue is who invokes `closeMarket()` every other block? There are actually two issues here: the issue of scheduling the function call and the issue of paying for it. For scheduling the function call, we can do one of the following: rely on market participants, who are eager to trade, to reopen the market, offer a bounty to reopen the market, or use an external service like Ethereum Alarm Clock.[8] Next we consider the second issue of who pays to close the market.

5.5 Who Pays to Close/Reopen the Market?

It is unlikely that a mining software will become aware of specific DApps, but miners are paid all price improvements in the market and so it is possible that running `closeMarket()` would pay for itself. However we consider two other (more realistic) scenarios. One solution requires a modified closing function, `closeMarket(n)`, that only processes n orders at a time (until the order book is empty). Once the time window for submitting orders is past, a new market is created (without settling the previous market). Every order submission on the new market also requires to run, say, `closeMarket(10)` on the older market, thus progressively closing the previous market while accepting orders to the new market. This solution pattern has two issues: first, amortizing the cost of closing the market amongst the early traders of the new market is an added incentive to not submit orders early to the market; the second related issue is if not enough traders submit orders in

[8] https://ethereum-alarm-clock-service.readthedocs.io/

the new market, the old market never closes (resulting in a backlog of old markets waiting to close).

The second solution is to levy a carefully computed fee against the traders for every new order they submit. These fees are accumulated by the DApp to use as a bounty. When the time window for the open market elapses, the user who calls `closeMarket()` receives the bounty. This is a better solution, although not perfect: `closeMarket()` cost does not follow a tight linear increase with the number of orders, and gas prices vary over time which could render the bounty insufficient for offsetting the `closeMarket()` cost. However an interested third party (such as the token issuer for a given market) might bailout the market when it halts on `closeMarket()` to facilitate further trading. If the contract can pay for its own functions, an interested party can also arrange for a commercial service (*e.g.,* any.sender[9]) to relay the `closeMarket()` function call on Ethereum (an approach called *meta-transactions*).

5.6 Off-chain Closing

We have two main motivations for designing an *on-chain* order book: (1) the trust model is superior to partially/fully centralized solutions, and (2) Lissy can be deployed without any delay or other arrangements. We have relaxed (2) already by considering services like Ethereum Alarm Clock and any.sender to facilitate easy market closings. In this section, we present a variant that only provides (1) (however one that can be deployed without any other market stakeholders).

A function can be computed off-chain and the new state of the DApp, called a *rollup*, is written back to the blockchain, accompanied by either (1) a proof that the function was executed correctly, or (2) a dispute resolution process that can resolve, on-chain, functions that are not executed correctly (*e.g.,* Arbitrum [**?**]). In the case of (1), validating the proof must be cheaper than running the function itself. There are two main approaches to (1): the first is to use cryptographic proof techniques (*e.g.,* SNARKS [**?**, **?**] and variants [**?**]), and the second is to execute the function in a trusted execution environment (TEE; *e.g.,* Intel SGX) and validate the TEE's quote on-chain (*e.g.,* Ekiden [**?**]).[10]

We implemented a variant of Lissy using rollups on Arbitrum.[11] In this model, the token issuer (or other interested party) will run a dedicated server to watch Ethereum for function invocations submitted and sequenced on-chain, perform the function call off-chain, and write the resultant state on-chain. Anyone capable of running EVM code can verify the result. If it is incorrect, a dispute can be filed (with the correct state) and an on-chain correction will be made. With or without disputes, participants that validate the function calls for themselves can proceed knowing that the correct state will eventually be finalized.

In our Lissy variant with rollups, the token issuer does all the computation (both enqueuing and dequeuing). Thus we switch the priority queue to use a heap with dynamic array, which balances the expense of both operations (instead of optimizing for dequeuing in `closeMarket()`). Recall that on-chain, such a priority queue can only match 38 pairs at a cost of 5,372,679 gas. With roll-ups, 38 pairs cost 38,863 gas. As the pairs increase, the cost is essentially constant (*e.g.,* 1000 pairs cost 38,851 as opposed to 457,326,935 on-chain). Submitting an order costs 39,169 gas on average as opposed to 207,932 on-chain.

