TBD

No Author Given

No Institute Given

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.,* 'roll-ups') in closing the performance gap?
To answer these questions, we 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 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). We also provide a thorough front-running evaluation between Lissy and other types of markets.

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 key components 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 favor 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 financial regulator (name withheld for anonymity) 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 (see Section 5 for in-depth front-running analysis). 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 6.5).

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. A broader consequence of our research is that any

on-chain application that requires the traversal of hundreds of elements in a data structure in a single transaction is not feasible on Ethereum today.

We also speculate that Ethereum is likely to improve vastly in the coming years and we demonstrate one avenue for improvement through 'roll-ups' [**?**] which reduce the gas costs of Lissy by more than 99.9%. While roll-ups are not fully on-chain and introduce new network participants, the roll-up architecture is still concerning to regulators because the participants (called validators) are not directly reachable by users; they only watch the Ethereum state and interact directly with Ethereum.

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 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.

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). Table 1 illustrates various trade execution systems and summerizes their advantages and disadvantages.

Exchange Platform	Description	Advantages	Disadvantages
Centralized Exchanges	Exchange acts as a trusted third party between traders	High Liquidity High trade performance Low price slippage Easy to regulate	Third party custodian Low security No instant settlement Server downtime
On-chain Order Books	Exchange operates fully on-chain as coded	High security Non-custodial No trusted third party Fully Autonomous No Impermanent Loss Low price slippage	Low trade performance Hard to regulate
On-chain Dealers	A dealer uses its own inventory to offer a price for assets being traded	High security Non-custodial No trusted third party Fully Autonomous High performance	Impermanent Loss High price slippage No support for limit orders
Hybrid Designs	Exchange operates partially on-chain	High trade performance Easy to regulate Non-custodial	Low security Server downtime Uncertain fair execution

Table 1: Comparison among different trade execution systems.

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, 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 regardless through front-running). Call markets largely side step front-running attacks from other traders because reordering trades has no impact (discussed more in section 5). The paper does not include an implementation, was envisioned as running on a custom blockchain (Ethereum was still in development in 2014) and 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 [?], with recent interest in expanding their use to combat high frequency trading [?,?]. 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.

Other Academic Literature. The most similar academic work to this paper is the Ethereum-based periodic auction by Galal *et al.* [?] and the continuous-time exchange TEX [?]. As with us, front-running is a main consideration of these works. In a recent SoK on front-running attacks in blockchain [?], three general mitigations are proposed. Both of these papers use the solution of providing confidentiality over the content of orders. 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 mitigation approach 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 with Lissy which uses *no cryptography* and is *submit-and-go* for traders. A detailed comparison of front-running is provided in Section 5.

Our paper also emphasizes implementation details: Galal *et al.* do not provide a full implementation, and TEX uses both on-chain and off-chain components, and thus does not answer our research question of how feasible an on-chain order book is.

Less related are papers on topics 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 [?]. Velocity [?] and Findel [?] consider structuring derivatives in smart contracts. Atomic swaps (*i.e.,* payment vs. delivery) are necessary for settling trades and some general approaches include Arwen [?] and Tesseract [?].

3 Call Market Design

Our proof of concept, Lissy, is a call market that 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 priority queue (discussed in Section 3.1). Our vision is the market would be open for a very short period of time, close, and then reopen immediately (*e.g.,* every other block). Lissy is open source and written in 336 lines

[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
depositToken()	Deposits ERC20 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 bids inside the priority queue
submitAsk()	Inserts the upcoming asks inside the priority queue
claimTokens()	Transfers tokens to the traders
claimEther()	Transfers ETH to the traders

Table 2: Primary operations of Lissy smart contract.

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 3: Operations for a generic priority queue.

(SLOC) of Solidity plus the priority queue (*e.g.,* a heap with dynamic array is 282 SLOC). 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. The Solidity source code for Lissy and Truffle test files are available in a GitHub repository.[3] 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]. We cross-checked for vulnerabilities with *Slither*[4] and *SmartCheck*[5] (it only fails some 'informational' warnings that are intentional design choices—*e.g.,* a costly loop). Table 2 summarizes Lissy's primary operations. Figuremindmap represents a design landscape and extensions.

3.1 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 are the most time-consuming steps. 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

[3] Github: Link removed for anonymity.
[4] https://github.com/crytic/slither
[5] https://tool.smartdec.net

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 3) 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 also be 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 it ends up being the most gas efficient data structure to dequeue. Thus, each node is its own contract and contains the incoming order's 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 the incoming order's 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.

