FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker

Michele Ciampi[1], Muhammad Ishaq[2], Malik Magdon-Ismail[3], Rafail Ostrovsky[4], and Vassilis Zikas[2]

[1] The University of Edinburgh
michele.ciampi@ed.ac.uk
[2] Purdue University
{ishaqm, vzikas}@cs.purdue.edu
[3] Rensselaer Polytechnic Institute (RPI)
magdon@gmail.com
[4] University of California, Los Angeles (UCLA)
rafail@cs.ucla.edu

Abstract. Frontrunning is a major problem in DeFi applications, such as blockchain-based exchanges. Albeit, existing solutions are not practical and/or they make external trust assumptions. In this work we propose a market-maker-based crypto-token exchange, which is both more efficient than existing solutions and offers provable resistance to frontrunning attack. Our approach combines in a clever way a game theoretic analysis of market-makers with new cryptography and blockchain tools to defend against all three ways by which an exchange might front-run, i.e., (1) reorder trade requests, (2) adaptively drop trade requests, and (3) adaptively insert (its own) trade requests. Concretely, we propose novel light-weight cryptographic tools and smart-contract-enforced incentives to eliminate reordering attacks and ensure that dropping requests have to be oblivious (uninformed) of the actual trade. We then prove that with these attacks eliminated, a so-called monopolistic market-maker has no longer incentives to add or drop trades. We have implemented and benchmarked our exchange and provide concrete evidence of its advantages over existing solutions.

Keywords: Front-running · Market Maker · Blockchain · Fairness

1 Introduction

Since Bitcoin's introduction in 2008, *crypto-currencies* have become a significant market in global finance[5]. Several tools and platforms have been built to facilitate crypto-currency trading. The early exchanges e.g. Binance, Coinbase, Bittrex, etc. have two undesirable properties. First, they were *custodial*, meaning that traders transfer their assets to the exchange, and trading activity translates to updating the exchange's internal mapping of traders and their assets. Second, they were *order-book* based i.e. they only match buyers with sellers (collectively, traders), so trading halts when there are no sellers whose ask-prices match the buyer bid-prices. The prices are fully controlled by traders and therefore can be volatile. The later exchanges e.g. Uniswap, Balancer, Curve, etc—collectively called Decentralized Exchanges or *DEXes*—are *non-custodial*, have their own inventory of assets, and use a market making (MM) algorithm to adjust prices. The latter type of exchanges, colloquially also known as *market makers*, leverage machine learning (ML) to increase liquidity along with additional desirable properties for the market maker (e.g., maximizing profit) or the market itself (e.g., stability by incentivizing traders to report true valuations).

A perennial problem with both types of exchanges (and traditional markets, too) is *frontrunning*, where an adversary reorders trades to gain a price advantage. For example, say a market maker (MM) is selling an asset t at \$100 each, and the pricing function is such that for each unit sold, the price increases by \$100. Say a trader P submits a trade T_P to buy one unit, a (possibly adversarial) MM A sees this pending trade and executes, before processing T_P, a different trade T_A to buy one unit, i.e. the adversary A "front-runs" the

[5] Market capitalization of approx. \$2 trillion during all of 2021.

trader P. Executing T_A before T_P raises the price to \$200 for P, \$100 more than he would otherwise pay. Trades should be executed in the order they were submitted.

Frontrunning penalizes honest players, and also has a detrimental effect on the health of the underlying crypto-currency blockchain. As an example, adversarial bots might flood the blockchain with frontrunning orders when an opportunity for profit arises, with only few of these orders being executed. This flooding creates a denial of service (DoS) attack. The above effects of frontrunning are worsened in decentralized exchanges. E.g., when the exchange is implemented by a smart contract on a chain like Ethereum, e.g., Uniswap, frontrunning raises transaction fees (*gas price* on Ethereum) as traders compete to get their trades in first. And, in theory, it can also affect the security of the underlying blockchain. Indeed, in DEXes, where the trade ordering is affected by miners through transaction reordering, frontrunning might create incentives for forking as the miners are more likely to pursue a chain on which they make more profit.

Traditional markets mitigate frontrunning through legislation. However, such legislation is tricky to enforce as most frontrunning attacks do not leave an indisputable evidence, which would be necessary to apply fines. As such, several mature markets have embedded certain controlled forms of frontrunning in their allowed operations. The classical example of this principle is embodied in traditional stock exchange markets, where high-profile clients might get so called *privileged access*, allowing them to react faster to market changes. The above solution is unsatisfactory for crypto-currency markets. For starters, legislation lags behind and is typically not tailored to crypto-currencies, making deterrence by regulation much harder. But more importantly, the egalitarian nature of these assets makes preferential frontrunning an undesirable feature for the majority of its users. The main question addressed (to the affirmative) by our work is

> Can we leverage the public-observability of blockchain ledgers together with cryptographic and machine learning tools to devise a practical frontrunning resistant market maker?

Informally, solving the above problem requires: (i) ensuring a strict ordering on the trades, and (ii) making sure the system is fast enough for high-frequency trading. The first requirement gets rid of frontrunning attack and the second ensures the solution's practicality. The first requirement can be further broken down into a) preventing traders from frontrunning each other and b) preventing MM from frontrunning. For private-state blockchains (e.g. ZCash, Dash, Monero, etc), preventing traders from frontrunning is easy because the traders cannot see each other's trades (due to transaction privacy). However, the mainstream blockchains (e.g. Bitcoin, Ethereum, Cardano, etc) are not private and preventing traders from frontrunning each other requires additional mechanisms.

Several such mechanisms have been proposed, e.g. commit-reveal, encryption, zk-rollups, speed-bumps/retro-active pricing, and commit secret sharing. However they all cause a slowdown of the system and are, therefore, in conflict with the second requirement (the system needs to be fast). Alternative approaches have proposed to simulate MM as a trusted third party (TTP) through secure multi-party computation (MPC), trusted execution environments (TEEs) and zero-knowledge proofs (ZKPs). These do solve the frontrunning problem, but once more, conflict with the second requirement and/or make additional trust assumptions (see also the related research section for a detailed comparison). Indeed, MPC and ZKPs are expensive cryptographic primitives that negatively affect speed of the system and, TEEs place additional demands on application hardware (and assumptions thereof). A viable solution needs to satisfy both of the above requirements, ideally without additional trust assumptions.

This work, FairMM, proposes a market maker (MM) that resolves the frontrunning problem using off-chain communication and inexpensive hash functions. This is done through a combination of techniques from cryptography, game theory, blockchain, and machine leaning for financial economics.

At a high level, FairMM operates as follows: Traders and MM communicate off-chain via secure communication channels (e.g. through TLS), traders form a queue, and a trade is processed as follows: 1) MM issues a ticket to the trader T_i at the front of the queue, this ticket is identified by a cryptographic hash and signed by the MM (think of the hash as a serial number), then, 2) trader T_i may decide to respond with trade (trade) or not trade ($\mathsf{no\_trade}$), 3) MM processes T_i's response and moves to the next trader T_{i+1}. The nature of this ticket hash (or serial number) is such that it incorporates the complete trading history up to that point. If the MM tries to talk to more than one trader at a time, all but one of the traders will get their trade incorporated into the trading history (trading history is linear, there is no way to keep all serial numbers valid without creating branches). MM posts the trading activity to a public bulletin board at regular intervals. The traders can read this bulletin board at any time and if they find any invalid tickets or that their ticket is missing, they can complain to a smart contract. This smart contract, established by the MM before its operation starts, locks a large collateral on behalf of MM. On a valid complaint, the complainant is rewarded (with a sufficiently large but less than the collateral amount) and MM loses all collateral. This ticketing mechanism is extremely efficient to compute.

The above ticketing mechanism already takes care of the worst-case scenario, i.e., reordering attacks. One could plug in any market making algorithm (to adjust asset prices) and obtain a reordering-resilient system. However, the MM can still drop trades, although it will be doing so without the knowledge of subsequent trades. We resolve this problem by carefully choosing a market maker—a *monopolistic profit seeking* market maker—that has economic incentive to not manipulate trading activity in this manner (e.g. by dropping trades obliviously of future trade requests). A monopolistic profit seeking MM uses trade requests as signals to determine where the true value of an asset lies (more buy trades $\implies$ true value of the asset is higher, more sell trades $\implies$ true value is lower). Its core principle is that, because a monopolistic profit seeking MM makes most profit when trading activity happens around the true value of an asset, it has no incentive to manipulate trading requests that would make the signals from trading activity less reliable. See Section 5 for formal discussion.

In addition to proving the security of FairMM, we have implemented and evaluated our design. We show that this design is extremely competitive. Concretely, we achieve a throughput of over 200 trades/minute. This is despite strict serialization of trade requests and the fact that we are running off-the-chain part on a relatively weak, consumer laptop (which communicates with an actual Ethereum node in test environment). These figures are about 50% higher than the maximum daily volume of Uniswap [36], arguably the most popular DEX and an order of magnitude higher than P2DEX [55], an order-book exchange implemented using MPC. We are also better than TEX [42], also an order-book exchange, which either does not support high frequency trading or is not frontrunning resilient. While Tesseract [38], another orderbook exchange, reports much higher throughput, it requires both trusted hardware and a consensus group assumption for its security guarantees. We, on the other had, require no such assumption.

In summary, our work makes the following contributions:

- We provide design of a non-custodial frontrunning resilient market-making crypto-currency exchange that does not require sophisticated cryptographic machinery (MPC, ZKPs, etc), special hardware (TEEs) or additional assumptions (e.g. an additional consensus group in addition to the blockchain).
- We extract a useful abstraction—Σ-trade protocols—for asset exchanges that facilitates modular design of blockchain trading systems.
- We provide an instantiation of the system on Ethereum blockchain and demonstrate that it is very fast, and practical for real world applications.

1.1 Related Works

Market Makers for Crypto-token Markets New emerging markets, e.g. prediction markets [10] or crypto-token markets, are typically thin and illiquid and often have to be bootstrapped through intelligent market makers to provide liquidity and price discovery [12]. A market maker algorithm aims at maximizing liquidity in the market and/or maximizing its own profit. The *zero-profit competitive market maker* model [4, 11, 1] considers multiple MMs that compete with each other by lowering their marginal profit to eliminate competition—such a system converges to a zero-profit. The *monopolist* market-maker, has been shown to provide greater liquidity than zero-profit competitive market makers [4, 11, 1, 13]. We adopt the extension by Das [11] (cf. Section 5).

Fair Exchange and Blockchains There is a large amount of literature on fair exchange including early MPC works [2, 3, 5, 6, 14], which has been re-ignited with the adoption of blockchains and cryptocurrencies [16, 21, 37, 20, 24, 39]. Due to the relevance of these works to ours, we include a detailed review in Appendix B. However, these works are not suitable for reuse in our design. Informally, the reason is that in our setting, fair exchange is a subroutine of the Market Maker (MM) protocol, and MM needs to know immediately whether a trade will settle or not on the blockchain. Therefore, we designed our own fair exchange protocol, a Σ-trade protocol, that we proved amenable to such composition.

Decentralized exchanges Popular decentralized exchanges e.g. Uniswap, Curve, Kyber, etc [36, 44, 28, 29, 30, 62, 33, 34] do not defend against frontrunning. To our knowledge, Tesseract by Bentov et al. [38] is the first work that addresses frontrunning in the crypto-currency space. It is orderbook based, custodial, and simulates a trusted third party (TTP) through trusted execution environments (TEEs). The assumption here is that since the exchange is a TTP, frontrunning does not happen. Since it relies on players to provide it with blockchain data, there is a check-pointing mechanism on trusted blocks, and if the exchange becomes unavailable, there is a *consensus group* of TEE backed nodes that can enforce/cancel transactions so that traders do not lose funds. In a similar vein, orderbook based P2DEX [55] by Baum et al. simulates TTP through outsourced MPC, their technique is similar to the work by Charanjit et al [19] for traditional markets. TEX (Trustless Exchange) by Khalil et al. [42] is another orderbook exchange. It uses Zero Knowledge Proofs (ZKPs) for its guarantees. ZKPs are an expensive primitive and, in TEX, there is a trade-off as it either does not fully support high frequency trading or does not provide frontrunning resilience. In contrast to the above works, our construction is a *market maker*, it is *non-custodial*, does away with expensive primitives (MPC, ZKPs), additional requirements on hardware and/or additional check-pointing/consensus mechanisms, and provides frontrunning defence in a high frequency trading environment (as demonstrated by our detailed comparison with Uniswap in Section 6.3). Note however, that Tesseract supports cross-chain trading, our work does not.

