FRED: A Frontrunning Resistant Darkpool

Jacob Chudnovsky
jchudnov@stanford.edu

Henry Friedlander
hnf035@stanford.edu

Ajay Vasisht
avasisht@stanford.edu

Federico Zalcberg
fedez@stanford.edu

***Abstract*—Darkpools are exchanges, in which prior to matching, a counterparty does not need to publicly broadcast their trading intentions. This is useful for traders wishing to dump a large amount of an asset quickly. Centralized darkpools have been demonstrated to not be equitable for the trades since the darkpool operators place priority on their own trades rather than external trades. In this paper, we introduce the FRED system architecture to ensure that the darkpool mechanism treats trades according to equitable matching rules. We achieve this by publishing a BFT public ledger of hashed transactions.**

***Index Terms*—darkpool, HFT, HotStuff, BFT, Blockchains**

I. Introduction

In today's electronic-dominant trading world, at any given point, about 12-15% of all US trades [1] are done on a purposefully-opaque Dark Pool. These darkpools match buy and sell orders anonymously so that traders can prevent their orders from being known before execution, and hence protect the spread of their executions, especially for large block orders [2].

While a darkpool's anonymity solves these problems, they still suffer from one main flaw — namely, that they are operated by a single entity with complete anonymity and loose regulatory oversight [2]. This leads to the problem of today's darkpools: frontrunning by the darkpools themselves. Since the inner workings and state of a dark pool are totally opaque, there is no mechanism to prevent a darkpool operator from frontrunning orders. Any naive solution that attempts to resolve the frontrunning problem would inevitably release data about the orders in the darkpool, which would in turn, negate the very purpose of the darkpool (namely, by putting information out in the clear, it opens the way for High Frequency Trading (HFT) firms to frontrun the large block orders.)

In this paper, we introduce a novel approach to this problem. Chiefly, a distributed mechanism by which we can keep a centralized darkpool accountable (and thus honest). To this end, we construct a distributed ledger that every participating node in our system needs to curate according to some Byzantine fault tolerance (BFT) protocol. There are many algorithms proposed to solve this problem, including pBFT [4], Honey Badger BFT [10], Streamlet [5], among others. However, these consensus protocols all do not satisfy the reasonable assumption of partial synchrony and, perhaps even more importantly, scale super linearly in the number of nodes in the system. Since our system needs to be robust to many nodes executing consensus to serve an arbitrary number of trades, we need linear scaling in the number of nodes. We leverage HotStuff [12], a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model, as an append-only blockchain that stores hash commitments of orders in the darkpool, which has the property of linear scaling in the number of nodes.

In this paper, we provide implementation details for a basic darkpool that can be either honest or dishonest (toggle-able via API call) as well as a basic client that is using the darkpools, which executes the HotStuff consensus protocol. We claim that the proposed solution allows users to monitor and hold a centralized darkpool accountable in a decentralized byzantine fault tolerant manner. It prevents a darkpool from doing internal, opaque frontrunning without leaking any information about orders in the darkpool. As well it prevents any client nodes from adverserially manipulating the ledger history. The implementation for FReD can be found at https://github.com/avasisht23/cs244b_fp.

II. Related Work

There has been a thread of literature analyzing from the perspective of market design and how incentives could give rie to front running. The seminal paper [3] in this field which demonstrated the incentives that drove the industry to compete on speeds of microseconds. They argue that markets could be ordered in batches to combat front running. Follow up work [9] has documented the role and negative externalities that the increasing infiltration of High Frequency Trading and front running has had on markets.

In the cryptocurrencies markets, there recently has been a lot of interest in construction of distributed exchanges, which has the potential to address some of these frontrunning problems. 0x [11] and Loopring [7] provide interfaces for atomic asset swaps between untrusted parties. Also, decentralized finance has made used of automated market makers to enable uses to bootstrap liquidity from liquidity pool ratios. However, both of these approaches are not intrinsically front-

running resistant. To combat front-running, Clockwork employs timelock puzzles to allow exchanges to commit to processing an order before the trader is able to see other orders.

