Lissy: Experimenting with on-chain order books

Mahsa Moosavi and Jeremy Clark

Concordia University, Montréal, Canada

Abstract. Financial regulators have long-standing concerns about fully decentralized exchanges that run 'on-chain' without any obvious regulatory hooks. The popularity of Uniswap, an automated market makers (AMM), made these concerns a reality. AMMs implement a lightweight dealer-based trading system, but they are unlike anything on Wall Street, require fees intrinsically, and are susceptible to front-running attacks. This leaves the following research questions we address in this paper: (1) are conventional (*i.e.,* order books), secure (*i.e.,* resistant to front-running and price manipulation) and fully decentralized exchanges feasible on a public blockchain like Ethereum, (2) what is the performance profile, and (3) how much do Layer 2 techniques (*e.g.,* Arbitrum) increase performance? To answer these questions, we implement, benchmark, and experiment with an Ethereum-based call market exchange called Lissy. We confirm the functionality is too heavy for Ethereum today (you cannot expect to exceed a few hundred trade executions per block) but show it scales dramatically (99.88% gas cost reduction) on Arbitrum.

1 Introductory Remarks

There are three main approaches to arranging a trade [19]. 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).

Conventional financial markets (*e.g.,* NYSE, NASDAQ) use both continuous time trading during open hours, and a call market before and during open hours to establish an opening price and a closing price. After early experiments at implementing continuous time trading on Ethereum (*e.g.,* EtherDelta, OasisDEX), it was generally accepted that conventional trading is infeasible on Ethereum for performance reasons. Centralized exchanges continued their predominance, while slowly some exchanges moved partial functionality on-chain (*e.g.,* custody of assets) while executing trades off-chain.

A clever quote-driven alternative, called an automatic market maker (AMM), was developed that only requires data structures and traversals with low gas complexity. This approach has undesirable price dynamics (*e.g.,* market impact of a trade, slippage between the best bid/ask and actual average execution price, *etc.*) which explains why there is no Wall Street equivalent, however, it is efficient on Ethereum and works 'good enough' to attract trading. First generation AMMs provide makers (called liquidity providers) with no ability to act on price information—they are uninformed traders that can only lose (called impermanent loss) on trades but make money on fees. Current generation AMMs (*e.g.,* Uniswap v3) provided informed makers with a limited ability (called concentrated liquidity) to act on proprietary information [31] without breaking Ethereum's performance limitations. Ironically, the logical extension of this is a move back to where it all started—a full-fledged order-driven exchange that allows informed makers the fullest ability to trade strategically.

Contributions. In this paper, we experiment with on-chain markets to understand in detail if they remain infeasible on Ethereum and what the limiting factors are. Some highlights from our research include answering the following questions:

- What type of exchange has the fairest price execution on balance? (A call market.)
- How many orders can be processed on-chain? (Upper-bounded by 152 per block.)

- How much efficiency can be squeezed from diligently choosing the best data structures? (Somewhat limited; turn 38 trades into 152.)
- To what extent can we mitigate front-running attacks? (Almost entirely.)
- Can we stop the exchange's storage footprint on Ethereum from bloating? (Yes, but it is so expensive that it is not worth it.)
- Are on-chain order books feasible on layer 2? (Yes! Optimistic roll-ups reduce gas costs by 99.88%.)
- Which aspects of Ethereum were encountered that required deeper than surface-level knowledge to navigate? (Optimizing gas refunds, Solidity is not truly object-oriented, miner extractable value (MEV) can be leveraged for good, and bridging assets for layer 2.)
- How hard is an on-chain exchange to regulate? (The design leaves almost no regulatory hooks beyond miners (and sequencers on layer 2).)

2 Preliminaries

2.1 Ethereum

We assume the reader is familiar with the following concepts: blockchain technology; smart contracts and decentralized applications (DApps) on Ethereum; how Ethereum transactions are structured, broadcast, and finalized; the gas model including the gas limit (approximately 11M gwei at the time of our experiments) per block. A **gas refund** is a more esoteric subject (not covered thoroughly in any academic work to our knowledge) that we use heavily in our optimizations. Briefly, certain EVM operations (`SELFDESTRUCT` and `SSTORE 0`) cost negative gas, with the follow caveats: 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). We provide full details of all of these topics in Appendix A.1.

2.2 Trade Execution Systems

Table 1 illustrates various trade execution systems and summarizes their advantages and disadvantages. Appendix A.2 provides a full justification for the table. Briefly, fully decentralized, on-chain exchanges require the lowest trust, provide instant settlement, and have transparent trading rules that will always execute correctly. Front-running attacks (see Section 5 for a very thorough discussion) are weaknesses inherent in blockchains that require specific mitigation.