Fairy by Stathakopoulou et al. [59] solve frontrunning for Byzantine Fault Tolerant (BFT) systems by augmenting Total Order Broadcast (TOB) protocols with *input causality* and *sender obfuscation*. They also require TEEs. Moreover, adapting this work to address frontrunning in crypto-currencies is non-trivial. GageMPC [53] by Almashaqbeh et al. tackles privacy preserving auctions using non-interactive MPC (NIMPC). This work could be adapted into an exchange, but it is unclear whether it could handle high frequency trading. A^2MM by Zhou et al. [60] optimizes onchain swaps to *mitigate* frontrunning attacks. They study two point arbitrages for two assets. Their analysis holds assuming that all exchanges on a blockchain will be handled by A^2MM. This assumption is too strict for practical applications. Flashbots [57] and Gnosis Protocol V2 [58] both claim to resolve frontrunning. Flashbots requires strong trust (in the players to follow protocol) and is therefore not comparable to our work. Gnosis Protocol V2 claims that it will have a defense against frontrunning when it is built but currently there is no description of how it will be achieved. We refer to Chainlink 2.0 [56] whitepaper for details on existing techniques to achieve strict ordering of transactions. It also proposes a Fair Sequence Service (FSS) for Distributed Oracle Networks (DONs) that should solve this problem in general. However, exactly how such FSS will be implemented is not specified in the whitepaper.

There are some complementary works to ours which studies frontrunning. Flash Boys 2.0 [45] by Daian et al. give evidence that frontrunning is a serious problem on Ethereum. Bartoletti et al. [54] provide a theoretical framework to maximize miner extractable value (MEV), Sobol et al [50] discuss frontrunning on proof of stake blockchains, and Zhou et al. [61] study sandwich attacks.

Next we dive into technical details of the paper, due to space constraints, we have provided relevant background in Appendix A.

2 Σ-Trade Protocols

A Σ-trade protocol Π is an interactive protocol run by a seller S and potentially many buyers $B_1, \ldots, B_m$ (seller S need not know m) where the exchange of tokens happens on blockchain E (We can think of E as the Ethereum blockchain).

Assume two tokens t_1 and t_2, each buyer wants to buy tokens of type t_2 in exchange of tokens of type t_1. Assume also that, each buyer B_i has an upper bound, denoted with $\mathbf{z}_i$, of type t_1 tokens that he can spend. The amount of t_2 tokens the buyer wants to buy is decided adaptively in the last round of interaction. A Σ-trade protocol Π consists of the following steps:

$\texttt{SC}_i$

State: $\texttt{z}_i \varXi$ locked for time T_i, the public keys $(\texttt{pk}_S^E, \texttt{pk}_S^T)$, $(\texttt{pk}_i^E, \texttt{pk}_i^T)$ and an initially empty list of identifier `usedIDs`
Input: $x, y, \texttt{ID}, \sigma_1, \sigma_2$. If $\texttt{ID} \notin \texttt{usedIDs}$ and $\texttt{Ver}(\texttt{pk}_i^E, \sigma_1, x||y||\texttt{pk}_S^E||\texttt{ID}) = 1$ and $\texttt{Ver}(\texttt{pk}_S^E, \sigma_2, x||y||\texttt{pk}_i^E||\texttt{ID}) = 1$ and there is a transaction with the identifier ID in its payload that moves $y\check{T}$ from $\texttt{pk}_S^{\check{T}}$ to $\texttt{pk}_i^{\check{T}}$ then move $x\varXi$ from $\texttt{pk}_i^E$ to $\texttt{pk}_S^E$, set $z_i \leftarrow z_i - x$ and add ID to `usedIDs`.

Fig. 1. Smart contract $\texttt{SC}_i$ for the case where B_i wants to buy $\check{T}$ for $\varXi$. The time T_i has to be set in such a way that the seller has time to create a transaction that pays B_i and to invoke the contract to get the $\varXi$ from B_i

1. Each buyer B_i creates a smart contract $\texttt{SC}_i$ on E that locks $\texttt{z}_i$ tokens of type t_1 (more details on $\texttt{SC}_i$ are provided later).
2. B_i and S exchange three off-chain messages. First, B_i sends his identities to the seller S. Note that B_i does not yet disclose his desired quantity t_2 tokens. In response, S proposes the exchange rate, `askedPrice`, for the tokens.
3. Let y be the quantity of tokens of type t_2 that the buyer wants to buy s.t. $y \cdot \texttt{askedPrice} \leq \texttt{z}_i$. If B_i agrees with `askedPrice`, then B_i sends a *certificate* c. This c can be used by S to invoke $\texttt{SC}_i$ and withdraw $x = y \cdot \texttt{askedPrice}$ tokens of type t_1 from B_i's account. However, $\texttt{SC}_i$ will move the x tokens from B_i's account if S has moved to B_i's account y tokens of type t_2. $\texttt{SC}_i$ ensures atomic transactions but can only be triggered by the seller.

Any instantiation of $\varSigma$-trade protocol can be used in our $\varPi^{\texttt{trade}}$ protocol. We now show an insantiation of $\varSigma$-trade protocol to trade $\check{T}$ for $\varXi$.

2.1 Selling tokens for ethers

For a buyer B_i, denote with $(\texttt{sk}_i^C, \texttt{pk}_i^C)$ the signing-verification keys associated with account $C \in \{E, \check{T}\}$, where E represents Ethereum and $\check{T}$, a token on Ethereum. $\varXi$ denotes Ethereum currency. Similarly for seller, $(\texttt{sk}_S^C, \texttt{pk}_S^C)$ denote the signing-verification keys associated with account $C \in \{E, \check{T}\}$.

A formal description of the smart-contract and our protocol $\varPi$ is in Fig. 1 and Fig. 2. Here, we give the intuition. The smart contract $\texttt{SC}_i$ locks for T_i rounds $\texttt{z}_i\varXi$ and manages a list of transaction identifiers. Upon receiving an input $(x, y, \texttt{ID})$ that has been authenticated by both the buyer and the seller, $\texttt{SC}_i$ moves $x\varXi$ to seller's account if 1) a transaction `trx` that moves $y\check{T}$ from the seller's account to the buyer's account has been made and 2) `trx` contains the identifier ID in its payload. In addition, to prevent replay attacks, $\texttt{SC}_i$ does not allow reusing ID. The same contract $\texttt{SC}_i$ can be used for multiple trades if $\texttt{z}_i$ is big enough.

We now describe the protocol. The buyer sends his Ethereum public key to the seller, who replies with the exchange rate, `askedPrice` between $\varXi$ and $\check{T}$. If the buyer agrees with `askedPrice` and wants to buy $y\check{T}$ tokens, he generates an identifier ID, computes $x = y \cdot \texttt{askedPrice}$ in $\varXi$, and signs $x||y||\texttt{ID}$. He sends the signed values (and signature) to the seller. The seller, 1) posts a transaction `trx` that pays $y\check{T}$ into the buyer's account, `trx` contains ID in its payload, and 2) signs $x||y||\texttt{ID}$, and uses the resulting signature, along with signature from the buyer, to invoke $\texttt{SC}_i$. Note that the seller could post `trx` and also sends it to the buyer to indicate that the trade will occur.

3 (Fair) Ordering of Transactions

Our main contribution is the Universally Composable (UC)[8] formalization and realization of the *trade functionality* $\mathcal{F}^{\texttt{trade}}$. $\mathcal{F}^{\texttt{trade}}$ formally specifies the only ways in which the market maker can reorder the trades. For simplicity, assume that there are only two assets: $\varXi$ and $\check{T}$. Denote with $\texttt{price}^{\check{T}\rightarrow\varXi}$ (and $\texttt{price}^{\varXi\rightarrow\check{T}}$)

Π

$\mathbf{B_i}$'s initial state:$(\mathtt{pk}_S^{\Xi}, \mathtt{pk}_S^{\check{T}})$, $(\mathtt{pk}_i^{\Xi}, \mathtt{pk}_i^{\check{T}})$, $(\mathtt{sk}_i^{\Xi}, \mathtt{sk}_i^{\check{T}})$, the smart contract $\mathtt{SC}_i$ (see Fig. 1) and a transaction identifier $\mathtt{ID}$ initialized to 0.
S's initial state: $(\mathtt{pk}_S^{\Xi}, \mathtt{pk}_S^{\check{T}})$, $(\mathtt{sk}_S^{\Xi}, \mathtt{sk}_S^{\check{T}})$.

B_i. Let y be the amount of $\check{T}$ that B_i wants buy using Ξ. Send $\mathtt{pk}_i^E$ to S

S. Let $\mathtt{askedPrice}$ be the price at which S is willing to sell $\check{T}$ for Ξ (i.e., $1\check{T} = \mathtt{askedPrice}\Xi$). Upon receiving $\mathtt{pk}_i^E$ from the party B_i do the following.
- If B_i has created a contract according to Fig. 1 then continue, otherwise stop interacting with B_i.
- Send $\mathtt{askedPrice}$ to B_i.

B_i. Upon receiving $\mathtt{askedPrice}$ from S, if $\mathtt{askedPrice}$ represents a good price (w.r.t. the strategy of B_i) then do the following steps, otherwise send $\mathtt{NO\text{-}TRADE}$ to S.
- Compute $\mathtt{ID} \leftarrow \mathtt{ID} + 1$ and $x \leftarrow y \cdot \mathtt{askedPrice}$ and $\sigma_1 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}_i^E, x||y||\mathtt{pk}_S^E||\mathtt{ID})$
- Send $x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}_i^{\check{T}}$ to S.

S. Upon receiving $x, y, \mathtt{ID}, \sigma_1$ from B_i do the following steps.
- If $\mathtt{Ver}(\mathtt{pk}_i^E, \sigma_1, x||y||\mathtt{pk}_S^E||\mathtt{ID}) = 1$ and $x = y \cdot \mathtt{askedPrice}$ then continue with the following steps, ignore the message of B_i otherwise.
- Compute $\sigma_2 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}_S^E, x||y||\mathtt{pk}_i^E||\mathtt{ID})$.
- Post a transaction $\mathtt{trx}$ with the identifier $\mathtt{ID}$ in its payload that moves $y\check{T}$ from $\mathtt{pk}_S^{\check{T}}$ toward $\mathtt{pk}_i^{\check{T}}$.
- Invoke $\mathtt{SC}_i$ using the input $(x, y, \mathtt{ID}, \sigma_1, \sigma_2)$.

Fig. 2. Π, B_i wants to sell Ξ for $\check{T}$.

the price at which MM sells $\check{T}$ (or Ξ) for Ξ (for $\check{T}$). Assume that trader P_i's trade information is encoded in $\mathtt{trade}_i$. That is, $\mathtt{trade}_i$ describes the type and the amount of assets, the prices, trade direction (sell or buy) and etc. Moreover, assume that all the parties share the procedure MMalgorithm (the MM algorithm), which on input of a trade outputs the updated prices ($\mathtt{price}^{\check{T}\to\Xi}$ and $\mathtt{price}^{\Xi\to\check{T}}$). At a high level, $\mathcal{F}^{\mathsf{trade}}$ works as follows. Upon receiving a request from a trader P_i, $\mathcal{F}^{\mathsf{trade}}$ sends the prices to P_i, and signals to MM that P_i wants to trade. If P_i agrees with the prices, he sends trade information, $\mathtt{trade}_i$, to $\mathcal{F}^{\mathsf{trade}}$. Upon receiving $\mathtt{trade}_i$, $\mathcal{F}^{\mathsf{trade}}$ forwards $\mathtt{trade}_i$ to MM who has two choices: 1) decide not to trade with P_i by sending a command NO-TRADE to $\mathcal{F}^{\mathsf{trade}}$, or 2) accept trade with P_i. If MM does any other action before doing one of these two (e.g., MM starts trading with a party other than P_i), $\mathcal{F}^{\mathsf{trade}}$ allows that but also sets a special flag abort to 1. This means that if the traders query $\mathcal{F}^{\mathsf{trade}}$ with the command getTrades (to get the list of trades accepted by MM), $\mathcal{F}^{\mathsf{trade}}$ would return $\perp$ to denote that MM has misbehaved. A corrupt MM can also decide to set the output of $\mathcal{F}^{\mathsf{trade}}$ to always be $\perp$. This captures the fact that MM can decide to stop working at his will. Moreover, MM can add any trade of a corrupted party to the list of trades using the command setAdvTrade, but this can be done only after MM has concluded any in-progress trades, as specified above.

$\mathcal{F}^{\mathsf{trade}}$ is parametrized by Δ, which denotes the maximum number of rounds per *epoch*. In each *epoch* MM should allow traders to see the entire list of trades. MM can make the list of trades accessible via a special command setOutput. If MM does not send this command at least every Δ rounds, $\mathcal{F}^{\mathsf{trade}}$ will return $\perp$ to any honest party who requests trades list.