In many of these systems, there is a mempool where traders submit their trades to be picked up by miners. However, for many of these trades, traders are able to ping the darkpool and potentially go to another exchange before the market is able to globally adjust its prices. In the context, of decentralized finance, the negative externalities come from a trading strategy called Maximum Extractable Value (MEV) where traders are able to deterministically front-run trades by bribing miners to order their trades in an advantageous manner. There are services like flashbots [6], which obfuscates access to the pending transactions mitigating the effects of MEV but also front-running. In principle, this paper's project is similar to their approach though the domain and techniques applied are quite different.

III. Problem

At first glance, this appears to be a catch-22. That is, having a totally opaque darkpool to prevent frontrunning on the open market allows for the darkpool operator to itself frontrun without anyone else knowing. Any information that is revealed to prevent this darkpool from doing this will in turn open the door for the initial malicious HFT firms to frontrun.

To better see this, we consider an opaque Darkpool that makes buy and sell limit orders from customers. At any given time, only the darkpool knows what bid and are in the pool waiting to be matched (denoted by $B_1, B_2, ...$) and what the asks are (denoted by A_1,A_2,...). For this example, we assume we only provide coverage over one asset ($AAPL$).

1) Customer $Alice$ submits a limit bid order B_1 at price $\$100$ for asset $AAPL$ to the darkpool
2) Customer Bob submits a limit ask order A_1 at price $\$99.5$ for asset $APPL$ to the darkpool

In theory, the fair thing for the darkpool to do would be to match these two orders and for Alice and Bob to meet in the middle (since the bid is higher than the ask). However, what the darkpool could do is the following:

1) Customer $Alice$ submits a limit bid order B_1 at price $\$100$ for asset $AAPL$ to the darkpool
2) Customer Bob submits a limit ask order A_1 at price $\$99.5$ for asset $AAPL$ to the darkpool
3) The darkpool gets one of its traders to submit a limit ask order A_2 at price $\$100$ for asset $AAPL$
4) The darkpool gets one of its traders to submit a limit bid order B_2 at price $\$99.5$ for asset $AAPL$
5) B_1 is matched with A_2 at strike price $\$100$
6) A_1 is matched with B_2 at strike price $\$99.5$

Note that here, although $Alice$ and Bob are contently matched and both are within their limits, they got suboptimal pricing and the darkpool/trader gains 0.05 in arbitrage. Since this is all opaque, there is no way for $Alice$ and Bob to know that A_1 and B_1 should've been matched first before A_2 and B_2 were created. There is also no way of knowing that this arbitrage took place unfairly.

Another example of wrongdoing by the darkpool could be:

1) Customer $Alice$ submits a limit bid order B_1 at price $\$100$ for asset $AAPL$ to the darkpool
2) Customer Bob submits a limit ask order A_1 at price $\$100$ for asset $AAPL$ to the darkpool
3) Customer $Charlie$ submits limit ask order A_2 at price $\$100$ for asset $AAPL$ to the darkpool
4) B_1 is matched with A_2 at strike price $\$100$

Note that the darkpook didn't do any arbitrage or create any new orders, but what did happen is that it didn't match Bob's order with Alice's although Bob's came in first. The reason for this might be that Charlie is a bigger client or Bob pays more commissions, etc. This is unfair behavior that can not be proved by any actor other than the darkpool since no one has enough information about others, only about their own orders.

IV. Proposed Solution

These are precisely the problems that we seek out to solve. Namely, a mechanism by which all orders in the darkpool are kept a secret from outside HFT firms, but that is also protected against wrongdoing on the darkpool operator's behalf.