2.3 Related Work

Call markets are studied widely in finance and provide high integrity prices (*e.g.,* closing prices that are highly referenced and used in derivative products) [20,30,15]. They can also combat high frequency trading [7,1]. An older 2014 paper [12] on the 'Princeton prediction market' [6] show that call markets mitigate most blockchain-based front-running attacks present in an on-chain continuous-trading exchange as well as other limitations: 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. 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.

The most similar academic work to this paper is the Ethereum-based periodic auction by Galal *et al.* [16] and the continuous-time exchange TEX [23]. As with us, front-running is a main consideration of these works. In a recent SoK on front-running attacks in blockchain [14], three general mitigations are proposed: confidentiality, sequencing, and design. Both of these papers use confidentiality over the content of orders (*cf.* [37,39,38,10,27]). 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 akin to first-in-first-out [22,25]. These are not available for experimentation on

Type	Description	Advantages	Disadvantages
Centralized Exchanges (CEX)	Order-driven exchange acts as a trusted third party (*e.g.,* Binance, Bitfinex)	Conventional Highest performance Low fees Easy to regulate Low price slippage Verbose trading strategies	Fully trusted custodian Slow withdrawals Server downtime Uncertain fair execution
Partially On-chain Exchange	Order-driven exchange acts as a semi-trusted party (*e.g.,* EtherDelta, 0x, IDEX, Loopring)	High performance Low fees Easy to regulate Low price slippage Verbose trading strategies Semi-custodial	Slow withdrawals Server downtime Front-running attacks Uncertain fair execution
On-Chain Dealers	Quote-driven decentralized exchange trades from inventory with public pricing rule (*e.g.,* Uniswap v3)	Non-custodial Instant trading Moderate performance Fair execution	Unconventional Impermanent loss High price slippage Intrinsic fees Front-running attacks Limited trading strategies Hard to regulate
On-chain Order-Driven Exchanges	Order-driven decentralized exchange executes trades between buyers and sellers (*e.g.,* Lissy)	Conventional Non-custodial Low price slippage Fair execution Verbose trading strategies Front-running is mitigable	Very low performance Hard to regulate

Table 1: Comparison among different trade execution systems.

Ethereum yet (although Chainlink has announced an intention[1]). 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.

3 Call Market Design

A call market opens for traders to submit bids and asks which are enqueued until the market closes. Trades are executed by matching the best priced bid to the best priced ask until the best bid is less than the best ask, then all remaining trades are discarded. See Appendix A.3 for a numeric example. If Alice's bid of $100 is executed against Bob's ask of $90, Alice pays $100, Bob receives $90 and the $10 difference (called a price improvement) is given to miners for reasons in explained in the front-running evaluation (Section 5).

For our experiments and measurements, we implement a call market from scratch. Lissy 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 (SLOC) of Solidity plus the priority queue (*e.g.,* we implement 5 variants, each around 300 SLOC). We tested it with the Mocha testing framework using Truffle [36] on Ganache-CLI [35] to obtain our performance metrics. Once deployed,

[1] 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.

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.[2] 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*[3] and *SmartCheck*[4] and it only fails some 'informational' warnings that are intentional design choices (*e.g.,* a costly loop). All measurements assume a block gas limit of 11 741 495 and 1 gas = 56 Gwei.[5] Table 2 summarizes Lissy's primary operations.

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. Assessing which data structures will perform best is hard (*e.g.,* gas refunds, a relatively cheap mapping data structure, only partial support for object-oriented programming) without actually deploying and evaluating several variants.

We first observe that orders are executed in order: highest to lowest price for bids, and lowest to highest price for asks. This means random access to the data structure holding the orders is unnecessary (we discuss cancelling orders later in Section 6.2). We can use a lightweight *priority queue* (PQ) which has only two functions: Enqueue() inserts an element into the priority queue; and Dequeue() removes and returns the highest priority element. 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.

As closing the market is very expensive with any PQ, we rule out sorting the elements while dequeuing and sort during each enqueue. We then implement the following 5 PQ variants:

1. **Heap with Dynamic Array.** A heap is a binary tree where data is stored in nodes in a specific order where the root always represents the highest priority item (*i.e.,* highest bid price/lowest ask price). Our heap stores its data in a Solidity-provided dynamically sized array. The theoretical time complexity is logarithmic enqueue and logarithmic dequeue.
2. **Heap with Static Array.** This variant replaces the dynamic array with a Solidity storage array where the size is statically allocated. This is asymptotically the same and marginally faster in practice.
3. **Heap with Mapping.** In this variant, we store a key for the order in the heap instead of the entire order itself. Once a key is dequeued, the order struct is drawn from a Solidity mapping (which stores key-value pairs very efficiently). This is asymptotically the same and faster with variable-sized data.
4. **Linked List.** In this variant, elements are stored in a linked list (enabling us to efficiently insert a new element between two existing elements during enqueue). Solidity is described as object-oriented but the Solidity equivalent of an object is an entire smart contract. Therefore, an object-oriented linked list must