Note that $\mathcal{F}^{\mathsf{trade}}$ allows the adversarial MM to misbehave (e.g., by completely reordering the trades) but this misbehavior will be notified to the honest parties. Moreover, the MM cannot modify the trades (e.g., change the quantity that a party P_i is willing to sell/buy). Therefore, even if the adversary reorders the trades (at the cost of being detected), all the trades will be consistent with the prices that $\mathcal{F}^{\mathsf{trade}}$ sent to the traders. The market maker still has the power to choose the parties he wants to trade with first, however, this choice has to be made obliviously of the trade information of the honest party. Luckily, we can also argue that for a relevant class of market-making algorithms, this does not constitute an additional useful power. We finally note that $\mathcal{F}^{\mathsf{trade}}$ does not allow any real exchange of assets. However, if the output of $\mathcal{F}^{\mathsf{trade}}$ is posted on a blockchain and if the trades are defined properly according to the language of the blockchain,

Auxiliary procedures

```
$\texttt{verifyPrices}(\texttt{Trades})$
**Constants**: $\texttt{SP}^{\check{T}\to\Xi}, \texttt{SP}^{\Xi\to\check{T}}$
    Set $p_1 \leftarrow \texttt{SP}^{\check{T}\to\Xi}$, $p_2 \leftarrow \texttt{SP}^{\Xi\to\check{T}}$ and for $j \leftarrow 1, \ldots, |\texttt{Trades}|$
        If $\texttt{checkTrade}(\texttt{trade}_j, p_1, p_2) = 0$ then return 0.[a]
        Compute $p_1, p_2 \leftarrow \texttt{MMalgorithm}(p_1, p_2, \texttt{trade}_j)$.
    return 1.
$\texttt{verification}(h_{\texttt{start}}, h', h, \texttt{pk}, \texttt{requests})$
    Set $\texttt{found} \leftarrow 0$.
    Set $h_1 \leftarrow h_{\texttt{start}}$ and for $j \leftarrow 1, \ldots, |\texttt{requests}|$
        Parse $\texttt{requests}[j]$ as $(\texttt{pk}_j, \texttt{trade}_j, \sigma_j)$.
        Compute $h_2 \leftarrow \mathcal{H}(h_1||\texttt{pk}_j)$, $h_1 \leftarrow \mathcal{H}(h_2||\texttt{trade}_j)$.
        If $h_2 = h$ and $\texttt{pk} = \texttt{pk}_j$ then
            $\texttt{found} \leftarrow 1$.
            If $\texttt{trade} \neq \texttt{NO-TRADE}$ and $\texttt{Ver}(\texttt{pk}_j, \sigma_j, \texttt{trade}) \neq 1$ then return 0.[b]
    If $h' \neq h_1$ or $\texttt{found} = 0$ then return 0.
    Return 1.
$\texttt{checkBB}(h_{\texttt{start}}, h', \texttt{requests}, \texttt{pk}_{\texttt{MM}}, \texttt{e})$
    Set $\texttt{found} \leftarrow 1$.
    For any value $t_i := (h_i, \sigma_i, \texttt{pk}_i, \texttt{e})$ that appears on the BB (posted on the behalf of a party which is not MM)
    do the following.
        If $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, h_i||\texttt{pk}_i||\texttt{e}, \sigma_i) = 0$ then ignore $t_i$ else $\texttt{found} \leftarrow \texttt{found} \wedge$
        $\texttt{verification}(h_{\texttt{start}}, h', h_i, \texttt{pk}_i, \texttt{requests})$
    Return $\texttt{found}$.
$\texttt{checkPrices}(\texttt{SP}^{\check{T}\to\Xi}, \texttt{SP}^{\Xi\to\check{T}}, \texttt{Trades})$
    Initialize $p_1 \leftarrow \texttt{SP}^{\check{T}\to\Xi}$, $p_2 \leftarrow \texttt{SP}^{\Xi\to\check{T}}$ and $\texttt{abort} \leftarrow 0$.
    For $j \leftarrow 1, \ldots, |\texttt{Trades}|$
        parse $\texttt{Trades}[j]$ as $(P, \texttt{trade})$.
        if $\texttt{checkTrade}(\texttt{trade}_j, p_1, p_2) = 0$ then $\texttt{abort} \leftarrow 1$.
        compute $p_1, p_2 \leftarrow \texttt{MMalgorithm}(p_1, p_2, \texttt{trade})$.
    Return $\texttt{abort}$
```

[a] In this case, everybody will detect an invalid computation of the prices since the algorithm MMalgorithm is public.

[b] Also in this case, everybody will detect an invalid signature and abort.

Fig. 3. Auxiliary procedures

then the MM can use the trades to trigger events on the blockchain that move the assets according to what is described by $\mathcal{F}^{\texttt{trade}}$. We can also disincentivize any malicious behavior of the adversary by means of the compensation paradigm over the blockchain. Indeed, given that in our protocol all the honest parties can detect a malicious behavior without using any private state, the same can be done by a smart contract.

To simplify the description of our protocol, we make use of the procedures checkTrade and checkPrices. checkTrade takes as input trade, $\texttt{price}^{\check{T}\to\Xi}$ and $\texttt{price}^{\Xi\to\check{T}}$, and outputs 1 if the description of a trade trade is consistent with the prices defined by $(\texttt{price}^{\check{T}\to\Xi}, \texttt{price}^{\Xi\to\check{T}})$. checkPrices takes as input a list of trades and verifies that trade prices are consistent with MMalgorithm. These procedure are formally specified in Fig. 3 and $\mathcal{F}^{\texttt{trade}}$ in Fig. 4.

3.1 Our Protocol: how to realize $\mathcal{F}^{\texttt{trade}}$

Assume all parties have access to a bulletin board BB, all parties know the MM's public key, and the procedure MMalgorithm is public. Our protocol realizes $\mathcal{F}^{\texttt{trade}}$ as follows. MM maintains a hash chain (that starts with a

Functionality $\mathcal{F}^{\texttt{trade}}$

$\mathcal{F}^{\texttt{trade}}$ is parametrized by party-set $P_1, \ldots, P_m$ and the market maker $\texttt{MM}$. The functionality also manages a flag $\texttt{abort} \leftarrow 0$ and an initially empty list $\texttt{Trades}$. It is also parametrized by τ, $\texttt{timer}$ (with $\texttt{timer}$ initialized to $-\tau$), the starting prices $\texttt{SP}^{\check{T} \to \varXi}$ and $\texttt{SP}^{\varXi \to \check{T}}$ at which $\texttt{MM}$ is willing to sell (and respectively buy) $\check{T}$ for $\varXi$, respectively, the integers $\varDelta$, R, and the epoch index $\texttt{e}$. The functionality initializes $\texttt{price}^{\varXi \to \check{T}} \leftarrow \texttt{SP}^{\varXi \to \check{T}}$, $\texttt{price}^{\check{T} \to \varXi} \leftarrow \texttt{SP}^{\check{T} \to \varXi}$, $R \leftarrow \varDelta$, $\texttt{e} \leftarrow 0$.

Let $T_{\texttt{now}}$ be the current round ($T_{\texttt{now}} > 0$), upon receiving any message from any party or from the adversary $\mathcal{A}$ act as follows:

- Upon receiving $(\texttt{request}, P_i)$ from a party P_i
  - If $\texttt{MM}$ is corrupted then send $(\texttt{request}, P_i)$ to $\mathcal{A}$ else
  - If $T_{\texttt{now}} - \texttt{timer} > \tau$ then[a] send $(\texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi})$ to P_i, send P_i to $\texttt{MM}$, set $\texttt{activep} \leftarrow P_i$ and $\texttt{timer} \leftarrow T_{\texttt{now}}$ else ignore the command.
- Upon receiving $(\texttt{setPrice}, P_i, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi})$ from a corrupted $\texttt{MM}$ then send $(\texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi})$ to P_i. If there is no entry $(P_j, \perp)$ with $j \in [m]$ in $\texttt{Trades}$ then add the entry $(P_i, \perp)$ to the list $\texttt{Trades}$, otherwise set $\texttt{abort} \leftarrow 1$.
- Upon receiving $(\texttt{ok}, \texttt{trade}_i)$ from P_i
  - If $\texttt{MM}$ is corrupted then send $(P_i, \texttt{trade}_i)$ to $\mathcal{A}$
  - else if $P_i \neq \texttt{activep}$ then ignore the input, else

    If $\texttt{checkTrade}(\texttt{trade}_i, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi}) = 1$ then add $(P_i, \texttt{trade}_i)$ to $\texttt{Trades}$ and send $\texttt{ok}$ to P_i else add $(P_i, \texttt{NO-TRADE})$ to $\texttt{Trades}$ and send $(\texttt{ko})$ to P_i.
- Upon receiving $(\texttt{setTrade}, P_i, y)$ from $\mathcal{A}$ do:
  - If the entry $(P_i, \perp)$ is on the top of the list $\texttt{Trades}$ and $(\texttt{ok}, \texttt{trade}_i)$ has been received from P_i then
    - if $y = 1$ then replace $(P_i, \perp)$ with $(P_i, \texttt{trade}_i)$ in $\texttt{Trades}$, otherwise replace $(P_i, \perp)$ with $(P_i, \texttt{NO-TRADE})$
  - else set $\texttt{abort} \leftarrow 1$.
- Upon receiving $(\texttt{setAdvTrade}, P_i, \texttt{trade})$ from $\mathcal{A}$, if P_i is not a corrupted party then ignore the message, otherwise do the following:
  - if there is an entry $(P_j, \perp)$ with $j \in [m]$ on the top of the list then set $\texttt{abort} \leftarrow 1$ else
  - add $(P_i, \texttt{trade})$ to $\texttt{Trades}$.
- Upon receiving $(\texttt{getTrades})$ from a party P_i at round $T_{\texttt{now}}$ do the following steps.
  - If $T_{\texttt{now}} \leq R$ and $\texttt{output} = 0$ then ignore the input of P_i.
  - If $T_{\texttt{now}} \geq R$ and $\texttt{output} = 0$ then return $\perp$ else
  - If $\texttt{output} = 1$ and $\texttt{abort} = 0$ and $\texttt{checkPrices}(\texttt{SP}^{\check{T} \to \varXi}, \texttt{SP}^{\varXi \to \check{T}}, \texttt{Trades}) = 0$ and there is no entry $(P_j, \perp)$ in $\texttt{Trades}$ then return $(\texttt{Trades}^\star, \texttt{e})$ (where $\texttt{Trades}^\star$ is a copy of $\texttt{Trades}$ with the expection that each entry $(P, \texttt{trade})$ is replaced with $\texttt{trade}$ in $\texttt{Trades}^\star$, in the same order, but) and set $R \leftarrow T_{\texttt{now}} + \varDelta$, $\texttt{e} \leftarrow \texttt{e} + 1$, $\texttt{output} = 0$.
  - else return $\perp$.

[a] This condition lets the functionality ignore any new request for at most τ rounds. Note that the first time the functionality receives the commnad $\texttt{request}$ the condition trivially holds.

Fig. 4. The trades functionality $\mathcal{F}^{\texttt{trade}}$.

$\mathit{\Pi}^{\texttt{trade}}$

1) P_i**: creation of a request.** Send $(\texttt{request}, \texttt{pk}_i)$ to $\texttt{MM}$.
2) $\texttt{MM}$**: waiting for a request.** Upon receiving $(\texttt{request}, \texttt{pk}_i)$ from the party P_i do the following.
   – Compute $h' \leftarrow \mathcal{H}(h||\texttt{pk}_i)$ and set $h \leftarrow h'$ and $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h||\texttt{pk}_i||\texttt{e})$.
   – Send $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi}, \texttt{pk}_i)$ to P_i and ignore any request that comes from any party $P_j \neq P_i$ for τ rounds.
3) P_i**: finalizing the request.** Upon receiving $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi}, \texttt{pk}_i)$ from $\texttt{MM}$, if the received prices are not satisfactory then send $\texttt{NO-TRADE}$. Else create $\texttt{trade}$ with all the required information with $\texttt{trade}.p_1 = \texttt{price}^{\varXi \to \check{T}}$ and $\texttt{trade}.p_2 = \texttt{price}^{\check{T} \to \varXi}$ and do the following:
   – If $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, \sigma, h||\texttt{pk}_i||\texttt{e}_i) = 1$ then compute $\sigma_i \leftarrow \texttt{Sign}(\texttt{sk}_i, h||\texttt{trade})$ and send $(\texttt{trade}, \sigma_i)$ to $\texttt{MM}$, else ignore the message received from $\texttt{MM}$
4) $\texttt{MM}$**: reply to the request of** P_i**.** If $(\texttt{trade}, \sigma_i)$ is received from P_i within τ rounds such that $\texttt{Ver}(\texttt{pk}_i, \sigma_i, h||\texttt{trade})) = 1$ and $\texttt{checkTrade}(\texttt{trade}_i, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi}) = 1$ then do the following.
   – Compute $h' \leftarrow \mathcal{H}(h||\texttt{trade})$ and set $h \leftarrow h'$.
   – Add $(\texttt{pk}_i, \texttt{trade}, \sigma_i)$ to $\texttt{requests}$.
   – Run $\texttt{MMalgorithm}(\texttt{trade}, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi})$ thus obtaining $\texttt{price}^{\varXi \to \check{T}'}, \texttt{price}^{\check{T} \to \varXi'}$ and set $\texttt{price}^{\varXi \to \check{T}} \leftarrow \texttt{price}^{\varXi \to \check{T}'}, \texttt{price}^{\check{T} \to \varXi} \leftarrow \texttt{price}^{\check{T} \to \varXi'}$.

   else do the following
   – Compute $h' \leftarrow \mathcal{H}(h||\texttt{NO-TRADE})$ and set $h \leftarrow h'$ and add $(\texttt{pk}_i, \texttt{NO-TRADE}, 0^\lambda)$ to $\texttt{requests}$.

