Trading on-chain: how feasible is regulators' worst-case scenario?

Mahsa Moosavi and Jeremy Clark

Concordia University, Montreal, Canada

Abstract

When consumers trade financial products, they typically use well-identified service providers that operate under government regulation. In theory, decentralized platforms like Ethereum can offer trading services 'on-chain' without an obvious entry point for regulators. Fortunately for regulators, most trading volume in blockchain-based assets is still on centralized service providers for performance reasons. However this leaves the following research questions we address in this paper: (i) is secure trading (*i.e.,* resistant to front-running and price manipulation) even feasible as a fully 'on-chain' service on a public blockchain, (ii) what is its performance benchmark, and (iii) what is the performance impact of novel techniques (*e.g.,* 'rollups') in closing the performance gap?

To answer these questions, we 'learn by doing' and custom design an Ethereum-based call market (or batch auction) exchange, Lissy, with favourable security properties. We conduct a variety of optimizations and experiments to demonstrate that this technology cannot expect to exceed a few hundred trade executions per block (*i.e.,* 13s window of time). However this can be scaled dramatically with off-chain execution that is not consumer-facing. We also illustrate, with numerous examples throughout the paper, how blockchain deployment is full of nuances that make it quite different from developing in better understood domains (*e.g.,* cloud-based web applications).

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.

Financial regulators seek to provide consumer protection during the issuing and trading of financial products and assets. They are concerned by their limited ability to intervene when trading is conducted on decentralized networks that, like Ethereum, run on the open internet. Order-driven trading can happen fully 'on-chain' and this was experimented within the early days of Ethereum, but it has been largely abandoned for performance reasons in favour of running on centralized servers. More specifically, the core functionality is performed off-chain (*e.g.,* matching orders) while other aspects (*e.g.,* loading accounts, order cancellation) might be performed on-chain. In this world, company names, employees, addresses, and publicly addressable servers all provide regulatory hooks.

Our research group is actively collaborating with our jurisdiction's (Quebec, Canada) financial regulator, the Autorité des Marchés Financiers (AMF), to help them forecast how trading can be impacted by blockchain technology. This paper addresses their concerns about the feasibility of a 'worst-case scenario' where a trading platform is anonymously deployed on a public blockchain, like Ethereum, and runs autonomously without any further intervention from an externally visible entity. Such a design appears feasible but is considered too slow. Together, we agreed 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 in practice) providing future efficiency gains for on-chain trading, (2) demand for on-chain trading is exemplified by the recent popularity of dealer quote-based trading like Uniswap and Curve Finance (reviewed below), and (3) stablecoins have become popular and allow on-chain trading with pricing in USD, alleviating another regulatory hook: the need for platforms to maintain traditional accounts for holding governmental currency and interfacing with the banking system.

Contributions. We study fully on-chain markets through design and implementation with Solidity—a high-level programming language for Ethereum that is syntactically similar to Java [**?**]. We choose Ethereum as a hostile environment for an order book: Ethereum is fully decentralized (hardest to regulate), network participation is open to anyone on the

internet (strongest adversarial model), and, as a result, it is slow (a lower-bound benchmark). Generally if an application is feasible on Ethereum, it will also be feasible and only run faster on a private (or permissioned) blockchain (*e.g.,* Hyperledger), which is a blockchain operated by authorized network nodes only, generally visible to regulators.

Our proof-of-concept, Lissy, is an extensible base class suitable for experiments. Lissy is designed from a security perspective: users only trust their assets to Lissy's auditable code (non-custodial) and not to a third party; all operations are transparent; and we nearly eliminate the ability for adversarial network nodes to profit from front-running orders. In fact, we believe we are the first to design a decentralized application that actually leverages front-running (*i.e.,* miner extractable value [**?**]) for the benefit of the system (see Section **??**).

