TradeIn: Revisiting fully on-chain exchanges

No Institute Given

Abstract. Abstract goes here

1 Introductory Remarks

Blockchain is a type of distributed database (or ledger) that is open to anyone who wants to participate, robust against a wide range of faulty and malicious behaviours, and runs without anyone in charge. When a participant looks at her local copy of the ledger, she is assured that (i) anyone has the exact same records and (ii) each record was validated by the "majority" (technically, a computational majority) of participants before it was written into the ledger. Blockchain technology has rapidly gained global interest and become one of the most significant technologies in recent years. It was first introduced in 2008 by Satoshi Nakamoto as an underlying technology of Bitcoin, a peer to peer electronic cash system [7], whose currency reached a market capitalization of $137 billion as of January 2020. In 2014, a new blockchain based application known as Ethereum was introduced by Buterin [2]. By implementing a decentralized virtual machine, Ethereum virtual machine (EVM), Ethereum allows network users to execute programmable smart contracts on it. So developers are now able to build decentralized applications (DApps) that are executed correctly according to the consensus rules of the Ethereum network.

Contributions.

Our design Tradine is intended as a simple module that developers can modify as they choose. By adding Tradine to an ERC20 (or similar) token contract, everything that is needed for buying/selling tokens is immediately available on chain without involving any third party exchanges. The main advantage of blockchain is low barriers of entry and this further lowers the barrier.

2 Preliminaries

Strong interest in blockchain and distributed ledger technologies has led into the high trading volume of digital assets and emergence of fast growing cryptocurrency exchanges all over the world. These crypto exchanges are similar to stock exchanges but, as the name suggests, facilitate the trading process of different cryptocurrency tokens. Basically, such exchanges provide a trading venue where market participants can buy and/or sell cryptocurrency assets among one another. In this section, we describe two types of crypto exchanges.

2.1 Blockchain

- **Transparency and Front-running.** Blockchains (in general) are transparent — transactions are first relay around the network before being selected by a miner, validated, and included in a block. This allows malicious nodes/miners to observe and front-run the transactions by dropping them or attempting to insert their own transactions before the observed transactions [5]. In the context of a DEX, front-running attacks enable malicious nodes to have their orders executed before others and benefit from it.
- **Speed.** Blockchains are not fast architectures — speed is one of the factors that diminishes the real world potential of blockchains. Blockchains are slow-moving as it takes block interval (*i.e.,* time it takes to mine a block) for transactions to be confirmed and included in valid blocks. So it is difficult to push a stock market on the blockchain as stock markets move extremely fast. High-frequency trading (HFT) is a trading method example that uses active computers to process a large number of trades in fractions of a second. In Ethereum 1.0 blocks are produced at intervals every 10 to19 seconds, note that this is not yet suitable for time sensitive transactions. However, capabilities of blockchains will be further enhanced ad they will become more efficient over time; Ethereum 2.0 has a shorter block time of 12 seconds.
- **Gas Parameter in Ethereum Transactions.**
  Gas is the Ethereum's pseudo-currency and a key variable for the execution of smart contracts. DApps written in high-level programming languages are compiled and translated into a compact representation (called 'bytecode') to be further executed on the Ethereum virtual machine or EVM. Each of these opcodes has a fixed amount of gas assigned and is a measure of computational effort. Separating ETH from gas prevents transactions cost from being too expensive as they are less impacted by ETH price fluctuations. Gas unit does not a have a monetary value and as mentioned, it only measures the computational work undertaken by miners. To pay miners, Ethereum introduces a `gas price` — small denomination of ether called **Gwei** attached to each gas unit. Essentially, gas price indicates how much users are willing to pay per unit of gas, clearly the higher gas price leads to the faster execution of a transaction. Miners also need to know the total amount of computational work a user is requesting, called the `gas limit`. `Gas limit` is a parameter that limits the amount of gas users would spend in a transaction. It protects users from spending unlimited ETH on their transactions and must be set carefully. If the `gas limit` is too low, the transaction will exceed the limit, all operations will be reverted while the user must pay for the computational work performed by the miner. By default the gas limit for an Ethereum transaction is 21,000 gas. An Ethereum block also has a `block gas limit` field which is set by the Ethereum miners and indicates the maximum amount of gas all the transactions in that block are allowed to consume, the Ethereum `block gas limit` is currently 11,741,495. [1]

[1]

Gas Refunds.