   Start accepting new requests from any party (i.e., goto step 2).
5) $\texttt{MM}$**: posting trades to the BB**. If R rounds have passed, post $(h, \sigma, \texttt{requests}, \sigma^\star, \texttt{e})$ to the BB, where $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h)$ and $\sigma^\star \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, \texttt{requests}||\texttt{e})$. Set $R \leftarrow R + \mathit{\Delta}$, update the epoch number $\texttt{e} \leftarrow \texttt{e} + 1$ and reinitialize $\texttt{requests}$.
6) P_i**: checking honest behavior of the** $\texttt{MM}$. In each round P_i does the following
   - If no message $(h', \sigma', \texttt{requests}, \sigma^\star, \texttt{e}_i)$ has been posted on the BB within the last $\mathit{\Delta}$ rounds such that $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, \sigma', h') = 1$ and $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, \sigma^\star, \texttt{requests}||\texttt{e}_i) = 1$ then output $\perp$, else compute $\texttt{e}_i \leftarrow \texttt{e}_i + 1$ and continue as follows.
   - If P_i has not received a new ticket $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\varXi \to \check{T}}, \texttt{price}^{\check{T} \to \varXi}, \texttt{pk}_i)$ during the epoch $\texttt{e}_i - 1$ then continue, else if $\texttt{verification}(h_{\texttt{e}_i - 1}, h', h, \texttt{pk}_i, \texttt{requests}) = 0$ then send $(h, \sigma, \texttt{pk}, \texttt{e}_{i-1})$ to the BB as a proof of cheating of $\texttt{MM}$ and set $\texttt{output}_i \leftarrow \perp$.
   - Set $h_{\texttt{e}_i} \leftarrow h'$.
7) P_i upon receiving $\texttt{getTrades}$, if $\texttt{output}_i = \perp$ then return $\perp$ else reinitialize $\texttt{Trades}$ and do the following.
   - For each message $(h'_j, \sigma'_j, \texttt{requests}_j, \sigma^\star_j, j)$ with $j \in \{0, \ldots, \texttt{e}_i - 1\}$ such that $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, \sigma'_j, h'_j) = 1$ and $\texttt{Ver}(\texttt{pk}_{\texttt{MM}}, \sigma^\star_j, \texttt{requests}_j||j) = 1$ posted on the BB, if $\texttt{checkBB}(h_j, h'_j, \texttt{requests}_j, \texttt{pk}_{\texttt{MM}}, j)$=0 then return $\perp$, else for each $(\texttt{pk}, \texttt{trade}, \sigma)$ in $\texttt{requests}_j$ add $\texttt{trade}$ to $\texttt{Trades}$.
   - If $\texttt{verifyPrices}(\texttt{Trades}) = 1$ then output $(\texttt{Trades}, \texttt{e}_i)$ else output $\perp$.

Fig. 5. $\mathit{\Pi}^{\texttt{trade}}$, the protocol that realizes $\mathcal{F}^{\texttt{trade}}$.

value $h_{\texttt{start}}$), all parties know $h_{\texttt{start}}$. Whenever MM receives a request from a trader P_i, he adds to the hash chain the public identity of P_i, signs the new head (say h_i), the public key of P_i and the current prices. We call this set of information a *ticket*. The MM then hands over the ticket to the trader. The trader checks that the signature is valid under the MM's public key, and if so, P_i defines the trade $\texttt{trade}_i$, signs it thus obtaining σ_i, and sends $(\texttt{trade}_i, \sigma_i)$ to MM (σ_i guarantees that MM cannot change $\texttt{trade}_i$). MM, upon receiving $\texttt{trade}_i$ and its signature, checks if $\texttt{trade}_i$ is well formed (i.e., the prices used to describe $\texttt{trade}_i$ are consistent with what MM sent in the previous round). If so, MM adds to the hash chain $\texttt{trade}_i$, adds $\texttt{trade}_i$ with σ_i to a list `requests`, run `MMalgorithm` on $\texttt{trade}_i$ and the current prices to get the new prices, and waits to receive next trade request.

In every epoch (at most Δ rounds) MM publishes to the bulletin board[6] the head h of the hash chain and the list `requests`, all authenticated with his signing key. If MM that does not post such authenticated information within Δ rounds then all the traders will understand this as an abort and output $\perp$. Each honest party that has access to the BB now: 1) checks that each trade in `requests` is either NO-TRADE or a correctly signed trade; 2) checks that all the prices are consistent with `MMalgorithm` and that the hash chain that starts at $h_{\texttt{start}}$ and finishes at h can be constructed using the trades in `requests`; and, 3) checks if the hash value h_i (received as part of the ticket) is part of the hash chain.

We observe that anyone (even traders who did not trade e.g. third parties) can check if the first and the second conditions hold. If either the first or the second condition does not hold, then all the honest traders output $\perp$. The third condition can be checked only by a trader who received a ticket. If a trader detects that the third condition does not hold, he can post his ticket on the bulletin board. At this point all the other parties who see BB can also determine that MM misbehaved and output $\perp$. Intuitively, our protocol realizes $\mathcal{F}^{\texttt{trade}}$ because once MM sends a ticket to a trader, he also commits to a set of trades. As long as MM cannot generate collisions for the hash function, he cannot include new trades in the hash chain. This protocol, $\Pi^{\texttt{trade}}$, is formally specified in Fig. 5. In the protocol, MM maintains $h \leftarrow 0^\lambda$, an initially empty list `requests` and the integers R, τ and Δ. Δ represents the maximum number of rounds after which MM has to post the trades on the BB, τ represents the timeout (e.g. number of seconds, or rounds) before which a party has to reply to MM (to avoid DoS attack) and R is initialized to Δ. Let also $\texttt{SP}^{\check{T}\to\Xi}$ and $\texttt{SP}^{\Xi\to\check{T}}$ be the starting prices. MM also maintains an integer called *epoch index* denoted with $\texttt{e}$, MM initializes $\texttt{price}^{\Xi\to\check{T}} \leftarrow \texttt{SP}^{\Xi\to\check{T}}$ and $\texttt{price}^{\check{T}\to\Xi} \leftarrow \texttt{SP}^{\check{T}\to\Xi}$ and $\texttt{e} = 0$. Each party maintains and initially empty list `Trades`, $h_0 \leftarrow 0^\lambda$ and a view of the current epoch index which we denote by $\texttt{e}_i$.

The protocol uses utility procedures to check misbehavior. A formal description of these procedures is presented in Fig. 3, a summary follows:

- `verifyPrices` takes as input a list of trades and checks each trade price is computed according to `MMalgorithm`.
- `verification` takes as input the ticket received by a trader, the head of the hash chain and the list of trades posted at the end of an epoch to the BB by MM, and checks whether the ticket appears in the hash chain and its consistency of trades list with hash chain.
- `checkBB` checks BB for valid tickets and runs `verification` for each of them. If `verification` outputs 0, the procedure outputs 0 as well.

In Appendix C we formally prove the following theorem:

Theorem 1. *Assuming that unforgeable signatures, and collision resistent hash functions exist, $\Pi^{\texttt{trade}}$ realizes $\mathcal{F}^{\texttt{trade}}$ in the $(\mathcal{F}_{RO}, BB)$-hybrid world.*

4 Combining $\mathcal{F}^{\texttt{trade}}$ with Σ-Exchange Protocols.

We observe that if, in the realization of $\mathcal{F}^{\texttt{trade}}$, we replace the BB with a blockchain that supports smart contracts, then a smart contract can act as a party registered to $\mathcal{F}^{\texttt{trade}}$ that can query $\mathcal{F}^{\texttt{trade}}$ with the

[6] publishing can be done cheaply e.g. by only posting the hash on the blockchain and providing hash-preimages on demand.

$\mathtt{SC}_{\mathtt{account}}$

State: The public key $\mathtt{pk}^{\varXi}_{\mathtt{MM}}$ and the integer Y. The contract remains active until round $T_{\mathtt{MM}}$.
On any payment toward $\mathtt{pk}^{\varXi}_{\mathtt{MM}}$: If the balance of $\mathtt{pk}^{\varXi}_{\mathtt{MM}}$ after the payment is less than Y then accept the payment, else reject the payment.

Fig. 6. The contract does not allow MM to gain more than $Y\varXi$,

command getTrades. We can program this smart contract in such a way that if the output of $\mathcal{F}^{\mathtt{trade}}$ is $\perp$ then the MM is penalized. In our final protocol the traders and MM run a Σ-trade protocol Π, and in parallel, invoke $\mathcal{F}^{\mathtt{trade}}$ with the same information as input i.e. the prices, quantity and the type of the trades used in the execution of Π. Once that the output of $\mathcal{F}^{\mathtt{trade}}$ is generated, we can rely on a smart contract to check that the trades are consistent with the transactions generated by Π. If this is not the case then MM can be penalized. More precisely, to punish a misbehaving MM we require MM to create a smart contract $\mathtt{SC}_{\mathtt{penalize}}$ which locks a collateral z. $\mathtt{SC}_{\mathtt{penalize}}$, if queried by any party, inspects the output of $\mathcal{F}^{\mathtt{trade}}$ and if it is $\perp$ then $\mathtt{SC}_{\mathtt{penalize}}$ burns the collateral of MM. Otherwise $\mathtt{SC}_{\mathtt{penalize}}$ checks whether the trades from $\mathcal{F}^{\mathtt{trade}}$ are consistent with the transactions generated by MM on the Ethereum blockchain with respect to the wallet addresses $(\mathtt{pk}^{\varXi}_{\mathtt{MM}}, \mathtt{pk}^{\tilde{T}}_{\mathtt{MM}})$. If they are not, $\mathtt{SC}_{\mathtt{penalize}}$ burns the collateral. We note that this contract is expensive to execute (in terms of gas cost). However, if MM and the traders follow the protocol nobody will ever invoke it. On the other hand, if MM misbehaves then a trader will detect it (from the output of $\mathcal{F}^{\mathtt{trade}}$) and will invoke $\mathtt{SC}_{\mathtt{penalize}}$. We incentivize the honest invocations of $\mathtt{SC}_{\mathtt{penalize}}$ by transferring a small portion of the locked collateral to the calling party before burning the rest of it.

$\mathtt{SC}_{\mathtt{penalize}}$

State: $z\varXi$ and reward$\varXi$ locked for time T, the public key $\mathtt{pk}^{\varXi}_{\mathtt{MM}}$
Checking that MM is advancing:
- Act as a party P registered to $\mathcal{F}^{\mathtt{trade}}$ which receives no input.

Upon receiving the input detected from a party P_i with public key $\mathtt{pk}^{\varXi}_{i}$:
- Send (getTrades) to $\mathcal{F}^{\mathtt{trade}}$.
- Upon receiving output from $\mathcal{F}^{\mathtt{trade}}$, if output $= \perp$ then
  - move reward$\varXi$ to $\mathtt{pk}^{\varXi}_{i}$ and destroy $z\varXi$, else

  Else, parse output as (Trades, e) and check that each trade in Trades corresponds to a transaction on the ethereum chain. If that is the case then do nothing, else move reward$\varXi$ to $\mathtt{pk}^{\varXi}_{i}$ and destroy the $z\varXi$.

Fig. 7. Smart contract $\mathtt{SC}_{\mathtt{penalize}}$ that penalize MM in the case where $\mathcal{F}^{\mathtt{trade}}$ outputs $\perp$. The gas fee required to run the contract on the input detected is payed by the contract's caller. We assume that this fee is strictly less than reward (e.g., reward is 10 times the gas fee).

To finish exposition, we need to introduce yet another smart contract, $\mathtt{SC}_{\mathtt{account}}$. This contract, too, is created by MM. It checks if the transactions that pay the MM's account exceed a certain value Y. If this is the case, then the contract blocks additional payment towards MM. Hence it bounds the amount of commodities that MM can trade, We do it to prevent a situation where profit of the MM exceeds the collateral and thus makes it rational to misbehave (and get penalized). Observe that no malicious (even irrational) MM can steal money from the traders. The worst that MM can do is to frontrun the traders (by letting $\mathcal{F}^{\mathtt{trade}}$ output $\perp$) or avoid posting transactions that allow the settling of the trades. Both these types of misbehavior is caught by $\mathtt{SC}_{\mathtt{penalize}}$ and $\mathtt{MM}$ loses collateral. Thus, if we set Y to be smaller than the collateral of $\mathtt{SC}_{\mathtt{penalize}}$, then it is not a viable strategy for a rational $\mathtt{MM}$ to be penalized by means of $\mathtt{SC}_{\mathtt{penalize}}$.

$\Pi^{\mathtt{full}}$

$\mathtt{MM}$'s initial state: public keys $(\mathtt{pk}^{\Xi}_{\mathtt{MM}}, \mathtt{pk}^{\check{T}}_{\mathtt{MM}})$ with the corresponding secret keys $(\mathtt{sk}^{\Xi}_{\mathtt{MM}}, \mathtt{sk}^{\check{T}}_{\mathtt{MM}})$ and the smart contracts $\mathtt{SC}_{\mathtt{penalize}}, \mathtt{SC}_{\mathtt{account}}$.