3.2 Priority Queue Evaluation

Enqueue Performance. We implemented, deployed, and tested each PQ using the Truffle development framework [**?**] 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 1. We 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.

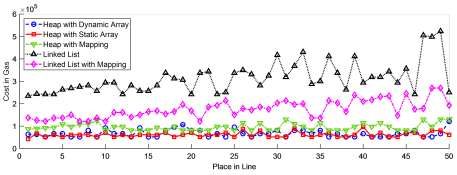

Fig. 1: Gas costs for enqueuing 50 random integers into five priority queue variants. For the x-axis, a value of 9 indicates it is the 9th integer entered in the priority queue. The y-axis is the cost of enqueuing in gas.

	Gas Costs (gwei)	*Refund (gwei)*	*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 4: The gas metrics associated with dequeuing 50 integers from five priority queue 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 4. These tests are performed using the following Ethereum gas metrics: block gas limit =11,741,495 and 1 gas = 56 Gwei.[6] 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.

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

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

	Gas Costs (*gwei*)	Potential Refund (*gwei*)	Full Refund?
Linked List without `SELFDESTRUCT`	721,370	0	◐
Linked List with `SELFDESTRUCT`	557,085	1,200,000	●
Linked List with Mapping and without `DELETE`	334,689	765,000	●
Linked List with Mapping and `DELETE`	731,514	3,765,000	●

Table 5: 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.

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 behavior 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.,* closing the market and matching the orders).

3.3 Cost/Benefit of Cleaning Up After Yourself

Gas Refunds. To incentivize for deleting unused storage on the blockchain, some EVM operations cost a negative amount of gas. These include `SELFDESTRUCT` that destroys contract and refunds 24,000 gas, and `SSTORE` that resets storage values from non-zero to zero and refunds 15,000 gas. 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 an 11M limit). 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 4, 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 or15,000 respectively.

Refunds and PQs. 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

	Max Trades (w.c.)	Gas for Max Trades (*gwei*)	Gas for 1000 Trades (*gwei*)	Gas for Submission (avg) (*gwei*)
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. Each consumes just under the block gas limit (∼11M gas) with a full refund of half of its gas, recall Ethereum gas refund is capped at 50% of the total gas cost of the transaction(see Gas Refunds:3.3).

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 5, 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 finish processing a node in the PQ. As it can be seen in the last two rows of Table 5, leaving the data on the blockchain 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 4 implement some cleanup.

4 Lissy Performance 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 PQ implementation is the main influence on performance. 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). Table 6 shows the performance results of Lissy for each PQ variant. In the first two columns, we determine the highest number of trades that can be executed and processed in a single call to the `closeMarket()` and not exceed the current Ethereum block gas limit of 11,741,495. Since Ethereum will become more efficient over time, we were also 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 cost 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).

5 Front-running Evaluation

As we illustrate in Table 7, call markets have a unique profile of resilience against *front-running attacks* [**?**,**?**,**?**] that differs somewhat from continuous-time markets and automated market makers. Traders are sometimes distinguished as *makers* (adds orders to a market) and *takers* (trades against a pre-existing, unexecuted orders). A continuous market has both. All traders using an automated market maker are takers, while the investors who provide tokens to the AMM (liquidity providers) are makers. Under our definition, a call market only has makers: the only way to have a trade executed is to submit an order. The front-running attacks in Table 7 are sub-categorized, using a recent SoK [**?**], as being *Insertion*, *Displacement*, and *Suppression*. To explain the difference, we will illustrate the first three attacks in the table.

In an *insertion attack*, Mallory learns of a transaction from Alice. Consider Alice submitting a bid order for 100 tokens at any price (market order). Mallory decides to add new ask orders to the book (limit orders) at the maximum price reachable by Alice's order given the rest of the asks in the book. Mallory must arrange for her orders to be added before Alice's transaction and then arrange for Alice's transaction to be the next (relevant) transaction to run (*e.g.,* before competing asks from other traders are added).