While Solidity and its compiled bytecode is like many common programming languages, it also has 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, *etc.*). 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 only a narrow set of markets today (low liquidity, small number of traders) which is good news for regulators. However we caution that the technology is likely to improve vastly in the coming years and we demonstrate one avenue for improvement through 'roll-ups' [**?**]. While roll-ups use centralized servers and are not fully on-chain, an architecture where these servers only interface with Ethereum and never interface directly with users is still concerning, as it side-steps the regulatory hook that is present in many other server-augmented trading platforms.

2 Preliminaries

2.1 Ethereum and Blockchain Technology

A public blockchain is an open peer-to-peer network that maintains a set of transactions without a single entity in charge. In Ethereum, *transactions* encode the bytecode of user-written *decentralized applications* (DApps) to be stored on the blockchain; and the function calls made to the DApp. Every execution of every function call is a validated by all honest, participating nodes to correct; a property that is robust against a fraction of faulty and malicious network nodes (or more precisely, their accumulated computational power). Once transactions are agreed upon, all honest participants will have identical sets of transactions in the same order. For Ethereum, this is conceptualized as the current state of a large *virtual machine (EVM)* that is running many DApps.

Transactions are broadcast by users to the blockchain network where they are propagated to all nodes. Nodes that choose to *mine* will collect transactions (in the order of their choosing) into a block, and will attempt to have the network reach a consensus that their block should be added to the set (or chain) of previous blocks. A transaction is considered finalized once consensus on its inclusion has held for several additional blocks.

Ethereum's Gas Model. Every transaction results in the participating nodes having to execute bytecode. This is not free. 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 Ethereum's internal currency *ETH* (typically quoted in units of Gwei which is one billionth of an ETH) per unit of gas, and miners are free to choose to execute that transaction or ignore it. The function caller is charged for exactly what the transaction costs to execute, 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 EVM (as if the function never ran).

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 approximately 11M gas. Essentially, our main research question is how many on-chain trades can be executed without exceeding that limit. Later we also discuss several bytecode operations (*opcodes*) that refund gas (*i.e.,* cost negative gas), which we heavily utilize in our optimizations.

2.2 Trade 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).

2.3 Trading Blockchain-based Assets

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 (*e.g.,* Bi-

nance, Bitfinex, *etc.*) operates the platform as a trusted third party in every aspect: custodianship over assets/currency being traded, exchanging assets fairly, offering the best possible price execution. Security breaches and fraud in centralized exchanges (*e.g.,* MtGox [**?**], QuadrigaCX [**?**], and many others) have become a common source of lost funds for users, while accusations of unfair trade execution have been levelled but are difficult to prove. Today, CEXes are often regulated as other money service businesses—this provides some ability for the government to conduct financial tracking but does little to provide consumer protection against fraud.

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. Since it is an open system, there is no obvious regulatory hook (beyond the blockchain itself).

In this paper, we focus on benchmarking an order book for the public blockchain Ethereum. Ethereum is widely-used and we stand to learn the most from working in a performance-hostile environment. Exchanges could be given their own dedicated blockchain, where trade execution logic can be coded into the network protocol. Trading systems on permissioned blockchains (*e.g.,* NASDAQ Linq, tZero) can also improve execution time and throughput, but they reduce user transparency and trust if unregulated.

On-chain Dealers. An advantage of on-chain trading 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 profit from the trades themselves but hopes their losses (termed 'impermanent losses') are offset over the long-term by trading fees. By contrast, an order book requires no upfront inventory and trading fees are optional. 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. Such decentralized exchanges (DEXes) 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 (*e.g.,* EtherDelta, 0x, IDEX, *etc.*): many avoid taking custodianship of assets off-chain, and virtually all (for order-driven markets) operate the order book itself off-chain (a regulatory hook). A non-custodial DEX solves the big issue of a CEX—the operator stealing the funds—however trade execution is still not provably fair, funds can still be indirectly stolen by a malicious exchange executing unauthorized trades, and server downtime is a common frustration for traders. An enhancement is to prove that trade execution is correct (*e.g.,* Loopring) but these proofs have blindspots (discussed in section **??**).

2.4 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, and 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 **??**). The paper does not include an implementation and was envisioned as running on a custom blockchain (Ethereum was still in development in 2014). Market operations are part of the blockchain logic.