Our Lissy variant is not the first rollup-based order book. Loopring 3.0[12] offers a continuous-time order book. The primary difference is that orders in Loopring 3.0 are submitted off-chain to the operator directly, whereas our variant requires on-chain submission. While on-chain submission is hampered by Ethereum's block limits, the threat model of reordering transactions is well-understood and our call market design mitigates most front-running attacks. In Loopring 3.0, the rollup proof does not ensure that the exchange did not reorder transactions, which is particularly problematic in a continuous-time order book. In short, we provide better protection against a malicious exchange while Loopring 3.0 provides even better performance.

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 host your own servers. 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. With future improvements to blockchain scalability, it could become the defacto trading method.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If

[9] https://github.com/PISAresearch/docs.any.sender

[10] The TEE-based approach is mired by recent attacks on SGX [**?**, **?**, **?**, **?**], however these attacks do not necessarily apply to the specifics of how SGX is used here, and safer TEE technologies like Intel TXT (*cf.* [**?**]) can be substituted.

[11] https://offchainlabs.com for more current details than the 2018 *USENIX Security* paper [**?**].

[12] https://loopring.org

you made your code or data available, it's worth mentioning this fact in a dedicated section.

A Clearing storage (extended discussion)

For space considerations, we kept the discussion in Section 3.2 brief but the subject could use more explanation. First, we discuss the Ethereum's gas model— (i) what are different gas parameters one needs to take into account when sending a transaction on the Ethereum blockchain, (ii) what is the gas refund, and (iii) and how it is calculated. Next, we discuss one challenge in Lissy, *clearing mappings*, that turns out to be more difficult than we initially expected.

A.1 Ethereum's Gas Model

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 (EVM). When a transaction is executed, each opcode in the execution path accrues a fixed, pre-specified amount of gas. The function caller will pledge to pay a certain amount of ETH (typically quoted in units of **Gwei**, where *wei* is the smallest transactional unit of ETH) per gas, and miners are free to choose to execute that transaction or ignore it. The function caller is charged for exactly the amount the transaction costs and they cap the maximum they are willing to be charged (*gas limit*)—if the cap is too low to complete the execution, the miner keeps the Gwei and reverts the state of the DApp (as if the function never ran).

In Ethereum, transactions are bundled into blocks. A miner can include as many transactions (typically preferring transactions that bid the highest for gas) that can fit under a pre-specified `block gas limit`, which is algorithmically adjusted for every block. As of the time of writing, the limit is around 11M gas.

A.1.1 Gas Refunds.

In order to reconstruct the current state of Ethereum's EVM, a node must obtain a copy of every variable change since the genesis block (or a more recent 'checkpoint' that is universally agreed to). For this reason, stored variables persist for a long time and, at first glance, it seems pointless to free up variable storage (and unclear what 'free up' even means). Once the current state of the EVM is established by a node, it can forget about every historical variable changes and only concern itself with the variables that have non-zero value (as a byte-string for non-integers) in the current state (uninitialized variables in Ethereum have the value zero by default). Therefore, freeing up variables will reduce the amount of state Ethereum nodes need to maintain going forward.

For this reason, some EVM operations cost a negative amount of gas. That is, the gas is refunded to the sender at the end of the transaction, however (1) the refund is capped at 50% of the total gas cost of the transaction, and (2) the block gas limit applies to the pre-refunded amount (*i.e.,* a transaction receiving a full refund can cost up to 5.5M gas with a 11M limit). Negative gas operations include:

- `SELFDESTRUCT`. This operation destroys the contract that calls it and refunds its balance (if any) to a designated receiver address. `SELFDESTRUCT` operation does not remove the initial byte code of the contract from the chain. It always refunds 24,000 gas. For example, if a contract A stores a single non-zero integer and contract B stores 100 non-zero integers, the `SELFDESTRUCT` refund for both is the same (24,000 gas).
- `SSTORE`. This operation loads a storage slot with a value. Using `SSTORE` to load a zero into a storage slot means the nodes can start ignoring it (recall that all variables, even if uninitialized, have zero by default). Doing this refunds 15,000 gas per slot.

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 users are not able to obtain the value of the EVM's refund counter from inside the EVM [**?**]. So in order to account for refunds in Tables 3, we calculate them manually. First we need to 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.