			Centralized Continuous Market (Coinbase)	Partially Off-chain Continuous Market (EtherDelta)	Partially Off-chain Continuous Market w/ Roll-up (Loopring)	On-chain Continuous Market (OasisDex)	On-chain Dark Continuous Market (TEX)	On-chain Automated Market Maker (Uniswap)	On-chain Call Market w/ Price Improvement	On-chain Call Market (**Lisy**)	On-chain Call Market w/ Roll-up (**Lisy** variant)	On-chain Dark Call Market (Galal *et al.*)
Who is Mallory? Authority, Trader, Miner, Sequencer			A	A,T,M	A,T,M,S	T,M	T,M	T,M	T,M	T,M	T,M,S	T,M
Attack Example	Mallory (*maker*) squeezes in a transaction before Alice's (*taker*) order	Ins.	○	○	○	○	●	○	●	●	●	●
	Mallory (*taker*) squeezes in a transaction before Bob's (*taker 2*)	Disp.	○	○	○	○	●	○	●	●	●	●
	Mallory (*maker 1*) suppresses a better incoming order from Alice (*maker 2*) until Mallory's order is executed	Supp.	○	○	○	●	●	●	◐	◐	◐	◐
	A hybrid attack based on the above (*e.g.,* sandwich attacks, scalping)	I/S/D	○	○	○	○	●	○	○	●	●	●
	Mallory suspends the market for a period of time	Supp.	○	○	○	◐	◐	◐	◐	◐	◐	◐
	Spoofing: Mallory (*maker*) puts an order as bait, sees Alice (*taker*) tries to execute it, and cancels it first	S&D	○	○	○	○	●	○	●	●	●	●
	Cancellation Griefing: Alice (*maker*) cancels an order and Mallory (*taker*) fulfills it first	Disp.	○	○	○	○	●	○	●	●	●	●

Table 7: An evaluation of front-running attacks (rows) for different types of order books (columns). Front-running attacks are in three categories: Insertion, displacement, and suppression. A full dot (●) means the front-running attack is mitigated or not applicable to the order book type, a partial mitigation (◐) is awarded when the front-running attack is possible but expensive, and we give no award (○) if the attack is feasible.

In a centralized exchange, Mallory would collude with the *authority* running the exchange to conduct this attack. On-chain, Mallory could be a fast *trader* who sees Alice's transaction in the mempool and adds her transaction with a higher gas fee to bribe miners to execute hers first (insertion is probabilist and not guaranteed). Finally, Mallory could be the *miner* of the block that includes Alice's transaction allowing her to insert with high fidelity. Roll-ups use *sequencers* discussed in Section 7.4.

A *displacement attack* is like an insertion attack, except Mallory does not care what happens to Alice's original transaction—she only cares about being first. If Mallory sees Alice trying to execute a trade at a good price, she could try to beat Alice and execute the trade first. Mallory is indifferent to whether Alice can then execute her trade or not. The analysis of both insertion and suppression attacks are similar. Call markets mitigate these basic insertion and displacement attacks because they do not have any time priority (e.g., if you were to shuffle the order of all orders submitted within the same call, the outcome would be exactly the same). A different way to mitigate these attacks is to seal orders with confidentiality (a *dark* market).

In a *suppression attack*, Mallory floods the network with transactions until a trader executes her order. Such selective denial of service is possible by

an off-chain operator. With on-chain continuous markets, it is not possible to suppress Alice's transaction while also letting through a transaction from a taker—suppression applies to all Ethereum transactions or none. A call market is uniquely vulnerable because it eventually times out (which does not require an on-chain transaction) and new orders cannot be added. We still award a call market partial mitigation since suppression attacks are expensive (*cf.* Fomo3D attack [**?**]). If the aim of suppression is a temporary denial of service (captured by attack 5 in the table), then all on-chain markets are vulnerable to this expensive attack.

Some attacks combine more than one insertion, displacement, and/or suppression attacks. AMMs are vulnerable to a double insertion called a sandwich attack [**?**] which bookends a victim's trade with the front-runner's trades (plus additional variants). 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* and it allows traders to submit at their best price. A hybrid front-running attack allows Mallory to extract any price improvements. Consider the case where Alice's ask crosses Bob's bid with a material price improvement. Mallory inserts a bid at Alice's price, suppresses Bob's bid until the next call, and places an ask at Bob's price. She buys and then immediately sells the asset and nets the price improvement as arbitrage. To mitigate this in Lissy, all price improvements are given to the miner (using `block.coinbase.transfer()`)traders would have to pay for that. 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.

Other front-running attacks use order cancellations (discussed in the next section) which Lissy mitigates by running short-lived markets with no cancellations.

There are two main takeaways from Table 7. Call markets provide strong resilience to front-running only bested slightly by dark markets like TEX [**?**], however they do it through design—no cryptography and no two-round protocols. A second observation is that dark call markets, like Galal *et al.* [**?**], are no more resilient to front-running than a lit market (however confidentiality could provide resilience to predatory trading algorithms that are reactionary but do not front-run).