[2] https://github.com/MadibaGroup/2020-Orderbook
[3] https://github.com/crytic/slither
[4] https://tool.smartdec.net
[5] EthStats (July 2020): `https://ethstats.net/`

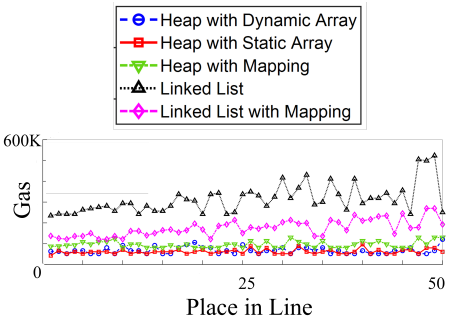

Fig. 1: Gas costs for enqueuing 50 random integers into five priority queue variants. For the x-axis, a value of 9 indicates it is the 9th integer entered in the priority queue. The y-axis is the cost of enqueuing in gas.

	Gas Used	*Refund*	*Full Refund?*
Heap with Dynamic Array	2,518,131	750,000	●
Heap with Static Array	1,385,307	750,000	●
Heap with Mapping	2,781,684	1,500,000	●
Linked List	557,085	1,200,000	●
Linked List with Mapping	731,514	3,765,000	●

Table 3: The gas metrics associated with dequeuing 50 integers from five priority queue variants. Full refund amount is shown but the actual refund that is applied is capped.

either (1) create each node in the list as a struct—but this is not possible as Solidity does not support recursive structs—or (2) make every node in the list its own contract. The latter option seems wasteful and unusual, but it surprisingly ends up being the most gas efficient data structure to dequeue. The theoretical time complexity is linear enqueue and constant dequeue.

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. Asymptotically, it is linear enqueue and constant dequeue.

We implemented, deployed, and tested each PQ. A simple test of enqueuing 50 integers chosen at random from a fixed interval is in Figure 1 and dequeing them all is in Table 3. Dequeuing removes data from the contract's storage resulting in a gas refund. Based on our manual estimates,[6] 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.

We observe that (1) the linked list variants are materially cheaper than the heap variants at dequeuing; (2) 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 (3) Ethereum will not permit more than hundreds of orders so asymptotic behaviour is not significant. For these reasons, we suggest using one of the linked list variants. 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 which balances a moderate gas cost for insertion (*i.e.,* order submission) with one for removal (*i.e.,* closing the market and matching the orders). In Section 4, we implement Lissy on Layer 2. There, the PQ variant does not change the layer 1 gas costs (as calldata size is the same) and the number of orders can be substantially increased. thus, we reconsider asymptotic and choose a heap (with dynamic array) to lower L2 gas costs across both enqueuing and dequeuing.

3.2 Cost/Benefit of Cleaning Up After Yourself

One consequence of a linked list is that a new contract is created for every node in the list. Beyond being expensive for adding new nodes (a cost that will be bared by the trader in a call market), it also leaves a

[6] EVM does not expose the refund counter. We determine how many storage slots are being cleared and how many smart contracts destroyed, then we multiply these numbers by 24,000 or 15,000 respectively.

	Gas Used	Potential Refund	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 4: The gas metrics associated with dequeuing 50 integers from four linked list variants. For the refund, (●) indicates the refund was capped at the maximum amount and (◐) means a greater refund would be possible.

large footprint in the active Ethereum state, especially if we leave the nodes on the blockchain in perpetuity (*i.e.,* we just update the head node of the list and leave the previous head 'dangling'). However in a PQ, nodes are only removed from the head of the list; thus the node contracts could be 'destroyed' one by one using an extra operation, `SELFDESTRUCT`, in the `Dequeue()` function. As shown in Table 4, the refund from doing this outweighs to the cost of the extra computation: gas costs are reduced from 721K to 557K. This suggests a general principle: cleaning up after yourself will pay for itself in gas refunds. Unfortunately, this is not universally true as shown by applying the same principle to the linked list with mapping.

Dequeuing in a linked list with mapping can be implemented in two ways. The simplest approach is to process a node, update the head pointer, and leave the 'removed' node's 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 trade. As it can be seen in the last two rows of Table 4, 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). In Appendix B, we present a second case study of the cost-benefit of clearing a mapping when it is no longer needed (including our idea to store the mapping in its own contract so it can `SELFDESTRUCT` with a single function call). The unfortunate takeaway is, again, that it is cheapest to leave the mapping in place. Cleaning up EVM state is a complicated and under-explored area of Ethereum in the research literature. For our own work, we strive to be good citizens of Ethereum and clean up to the extent that we can—thus all PQs in Table 3 implement some cleanup.