One solution might seek to have some kind of gossip mechanism in which $Alice$ and Bob tell each other what order they put in and what time they submitted each other. However, this would negate the entire purpose of the darkpool since Alice and Bob would have to reveal details about orders prior to executing (exposes them to being front run by High Frequency Trading Algorithms). We somehow need a trusted, non-centralized mechanism in which $Alice$ and Bob could prove their order timing safely without revealing anything about them.

A. Overview

To do this, we introduce a distributed additional component that will act as a source of truth in any debates. Specifically, we introduce a Byzantine Fault Tolerant, Append-Only ledger (in our case, the blockchain maintained by the Hotstuff protocol) that will contain the hash commitments of orders submitted to the darkpool. Note that we need this append-only ledger to not leak any data about any unexecuted orders

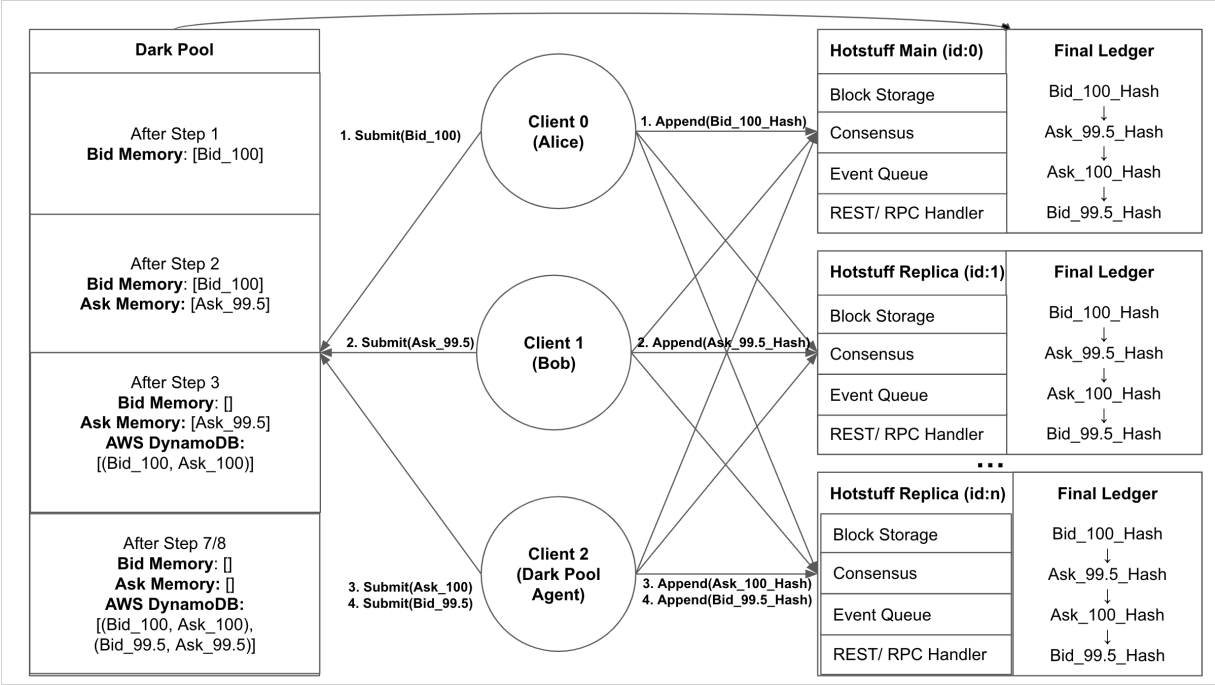

Fig. 1. System diagram for the darkpool solution setup.

(as to prevent front running).