Large exchanges, like the NYSE and NASDAQ, run two call markets every trading day in parallel with a continuous-time market. One call market closes at market open to produce the opening price for each stock, while the other closes at the end of the day to produce the closing price. Other exchanges, called crossing networks, also operate as a call market at various times throughout the trading day.[1] Call markets 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 [**?**].

The most similar academic work to this paper is the TEX exchange [**?**]. TEX is also an implementation of an order book, however it uses on-chain and off-chain components, and does not answer our research question of how feasible an on-chain order book is. We later compare it to our variant with roll-ups in Section **??**. The main overlap is our common focus on resisting front-running attacks. In a recent survey on front-running attacks in blockchain [**?**], three solutions are proposed. One solution is to provide confidentiality over the content of orders which is the approach taken by TEX. The main downside is that honest traders cannot submit their orders and leave, they must interact in a second round to reveal their orders. The second solution is to sequence transactions according to some rule, either at the protocol level [**?**] or as a third party service [**?**] (a regulatory hook). These are very recent works and are not available for experimentation on Ethereum yet (although Chainlink has announced an intention[2]). The third solution is to design the service in a way that front-running attacks are not profitable—this is the approach we take here.

[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.

[2] A. Juels. blog.chain.link, 11 Sep 2020.

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.

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 **??**) 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 PQ 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 PQ 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

Enqueue Performance. We implemented, deployed, and tested each PQ using Truffle and Ganache. 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 **??**. 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 cost for fully dequeuing the PQ variants is outlined in Table **??**. 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 worse based on how much of a refund they generate.

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

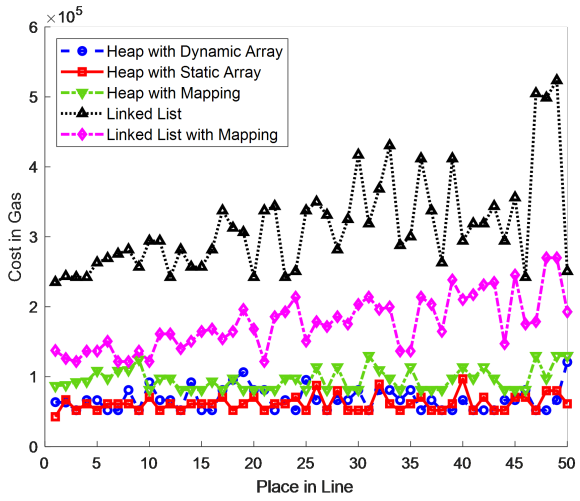

Figure 1: Gas costs for enqueuing 50 random integers into five priority queue variants. For the x-axis, a value of 9 indicates it is the 9th integer entered in the priority queue. The y-axis is the cost of enqueuing in gas.

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 **??**, 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 DApp. 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

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

	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 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.

	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 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.

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 **??**, 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.

Refunds and PQs. One consequence of a linked list is that a new DApp 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 **??**, 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 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. As it can be seen in the last two rows of Table **??**, 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,

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 4: Primary operations of Lissy smart contract.

we strive to be good citizens of Ethereum and clean up to the extent that we can—thus all PQs in Table **??** implement some cleanup.

4 Call Market Design

Lissy is proof of concept code. We try to simplify the design at every step to make it an extensible base class but still functional without any extensions. A call market will open for a specified period of time during which it will accept a capped number of orders (*e.g.,* 100 orders—parameterized so that all orders can be processed), 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 which require a second transaction and front-running attacks apply to cancellation orders [**?**]. As markets are relatively short-lived, orders simply expire when the market call period ends.

Another simplifying assumption is to implement a *collateralized* call market. We assume all trades are between ETH and an ERC20 token, all orders are pre-funded in the DApp 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 open source and written in 336 lines (SLOC) of Solidity plus the priority queue (*e.g.,* a heap with dynamic array is 282 SLOC). Refer to the availability section at the end of the paper. We tested it with the Mocha testing framework using Truffle on Ganache-CLI to obtain our performance metrics. Once deployed, the bytecode of Lissy is 10,812 bytes plus the constructor code (6,400 bytes) which is not stored. We cross-checked for vulnerabilities with *Slither*[4] and *SmartCheck*[5] (it only fails some 'informational' warnings that are intentional

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

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

	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. Each consumes just under the block gas limit (∼11M gas) with a full refund of half of its gas.

design choices—*e.g.,* a costly loop). Table **??** summarizes Lissy's primary operations.