3.3 Lissy Performance Measurements

The main research question is how many orders can be processed under the Ethereum block gas limit. The choice of PQ implementation is the main influence on performance and the results are shown in Table 5. These numbers are for the *worst-case*—when every submitted bid and ask is marketable (*i.e.,* will require fulfillment). In practice, once `closeMarket()` hits the first bid or ask that cannot be executed, it can stop processing all remaining orders. Premised on Ethereum becoming more efficient over time, we 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 cost will vary depending on how many orders are already submitted (recall Figure 1), we average the cost of 200 order submissions.

The main takeaway is that call markets appear to be limited to processing about a hundred orders per transaction and even that is at the enormous cost of monopolizing an entire Ethereum block just to close the market. Perhaps 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).

	Max Trades (w.c.)	Gas Used for Max Trades	Gas Used for 1000 Trades	Gas Used for Submission(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.

	Layer1 gasUsed	**Layer2 ArbGas**
Lissy on Ethereum	5,372,679	N/A
Lissy on Arbitrum	6,569	508,250

Table 6: Gas costs of closing a market on Ethereum and on Arbitrum. ArbGas corresponds to Layer 2 *computation used*.

4 Lissy on Arbitrum

Layer 2 (L2) solutions [18] are a group of scaling technologies proposed to address specific drawbacks of executing transactions on Ethereum, which is considered *Layer 1 (L1)*. Among these proposals, *roll-ups* prioritize reducing gas costs (as opposed to other valid concerns like latency and throughput, which are secondary for Lissy). We review two variants, *optimistic roll-ups* and *zk roll-ups*, in Appendix A.1. Briefly, in a roll-up, every transaction is stored (but not executed) on Ethereum, then executed off-chain, and the independently verifiable result is pushed back to Ethereum, with some evidence of being executed correctly. In the Appendix, we also compare Lissy on Arbitrum to Loopring 3.0.

We choose to experiment with Lissy on the optimistic rollup Arbitrum.[7] To deploy a DApp on Arbitrum, or to execute a function on an existing Arbitrum DApp, the transaction is sent to an *inbox* on L1. It is not executed on L1, it is only recorded (as calldata) in the inbox. An open network of *validators* watch the inbox for new transactions. Once inbox transactions are finalized in an Ethereum block, validators will execute the transactions and assert the result of the execution to other validators on a sidechain called ArbOS. 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 correctly reported on ArbOS. Disputes are adjudicated by Ethereum itself and require a small, constant amount of gas, invariant to how expensive the transaction being disputed is. When the dispute challenge period is over, the new state of ArbOS is stored as a checkpoint on Ethereum.

4.1 Lissy Performance Measurements on Arbitrum

Testing Platforms. We implement Lissy using the Arbitrum Rollup chain hosted on the Rinkeby testnet. It is visible and can be interacted with here: [Arbitrum Explorer]. To call functions on Lissy, traders can (1) send transactions directly to the Inbox contract, or (2) 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).

[7] See `https://offchainlabs.com` for more current details than the 2018 *USENIX Security* paper [21].

In our Lissy variant on Arbitrum, the validators do all computations (both enqueuing and dequeuing) so we choose to use a heap with dynamic array for our priority queue, which balances the expense of both operations. Heaps are 32% more efficient than linked lists 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. Table 6 shows that 38 pairs cost only 6,569 in L1 gas (a 99.88% savings). This is the cost of submitting the `closeMarket()` transaction to the Inbox to be recorded, which is 103 bytes of calldata. Most importantly, recording `closeMarket()` in the Inbox will always cost around 6,569 even as the number of trades increases from 38 pairs to thousands or millions of pairs. Of course, as the number of trades increase, the work for the validators on L2 increases, as measured in ArbGas. The price of ArbGas in Gwei is not well established but is anticipated to be relatively cheap. Arbitrum also reduces the costs for traders to submit an order: from 207,932 to 6,917 in L1 gas. In Appendix A.1, the full interaction is shown in Figure 3, which illustrates how traders interact with Lissy on Arbitrum including bridges, inboxes, sequencers and validators.

Running Lissy on Arbitrum has one large caveat. If the ERC20 tokens being traded are not issued on ArbOS, which is nearly always the case today, they first need to be *bridged* onto ArbOS, as does the ETH. Traders send ETH or tokens to Arbitrum's bridge contracts which create the equivalent amount at the same address on L2. Withdrawals work the same way in reverse, but are only final on L1 after a dispute challenge period (currently 1 hour).[8]

5 Front-running Evaluation

As we illustrate in Table 7, call markets have a unique profile of resilience against *front-running attacks* [12,14,13] 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 subcategorized, using a recent SoK [14], 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).

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 5.1.

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

[8] L1 users might accept assets before they are finalized as they can determine their eventual emergence on L1 is indisputable (*eventual finality*).