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

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

SSTORE. This operation clears the Ethereum storage and has15,000 gas refunds.
Note that in order to urge miners to process smart contracts with refunds, the accumulated gas refund can never exceed half the gas used up during computation [12]. So at the end of a successful transaction, the amount of gas in the refund counter (capped at half the net gas used) is returned to the caller. At the time of this writing, Ethereum transaction receipts only account for the `gasUsed`, which is the total amount of gas units spent during a transaction, and we are not able to obtain the value of the EVM's refund counter from inside the EVM [11]. So in order to account for refunds, we decide to calculate them manually; first we figure out exactly how much storage is being cleared or how many smart contracts are being destroyed, then we multiply these numbers by 24,000 and 15,000 respectively.
Based on these issues blockchains cannot solve this particular problem. Many stock exchanges that currently operate on blockchains tackle this issue by using a private network (where not every network participant can be a node or miner) together with a central time-server that establishes the time priority. We propose a different method of implementing a fully decentralized exchange using an alternative data structure known as *a call market* or *frequent batch auction* instead of a time-sensitive order book [4].

2.2 Exchanges

Centralized Exchanges (CEX) This group of exchanges operate in a traditional manner by providing a centralized platform controlled by exchange operators. Essentially, the exchange acts as a trusted party whom the market participants must trust to handle their assets. Traders transfer the ownership of their assets to the exchange who safeguards customer funds using different methods. Centralized exchanges ease trading process of digital assets as they do not place the burden of safeguarding of the assets onto individualst, however, traders must

https://ethstats.net/

fully trust the exchange operator in taking custody of their assets (*e.g.,* Coinbase). [2] Similar to any centralized platform these exchanges are not completely immune to malicious activities. For example in February 2014, a famous centralized Tokyo-based Bitcoin exchange called MtGox was hacked which led to loss of 650,000 Bitcoins [8]. Also, a malicious exchange operator can simply steal users' funds; in February 2019, QuadrigaCX, Canada's well-known centralized cryptocurrency exchange, claimed that it cannot access to $190-million worth of customers' funds [10].

Decentralized Exchanges (DEX) Unlike centralized exchanges, DEXes facilitate a secure automated peer-to-peer trading process on blockchains (*e.g.,* Ethereum) for market participants with no third parties involved, hence solving the issues that are inherent in centralized exchanges. This group of exchanges provide a better security and privacy by enabling traders to remain fully authoritative over their funds and personal data. With rising of Decentralized Finance (DeFi) or Open Finance movements, the number of DEX platforms on Ethereum have recently exploded and large volume of trading now occurs on DeFi exchanges (*e.g.,* Uniswap). [3]

However due to the high cost and technical limitations introduced by blockchain technologies, building a fully decentralized exchange has remained a major challenge. In the following section, we explore the idea of building a decentralized exchange on Ethereum.

2.3 Order Books

Order books are electronic data structures that maintain lists of bid and ask orders for various assets (*e.g.,* currencies, stocks, bonds *etc.*) in specific markets. The most common version of order books is a *double auction*, where market participants submit their bid and ask orders and the market clearing price will be calculated as the average between the best bid and ask prices. Order books often sort orders based on their price and submission time, this order allocation technique is called price-time priority [9] where orders are prioritized from highest price to lowest and given any two orders with the same price, they will be sorted based on their submission timestamps.

Looking more closely, order books are rather electronic ledgers that get updated over time. Given the definition of the Ethereum blockchain, that is a distributed ledger, the idea of implementing an order book in the form of a smart contract will seemingly resolve the existing issues with centralized exchanges (see Section 2.2). However, this design is not feasible due to some technical limitations that exist within blockchains:

- **Speed.** All blockchains are slow-moving as it takes block intervals for transactions to get confirmed. On the other hand stock markets move very fast;

[2] https://www.coinbase.com
[3] https://uniswap.org

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

Table 1: Operations for a generic Priority Queue.

high-frequency trading (HFT) is a trading method example that uses active computers to process a large number of trades in fractions of a second.