$\mathbf{P_i}$'s initial state: the public keys of the MM $(\mathtt{pk}^{\Xi}_{\mathtt{MM}}, \mathtt{pk}^{\check{T}}_{\mathtt{MM}})$, his own public keys $(\mathtt{pk}^{\Xi}_{i}, \mathtt{pk}^{\check{T}}_{i})$ with the corresponding secret keys $(\mathtt{sk}^{\Xi}_{i}, \mathtt{sk}^{\check{T}}_{i})$, the smart contract $\mathtt{SC}_i$ which can be invoked by $\mathtt{MM}$ up to round T_i with $T_i << T$[a] and a transaction identifier $\mathtt{ID}$ initialized to 0.

P_i. Before interacting with $\mathtt{MM}$ check that $\mathtt{MM}$ has created a smart-contracts $\mathtt{SC}_{\mathtt{penalize}}, \mathtt{SC}_{\mathtt{account}}$ prescribed in Figs. 6 and 7. Let x be the amount of Ξ that P_i wants exchange for $\check{T}$. Send $(\mathtt{request}, \mathtt{pk}^{E}_{i})$ to $\mathcal{F}^{\mathtt{trade}}$. Upon receiving $(\mathtt{price}^{\Xi\rightarrow\check{T}}, \mathtt{price}^{\check{T}\rightarrow\Xi})$ from $\mathcal{F}^{\mathtt{trade}}$, if the prices are not good accordingly to the P_i's strategy, then send $\mathtt{NO\text{-}TRADE}$ to $\mathcal{F}^{\mathtt{trade}}$, else do the following

- Compute $\mathtt{ID} \leftarrow \mathtt{ID} + 1$ and $y \leftarrow x \cdot \mathtt{askedPrice}$ and $\sigma_1 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}^{E}_{i}, x||y||\mathtt{pk}^{E}_{B}||\mathtt{ID})$
- Define $\mathtt{trade}_i$ as the concatenation of $(x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}^{\check{T}}_{i})$ and send $(\mathtt{ok}, \mathtt{trade}_i)$ to $\mathcal{F}^{\mathtt{trade}}$.

P_i. On each round send $\mathtt{getTrade}$ to $\mathcal{F}^{\mathtt{trade}}$. If $\mathcal{F}^{\mathtt{trade}}$ replies with $\perp$ then invoke $\mathtt{SC}_{\mathtt{penalize}}$ with the input $\mathtt{detected}$, else let $(\mathtt{Trades}, \mathtt{e}_i)$ be the output of $\mathcal{F}^{\mathtt{trade}}$ and do the following.

- If T_i rounds have passed (i.e., $\mathtt{SC}_i$ cannot be invoked anymore), $\mathtt{trade}_i$ belongs to $\mathtt{Trades}$ and $\mathtt{SC}_i$ has never stored in $\mathtt{usedIDs}$ the identifier $\mathtt{ID}$ then invoke $\mathtt{SC}_{\mathtt{penalize}}$ with the input $\mathtt{detected}$.[b]

$\mathtt{MM}$. Upon receiving $\mathtt{pk}^{E}_{i}$ from $\mathcal{F}^{\mathtt{trade}}$ do the following steps.

- If P_i has created a contract $\mathtt{SC}_i$ accordingly to Fig. 1 then continue, otherwise stop interacting with P_i.
- Sends $(\mathtt{setPrice}, \mathtt{price}^{\Xi\rightarrow\check{T}}, \mathtt{price}^{\check{T}\rightarrow\Xi})$ to $\mathcal{F}^{\mathtt{trade}}$.

$\mathtt{MM}$. Upon receiving $(\mathtt{ok}, \mathtt{trade}_i)$ from $\mathcal{F}^{\mathtt{trade}}$, parse $\mathtt{trade}_i$ as $x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}^{\check{T}}_{i}$ and do the following steps.

- If $\mathtt{Ver}(\mathtt{pk}^{\Xi}_{i}, \sigma_1, x||y||\mathtt{pk}^{\Xi}_{\mathtt{MM}}||\mathtt{ID}) = 1$ and $y = x \cdot \mathtt{askedPrice}$ and there are enough rounds to post a transaction and invoke $\mathtt{SC}_i$ then continue with the following steps, else send $(\mathtt{setTrade}, P_i, 0)$ to $\mathcal{F}^{\mathtt{trade}}$.
- Compute $\sigma_2 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}^{\Xi}_{\mathtt{MM}}, x||y||\mathtt{pk}^{\Xi}_{i}||\mathtt{ID})$.
- Post a transaction $\mathtt{trx}$ with the identifier $\mathtt{ID}$ in its payload that moves $y\check{T}$ from $\mathtt{pk}^{\check{T}}_{\mathtt{MM}}$ toward $\mathtt{pk}^{\check{T}}_{i}$.
- Invoke $\mathtt{SC}_i$ using the input $(x, y, \mathtt{ID}, \sigma_1, \sigma_2)$.
- Send $(\mathtt{setTrade}, P_i, 1)$ to $\mathcal{F}^{\mathtt{trade}}$.

$\mathtt{MM}$. Every Δ rounds $\mathtt{MM}$ sends $\mathtt{setOutput}$ to $\mathcal{F}^{\mathtt{trade}}$.

[a] We require T_i to be smaller than T (the time-lock of $\mathtt{SC}_{\mathtt{penalize}}$) to give time to a party to trigger the complain mechanism of $\mathtt{SC}_{\mathtt{penalize}}$. The exact relation between T_i and T is given by the liveness parameter of the underling blockchain.

[b] This capture the scenario where $\mathtt{MM}$ is honest in the execution of $\mathcal{F}^{\mathtt{trade}}$ but he does not post the transaction that triggers the actual trade on-chain.

Fig. 8. Full market-maker protocol with identifiable (and punishable) misbehaviour.

The formal description of our final protocol $\Pi^{\mathtt{full}}$ is in Fig. 8. We describe the case when traders only want to buy $\check{T}$ for Ξ. $\Pi^{\mathtt{full}}$ combines the functionality $\mathcal{F}^{\mathtt{trade}}$ and the Σ-trade protocol. We specify $\mathtt{SC}_{\mathtt{penalize}}$ in Fig. 7. $\mathtt{SC}_{\mathtt{penalize}}$ acts like a party registered to $\mathcal{F}^{\mathtt{trade}}$ who, when queried sends $\mathtt{getTrades}$ to $\mathcal{F}^{\mathtt{trade}}$ and decides whether the MM misbehaved. Let T be the number of rounds for which $\mathtt{SC}_{\mathtt{penalize}}$ has locked the collateral, we can claim the following:

Theorem 2. *If there is at least one honest party P_i then, within the first T rounds one of the following occurs with overwhelming probability:*

1. *the $\mathcal{F}^{\mathtt{trade}}$ outputs $\perp$ and the collateral locked in $\mathtt{SC}_{\mathtt{penalize}}$ by $\mathtt{MM}$ is burned;*

2. *the $\mathcal{F}^{\texttt{trade}}$ is not $\perp$ but there is not a perfect correspondence between the trades contained in the output of $\mathcal{F}^{\texttt{trade}}$ and the transactions that appear on the blockchain E with respect to* MM*'s public keys. Moreover, the collateral locked in* $\texttt{SC}_{\texttt{penalize}}$ *is burned;*
3. *the $\mathcal{F}^{\texttt{trade}}$ is not $\perp$, there is a perfect correspondence between the trades contained in the output of $\mathcal{F}^{\texttt{trade}}$ and the transactions that appear on the blockchain E with respect to* MM*'s public keys and all the collateral remains locked in* $\texttt{SC}_{\texttt{penalize}}$ *for T rounds.*

For appropriate parameters in the smart contracts, and assuming the market-maker maximizes his amount of $\varXi$, we can argue that the first two cases in Theorem 2 happen with negligible probability. Indeed, let α be the gas cost to run $\texttt{SC}_{\texttt{penalize}}$ with the input `detected`, let `reward` be reward that could be given to a party calling $\texttt{SC}_{\texttt{penalize}}$, let z be the locked collateral in $\texttt{SC}_{\texttt{penalize}}$ and let Y be the maximum amount of $\varXi$ that MM can earn at $\texttt{pk}^{\varXi}_{\texttt{MM}}$. If there is at least one honest party P_i, $\texttt{reward} > \alpha$, and $z > Y$ then for every rational market-maker the probability of occurrence of the first two cases of Theorem 2 is negligible.

5 Incentive Compatibility of Market Maker (MM)

We use myopic-greedy market maker from [13] in our construction. Here we provide an overview, see Appendix D for details and proofs. Let market maker's distribution $p_t(v)$ quantify market maker's information on the true value V after t trades, our market maker has the following properties:

– The market discovers the originally unknown true value of the commodity based on trades with traders who arrive with imperfect information. Empirically, the speed of this convergence is illustrated in [13] and follows the standard $1/t$ convergence for Bayesian updates.
– The market maker uncertainty converges to 0. The market maker recovers the true value in expectation, and also becomes more certain of it. Again, this convergence is standard for Bayesian updates.
– In equilibrium, the market maker spread that produces maximum single step profit monotonically increases with the variance of its distribution, which converges to zero. Hence the bid-ask spread converges to a minimum possible for a profit maximizing market maker. A market maker who knows V can always make more expected profit than a market maker who does not.

The last bullet above is essentially the intuition behind why an optimal market maker has no incentive to manipulate prices. The maximum profit is made when the market maker knows the true value V. Hence the market maker is incentivized to discover the true value V as quickly as possible. The only information available on the V is through the un-manipulated trader signals x_t.

We now present the main theorem (proved in Appendix D):

Theorem 3 (Incentive compatibility). *A rational profit-seeking market maker has no incentive to manipulate the price given knowledge that some trader wishes to place a trade and the direction (buy/sell) of the trade being known.*

The following lemma states that it is suboptimal for the market maker in our setting to ignore trades without knowledge of other trades.

Lemma 1. *A rational profit-seeking market maker which receives sequential trades, has no incentive to disregard completed trades, even when the direction of the following trade is known.*

6 Evaluation

We implemented $\varPi^{\texttt{trade}}$ to trade Ether and ERC20 tokens on Ethereum (see Appendix E for implementation details). Table 1 lists the gas costs. Note that the cost of executing one trade is the sum of the costs of *execute* methods of the *SellerContract* and the *BuyerContract*.

6.1 Experiment Setup

To measure throughput, we ran several experiments on a consumer laptop equipped with Core i7-10510U 1.80 GHz CPU and 8GB of RAM running Ubuntu 20.04. Recall that in our fair trade protocol $\Pi^{\texttt{trade}}$ (see Fig. 5), the buyer P_i first sends its public keys to the seller MM. Then the seller responds by sending a ticket and the current prices. Both of these messages can be computed very cheaply. Concretely creating the first message takes less than 50ms (for each party) in our setup. Then the buyer either responds with NO-TRADE or trade. This is still cheap and can be done in less than 50ms. Now the seller must respond to the trade offer. If this offer is NO-TRADE, the buyer needs to perform very little work (concretely less than 50ms). However, if the offer is trade, the seller must verify and create signatures, perform balance checks on appropriate assets and create/broadcast a transaction for the trade. These operations are slow (especially the ones that involve communicating with an Ethereum node). Concretely, it takes $\approx$ 350ms to prepare this message. Lastly, we observed that the typical round trip time from buyer $\rightarrow$ seller $\rightarrow$ buyer is less than 100ms.

Our goal was to observe the system's throughput in the following adversarial scenarios. The first is the *Standard DoS attack*. Here, a malicious buyer floods the system with ticket requests and then stops responding, slowing the system down. To this end, we performed the following experiment: n buyers connect to the seller. The seller responds (with ticket and prices) to them in the order they connect. Upon receiving the ticket (and prices) from the seller, an honest buyer will execute a trade (i.e., the trade scenario). On the other hand, a corrupt buyer will stop responding. After a timeout τ, the seller will assume a NO-TRADE response, execute the NO-TRADE scenario, and move to the next buyer. We ran experiments with $n = 300$ buyers, repeating 5 times and reporting the average measurement. Note that relatively few repetitions of the experiments are not a concern because of low variance of the measured values.

The other attack scenario is a *Worst-case Throughput attack*. The setting remains the same as above with one difference: the malicious buyer now waits until just before the timeout and then responds with a trade response. This strategy is more effective at slowing down the system than the standard DoS attack. The reason why it is the case is discussed in the next section.

6.2 Analysis of Results

The results of the experiments for *Standard DoS attack* are summarized in Fig. 9, and for *Worst-case Throughput attack*, in Fig. 10. We observe that in Standard DoS attack (cf. Fig. 9) with no corruptions, throughput is over 200 trades/min. Recall that the gas cost of a trade is $101K$, Assuming *block gas limit* is $12M$ (i.e. the current limit) and block generation delay of $15s$, Upper bound on throughput is 475 trades/min. This upper bound assumes no other application (except ours) competes for block space. Keeping this in mind, achieving over 200 trades/min is an excellent throughput. This number is higher than Uniswap's[36] average throughput/min on its highest daily volume (cf. Section 6.3). Recall also that this throughput is achieved on a consumer laptop. A high-end server (typical machine for such use-cases) will yield higher throughput.