Our new mechanism will consist of hashing an order before submitting it to a darkpool (using a Collision Resistant Hash Function), and appending this hash value to the append-only ledger, prior to sending the order to the darkpool. Our darkpool is still centralized, but when a customer sends an order to the darkpool, they also show that the hash of their order is in the ledger (which darkpool can calculate itself and verify). Then when the darkpool does a match, anyone who feels like they had a better match can verify and prove this. We have outlined an example system execution diagram in Figure 1 above. To better understand this, let's revisit our initial example:

1) Customer $Alice$ creates a limit bid order B_1 at price $\$100$ for asset $AAPL$ to the darkpool. She computes the hash of B_1 as h_{b1} and appends this to the distributed ledger. She waits for confirmations (for liveness), and then submits B_1 to the darkpool. The darkpool, if it wants, can verify that this is on the ledger by computing the hash itself and checking the ledger, although this isn't really necessary.
2) Customer Bob creates a limit ask order A_1 at price $\$99.5$ for asset $AAPL$ to the darkpool. He computes the hash of A_1 as h_{a1} and appends this to the distributed ledger. He waits for confirmations (for liveness), and then submits A_1 to the darkpool.
3) The darkpool gets one of its traders to submit a limit ask order A_2 at price $\$100$ for asset $AAPL$. Even if the trader computes the hash of A_2 as h_{a2} and appends it to the distributed ledger, it will still occur AFTER h_{b1} and h_{a1} on the chain.
4) The darkpool gets one of its traders to submit a limit bid order B_2 at price $\$99.5$ for asset $AAPL$. Again, even if the trader computes the hash of B_2 as h_{b2} and appends it to the distributed ledger, it will still occur AFTER h_{b1} and ch_{a1}.
5) B_1 is matched with A_2 at strike price $\$100$
6) A_1 is matched with B_2 at strike price $\$99.5$

Note that the orders that have been matched are publicly revealed after they are matched (part of settlement process in traditional finance). However, this time we can prove any wrongdoing. For example, when Bob sees that A_2 was matched with B_1 instead of his A_1, he can compute the hash of A_2 as h_{a2} and look for it on the ledger. When he sees that his h_{a1} occurs BEFORE h_{a2} on the ledger, he can very clearly prove that the darkpool was unfair. Likewise, Alice can do a similar thing with B_2 being matched with A_1 instead of her B_1.

B. Dark Pool Algorithm and Client Bid-Ask Submissions

The Centralized darkpool server has the following endpoints:

1) GET
    a) `/getState` - this returns a JSON dump of the entire state of the darkpool which includes pending BID / ASK orders

b) `/getNewClientId` - this returns the latest id (which is just a simple incremented counter) to be used as the newest client id. For any client to interact with the Dark Pool, they must receive an unused nonce from the Darkpool using `/getClientId`. Serving as their auth token, they append this client id when submitting an order through the `/sendOrder` endpoint.

2) POST

a) `/setFairness` - this sets the fairness of the darkpool. This coefficient is solely for internal testing purposes to demonstrate how a darkpool can be on a spectrum of "fairness". In production this would be permissioned only for admin.

b) `/sendOrder` - this sends a new pending order with `asset`, `limit price`, `side`, and `client id` details to the darkpool to be processed. It will do some basic input validation, then check if the order can be exactly matched to any other order on the other side of the order book based on limit prices. If so, the order is matched and stored as a bid-ask tuple in a public AWS DynamoDB table for Clients to reference in the future to validate if the darkpool had frontrunned this order. On the client side, this is called after sending the identical hashed order to the hotstuff ledger

The Centralized Dark Pool is a single server implemented in Node.js and Express. The Clients we used to test written in JavaScript, and make calls to the Centralized Dark Pool and the Replicated State Machines that forms consensus through Hotstuff.