- **Front-running and Censorship.** Traders must broadcast their order messages to the network before they can be placed in the block (*i.e.,* order book). In such scenario, privileged nodes and miners can front-run transactions or completely drop (censor) those that are competing with their own [5]. Also, miners could almost control how the order book gets updated by inserting their own order messages after viewing all other orders and before mining the block .
- **Enforcing Time.** There is no notion of time on blockchains.I if two orders arrive at the same price, in the normal world whoever is the first wins. However, we cannot enforce the priority of orders in the blockchain space; if Alice is more well-connected to the network her order gets broadcast better and executed first although she sends it after Bob.

3 Priority Queue

In designing Tradine, performance within Ethereum's limited gas model became the main bottleneck. Within a call market, recording trades and accessing them to close the market is the most time consuming step. For this reason, we decided to do a deep study of the best data-structure for this specific task. While data structures are well studied for many languages, Solidity/EVM have their own quirks (*e.g.,* gas refunds, a relatively cheap mapping data structure, and no actual support for object oriented programming) that make them difficult to assess which will perform best without actually deploying and evaluating each variant.

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

There are numerous ways of implementing a PQ. A PQ has an underlying list—common options include a static array, dynamic array, and linked list. The most expensive operation is keeping the data sorted—common options include

(i) sorting during each enqueue, (ii) sorting for each dequeue, or (iii) splitting the difference by using a heap as the underlying data structure. Respectively, the time complexities are (i) linear enqueue and constant dequeue, (ii) constant enqueue and linear dequeue, and (iii) logarithmic enqueue and logarithmic dequeue. This said, Solidity/EVM has its own specific considerations to take into account. As closing the market is very expensive with any PQ, we rule out using (ii) as fully sorting while dequeuing would be prohibitive. We experiment with the following 5 options for (i) and (iii):

1. **Heap with Dynamic Array.** A heap is a type of binary tree data structure that comes in two forms of a (i) Max-Heap and (ii) Min-Heap. All the nodes of tree are in a specific order and the root of the tree always represents the highest priority item of the data structure (the largest and smallest values in the Max-Heap and Min-Heap respectively). We implemented a priority queue with a heap that stores its data in a dynamically-sized array.
2. **Heap with Static Array.** A heap can be also represented by a Solidity storage array in which the storage is statically allocated. To do this, we pass the required size of the array as a constructor parameter to the priority queue smart contract.
3. **Heap with Mapping.** In the above implementations, the heap stores the entire order (as a struct) in the heap. In this variant, we store the order struct in a Solidity mapping and store the mapping's key in the heap.
4. **Linked List.** In this variant, we insert a new element into its correct position (based on its price) when running enqueue. The PQ itself stores elements in a linked list (enabling us to efficiently insert a new element between two existing elements). Solidity is described as object-oriented but the equivalent of an object is an entire smart contract. Therefore an object-oriented linked list must either (i) create each node in the list as a struct—but this is not possible as Solidity does not support recursive structs—or (ii) make every node in the list its own contract. The latter option seems wasteful and unusual, but we try it out anyways. Thus each node is its own contract and contains the order data and a pointer to the address of the next contract in the list.
5. **Linked List with Mapping.** Finally, we try a variant of a linked list using a Solidity mapping. The value of the mapping is a struct with order data and the key of the next (and previous) node in the list. The contract stores the key of the first node (head) in the list.

3.1 Priority Queue Evaluation

Enqueue performance. We implemented, deployed, and tested each priority queue using Truffle and Ganache.[4] We tried a variety of tests (including testing the full call market with each varient) with consistent results in performance. A simple test to showcase the performance profile is shown in Figure 1. We

[4] Github: Link removed for anonymity.