4.1 Measurements

Previous Table (before adding Arbitrum data):

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 PQ implementation is the main influence on performance (see Table **??**). 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 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 **??**), 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 Front-running

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.

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 that they quote in their orders—and it removes any incentive for miners to front-run these profits.

Potential for front-running still exists around ties on price and order cancellations (both discussed in the next section). Finally, *displacement attacks* [**?**] are possible where competitive orders are delayed long enough for the market to close without them or the adversary fills up the current market (which caps the number of orders) to ensure other orders cannot be added. Both are expensive attacks. For all these reasons, we say Lissy reduces front-running attacks but stop short of saying this issue is completely solved by Lissy.

4.3 Cleaning Up Revisted

Beyond the cleaning up issues with priority queues in Section **??**, Lissy also uses mappings with each market. Traders preload their account with tokens to be traded (which comply with a common token standard called ERC20) 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 balances are updated for each trade that is executed). 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.

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) the calls need access control to ensure only the market contract can write to it (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.

5 Lissy on Layer 2

5.1 Roll-ups

We have avoided augmenting Lissy with centralized components and third party services as our research question con-

cerns the feasibility of a system with a minimum of regulatory hooks. However from a regulatory stance, there is a big difference between an architecture where the centralized component is publicly visible and interacted with by users (*e.g.,* most DEXes, roll-up architectures like Loopring, and commit-chain solutions like TEX). We consider an alternative design that is almost as difficult to regulate as a fully on-chain solution. In this design, an off-chain component is introduced to boost performance but it only interacts with the Ethereum network and never directly with traders. Traders still only interact with Ethereum.

Layer 2 solutions are a group of technologies that are designed and proposed to address specific drawbacks of executing transactions on *Layer 1* (*i.e.,* Ethereum and other blockchains) [**?**]. These technologies focus on fast transaction throughput, reducing gas costs, or educing transaction latency. When using Lissy, we strive to reduce the gas cost as performance is the main bottleneck. Thus, we choose a Layer 2 technology called *roll-up* which aims at reducing the gas cost for operating on Layer 1 by taking the transaction executions off-chain and only using the Ethereum blockchain for storing data. In a roll-up, every transaction is executed by a server or cluster of servers known as *validators* that can be run by a collection of users or third party operators (here they can be run by the token issuer). These validators then push the result of the executions (*i.e.,* updates in the EVM state) back to the Ethereum and assure the Ethereum network that the transactions have been executed correctly.

There are two broad roll-up categories (i) *zero knowledge* and (ii) *optimistic* roll-ups, each taking a different approach to ensure the validity of off-chain transaction executions. A zero knowledge roll-up (*a.k.a.* ZK-rollup) uses cryptographic validity proofs (SNARKS [**?**, **?**]) which can be later verified on-chain. These proofs are heavy to compute (introduce a burden to the validators who generate them) but considered valid once posted to the Ethereum. An optimistic roll-up uses a side chain (a roll-up chain) that is run on top of Layer 1. Once the executions are performed, validators assert the new states of the roll-up chain to the Ethereum and wait for anyone to dispute these assertions. Although the dispute time delays result in a slower transaction finality, optimistic roll-ups substantially increase the performance by decreasing the gas cost. Therefore, we choose this roll-up solution to for the Layer 2 implementation of Lissy.