C. Client Consensus Mechanism

The Client Consensus Mechanism can be used to verify that the darkpool operators aren't frontrunning orders. The client side algorithm is as follows:

Algorithm 1 Client procedure

procedure $(asset, limitPrice, side)$
  clientID := getClientId()
  hashedOrder := sha256(asset, limitPrice, side)] + clientID
  index := append(hashedOrder)
  send identical hashedOrder to darkpool
  dynDBTable = $DYNAMO\_DB()$
  while true **do**
    if dynDBTable[hashedOrder] exists **then**
      filledIndex := getIndex(hashedOrder+dynDBTable[key])
      if index $<$ filledIndex **then**
        FRONTRUNNING OCCURRED
      Exit
    sleep()

When a client submits an order to the Centralized Dark Pool through `/sendOrder`, they will also submit the SHA-256 hash of that order to the Distributed System of State Machines making up the Hotstuff Protocol through the `append` REST endpoint on all replicas. Through the Hotstuff Consensus, the order hash will be replicated onto multiple state machines that make the protocol, and avoiding Byzantine Fault Tolerance as well. New orders submitted to this distributed system will be appended to the ledger (chain of blocks) of order hashes. That way, if a client sees a new match come out of the darkpool where they for sure believe they should have been matched given they submitted their order side before the announced match, they can cross reference their order hash and the matched order hash to see which one came first, by calling `getIndex` on the Hotstuff protocol. If the index of their order hash indeed came before the matched order hash on the same side of the order book, then it's clear the darkpool had frontrunned the order.

It should be noted that the darkpool is also responsible for verifying that there is no bad client actors - that is, clients who submit to the darkpool but do not append to the distributed ledger through a `getIndex` call to the replicated state machines. In this event, the client to matched to the bad actor client could prove frontrunning even in the case that the darkpool acted totally fairly. Thus, a darkpool should probably have a way to verify that the orders they are processing have confirmed entries in the distributed ledger. Although this isn't really needed for our mechanism, it protects the darkpool against false fraud accusations.

V. Byzantine Fault Tolerant Ledger

The Replicated State Machines are each servers written in C++, and each contain a chained ledger of blocks representing data around submitted proposals that have undergone HotStuff Consensus.

A. Data Structures Within HotStuff

HotStuff [12] has a couple concepts central to its implementation. Many are standard in the literature on Byzantine fault tolerance.

Quorum Certificates. A Quorum Certificate over a tuple (`type, viewNumber, node`) aggregates signatures for the $(n - f)$ replicas.

Messages. A message m controls four state variables. The first, `m.type` $\in$ { `NEW-VIEW`, `PREPARE`, `PRE-COMMIT`, `COMMIT`, `DECIDE`}, keeps track of the current phase the message is referring to. The second, $m.node$, contains a proposed node, which is the leaf node of a proposed branch. The third, $m.justify$, is used by the leader to carry the QC for different phases, and replicas use the variable to carry the highest $prepareQC$. And the replicas hold the $m.partialSig$ over the previous variables.

Tree and branches. Each node hold a history of the commands. During the protocol, a replica delivers a message only after the branch led by the node is in its local tree. These replicas recover from falling behind through acquiring information from nodes who are farther ahead.

Concretely, each node keeps track of the following state variables

- $V[\cdot]$: mapping from a node to its votes.
- $vheight$: height of the last voted node.
- b_{lock}: last executed node.
- qc_{high}: the highest known QC kept by a Pacemaker
- b_{leaf}: leaf node kept by a Pacemaker

B. Algorithmic Overview

Since the service of an exchange requires the infrastructure to handle many traders and many trades, our approach to solving the problem of constructing a Byzantine fault tolerance needs to scale up to many states. We chose to integrate and implement the HotStuff BFT protocol. In this section, we will go over the various steps of this protocol. In the original paper, the authors introduce three variants to the protocol. We will discuss the Chained HotStuff implementation.

There are 5 steps to the protocol.

The first is the prepare phase. The protocol for a new leader starts by collecting `NEW-VIEW` messages from the $(n-f)$ replicas. The `NEW-VIEW` message is sent by a replica as it transitions into $viewNumber$ and carries the highest $prepareQC$ that the replica received. The leader processes these messages in order to select a branch that has the highest preceding view in which a $prepareQC$ was formed.