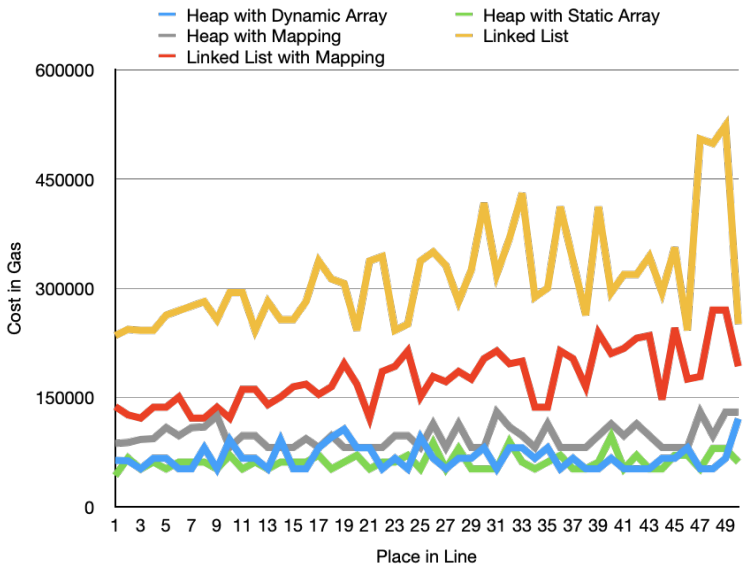

Fig. 1: Gas costs for enqueuing random items into five different priority queue variants. For the x-axis, a value of 10 indicates it is the 10th number entered in the priority queue. The y-axis is the cost of enqueue in gas.

simply enqueue 50 integers chosen at random from a fixed interval in each PQ variant. The bigger the PQ gets, the longer enqueue takes—a linear increase for the linked list variants, and logarithmic for the heap variants.

Dequeue performance. For each PQ variant storing 50 random integers, the `Dequeue()` function is iterated until the data structure is empty. The total gas cost for fully dequeuing the priority queue variants is outlined in Table 2. These tests are performed using the following Ethereum gas metrics: block gas limit =11,741,495 and 1 gas = 56 gwei.[5] Dequeuing removes data from the contract's storage. Recall this results in a gas refund. Based on our manual estimates (EVM does not expose the refund counter), every variant receives the maximum gas refund possible (*i.e.,* half the total cost of the transaction). In other words, each of them actually consumes twice the `gasUsed` amount in gas before the refund, however none of them are better or worst based on how much of a refund they generate.

Decision. Based on enqueuing, the heap variants are the cheapest in terms of gas, while based on dequeuing, the link list variants are cheapest. This is in

[5] https://ethstats.net/ *July*2020 :

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

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

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

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

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

3.2 Cost/Benefit of cleaning up after yourself

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

Dequeuing in a linked list with mapping can be implemented in two ways. The simplest approach is process a node, update the head pointer, and leave the 'removed' node's data behind in the mapping untouched (where it will never be referenced again). Alternatively we can call `DELETE` on each mapping entry once we are done with a node in the PQ. Deleting a storage variable is identical to setting a non-zero variable to zero (`SSTORE` 0) that costs 20,000 gas but with 15,000 refunded—a net positive gas cost [12]. As it can be seen in the last two rows of Table 3, leaving the data on chain is cheaper then cleaning it up.

The lesson here is that gas refunds incentivize developers to clean up storage variables they will not use again, but it is highly contextual as to whether it will pay for itself. Further the cap on the maximum refund means that refunds are not fully received for large cleanup operations (however removing the cap impacts the miners' incentives to include the transaction). This is a complicated and under-explored area of Ethereum in the research literature. For our own work, we strive to be good citizens of Ethereum and cleanup to the extent that we can—thus all PQs in Table 2 implement some cleanup and we select Linked List with Mapping and `DELETE` for Tradine.

4 Call Market Design

Our design Tradine is intended as a module that developers can modify as they choose—thus we tried to simplify the design at every step to make it highly extensible but still functional without any extensions. A call market will open for a specified period of time during which it will accept orders. Orders are added to a PQ. Our vision (discussed below) is the market would be open for a very short period of time, close, and then reopen immediately (*e.g.,* every other block). This is akin to how a crossing network operates in traditional finance (*i.e.,* some explanation). We keep the design simple by not allowing cancellations. Because markets are relatively short-lived, cancelations require a second transaction (and front-running attacks apply to cancellation orders [5]), orders simply expire when the market call period ends.