			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 (Lissy)	On-chain Call Market w/ Roll-up (Lissy 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.

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 [14]). 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 [41] 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

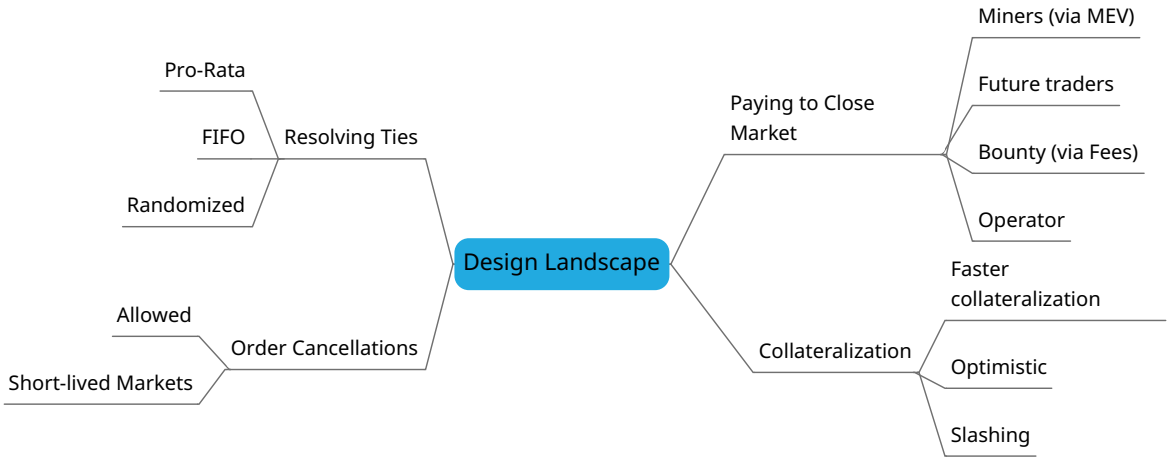

Fig. 2: A design landscape for on-chain call markets.

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.

Other front-running attacks use order cancellations (see Section 6.2) 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 [23], 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.* [16], are no more resilient to front-running than a lit market (however confidentiality could provide resilience to predatory trading algorithms that react quickly to trades without acutally front-running).

5.1 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 (called a *MEV auction*). 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.

6 Design Landscape

Lissy is a simple base class that implements the core functionality of a call market. To use it in the real world, design decisions need to be made about how it will be used. Figure 2 provides a design landscape for Lissy deployment, with possible extensions and customization.

6.1 Token Divisibility and Ties

A common trading rule is to fill ties in proportion to their volume (*i.e., pro rata* allocation)[9]. This can fail when tokens are not divisible. Consider the following corner case: 3 equally priced bids of 1 non-divisible token and 1 ask at the same price: (1) the bid could be randomly chosen (*cf.* Libra [28]), or (2) the bid could be prioritized based on time. In Lissy, tokens are assumed to be divisible. 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. 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. (1) is the fairest solution with one main drawback: on-chain sources of 'randomness' are generally deterministic and manipulatable by miners [5,9], while countermeasures can take a few blocks to select [4]. We implement (2) which means front-running attacks are possible in this one improbable case.

6.2 Order Cancellations

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 search tree instead of a heap; or methods to traverse a linked list rather than only pulling from the head). 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.

6.3 Who Pays to Close/Reopen the Market?

In the Princeton paper [12], the call market is envisioned 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). In Lissy, someone needs to execute `closeMarket()` at the right time and pay for it, which is probably the most significant design challenge for Lissy.

Since price improvements are paid to the miners, the miner is incentivized to run `closeMarket()` if it pays for itself. Efficient algorithms for miners to automatically find 'miner extractable value (MEV)' opportunities [13] is an open research problem. Even if someone else pays to close the market, MEV smooths out some market functionality. Assume several orders are submitted and then `closeMarket()`. A naive miner might order the `closeMarket()` before the submitted orders, effectively killing those orders and hurting its own potential profit. MEV encourages miners to make sure a profitable `closeMarket()` in the mempool executes within its current block (to claim the reward for itself) and that it runs after other orders in the mempool to maximize its profit.

Without MEV, markets should 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.

Another option is to have traders in the next call market pay to incrementally close the current market. For example, each order in the next market needs to pay to execute the next x orders in the current market until the order book is empty. This has two issues: first, amortizing the cost of closing the market amongst the early traders of the new market disincentives trading early in 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).

[9] 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.

A closely related option 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 sender of the first `closeMarket()` function to be confirmed receives the bounty. This is still 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. 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 called *meta-transactions*). This creates a regulatory hook.

The final option is to rely on an interested third party (such as the token issuer for a given market) to always close the market, or occasionally bailout the market when one of the above mechanisms fails. An external service like Ethereum Alarm Clock[11] (which also creates a regulatory hook) can be used to schedule regular `closeMarket()` calls.