The leader processes these messages to find a branch with the highest preceding view where a $prepareQC$ was constructed. Then, the leader selects the $prepareQC$ with the highest view to make the $highQC$ out of all the `NEW-VIEW` messages. And since there aren't any higher quorum certificates among the $(n-f)$ replicas and thus no higher view could have reached a commit decision, the branch with $highQC.node$ must be safe. Now that the leader knows that the tail of $highQC.node$ is safe, it calls `createLeaf` to extend this safe tail with his new proposal. Finally, when the nodes receive the `PREPARE` message, a replica r uses the `safeNode` predicate to determine whether or not to accept this message. If it is accepted, the replica sends a `PREPARE` vote with a partial signature to the leader.

This `safeNode` predicate examines the proposal message m carrying a quorum certificate justification $m.justify$ and determines whether $m.node$ is safe to accept. There are two conditions for a node to determine whether a predicate is true. Either the node personally has enough information to verify if it is true, which concretely means that the branch of $m.node$ extends from the currently locked node $lockedQC.node$. Otherwise, the node might have missed some information and would be able to verify its safety through the liveness rule. That is, the replica will accept m if $m.justify$ has a higher view than the current $lockedQC$.

The next phase of the protocol is the `PRE-COMMIT` phase. The receipt of the $(n-f)$ `PREPARE` votes for the current proposal $curProposal$, it combines them into a $prepareQC$. The leader then broadcasts $prepareQC$ in the `PRE-COMMIT` messages. A replica responds to the leader with `PRE-COMMIT` messages. A replica responds to the leader with a `PRE-COMMIT` vote after signing the proposal. Again, in a similar step, the algorithm executes the `COMMIT` phase. When the leader receives $(n-f)$ `PRE-COMMIT` votes, it combines them into a $precommitQC$ and broadcasts it in the `COMMIT` messages. The important difference with this step is that the replicas all become locked on their $precommitQC$ vote.

Finally, in the `DECIDE` phase, the leader receives $(n-f)$ `COMMIT` votes, and it combines them into a $commitQC$ then sending a `DECIDE` message to all the other replicas. When a replica receives this message, it considers and executes the proposal in the $commitQC$ and executes the commands in the committed branch. Closing this round, the replicas increment the $viewNumber$ and start the next view.

C. Chained HotStuff

An astute reader might notice that each of the three phases of the HotStuff have similar operating procedures aside from the naming details of the vote collection. The HotStuff protocol leverages this phase similarity to allow for the bundling of messages and pipelining of decisions.

In the Chained HotStuff, the votes over a `PREPARE` phase are collected in a view by the leader into a $genericQC$. However, rather than waiting for the entire round to end to initiate the next proposal, the next leader is able to bootstrap the `PRE-COMMIT` phase using the next `PREPARE` phase. That is, the `PREPARE` phase for the view $v+1$ simultaneously serves as the `PRE-COMMIT` for the view v. Similarly, the `COMMIT` phase for the view v and the `PRE-COMMIT` phase for the view $v+1$ become bundled with the `PREPARE` phase for the view v.

Standard to the literature, the hotstuff protocol uses a Pacemaker to guarantee progress after the GST. The designed pacemaker achieves this through two avenues. First, the pacemaker ensures that all honest replicas and a unique leader are bought into a a common height for a sufficiently long period of time. Then the Pacemaker provides a way to choose a proposal that will be sup-

ported by the honest replicas. This feature is especially important in the context of our darkpool implementation.

D. HotStuff API

1) GET
   a) `/index?order=${hashedOrder}` - this returns the index of the hashedOrder in a particular node ledger in hotstuff. The client floods all nodes with getIndex to get the correct index.
2) POST
   a) `/append` - this appends the hashedOrder to the end of the ledger of a particular node in hotstuff. The client floods all nodes with append.