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

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

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

Table 5: Performance of Tradine for each PQ variant.

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

Tradine is written in 225 lines (SLOC) of Solidity (including 3 external libraries),[6] a high level programming language that is syntactically similar to Java [1]. We tested it with the Mocha testing framework (tests included in the codebase), as deployed using Truffle on Ganache-CLI to obtain our performance metrics. Table 4 represents the call market's primary operations.

4.1 Performance

The main research question is how many orders can be processed in a single Ethereum transaction when the closing the call market, using Ethereum today. As our previous experiments indicated, the choice of priority queue (PQ) implementation is the main influence on performance (see Table 5). We implemented

[6] Github: linked removed for anonymity

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

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

Optimize for closing market, even at cost of higher submission.

5 Call Market Different Design and Implementation Details

5.1 Clearing Mappings

To maintain the collateral balance of each market participant, we use two Solidity type one-to-one mappings that map Ethereum addresses to 256 bits unsigned integers; `TotalBalance` and `UnavailableBalance`. Once market closes and orders are matched, the `UnavailableBalance` needs to be cleared. However, since it is not possible to delete the entire mapping without knowing the keys [7], clearing the `UnavailableBalance` mapping remains a challenging issue to solve. Here we provide a landscape of solutions for that.

- **Creating a New Mapping Every Time the Market Opens.** Instead of clearing the `UnavailableBalance` of traders at the end of a matching process, we could create a new mapping every time the market opens. Note that in this scheme traders can claim their funds back (using the `ClaimEther()` and/or `ClaimToken` methods) only when the market is closed (*i.e.,* the market is in the `Closed` state).

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

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

5.2 Closing and Reopening the Market

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

- **Enforcing Time on the Blockchain.**
- **Who Pays the Cost for Closing and Reopening the Market?**
  We think it is useful to explore the landscape of possible designs for closing the market.
  - **Miners Close and Reopen the Market.** The difference between the best bid and ask prices is called the *bid-ask spread*. In our design, when the trade occurs between the the highest bid (the highest amount a buyer is willing to pay for an asset) and lowest ask (the lowest amount a seller is willing to accept for an asset) orders, the bid-ask spread is paid to the miner. There are possibilities that (i) no trade occurs or (ii) the bid-ask spread is zero (*i.e.,* the best bid and best ask prices are identical). So there is enough economic incentive for the the miner to execute the `CloseMarket()` function and get refunded as the refund amount could be potentially higher than the bid-ask spread.
  - **Processing Orders in Groups.** Another solution pattern is to process certain number of bid and ask orders upon every execution of the `CloseMarket()` function rather than treating them as one substantial transaction. A market participant P_i would process n orders from the previous market (`CloseMarket(n)`) when sending new orders to the current market. This process continues until all the orders in the previous markets are processed.
  - **Using the Meta Transactions.** Meta transactions enable users to execute Ethereum functions without paying the gas. Rather than spending

gas, users sign their intended action using their private keys and broadcast it to the network with no cost. A third party process (*a relayer*) then crafts the actual transaction on user's behalf, sends the transaction to the Ethereum blockchain, and charges the base contract with the associated fees (see Figure 2). The required gas to pay for the `Match()` function could be collected as fees. So market participants are charged with certain amounts of fees every time they submit an order, these fees are accumulated in the CallMarket contract and will be used to pay for executing the `CloseMarket()` function.

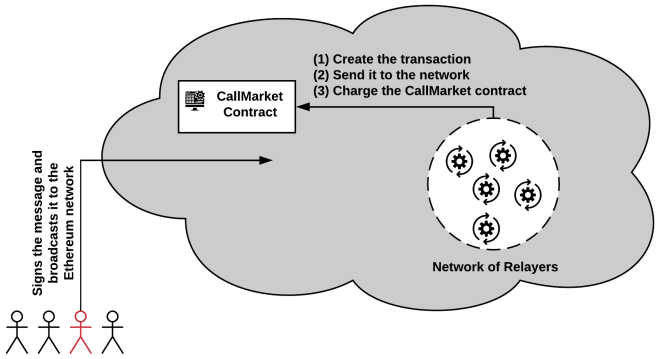

Fig. 2:

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

ticipants are incentivized to do so as they may receive more ethers than they have spent.