6.4 Collateralization Options

In Lissy, both the tokens and ETH that a trader wants to potentially use in the order book are preloaded into the contract. We discuss some alternative designs in Appendix C.

7 Concluding Remarks

Imagine you have just launched a token on Ethereum. Now you 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.

Acknowledgements. The authors thank 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).

Acknowledgements. The authors thank 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).

References

1. M. Aquilina, E. B. Budish, and P. O'Neill. Quantifying the high-frequency trading "arms race": A simple new methodology and estimates. *Chicago Booth Research Paper*, (20-16), 2020.

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

2. E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO*, 2019.
3. E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.
4. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO*, 2018.
5. J. Bonneau, J. Clark, and S. Goldfeder. On bitcoin as a public randomness source. `https://eprint.iacr.org/2015/1015.pdf`, 2015. Accessed: 2015-10-25.
6. R. Brandom. This princeton professor is building a bitcoin-inspired prediction market, Nov 2013.
7. E. Budish, P. Cramton, and J. Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.
8. J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, Baltimore, MD, Aug. 2018. USENIX Association.
9. B. Bünz, S. Goldfeder, and J. Bonneau. Proofs-of-delay and randomness beacons in ethereum. In *IEEE S&B*, 2017.
10. J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. Mpc joins the dark side. In *ASIACCS*, pages 148–159, 2019.
11. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE EuroS&P*, pages 185–200. IEEE, 2019.
12. J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, and A. Narayanan. On decentralizing prediction markets and order books. In *WEIS*, 2014.
13. P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. In *IEEE Symposium on Security and Privacy*, 2020.
14. S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. In *WTSC*, pages 170–189. Springer, 2019.
15. E. Félez-Viñas and B. Hagströmer. Do volatility extensions improve the quality of closing call auctions? *Financial Review*, 56(3):385–406, 2021.
16. H. S. Galal and A. M. Youssef. Publicly verifiable and secrecy preserving periodic auctions. In *WTSC*. Springer, 2021.
17. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, 2013.
18. L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. Sok: Layer-two blockchain protocols. In *Financial Cryptography*, pages 201–226. Springer, 2020.
19. L. Harris. *Trading and exchanges: market microstructure for practitioners*. Oxford, 2003.
20. P. Hillion and M. Suominen. The manipulation of closing prices. *Journal of Financial Markets*, 7(4):351–375, 2004.
21. H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*, pages 1353–1370, 2018.
22. M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. Order-fairness for byzantine consensus. In *CRYPTO*, pages 451–480. Springer, 2020.
23. R. Khalil, A. Gervais, and G. Felley. Tex-a securely scalable trustless exchange. *IACR Cryptol. ePrint Arch.*, 2019:265, 2019.
24. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, pages 1–19, 2019.
25. K. Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *ACM AFT*, 2020.
26. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association.
27. F. Massacci, C. N. Ngo, J. Nie, D. Venturi, and J. Williams. Futuresmex: secure, distributed futures market exchange. In *IEEE Symposium on Security and Privacy*, pages 335–353. IEEE, 2018.
28. V. Mavroudis and H. Melton. Libra: Fair order-matching for electronic financial exchanges. In *ACM AFT*, 2019.
29. A. Norry. The history of the mt gox hack: Bitcoin's biggest heist. `https://blockonomi.com/mt-gox-hack/`, June 2019. (Accessed on 12/31/2019).

30. M. S. Pagano and R. A. Schwartz. A closing call's impact on market quality at euronext paris. *Journal of Financial Economics*, 68(3):439–484, 2003.
31. A. Park. The conceptual flaws of constant product automated market making. *Available at SSRN 3805750*, 2021.
32. H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security and Privacy*, 2021.
33. Securities and E. B. of India. Sebi | order in the matter of nse colocation. `https://www.sebi.gov.in/enforcement/orders/apr-2019/order-in-the-matter-of-nse-colocation_42880.html`, 2019. (Accessed on 11/11/2019).
34. C. Signer. Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science, 2018.
35. T. Suite. Ganache. `https://www.trufflesuite.com/ganache`, May 2021. (Accessed on 05/26/2021).
36. T. Suite. Truffle. `https://www.trufflesuite.com/docs/truffle/overview`, May 2021. (Accessed on 05/26/2021).
37. C. Thorpe and D. C. Parkes. Cryptographic securities exchanges. In *Financial Cryptography*, 2007.
38. C. Thorpe and S. R. Willis. Cryptographic rule-based trading. In *Financial Cryptography*, 2012.
39. W. Yuen, P. Syverson, Z. Liu, and C. Thorpe. Intention-disguised algorithmic trading. In *Financial Cryptography*, 2010.
40. L. Zhao, J. I. Choi, D. Demirag, K. R. B. Butler, M. Mannan, E. Ayday, and J. Clark. One-time programs made practical. In *Financial Cryptography*, 2019.
41. L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais. High-frequency trading on decentralized on-chain exchanges. In *IEEE Symposium on Security and Privacy*, 2021.