6 Design Alternatives and Extensions

Lissy is designed as a base class that can be extended and customized. Here, we evaluate potential modifications.

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. A common trading rule (that does resist

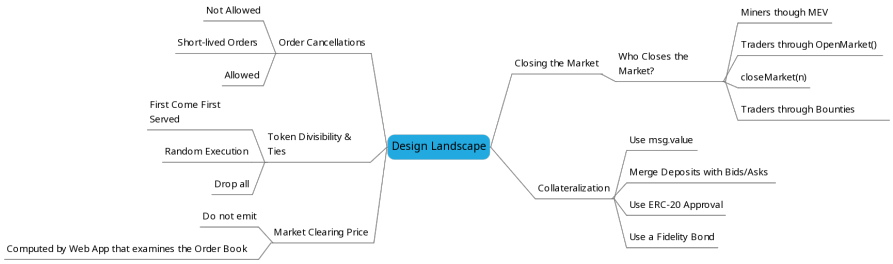

Fig. 2: Mind Map Draft

front-running) is to fill ties in proportion to their volume (*i.e., pro rata* allocation)[7], 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, (2) the bids could be prioritized based on time, or (3) the bid could be randomly chosen (*cf.* Libra [**?**]).

Evaluation. We avoid (1) which is fair but not market efficient. We note the conditions under which pro rata allocation fails (*i.e.,* non-divisible assets, an exact tie on price, and part of the final allocation) are improbable. In Lissy, tokens are assumed to be divisible but we otherwise default to (2). This means insertion front-running attacks are possible for this corner case. However if these attacks proved to be a problem in practice, (3) is a better solution with one main drawback: on-chain sources of 'randomness' are generally deterministic and manipulatable by miners [**?**,**?**], countermeasures can 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.

Evaluation. Support for cancellation 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. Also, 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 self-balancing binary

[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.

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.

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 derivatives 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 median 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

Lissy is a one-shot market. However it can be extended to reopen 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 on every block. Someone needs to call the `closeMarket()` for every block (we return to who this is later), 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 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).[8] 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 has 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.

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 the `closeMarket()` function receives the bounty.

Evaluation. The first 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 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 better, 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[9]) to relay the `closeMarket()` function call on Ethereum (an approach called *meta-transactions*) which introduces another regulatory hook.

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

6.6 Collateralization Options

In Lissy, both the tokens and ETH that a trader wants 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 options 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 ERC20 tokens for ETH, but would not work for ERC20 to ERC20 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 ERC20 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 number 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 allows traders to 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 ERC20 contract instead of simply updating its internal ledger.

7 Lissy on Layer 2

7.1 Roll-ups

We consider an alternative design for Lissy that is almost as difficult to regulate as a fully on-chain solution. In this design, an off-chain component is introduced to boost the 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 reducing 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*. This technology 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.

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: (1a) the first is to use cryptographic proof techniques (*e.g.,* SNARKS [**?**,**?**] and variants [**?**]). This is called a *zk-rollup*. Note that the proofs are heavy to compute (introducing a burden to the validators who generate them) but considered valid once posted to the Ethereum. The second approach (1b) 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] Approach (2) is called an *optimistic roll-up*. Although the dispute time delays result in a slower transaction finality, optimistic roll-ups substantially increase the performance by decreasing the gas cost.

7.2 Arbitrum

Arbitrum is an Ethereum Layer 2 scaling solution that operates as an optimistic roll-up [**?**]. Arbitrum runs entire DApps inside its environment, which acts like a container and requires special bridge contacts to interact with 'normal' (or Layer 1) Ethereum DApps. To deploy a DApp on Arbitrum, or to execute a function on an existing Arbitrum DApp, the transaction is sent to an *Inbox contract* on Layer 1. The transaction is not executed, it is only recorded (as calldata) in the Inbox. Like any other transactions on Ethereum, miners determine the order of the transactions sent to this contract. A set of validators (open to anyone) are watching the Inbox contract for new transactions. Once an Inbox transaction is finalized in an Ethereum block, validators will execute the transaction and assert the result of the execution to other validators on a sidechain called ArbOS. Once the dispute challenge period is over, the new state of ArbOS will be inserted into

[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.