VI. Future Steps

A. Transaction Costs

As common in various decentralized ecosystem, we will need to add a transaction fee structure for the ledger. The purpose of this is two-fold:

1) Since each client will have to pay a small fee prior to appending a hash to the darkpool, it prevents them from submitting an arbitrarily long amount of fake hashes. Consider the case where a client submits hashes to the ledger for the top 10,000 most common orders they might want to place in the future. In the future, when they do want to place one of these orders, they can unfairly claim a much earlier spot in the darkpool then when they actually submitted their order. By having to pay a fee every time they want to append to the darkpool, clients will be disincentivize from this behavior.
2) The transaction fees will be divided amongst the hosts operating the HotStuff nodes. This provides a financial incentives for them to spin up and run these nodes, which are the backbone of this entire project.

B. Penalization of an dishonest Darkpool

The whole premise of the approach described herein is to provably and safely discover when a darkpool is not being totally fair to its clients. As described in section IV, once we discover that a darkpool is being dishonest, we will need some way of penalizing it, both to incentives fair darkpool behavior and to provide confidence to consumers (clients). Some potential avenues to pursue here include:

1) The darkpool puts up a collateral to a trusted third-party (or even a Decentralized autonomous organization) that slashes the collateral whenever a proof of unfairness is submitted
2) The proofs of unfairness are just submitted to regulartory agencies (such as the SEC) which can fine and handle it directly with the force of the Government
3) The darkpool itself is implemented as smart contract that has a function for taking in proofs of unfairness and will payout upon receiving a proof of unfairness. Although we concede that this might be better in theory than in practice, as it would lead to a variety of potential issues.
4) We create another decentralized consensus entity that maintains a whitelist of trusted darkpools. When a proof of unfairness is submitted, the darkpool is removed from this list. Traders can check this list prior to routing their orders (similar to how Google's Transparency Report works [8])

VII. Conclusion

Modern day darkpools have a purpose in financial markets, but with a lack of regulatory oversight, they can fall victim to operator frontrunning. In hopes of providing at-trade-time transparency, we're leveraging a Byzantine Fault Tolerant consensus mechanism across replicated state machines to programmatically verify the absence of frontrunning in a darkpool, in which each party keeps each other accountable via a publicly accessible append-only ledger.

We achieve this by ensuring that before clients submit their orders to a darkpool, the submit a hash of their order to an append-only distributed ledger. Clients can verify that the order they matched to is the earliest in the append-only ledger, or otherwise announce the darkpool's unfair frontrunning. The darkpool must also verify that orders submitted to it have also been recorded in the append-only ledger, otherwise they would be liable to the ramifications of an unverified match. All parties can be incentivized by transaction fees: clients for submitting to both the darkpool and replicated state machine, the darkpool for ensuring good matches, and the nodes for running consensus protocol to ensure Byzantine Fault Tolerance.

The obvious next step would be to implement this in practice and see how clients react to this new darkpool paradigm. This paper presents just a new idea, in part for a class project, and the authors would love to see how something like this might be implemented in a production system.

VIII. Acknowledgement

We would like to acknowledge and thank our CA Geoff Ramseyer for his invaluable help in both understanding and implementing HotStuff. We are also grateful to our instructor, David Mazières, for teaching us everything distributed systems we needed for this project.

REFERENCES

[1] Dark pool trading system amp; regulation.

[2] Commissioner Luis A. Aguilar. Shedding light on dark pools, 2020.

[3] Eric Budish, Peter Cramton, and John 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.

[4] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[5] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[6] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[7] Alex Wang Matthew Finestone Daniel Wang, Jay Zhou. A decentralized token exchange protocol., 2017.

[8] Google. Working together to detect maliciously or mistakenly issued certificates.

[9] Andrei Kirilenko, Albert S Kyle, Mehrdad Samadi, and Tugkan Tuzun. The flash crash: High-frequency trading in an electronic market. *The Journal of Finance*, 72(3):967–998, 2017.

[10] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[11] Will Warren and Amir Bandeali. 0x: An open protocol for de- centralized exchange on the ethereum blockchain, 2017.

[12] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.