5.2 Arbitrum

Arbitrum is an Ethereum scaling Layer 2 solution that operates an optimistic roll-up [**?**]. The roll-up chain is run on top of the Ethereum blockchain and contains all the Arbitrum contracts (called ArbOS). There is a bridge contract (called the EThBridge) that resides on the Ethereum and serves as a bridge between two chains. To execute a transaction on a DApp that is running on Arbitrum, a user first sends a transaction to the bridge contract. This transaction only contains the instructions for calling a function and the data required for it. As as soon as this transaction is included in the Ethereum block, it will get timestamped and ordered. A validator, who is constantly watching the bridge contract for new transactions, sees the transaction, executes it and asserts the result of the execution to the ArbOS. Once the dispute challenge period is over, the transaction is final and the new state of the ArbOS will be inserted into the Ethereum. The bridge contract maintains all the data required for verification using which anyone can compute the state of the ArbOS and assure the executions have been performed correctly.

5.3 Lissy on Arbitrum

Testing Platforms. Arbitrum currently operates a test newtork with a bridge contract on the Ethereum Kovan testnet. To experiment with Arbitrum, we implemented a variant of Lissy and run the same experiments that we previously performed on Layer 1 (Ethereum). To call functions on Lissy, traders can (i) send transactions directly to the Arbitrum EthBridge contract, or (ii) use a relay server (called an *Aggregator*) provided by the Arbitrum. These aggregators will group and send all pending transactions together as a single Kovan transaction to the EthBridge contract (and pay the gas).

Performance Evaluations. Table **??** compares the cost of executing the maximum number of trades on Ethereum (Kovan testnet) and executing the same number of trades when running Lissy on Layer 2 with Arbitrum. The fourth column of the Table represents the Ethereum gas cost for relaying the transaction to the Arbitrum EthBridge contract and is paid in ETH. Note that here we report the cost of sending the transaction directly to the Arbitrum EthBridge contract and not through the aggregator [6]. Th last column of the Table shows the actual cost of executing the transaction that is measured in ArbGas paid by the validators. ArbGas is basically used to measure the validators' resource usage and is significantly cheaper than Layer 1 gas (currently the testnet has 100 ArbGas for every Layer 1 gas unit, this ratio is even likely to improve in Arbitrum's next versions).

Table **??** shows that Arbitrum reduces the cost for executing the trades by around 99.9% (a Lissy variant implemented with linked list can handle 152 pairs of trades at most with the cost of 5,495,265 gas, while on Arbitrum the cost is 1,325 gas). Note that when using Lissy on Arbitrum, we can basically process an infinite number of orders. In our Lissy variant on Arbitrum, 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

[6] The Layer 1 cost reported on the Kovan block explorer represents the gas cost related to the entire batch of transactions.

	Ethereum		Arbitrum		
	Max trades (w.c.)	*L1 Gas for max trades*	*L1 Gas for max trades*	*L2 ArbGas for max trades*	*Size*
Heap with Dynamic Array	38	5,372,679	1,544	46,705,693	103
Heap with Static Array	42	5,247,636	1,402	46,311,418	103
Heap with Mapping	46	5,285,275	1,339	50,395,993	103
Linked List	152	5,495,265	1,325	99,514,286	103
Linked List with Mapping	86	5,433,259	1,319	58,035,156	103

Table 6: Comparison between executing trades on Ethereum and Arbitrum. Size is the calldata in bytes.

Arbitrum roll-ups, 38 pairs cost 1,544 gas and ass the pairs increase, the cost is essentially constant. Submitting an order costs 2,005 gas on average as opposed to 207,932 on-chain.

Our Lissy variant is not the first roll-up based order book. Loopring 3.0[7] 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 uses on-chain submission so that the roll-up server does not need to be publicly reachable. Loopring 3.0 can operate near high frequency trading as order submission is unhampered by Ethereum, however its roll-up proof does not ensure that the exchange did not reorder transactions, which is particularly problematic in a continuous-time order book. Traders who prioritize trade fairness might opt for a solution like our variant, while traders who want speed would vastly prefer the Loopring architecture which offers near-CEX speed while being non-custodial. Loopring leaves a regulatory hook whereas our variant could be nearly as difficult to regulate as a fully on-chain solution if the roll-up server was kept anonymous: Ethereum and Arbitrum themselves would be the only regulatory hooks.