A Additional background

A.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.

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.

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 with a non-zero value 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 [34]. So in order to account for refunds in Table 3, we calculate them manually. First, we determine exactly how many storage slots are being cleared or how many smart contracts are being destroyed, then we multiply these numbers by 24,000 or 15,000 respectively.

Optimistic Roll-Ups. *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) [18]. 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 [21]). 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 [3,17] and variants [2]). 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 [11]).[12] 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.

[12] The TEE-based approach is mired by recent attacks on SGX [24,26,8,32], however these attacks do not necessarily apply to the specifics of how SGX is used here, and safer TEE technologies like Intel TXT (*cf.* [40]) can be substituted.

Arbitrum and Ethereum Optimism are the two prominent deployments of an optimistic roll-up. Arbitrum uses a multi-round dispute process that results in very minimal L1 gas costs to resolve a dispute. Specifically, if a dispute over a transaction arrises, the L1 cost of resolving the dispute is a small fraction of the cost of executing the transaction itself (whereas in Optimism, the dispute resolution cost is essentially the same as executing the transaction).

Figure 3 shows how traders interact with Lissy on Arbitrum. First, a trader sends a `depositETH` transaction on Ethereum to the Inbox contract to deposit X amount of ETH to the Arbitrum chain. Once the transaction is confirmed, X amount of ETH will be credited to the trader's address on the Arbitrum chain. Trader can now interact with Lissy and execute its functions by sending the instruction and data required for those executions to either (1) the Arbitrum regular Inbox on Ethereum, or (2) the sequencer. In our example, trader uses the regular Inbox to execute `depositEther()` and the sequencer to execute `submitBid()` from Lissy that lives entirely on Arbitrum chain. Accordingly, trader deposits ETH to Lissy smart contract by sending the instruction and data for executing the `depositEther()` to the Arbitrum Inbox contract that lives on Ethereum. A validator fetches this transaction from the Inbox, executes it, and asserts the result to ArbOS. Next, trader sends the instruction and data for execution of `submitBid()` to the sequencer. The sequencer then inserts this message into the Inbox that it owns. This Inbox contract has the same interface as the regular Inbox contract, however, it is *owned* by the sequencer. A validator sees the transaction in the sequencer Inbox of the bridge, executes it, and asserts the result to ArbOS. Periodically, the entire state of ArbOS is committed back to Ethereum.

Our Lissy variant is not the first roll-up-based order book. Loopring 3.0[13] 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.

A.2 Trade Execution Systems

Centralized 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.,* Binance, 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 [29], QuadrigaCX [33], and many others) have become a common source of lost funds for users, while accusations of unfair trade execution have been leveled 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 in 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). 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

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

Time	Trader	Order Type	Order Price	Volume
09:10	Mehdi	Ask	10.18	4
09:12	Avni	Bid	12	3
09:15	Kritee	Bid	13	3
09:18	Bob	Bid	12.15	1
09:26	Navjot	Ask	10.15	4
09:30	Alice	Ask	10	1

Table 8: Example orders that are submitted to a call market.

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 these 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 fueled a resurgence in on-chain exchange but through a quote-driven design rather than an order-driven one. Automated market makers (*e.g.,* Uniswap v3) have all the trust advantages of an on-chain order book, plus they are relatively more efficient. 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 favorable 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 functionalities. 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 blind spots (discussed above in Appendix A.1).

A.3 Call Markets

Assume traders submit their orders in Table 8 to a call market when it is open. In the following, we explain how these orders are executed:

- The call market first matches Alice's ask order to sell 1 at 10 with Avni's bid order to buy 3 at 12. Trade occurs at the price Alice asks for; 10, and 2 will be given to the miner as a price improvement. This trade fills Alice's order and leaves Avni with a remainder of 2 to buy at 12.

- Next, the call market matches Avni's remainder of 2 with the next highest priority ask order in the list which is Navjot's order to sell 4 at 10.15. Trade occurs at 10.15 and 1.85 will be given to the miner as a price improvement. This trade fills the remainder of Avni's bid order and leaves Navjot with a remainder of 2 to sell at 10.15.
- The market now matches the next highest bid order in the list, Bob's bid order to buy 1 at 12.15, with the remainder of Navjot's ask order to sell 2 at 10.15. Trader occurs at 10.15 and 2 will be given to miner as a price improvement. This trade fills Bob's bid order and leaves Navjot with a remainder of 1 to sell at 10.15.
- Next, the market matches Kritee's bid order to buy 3 at 13 with the remainder of Navjot's ask order to sell 1 at 10.15. Trade occurs at 10.15 and 2.85 will be given to miner as a price improvement. This trade fills Navjot's order and leaves Kritee with a remainder of 2 to buy at 13.
- The market then matches Mehdi's ask order to sell 4 at 10.18 with the remainder of Kritee's bid order to buy 2 at 13. Trade occurs at 10.18 and 2.82 is given to miner as a price improvement. This trade fills Kritee's order and leaves Mehdi with a remainder of 2 to sell at 10.18 unfilled.