A.2 Clearing Mappings

Now let us put this theory into practice. In Lissy, traders preload their account with ERC-20 tokens and/or ETH. Lissy tracks what they are owed using a mapping called `totalBalance` and allows traders to withdraw their tokens at any time. However if a trader submits an order (*i.e.,* ask for their tokens), the tokens are committed and not available for withdrawal until the market closes (after which, the tokens are either transferred to the buyer or, if the trade is not executed, returned to the trader). Committed tokens are also tracked in a mapping called `unavailableBalance`. Sellers can request a token withdrawal up to their total balance subtracted by their unavailable balance.

As the DApp runs `closeMarket()`, it starts matching the best bids to the best asks. As orders execute, `totalBalance` and `unavailableBalance` are updated. At a certain point, the bids and asks will stop matching in price. At this point, every order left in the order book cannot execute (because the priority queue sorts orders by price, and so orders deeper in the queue have worst prices than the order at the head of the queue). Therefore all remaining entries in `unavailableBalance` can be cleared.

In Solidity, it is not possible to delete an entire mapping without individually zero-ing out each entry key-by-key. At the same time, it is wasteful to let an entire mapping sit in the EVM when it will never be referenced again. The following are some options for addressing this conflict.

1. **Manually Clearing the Mapping.** Since mappings cannot be iterated, a common design pattern used by DApp developers is to store keys in an array and iterate over the array to zero out each mapping and array entry. Clearing a mapping this way costs substantially more to clear than what is refunded (15,000 for each element).

2. **Store the mapping in a separate DApp.** We could wrap the mapping inside its own DApp and when we are done with the mapping, we can run `SELFDESTRUCT` on the contract. This refunds us 24,000 gas which is less than the cost of deploying the extra contract. Additionally, every call to the mapping is more expensive because (1) it is an external function call, and (2) function calls to be evaluated to ensure they only come from the market contract (if a mapping is a local variable, you get private access for free).

3. **Leave and Ignore the Mapping.** The final option is to not clear the mapping and just create a new one (or create a new prefix for all mapping keys to reflect the new version of the mapping). Unfortunately, this is the most economical option for DApp developers even if it is the worst option for Ethereum nodes.

Clearing storage is important for reducing EVM bloat. The Ethereum refund model should be considered further by Ethereum developers to better incentivize developers to be less wasteful in using storage.

B Collateralization Options

In Lissy, both the tokens and ETH that a trader want to potentially use in the order book are pre-loaded into the contract. Consider Alice who holds a token and decides she wants to trade it for ETH. In this model, she must first transfer the tokens to the contract and then submit an ask order. If she does this within the same block, there is a chance that a miner will execute the ask before the transfer and the ask will revert. If she waits for confirmation, this introduces a delay. This delay seems reasonable but we point out a few ways it could be addressed:

1. **Use `msg.value`.** For the ETH side of a trade (*i.e.,* for bids), ETH could be sent with the function call to `submitBid()` to remove the need for `depositEther()`. This works for markets that trade ERC-20 tokens for ETH, but would not work for ERC-20 to ERC-20 exchanges.

2. **Merge Deposits with Bids/Asks.** Lissy could have an additional function that atomically runs the functionality of `depositToken()` followed by the functionality of `submitAsk()`. This removes the chance that the deposit and order submission are ordered incorrectly.

3. **Use ERC-20 Approval.** Instead of Lissy taking custody of the tokens, the token holder could simply approve Lissy to transfer tokens on her behalf. If Lissy is coded securely, it is unconcerning to allow the approval to stand long-term and the trader never has to lock up their tokens in the DApp. The issue is that there is no guarantee that the tokens are actually available when the market closes (*i.e.,* Alice can approve a DApp to spend 100 tokens even if she only has 5 tokens, or no tokens). In this case, Lissy would optimistically try to transfer the tokens and if it fails, move onto the next order. This also gives Alice an indirect way to cancel an order, by removing the tokens backing the order—this could be a feature or it could be considered an abuse.

4. **Use a Fidelity Bond.** Traders could post some amount of tokens as a fidelity bond, and be allowed to submit orders up to 100x this value using approve. If a trade fails because the pledged tokens are not available, the fidelity bond is slashed as punishment. This lets traders side-step time-consuming transfers to and from Lissy while still incentivizing them to ensure that submitted orders can actually be executed. The trade-off is that Lissy needs to update balances with external calls to the ERC-20 contract instead of simply updating its internal ledger.