5.4 Front-running on Arbitrum

Recall for running functions on Lissy on Arbitrum, traders must first instantiate the truncation by sending it to the Arbitrum EthBridge contract. As as soon as this transaction is included in the Layer 1 (Ethereum) block it will get timestamped and ordered. Thus, potential for front-running on Arbitrum is same as those explained in **??**.

6 Design Alternatives and Extensions

Lissy is designed as base class that can be extended and customized. We discuss potential modifications here.

[7] https://loopring.org

6.1 Token Divisibility and Ties

When executing trades, if the volume of the current best bid does not match the best ask, the larger order is partially filled and the remaining volume is considered against the next best order. In Lissy, tokens are assumed to be divisible. We 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)[8] however this approach does not always work. 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 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 [**?**]).

6.2 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

[8] 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.

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.

6.3 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.

6.4 Scheduling Events

As a simple base class, 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 actually incentivized to run `closeMarket()` last to make the most profit. This pattern is called miner extractable value (MEV) [**?**] and is usually considered in the context of attacks. However in our case, MEV is a feature. Efficient algorithms for miners to automatically find MEV opportunities is an open research problem.

A close alternative is to allow markets to open and close on different blocks. In this alternative, 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 (which creates a regulatory hook).[9] Next we consider the second issue of who pays to close the market.

6.5 Who Pays to Close/Reopen the Market?

As miners are paid all price improvements in the market, it is possible that a miner might run `closeMarket()` and it would pay for itself. However we consider two other scenarios that do not assume miners can automatically find MEV opportunities. One solution requires a modified closing function, `closeMarket(n)`, that only processes *n* orders at a time until the order book is empty (this is sensible design in any case to safeguard against the order book from locking up because the number of orders exceeds the gas limit to process them). 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 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 occasionally bailout the market when it halts on `closeMarket()` to facilitate further trading. If the DApp can pay for its own functions, an interested party can also arrange for a commercial service (*e.g.,* any.sender[10]) to relay the `closeMarket()` function call on Ethereum (an approach

[9] https://ethereum-alarm-clock-service.readthedocs.io/
[10] https://github.com/PISAresearch/docs.any.sender

called *meta-transactions*) which introduces another regulatory hook.

6.6 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.

7 Concluding Remarks

Imagine you have just launched a token on Ethereum. Now want 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 an on-chain order book, you just deploy the code alongside your token and trading is immediately supported. This should concern regulators. Even if it is too slow today, there is little reason for developers not to offer it as a fallback solution that accompanies every token. With future improvements to blockchain scalability, it could become the defacto trading method.

It may seem paradoxical or unethical to build for regulators exactly what they worry about, however we agreed that it was too difficult to answer our research questions without actual implementation and experimentation. Lissy is proof of concept code that implements only enough to understand the feasibility of on-chain trading and we release the code for reproducibility. However it is not production code, it is unpolished, it has no UI, and we have no intention of promoting it for adoption. Finally, by understanding the 'pain points' in the design, we found we were constantly tugged toward centralized components (Ethereum alarm clock, meta-transactions, rollup servers, *etc.*) which could serve as regulatory hooks even if the service is mainly on-chain.

Availability

The Solidity source code for Lissy, Truffle test files, and Arbitrum dependencies are available in a GitHub repository[11]. We have also deployed Lissy on Ethereum's testnet Rinkeby with flattened (single file) source code of just the Lissy base class and priority queue implementations. It is visible and can be interacted with here: [etherscan.io]; while the Arbitrum variant is here: [explorer.offchainlabs.com].

Acknowledgements.

The authors thanks the AMF (Autorité des Marchés Financiers) for supporting this research project. J. Clark also acknowledges partial funding from the National Sciences and Engineering Research Council (NSERC)/Raymond Chabot Grant Thornton/Catallaxy Industrial Research Chair in Blockchain Technologies, as well as NSERC through a Discovery Grant. M. Moosavi acknowledges support from Fonds de Recherche du Québec - Nature et Technologies (FRQNT).

[11] https://github.com/MadibaGroup/2020-Orderbook