B Cleaning-Up Revisited: Clearing Mappings

Beyond the cleaning up issues with priority queues in Section 3.2, 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.

C Collateralization Options in Call Markets

in Lissy, both the tokens and ETH that a trader wants to potentially use in the order book are preloaded 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.

D Market Clearing Prices

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.

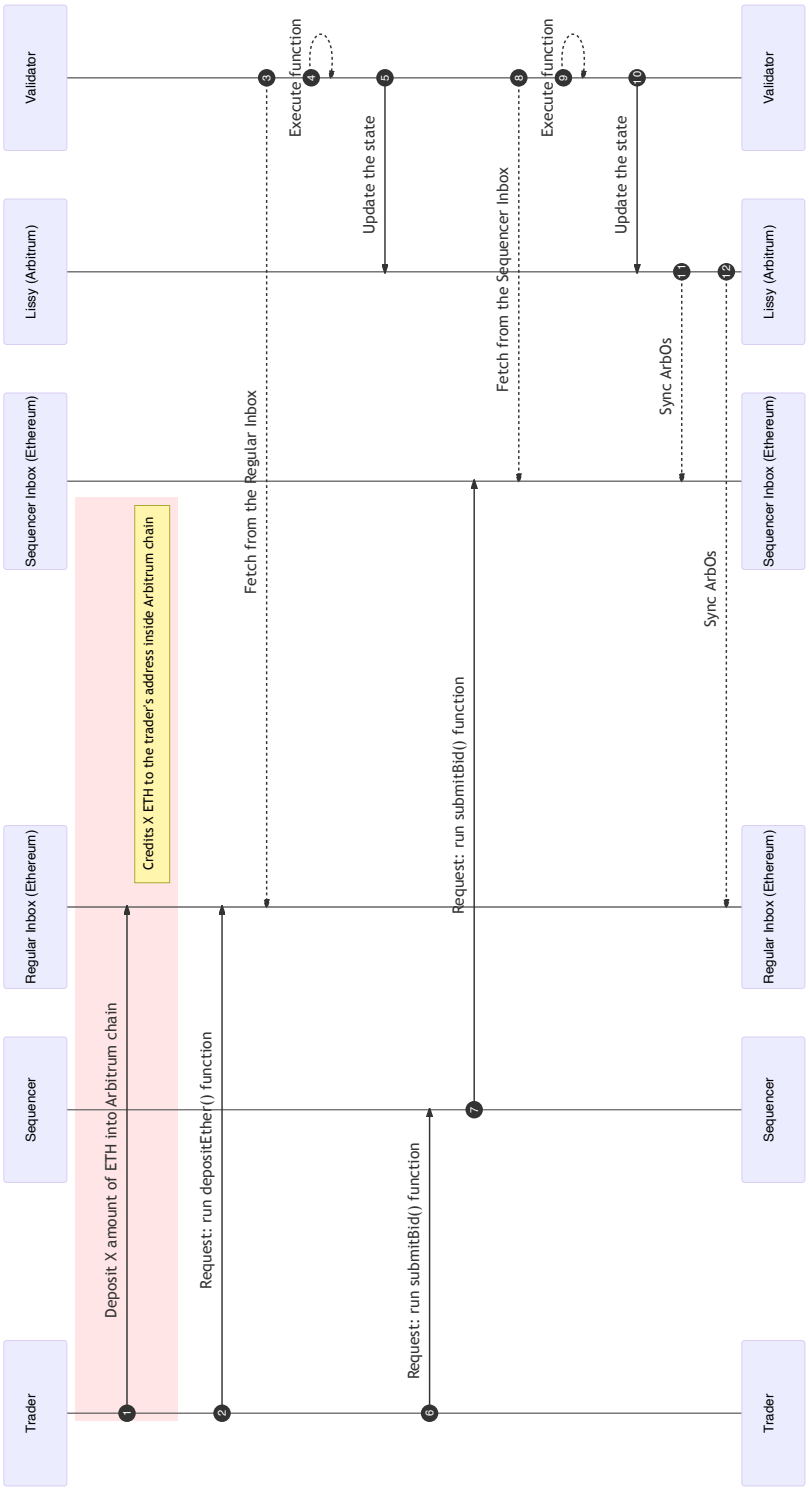

Fig. 3: Overview of Lissy on Arbitrum.