- **Using Rollups.** Rollup is a scaling method that moves the storage and computation of the smart contracts off-chain while maintaining the transaction data on the main chain as call-data. In this technique, any Ethereum user can act as a validator; they can execute the `CloseMarket()`function and only post the new state of the contract (the updated balance of traders) in the form of *assertions* to the main chain. Rollups improve scalability, provide faster and cheaper execution of the contracts, and eliminate the gas limit as the contract is no longer executed on-chain. In the followings we briefly discuss different rollup proposals and techniques. Each approach uses a different method to ensure correction of assertions:
  - **Non-interactive Rollups.** In this rollup technique, assertions are posted together with a validity proof that would be later used by validators to check if the `CloseMarket()` function has been executed correctly. ZK-Rollup scheme is one of the solutions that uses ZK-SNARKs to prove the validity of the assertions in zero-knowledge. ZK proofs are quick and cheap to verify but they are expensive and time consuming to generate. These proofs could be generated (i) for free or (ii) the CallMarket contract could collect proportional fees for every trade that is successfully executed.
  - **Optimistic Rollups.** In this scheme, assertions are assumed to be valid if there is no dispute posted about them with a certain window of time (a.k.a. "the challenge period'). Here, dispute resolution is a gas-costly method as the CallMarket contract would have to emulate the transaction on-chain to ensure the correctness the assertion. This scheme introduces a tradeoff between privacy and performance as all the assertions are publicly available and accessible. However, here the new state only reflects the updated balances of traders and no secret is involved.
  - **Multi-round Interactive Rollups.** In this design paradigm, *pending assertions* are posted on-chain and they are open to dispute. Once the challenge period is over and no challenge is submitted, the assertion is confirmed and the CallMarket contract transitions to the new state (*i.e.,* updates traders' balances). This scheme takes the overhead for the CallMarket contract to execute the `Close()` on-chain by using rounds to the dispute resolutions. The two parties (asserter and challenger) must run an interactive protocol and the CallMarket smart contract would have to act as a referee and decides which party's claim is true. Arbitrum is an example of multi-round interactive rollups that uses an efficient challenge-based protocol to penalize the dishonest parties [6].
- **Using Trusted Execution Environments.** Another way of achieving execution of the `CloseMarket()` function is incorporating the Ethereum blockchain into the Trusted Execution Environments (TEEs) and decou-

pling the contract execution from consensus mechanism. TEEs enable secure execution of applications in an isolated processing environment called the *enclave*. Here, the enclave could execute the `CloseMarket()` function off-chain in TEEs and publish an on-chain attestation Quote to the CallMarket contract. The contract then verifies the correctness of the Quote and if validated correctly, it transitions to the new state. Ekiden is an example that uses Intel SGX to solve the scalability and confidentiality issues with the smart contract execution [3]. A drawback of this scheme is in order to achieve confidentiality-preserving smart contracts we have to trust a trusted party in the form of the hardware manufacturer (*e.g.,* Intel).

5.3 Transferring the Collateral back to Traders.

6 Concluding Remarks

References

1. Ethereum development tutorial · ethereum/wiki wiki. `https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial`. (Accessed on 24/08/2020).
2. V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.
3. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
4. J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, and A. Narayanan. On decentralizing prediction markets and order books. In *Workshop on the Economics of Information Security, State College, Pennsylvania*, 2014.
5. S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. 2019.
6. H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.
7. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.
8. 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).
9. T. Preis. Price-time priority and pro rata matching in an order book model of financial markets. In *Econophysics of Order-driven Markets*, pages 65–72. Springer, 2011.
10. 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).
11. C. Signer. Gas cost analysis for ethereum smart contracts. Master's thesis, ETH Zurich, Department of Computer Science, 2018.
12. G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.