Interestingly, at low values of τ, the throughput of the system goes up with the number of corruptions. This is not an anomaly. If a malicious trader does not respond within the timeout τ, the seller assumes a NO-TRADE response which takes about one-seventh of the time it takes to execute trade response (cf. Section 6.1). This means at low values of τ ($\tau <$ execution-timet of trade) and some corruptions, some (i.e. the corrupt) trades are cheaper to execute compared to when there are no corruptions (because all honest players trigger the trade scenario). This effect disappears as soon as the value of the timeout τ goes near and above the execution-time of the trade scenario. While setting a low timeout τ may seem a good idea to defend against malicious parties, it should not be less than the typical round-trip time (100ms in our trials), otherwise it will cause timeouts for honest players. Note also that a trader needs to setup a smart contract and register with the market maker before commencing trading. Therefore, Sybil attack is not trivial and repeat offenders may be blacklisted.

A better attack strategy would be for a malicious buyer to wait until just before the timeout (for maximum slowdown) and then respond with trade response; to trigger the more expensive (in running time) scenario for seller. This strategy removes the above mentioned advantage. Concretely, a malicious seller would wait

Table 1. Gas Costs of Seller and Buyer Contracts.

Methods		**Gas**	**USD†**
Contract	Method	47gwei/gas∗	3,284.20 usd/eth∗
BuyerContract	claimExpiry	31,619	4.88
	execute	67,984	10.50
SellerContract	execute	33,456	5.16
Deployments			
BuyerContract		1,082,529	167.09
SellerContract		836,341	129.09

∗ Prices taken from https://coinmarketcap.com/ on 2021-09-10.
† USD cost is a bad measure of contract complexity. We list it to be consistent with other work.

until he has has just enough time left for one round-trip (100ms in our setup). Thus the amount of time he should wait, `delayBudget`, can be computed as `delayBudget` $= \tau -$ `RoundTripTime`. The negative effect of such attack is seen in Fig. 10. The throughput has gone down for all values of τ. Importantly though, observe that the x-axis in Fig. 10 starts at $\tau = 200$. This is because at $\tau = 100$, the `delayBudget` of the adversary is 0 i.e., he has to respond immediately and there is no longer a difference between an honest buyer and a malicious buyer.

In conclusion, the choice for value of τ should be the typical round-trip time (with some noise). This prevents throughput-loss even against a determined adversary who wants to pay (via `trade` responses) to slow down the system. Finally, consider that in real life some honest sellers may also respond with `NO-TRADE` e.g., if the prices are not favorable. Hence, the value of 205 trades per minute at $\tau = 100$ should be considered the lower bound.

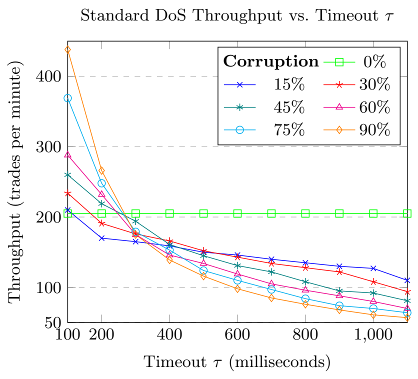

Fig. 9. Standard DoS Attack Throughput (at 0% to 90% corruption). Values average of 5 runs. Note that x-axis starts at ms, this is the typical RTT, and any $\tau < 100$ may cause timeouts for honest players.

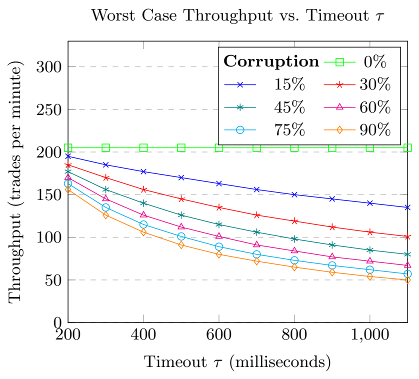

Fig. 10. Worst-case Throughput (at 0% to 90% corruptions). Note: x-axis starts at 200ms because malicious player needs a budget of at least RTT (100ms here) to respond without risking timeout.

6.3 Comparison with Uniswap

A summary of FairMM and Uniswap is presented in Table 2. See Appendix F for our analysis of Uniswap gas costs. At the time of this writing, the throughput values in v2 are higher than v3 e.g. the highest daily volume 3 times more for v2 (251K txns) than v3 (71K txns). So, we compare against v2.

Table 2. Comparison Summary

Feature	FairMM	Uniswap
Front Running Resilience	**Yes**	No
Gas Price Auctions	**No**	Yes
Miner Influence	**No**	Yes
Trade Execution (seconds)	$\approx$ **0.30**	≥ 15
Average Trade Cost (K)	$\approx$ **101**	$\approx 141^*$
Max Trade Cost (K)	$\approx$ **101**	$\approx 1,316^\dagger$
Max Throughput‡	$\approx$ **475**	≈ 340

$*$ Based on average cost of 1M trade transactions (block 12,162,664 to 12,231,464). Trades are calls to swap methods of V2Router02.

† Txn: https://etherscan.io/tx/0xa87b492f2945d2a99ca1f8e2d9530599c040f00c3257f989f9c2822e20b2ed5e). There may be more expensive transactions outside our dataset.

‡ in trades/minute. Theoretical upper bound on throughput based on average trade cost, assuming $12M$ block gas limit on Ethereum network.

First, Uniswap (or any existing market maker, centralized or decentralized) has no defense against front-running attacks without additional trust/hardware assumptions. Our construction resolves this long-standing problem by ensuring that the market maker cannot reorder trades without getting caught.

Second, our trade execution time is bounded by the round trip time of the network, about 350ms. In contrast, Uniswap trades are executed by the miners as part of mining a block. At the time of this writing, etherscan shows high fees transactions (ones that get picked up the soonest) take about 30 seconds. One can safely say that a trade in Uniswap takes at least 15 seconds (half of the value on etherscan). This is much larger than the approx. 0.3 seconds in our system.

Third, in Uniswap and similar systems, miners are free to reorder trades. This gives them a profit opportunity e.g. including favorable trades first. On the contrary, in our system the trade order is fixed before the corresponding transactions are broadcast to the blockchain, nullifying miners' influence. Moreover—because of the above mentioned miners' influence—traders on Uniswap have an incentive to pay high *gas price* to get their trade included sooner. In fact, since the traders can see other traders' activity, they can actively compete with one another. Such trading behavior induces the, so called, *gas price auction*s attack. Gas price auctions needlessly raise transaction cost for everyone (not just the traders). Transactions in our system are merely moving the funds and may be mined in any order. There is no incentive to pay higher than usual gas price.

Fourth, gas cost in Uniswap is variable. We observed an average gas cost of $141K$. It can be much higher depending on the trade e.g. over $1,316K$ for txn 0xa87b492f2945d2a99ca1f8e2d9530599c040f00c3257f989f9c2822e20b2ed5e. Recall that Uniswap is specifically designed and optimized for Ethereum. On the other hand, our system design is general and lacks aggressive optimizations. Yet, the gas cost of our system is constant at $101K$. Notwithstanding, even if the gas cost of Uniswap transactions were much lower than ours, Uniswap's transactions would still be more costly in Ethers because of the gas price auctions mentioned above.

Finally, based on the average trade gas cost and assuming a block gas limit of $12M$, the maximum throughput of Uniswap is ≈ 340 trades per minute. This is less than our upper bound of 475. Concretely, highest daily volume[7] on Uniswap has been $\approx 251K$ transactions. On average, this means about 174 trades per minute. Importantly, this throughput is achieved in a scenario where all trade data is locally available. Our construction on the other hand, communicates with the traders in real time. The fact that this communication

[7] https://etherscan.io/address/0x7a250d5630b4cf539739df2c5dacb4c659f2488d#analytics

happens sequentially—on first come first served basis—negatively affects our throughput. Despite this, we achieve at least 200 trades per minute (higher than the highest volume Uniswap). We stress that this throughput was achieved on a mid-range consumer machine. A computationally powerful server will increase throughput further. Therefore, we do not see it as a major problem in practice.

References

[1] Lawrence R. Glosten and Paul R. Milgrom. “Bid, ask and transaction prices in a specialist market with heterogeneously informed traders”. In: *Journal of Financial Economics* 14.1 (1985), pp. 71–100.

[2] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th FOCS*. IEEE Computer Society Press, Oct. 1986, pp. 162–167.

[3] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, May 1987, pp. 218–229.

[4] Lawrence R. Glosten. “Insider Trading, Liquidity, and the Role of the Monopolist Specialist”. In: *The Journal of Business* 62.2 (1989), pp. 211–235.

[5] N. Asokan, Victor Shoup, and Michael Waidner. “Optimistic Fair Exchange of Digital Signatures (Extended Abstract)”. In: *EUROCRYPT’98*. Ed. by Kaisa Nyberg. Vol. 1403. LNCS. Springer, Heidelberg, May 1998, pp. 591–606.

[6] Christian Cachin and Jan Camenisch. “Optimistic Fair Secure Computation”. In: *CRYPTO 2000*. Ed. by Mihir Bellare. Vol. 1880. LNCS. Springer, Heidelberg, Aug. 2000, pp. 93–111.

[7] Ran Canetti. “Security and Composition of Multiparty Cryptographic Protocols”. In: *Journal of Cryptology* 13.1 (Jan. 2000), pp. 143–202.

[8] Ran Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[9] Ran Canetti. *Universally Composable Signatures, Certification and Authentication*. Cryptology ePrint Archive, Report 2003/239. https://eprint.iacr.org/2003/239. 2003.

[10] Justin Wolfers and Eric Zitzewitz. “Prediction Markets”. In: *Journal of Economic Perspectives* 18.2 (June 2004), pp. 107–126.

[11] Sanmay Das. “A learning market-maker in the Glosten–Milgrom model”. In: *Quantitative Finance* 5.2 (2005), pp. 169–180.

[12] D. Pennock and R. Sami. “Computational aspects of prediction markets”. In: *Algorithmic Game Theory*. Cambridge University Press, 2007.

[13] Sanmay Das and Malik Magdon-Ismail. “Adapting to a Market Shock: Optimal Sequential Market-Making”. In: *Proc. Advances in Neural Information Processing Systems (NIPS)*. Dec. 2008, pp. 361–368.

[14] Alptekin Küpçü and Anna Lysyanskaya. “Usable Optimistic Fair Exchange”. In: *CT-RSA 2010*. Ed. by Josef Pieprzyk. Vol. 5985. LNCS. Springer, Heidelberg, Mar. 2010, pp. 252–267.

[15] Jonathan Katz et al. “Universally Composable Synchronous Computation”. In: *TCC 2013*. Ed. by Amit Sahai. Vol. 7785. LNCS. Springer, Heidelberg, Mar. 2013, pp. 477–498.

[16] Iddo Bentov and Ranjit Kumaresan. “How to Use Bitcoin to Design Fair Protocols”. In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, Aug. 2014, pp. 421–439.

[17] Christian Decker and Roger Wattenhofer. “A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels”. In: *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 9212*. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 3–18.

[18] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. “The Bitcoin Backbone Protocol: Analysis and Applications”. In: *EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. LNCS. Springer, Heidelberg, Apr. 2015, pp. 281–310.

[19] Charanjit S. Jutla. *Upending Stock Market Structure Using Secure Multi-Party Computation.* Cryptology ePrint Archive, Report 2015/550. https://eprint.iacr.org/2015/550. 2015.

[20] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. “Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts”. In: *ESORICS 2016, Part II.* Ed. by Ioannis G. Askoxylakis et al. Vol. 9879. LNCS. Springer, Heidelberg, Sept. 2016, pp. 261–280.

[21] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. “Fair and Robust Multi-party Computation Using a Global Transaction Ledger”. In: *EUROCRYPT 2016, Part II.* Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. LNCS. Springer, Heidelberg, May 2016, pp. 705–734.

[22] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* 2016.

[23] Christian Badertscher et al. “Bitcoin as a Transaction Ledger: A Composable Treatment”. In: *CRYPTO 2017, Part I.* Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, Aug. 2017, pp. 324–356.

[24] Matteo Campanelli et al. “Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services”. In: *ACM CCS 2017.* Ed. by Bhavani M. Thuraisingham et al. ACM Press, Oct. 2017, pp. 229–243.

[25] Arka Rai Choudhuri et al. “Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards”. In: *ACM CCS 2017.* Ed. by Bhavani M. Thuraisingham et al. ACM Press, Oct. 2017, pp. 719–728.

[26] Matthew Green and Ian Miers. “Bolt: Anonymous Payment Channels for Decentralized Currencies”. In: *ACM CCS 2017.* Ed. by Bhavani M. Thuraisingham et al. ACM Press, Oct. 2017, pp. 473–489.

[27] Rafael Pass, Lior Seeman, and abhi shelat. “Analysis of the Blockchain Protocol in Asynchronous Networks”. In: *EUROCRYPT 2017, Part II.* Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10211. LNCS. Springer, Heidelberg, Apr. 2017, pp. 643–673.