Ethereum. As the Inbox contract maintains all Arbitrum transactions, anyone can recompute the entire current state of the ArbOS and file a dispute if executions are not performed correctly. Disputes are adjudicated by Ethereum itself and require essentially constant gas costs, regardless of the size of the gas costs of the transaction being disputed

7.3 Lissy on Arbitrum

Testing Platforms. To experiment with Arbitrum, we used the Arbitrum Rollup chain hosted on the Rinkeby testnet. We implemented a variant of Lissy ([https://rinkeby-explorer.arbitrum.io/]) and ran the same experiments that we previously performed on Layer 1. To call functions on Lissy, traders can (i) send transactions directly to the Inbox contract, or (ii) use a relay server (called a *Sequencer*) provided by the Arbitrum. The sequencer will group, order, and send all pending transactions together as a single Rinkeby transaction to the Inbox (and pays the gas).

Recall that Lissy implements an exchange between ETH and tokens. To use Layer 1 assets on Layer 2, Arbitrum provides two bridge contracts: one for ETH, and one for tokens compliant with ERC20 or other standard interfaces. Prior to trading, a trader uses the bridge to move their ETH to the same address on Layer 2 and then move the ETH to Lissy with a Layer 2 transaction. From the user's perspective, tokens work similarly although the backend details are more complex. Finally, withdrawals work essentially same way in reverse, but are only final on Layer 1 after a dispute challenge period (currently 1 hour).[11]

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()`). Our experiments show that on Arbitrum, heaps are 32% more efficient than non-Heaps for submitting orders and 29% less efficient for closing. Recall that without a roll-up, such a priority queue can only match 38 pairs at a cost of 5,372,679 gas. With Arbitrum roll-ups, 38 pairs cost only 1,632 in Layer 1 gas (a 99.9% savings). This is effectively the cost of recording the `closeMarket()` function call in the Inbox, which is 103 bytes of calldata. As the number of pairs increases, Layer 1 gas cost is constant. For traders submitting an order, the cost is reduced from 207,932 to 2,656.

While the runtime of a transaction is no longer an issue for Ethereum, it is still an issue for the validators. Similar to Ethereum miners, validators do measure the amount of gas (called ArbGas in Arbitrum) a transaction takes, although Arbitrum emulates EVM code with its own different opcodes. Thus, ArbGas with a roll-up and Layer 1 gas without a roll-up are not directly comparable but both increase and decrease proportionately with the complexity of running the function. Closing a market with 38 pairs costs 46,705,693 ArbGas.

[11] Layer 1 users might accept assets before they are finalized if they determine their eventual emergence on Layer 1 is indisputable (*eventual finality*).

Our Lissy variant is not the first roll-up-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 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.

7.4 Front-running on Arbitrum

In our Lissy variant on the Arbitrum, traders can submit transactions to the Layer 1 Inbox contract instead of directly to the Lissy DApp. This has the same front-running profile as Lissy itself; only the Layer 1 destination address is different. If a sequencer is mandatory, it acts with the same privilege as a Layer 1 Ethereum miner in ordering the transactions it receives. Technically, sequencers are not limited to roll-ups and could be used in the context of normal Layer 1 DApps, but they are more apparent in the context of roll-ups. A sequencer could be trusted to execute transactions in the order it receives them, outsource to a fair ordering service, or (in a tacit acknowledge of the difficulties of preventing front-running) auction off permission to order transactions to the highest bidder. As shown in Table 7, a sequencer is an additional front-running actor but does not otherwise change the kinds of attacks that are possible.

8 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 roll-ups, 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 de facto trading method.

[12] `https://loopring.org`

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 a 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 user interface (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, roll-up servers, *etc.*) which could serve as regulatory hooks even if the service is mainly on-chain.

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.

A 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 0 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 an 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. The `SELFDESTRUCT` operation does not remove the initial byte code of the contract from the chain. It always refunds 24,000 gas. For example, if 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.

B Roll-ups: Additional Background

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

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: (1a) the first is to use cryptographic proof techniques (*e.g.,* SNARKS [**?**,**?**] and variants [**?**]). This is called a *zk-rollup*. Note that the proofs are heavy to compute (introducing a burden to the validators who generate them) but considered valid once posted to the Ethereum. The second approach (1b) 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 [**?**]).[13] Approach (2) is called an *optimistic roll-up*. Although the dispute time delays result in a slower transaction finality, optimistic roll-ups substantially increase the performance by decreasing the gas cost.

[13] 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.

C Cleaning Up Revisited

Beyond the cleaning up issues with priority queues in Section 3.3, 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 zeroing 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.