[28] Will Warren and Amir Bandeali. *0x: An open protocol for decentralized exchange on the Ethereum blockchain.* 2017.

[29] AirSwap. *AirSwap.* 2018.

[30] Ether Delta. *EtherDelta.* 2018.

[31] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. “FairSwap: How To Fairly Exchange Digital Goods”. In: *ACM CCS 2018.* Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 967–984.

[32] Maurice Herlihy. “Atomic Cross-Chain Swaps”. In: *37th ACM PODC.* Ed. by Calvin Newport and Idit Keidar. ACM, July 2018, pp. 245–254.

[33] IDEX. *IDEX.* 2018.

[34] Kyber. *Kyber.* 2018.

[35] Raiden Network. *What is Raiden Network?* 2018.

[36] Uniswap. *Uniswap Exchange Protocol.* 2018.

[37] Bitcoin Wiki. *Zero Knowledge Contingent Payment.* 2018.

[38] Iddo Bentov et al. “Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware”. In: *ACM CCS 2019.* Ed. by Lorenzo Cavallaro et al. ACM Press, Nov. 2019, pp. 1521–1538.

[39] Georg Fuchsbauer. *WI Is Not Enough: Zero-Knowledge Contingent (Service) Payments Revisited.* Cryptology ePrint Archive, Report 2019/964. https://eprint.iacr.org/2019/964. 2019.

[40] Runchao Han, Haoyu Lin, and Jiangshan Yu. *On the optionality and fairness of Atomic Swaps.* Cryptology ePrint Archive, Report 2019/896. https://eprint.iacr.org/2019/896. 2019.

[41] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. “Cross-Chain Deals and Adversarial Commerce”. In: *Proc. VLDB Endow.* 13.2 (Oct. 2019), pp. 100–113.

[42] Rami Khalil, Arthur Gervais, and Guillaume Felley. *TEX - A Securely Scalable Trustless Exchange.* Cryptology ePrint Archive, Report 2019/265. https://eprint.iacr.org/2019/265. 2019.

[43] Alexei Zamyatin et al. “XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets”. In: *2019 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2019, pp. 193–210.

[44] Curve. *Curve.* 2020.

[45] Philip Daian et al. "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability". In: *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 910–927.

[46] Apoorvaa Deshpande and Maurice Herlihy. "Privacy-Preserving Cross-Chain Atomic Swaps". In: *FC 2020 Workshops*. Ed. by Matthew Bernhard et al. Vol. 12063. LNCS. Springer, Heidelberg, Feb. 2020, pp. 540–549.

[47] Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. "OptiSwap: Fast Optimistic Fair Exchange". In: *ASIACCS 20*. Ed. by Hung-Min Sun et al. ACM Press, Oct. 2020, pp. 543–557.

[48] Joël Gugger. *Bitcoin-Monero Cross-chain Atomic Swap*. Cryptology ePrint Archive, Report 2020/1126. https://eprint.iacr.org/2020/1126. 2020.

[49] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. "The Arwen Trading Protocols". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Cham: Springer International Publishing, 2020, pp. 156–173.

[50] Andrey Sobol. *Frontrunning on Automated Decentralized Exchange in Proof Of Stake Environment*. Cryptology ePrint Archive, Report 2020/1206. https://eprint.iacr.org/2020/1206. 2020.

[51] Bitcoin Wiki. *Atomic Swap*. 2020.

[52] Bitcoin Wiki. *Hash Time Locked Contracts*. 2020.

[53] Ghada Almashaqbeh et al. *Gage MPC: Bypassing Residual Function Leakage for Non-Interactive MPC*. Cryptology ePrint Archive, Report 2021/256. https://eprint.iacr.org/2021/256. 2021.

[54] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. "Maximizing Extractable Value from Automated Market Makers". In: *CoRR* abs/2106.01870 (2021).

[55] Carsten Baum, Bernardo David, and Tore Frederiksen. *P2DEX: Privacy-Preserving Decentralized Cryptocurrency Exchange*. Cryptology ePrint Archive, Report 2021/283. https://eprint.iacr.org/2021/283. 2021.

[56] Lorenz Breidenbach et al. *Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks*. 2021.

[57] Flashbots. *Flashbots*. 2021.

[58] Gnosis. *Introducing Gnosis Protocol V2 and Balancer-Gnosis-Protocol*. 2021.

[59] Chrysoula Stathakopoulou et al. "Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs". In: *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 2021, pp. 34–45.

[60] Liyi Zhou, Kaihua Qin, and Arthur Gervais. "A2MM: Mitigating Frontrunning, Transaction Reordering and Consensus Instability in Decentralized Exchanges". In: *CoRR* abs/2106.07371 (2021).

[61] Liyi Zhou et al. "High-Frequency Trading on Decentralized On-Chain Exchanges". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 428–445.

[62] Bancor. *Bancor Network*.

A Preliminaries and Notation

We use "=" to check equality of two different elements (i.e. if $a = b$ then...) and "$\leftarrow$" as the assigning operator (e.g. to assign to a the value of b we write $a \leftarrow b$). A randomized assignment is denoted with $a \overset{\$}{\leftarrow} A$, where A is a randomized algorithm and the randomness used by A is not explicit. We call a function $\nu : \mathbb{N} \rightarrow \mathbb{R}^+$ *negligible* if for every positive polynomial $p(\lambda)$, a $\lambda_0 \in \mathbb{N}$ exists, such that for all $\lambda > \lambda_0 : \nu(\lambda) < 1/p(\lambda)$.

A.1 Signatures

Definition 1 (Signature scheme [9]). *A triple of* PPT *algorithms* $(\texttt{Gen}, \texttt{Sign}, \texttt{Ver})$ *is called a* signature scheme *if it satisfies the following properties.*

Completeness: *For every pair* $(s, v) \overset{\$}{\leftarrow} \texttt{Gen}(1^\lambda)$, *and every* $m \in \{0,1\}^\lambda$, *we have that* $\Pr[\texttt{Ver}(v, m, \texttt{Sign}(s, m)) = 0] < \nu(\lambda)$.

Consistency (non-repudiation): *For any m, the probability that* $\mathtt{Gen}(1^\lambda)$ *generates* (s, v) *and* $\mathtt{Ver}(v, m, \sigma)$ *generates two different outputs in two independent invocations is smaller than* $\nu(\lambda)$.

Unforgeability: *For every* PPT $\mathcal{A}$*, there exists a negligible function* ν*, such that for all auxiliary input* $z \in \{0,1\}^\star$ *it holds that:*

$$\Pr[(s, v) \xleftarrow{\$} \mathtt{Gen}(1^\lambda); (m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathtt{Sign}(s,\cdot)}(z, v) \wedge \\ \mathtt{Ver}(v, m, \sigma) = 1 \wedge m \notin Q] < \nu(\lambda)$$

where Q denotes the set of messages whose signatures were requested by $\mathcal{A}$ to the oracle $\mathtt{Sign}(s, \cdot)$.

A.2 Blockhain and Smart-Contracts

Ethereum is arguably the most popular blockchain for smart-contracts. Its protocol keeps track of each address' balance. *Transaction*s are used to move funds between the addresses and to execute code of the smart-contracts. A *Smart-Contract* is some code that lives on the blockchain. Storing and running contracts requires resources from the miners. In order to pay them for their expenses, Ethereum has the concept of *Gas*. Each instruction in a smart-contract costs some gas units proportional to what it does. Transaction senders can specify the amount of *Wei* they are willing to pay per gas unit. This is called *Gas Price*.

A.3 Blockchain as a Bulletin Board

For convenience, whenever the blockchain is just used for recording events, we treat it as a *bulletin board* (BB). The bulletin board has a sequential-writing pattern where every string published on the bulletin board has a counter (its position) associated to it. We do not make any assumptions about the order in which issued transactions are recorded, other than what is implied by the standard chain quality and transaction liveness properties of ledgers (cf. [18, 27, 23].

Bulletin Board The bulletin board `BB` allows the following queries (all of which can be emulated in modern blockchains; cf. [25] for a formal description):

– `getCounter`. The bulletin board returns the current value of the counter t: $t \leftarrow \mathtt{BB}(\mathtt{getCounter})$.
– `post`. Upon receiving a value x, the bulletin board posts x and increments the counter t by 1. The value can be retrieved by querying the bulletin board on t: $t \leftarrow \mathtt{BB}(\mathtt{post}, x)$.[8]
– `getContent`. Upon receiving the input t, it returns the value stored at counter value t. If t is greater than the current counter value, it returns $\perp$, else, $x \leftarrow \mathtt{BB}(\mathtt{getContent}, t)$.

A.4 Security framework

The Universal Composability (UC) framework introduced by Canetti in [8] is a security model capturing the security of a protocol Π under the concurrent execution of arbitrary other protocols. All those other protocols and processes not related to the protocol Π go through an environment $\mathcal{Z}$. The environment has the power to decide the input that the parties should use to run the Π, and to see the output of these parties. In this framework there is also an adversary $\mathcal{A}$ for the protocol Π that decides the parties to be corrupted and can communicate with $\mathcal{Z}$ (who knows which parties have been corrupted by $\mathcal{A}$). The security in this model is captured by the simulation-based paradigm. Let $\mathcal{F}$ be the ideal functionality that should be realized by Π. The ideal functionality $\mathcal{F}$ can be seen as a trusted party that handles the entire protocol execution and tells the parties what they would output if they executed the protocol correctly. We consider the ideal process where the parties simply pass on inputs from the environment to $\mathcal{F}$ and hand what they receive to the environment. In the ideal process, we have an ideal process adversary $\mathcal{S}$. $\mathcal{S}$ does not learn

[8] In [25] the output of `BB` consists also of a tag, that can be used to prove that a value is part of the BB, without inspecting the BB using the command `getContent` defined below.

the content of messages sent from $\mathcal{F}$ to the parties, but is in control of when, if ever, a message from $\mathcal{F}$ is delivered to the designated party. $\mathcal{S}$ can corrupt parties and at the time of corruption it will learn all inputs the party has received and all outputs it has sent to the environment. As the real world adversary, $\mathcal{S}$ can freely communicate with the environment. We compare running the real protocol with running the ideal process and say that $\mathit{\Pi}$ UC-realizes $\mathcal{F}$ if no environment can distinguish between the two worlds. This means that the protocol is secure, if for any polynomial time $\mathcal{A}$ running in the real world with $\mathit{\Pi}$, there exists a polynomial time $\mathcal{S}$ running in the ideal process with $\mathcal{F}$, so no non-uniform polynomial time environment can distinguish the two worlds.

For our formal security arguments we use the simulation paradigm. The advantage of using simulation based security is that it supports composition which allows us to employ a constructive approach to protocol design.Our constructions are secure in Canetti's Universal Composability (UC) framework [8] (in fact, its synchronous version from [15, 21, 23].) Nonetheless to make the presentation more accessible to a non-UC expert we often use the convention and language similar to [7]. Concretely, we assume that all protocols proceed in rounds, where in each round: the uncorrupted parties generate their messages for the current round, as described in the protocol; then the messages addressed to the corrupted parties become known to the adversary; then the adversary generates the messages to be sent by the corrupted parties in this round; and finally, each uncorrupted party receives all the messages sent in this round. At the end of the computation all parties locally generate their outputs. Using [15] it is easy to project our statement to (synchronous) UC.

A.5 The Random Oracle Functionality

The Random Oracle Functionality $\mathcal{F}_{RO}$ As typically in cryptographic proofs the queries to hash function are modelled by assuming access to a random oracle functionality: Upon receiving a query $(\texttt{EVAL}, \texttt{sid}, x)$ from a registered party, if x has not been queried before, a value y is chosen uniformly at random from $\{0,1\}^\lambda$ (for security parameter λ) and returned to the party (and the mapping (x, ρ) is internally stored). If x has been queried before, the corresponding ρ is returned.

B Works on Fair Exchange

Several early works [2, 3, 5, 6, 14] have discussed the idea of fair exchange. In recent years, with the increasing popularity of crypto-currencies, protocols have been designed around them to take advantage of the blockchain guarantees [16, 21]. An interesting use-case of fair exchange is to enable sale/purchase of assets with cryptocurrency [37, 20, 24, 39]. The general scheme is to construct smart-contracts in such a way, that claiming payment would reveal the key to the buyer. Unfortunately, the large (often prohibitive) off-chain cost of this technique (when the predicate is complex and/or the asset is large) hinders their adoption in many practical applications. FairSwap [31] provided a more efficient solution in terms of off-chain costs. OptiSwap [47] further improved over [31] for the optimistic case and also provided a defense against grieving attack.

In addition to generic solutions for exchange assets, there has been interest in creating more efficient solutions for specific subset of asset types. The most popular choice for this subset has been other cryptocurrencies or tokens. These works broadly fall into the following two categories; 1) The works that deal with assets on the same blockchain e.g. Uniswap [36], 0x [28], AirSwap [29], EtherDelta [30], Bancor[62], Idex [33], Kyber [34], Curve [44], etc and 2) the ones that operate across the blockchains or crosschain. The crosschain research has mostly been restricted to 2 parties (on 2 blockchains). Transactions in this setting are generally called *crosschain swaps* and the problem is usually solved with Hash Time Lock Contracts (HTLCs) [51, 52, 32, 43, 40, 48, 46]. HTLC in the nutshell is similar to (the on-chain contract of) ZKCPs. The buyer locks assets for specified time, and before the lock expires, the seller produces a pre-image (or key) of a hash. The notion of *crosschain swap* was generalized—with definitions and first constructions—to *crosschain deals* among n parties (and m blockchains) in [41]. HTLCs technique has also been used to workaround the scalability problem of blockchains [17, 26, 49, 35, 22].

C Proof of Theorem 1

Proof. We divide the proof in two parts: the first is to deal with the adversarial MM (that we denote with $\mathtt{MM}^\star$) and the second to deal with the case where $\mathtt{MM}$ is honest. We denote with Q the set containing all couples of query-answer performed using the hash function (that we model as a random oracle) and propose the formal description of the ideal world market-maker adversary $\mathcal{S}_{\mathtt{MM}}$. The simulator $\mathcal{S}_{\mathtt{MM}}$ works as follows.

- Let $h_{\mathtt{start}} = h_{\mathtt{temp}} = h^\star = 0^\lambda$. Initialize $\mathtt{price}^{\Xi\to\check{T}} \leftarrow \mathtt{SP}^{\Xi\to\check{T}}$ and $\mathtt{price}^{\check{T}\to\Xi} \leftarrow \mathtt{SP}^{\check{T}\to\Xi}$. Initialize also two empty lists $\mathtt{reqList}$ and $\mathtt{Tickets}$.
- Upon receiving $(\mathtt{request}, P_i)$ from $\mathcal{F}^{\mathtt{trade}}$, send $(\mathtt{request}, \mathtt{pk}_i)$ to $\mathtt{MM}^\star$.
- Upon receiving $\mathtt{ticket}_1 = (h, \sigma, \mathtt{price}^{\Xi\to\check{T}}, \mathtt{price}^{\check{T}\to\Xi}, \mathtt{pk}_i)$ from $\mathtt{MM}^\star$ do the following steps.
  - If $\mathtt{Ver}(\mathtt{pk}_{\mathtt{MM}}, \sigma, h||\mathtt{price}^{\Xi\to\check{T}}||\mathtt{price}^{\check{T}\to\Xi}||\mathtt{pk}_i) = 0$ then ignore the message received from $\mathtt{MM}^\star$, continue as follows otherwise.
  - Add $\mathtt{ticket}_1$ to the list $\mathtt{Tickets}$ and send $(\mathtt{setPrice}, P_i, \mathtt{price}^{\Xi\to\check{T}}, \mathtt{price}^{\check{T}\to\Xi})$ to $\mathcal{F}^{\mathtt{trade}}$.
  - If P_i is an honest party then do the following.
    - Add (P_i, h) to $\mathtt{reqList}$.
    - Inspect Q to check if $h_{\mathtt{temp}}$ is a prefix of the chain with head h. If it is not then send $\mathtt{setAbort}$ to $\mathcal{F}^{\mathtt{trade}}$, else continue as follows.
    - For each couple of items $(\mathtt{pk}_j, \mathtt{trade}_j)$ encoded in the hash chain[9] that starts from $h_{\mathtt{temp}}$ and finishes in h do the following

      if P_j is an honest party then send $(\mathtt{setTrade}, P_j, \mathtt{trade}_j)$ to $\mathcal{F}^{\mathtt{trade}}$.

      else send $(\mathtt{setAdvTrade}, P_j, \mathtt{trade}_j)$ to $\mathcal{F}^{\mathtt{trade}}$.

    Set $h_{\mathtt{temp}} \leftarrow h$.
- Upon receiving (P_i, y) from $\mathcal{F}^{\mathtt{trade}}$, if P_i is honest then do the following.

  if $y = \mathtt{NO\text{-}TRADE}$ then send $\mathtt{NO\text{-}TRADE}$ to $\mathtt{MM}$

  else get (P_i, h) from $\mathtt{reqList}$, and compute $\sigma \leftarrow \mathtt{Sign}(\mathtt{sk}_i, h||y)$ and send (y, σ) to $\mathtt{MM}$.
- Upon receiving any command, if no message $(h', \sigma', \mathtt{requests}, \sigma^\star, \mathtt{e})$ is posted on the BB within Δ rounds such that $\mathtt{Ver}(\mathtt{pk}_{\mathtt{MM}}, \sigma', h')) = 1$ and $\mathtt{Ver}(\mathtt{pk}_{\mathtt{MM}}, \sigma^\star, \mathtt{requests}||\mathtt{e})) = 1$ then send $\mathtt{setAbort}$ to $\mathcal{F}^{\mathtt{trade}}$, else do the following.
  - $\mathtt{notAbort} \leftarrow 1$.
  - For each (P_i, h) in $\mathtt{reqList}$ compute $\mathtt{notAbort} \leftarrow \mathtt{notAbort} \wedge \mathtt{verification}(h^\star, h', h, \mathtt{pk}_i, \mathtt{requests})$

  $\mathtt{notAbort} \leftarrow \mathtt{notAbort} \wedge \mathtt{checkBB}(h^\star, h', \mathtt{requests}, \mathtt{pk}_{\mathtt{MM}})$

  If $\mathtt{notAbort} = 0$ then send $\mathtt{setAbort}$ to $\mathcal{F}^{\mathtt{trade}}$.

  Set $h^\star \leftarrow h'$ and send $(\mathtt{setOutput})$ to $\mathcal{F}^{\mathtt{trade}}$.

$\mathcal{S}_{\mathtt{MM}}$ can fail only if one of the following occurs:

1. $\mathcal{F}^{\mathtt{trade}}$ aborts because the simulator sees (with respect to a party P_i) in the hash chain a value $\mathtt{trade}_i$ that is not consistent with the trade chosen by an honest party P_i, but in the real world P_i does not abort. If this is the case then we can break the signature scheme since the adversary is able to provide a new signature for $\mathtt{pk}_i$.
2. $\mathcal{F}^{\mathtt{trade}}$ aborts because the simulator is unable to reconstruct the hash-chain that goes form h' to h'', where h' and h'' are part of two different tickets given to two honest parties, whereas in the real world at least one honest party does not abort. If this is the case, then the adversary knows how to invert a RO value.
3. Given two hash values h' to h'', where h' and h'' part of two different tickets, there are multiple hash chains that connect h' to h''. If this is the case, then we can construct an adversary that finds a collision for $\mathcal{F}_{RO}$.

We denote with $\mathcal{S}_P$ the simulator for the case when $\mathtt{MM}$ is honest and an arbitrary set of traders can be corrupted. For we consider only the case where MM and at exactly one party P_k is honest. Formally, $\mathcal{S}_P$ acts as follows.

[9] We note that this values can be computed by inspecting Q.

Initialize $h \leftarrow 0^\lambda$, τ and $R \leftarrow \varDelta$ where τ represents the upper bound on the time that a party has to reply to $\mathtt{MM}$ (this is to avoid DoS attack) and $\varDelta$ be the maximum number of rounds after which $\mathtt{MM}$ should post the accumulated trades on the BB.

– Upon receiving $(\mathtt{request}, \mathtt{pk}_i)$ from a corrupted party P_i send $(\mathtt{request}, P_i)$ to $\mathcal{F}^{\mathtt{trade}}$.
– Upon receiving $(\mathtt{price}^{\varXi \to \check{T}}, \mathtt{price}^{\check{T} \to \varXi})$ from $\mathcal{F}^{\mathtt{trade}}$ do the following.
  - Compute $h' \leftarrow \mathcal{H}(h||\mathtt{pk}_i)$ and set $h \leftarrow h'$.
  - Compute $\sigma \leftarrow \mathtt{Sign}(\mathtt{sk}_{\mathtt{MM}}, h||\mathtt{price}^{\varXi \to \check{T}}||\mathtt{price}^{\check{T} \to \varXi}||\mathtt{pk}_i||\mathtt{e})$.
  - Send $\mathtt{ticket}_1 := (h, \sigma, \mathtt{price}^{\varXi \to \check{T}}, \mathtt{price}^{\check{T} \to \varXi}, \mathtt{pk}_i)$ to P_i.
– Upon receiving $(\mathtt{trade}, \sigma_i)$ from a corrupted P_i such that $\mathtt{Ver}(\mathtt{pk}_i, \sigma_i, h||\mathtt{trade}_i)) = 1$ then send $(\mathtt{ok}, \mathtt{trade}_i)$ to $\mathcal{F}^{\mathtt{trade}}$. If $\mathcal{F}^{\mathtt{trade}}$ replies with $\mathtt{ok}$ then do the following
  - Compute $h' \leftarrow \mathcal{H}(h||\mathtt{trade})$ and set $h \leftarrow h'$.
  - Compute $\sigma \leftarrow \mathtt{Sign}(\mathtt{sk}_{\mathtt{MM}}, h)$.
  - Set $\mathtt{requests}[k] \leftarrow (\mathtt{pk}_i, \mathtt{trade}, \sigma_i)$.
  - Run $\mathtt{MMalgorithm}(\mathtt{trade}, \mathtt{price}^{\varXi \to \check{T}}, \mathtt{price}^{\check{T} \to \varXi})$ thus obtaining $\mathtt{price}^{\varXi \to \check{T}'}, \mathtt{price}^{\check{T} \to \varXi'}$ and set $\mathtt{price}^{\varXi \to \check{T}} \leftarrow \mathtt{price}^{\varXi \to \check{T}'}, \mathtt{price}^{\check{T} \to \varXi} \leftarrow \mathtt{price}^{\check{T} \to \varXi'}$.

  If $\mathcal{F}^{\mathtt{trade}}$ replies with $\mathtt{ko}$ then do the following
  - Compute $h' \leftarrow \mathcal{H}(h||\mathtt{NO\text{-}TRADE})$ and set $h \leftarrow h'$.
  - Set $\mathtt{requests}[k] \leftarrow (\mathtt{pk}_i, \mathtt{NO\text{-}TRADE}, 0^\lambda)$.
  - Compute $\sigma \leftarrow \mathtt{Sign}(\mathtt{sk}_{\mathtt{MM}}, h)$.
– Upon receiving $(\mathtt{getTrades})$ forward it to and $\mathcal{F}^{\mathtt{trade}}$. If $(\mathtt{Trades}, \mathtt{e})$ is received from $\mathcal{F}^{\mathtt{trade}}$ then post $(h, \sigma, \mathtt{Trades}, \sigma^\star, \mathtt{e})$ to the BB, where $\sigma \leftarrow \mathtt{Sign}(\mathtt{sk}_{\mathtt{MM}}, h)$ and $\sigma^\star \leftarrow \mathtt{Sign}(\mathtt{sk}_{\mathtt{MM}}, \mathtt{Trades})$, else ignore the command[10].

Our simulator can fail only if one of the following occurs:

1. A malicious party posts on the BB a proof of cheating $(h, \sigma, \mathtt{pk})$ such that $\mathtt{Ver}(\mathtt{pk}_{\mathtt{MM}}, h||\mathtt{pk}, \sigma) = 1$, where $h||\mathtt{pk}$ has never been signed by the simulator.
2. The simulator does not manage to post a valid message on the BB within $\varDelta$ rounds.

We can argue that due to the unforgeability of the signature scheme neither the first nor the second case can occur. The third case instead cannot occur due to the security of the BB.

D Monopolist Profit Seeking MM

The Glosten and Milgrom model [1] has become a standard model of a zero knowledge market-maker who trades against an informed trader. We adopt the extension by Das [11], which is also the model used in Das and Magdon-Ismail [13]. In this model, the true price (value) of the commodity is an unknown V and we assume the MM has a prior over V at time 0, $p_0(v)$. The prior represents all the starting information the MM knows, and we can think of the prior as some very high-entropy distribution, for example a Gaussian with huge variance. Number the traders $t = 1, 2, \ldots$ in the sequence they arrive. Trader t has an estimate of the value $w_t = V + \epsilon$, where ϵ is a random perturbation (noise) of the true value which quantifies how informed the trader is. Let us denote the cumulative distribution function (CDF) of ϵ by F_ϵ, which is known to the market maker. For simplicity, we assume the noise is symmetric, so $F_\epsilon(-x) = 1 - F_\epsilon(x)$, for example zero mean Gaussian noise. We also assume that different traders are independent, which means their noisy perturbations of V are independent. We now consider the MM actions for trader t, which is to set bid and ask prices, $a_t > b_t$ for the trader who will arrive at time-step t. The trader will either trade or not depending on how their signal w_t relates to a_t, b_t. Specifically, the trader buys from the market maker if $w_t > a_t$, sells to the market maker if $w_t < b_t$ and makes no trade otherwise. If the trader buys, the market maker receives the signal $x_t = +1$, if the trader sells, the signal is $x_t = -1$ and otherwise the signal $x_t = 0$. When trader $t+1$ arrives, we already have a sequence of trades $x_1, x_2, \ldots, x_t$. Let us assume by induction that the market maker has correctly updated its distribution over V to the posterior $p_t(v)$ at time t. Here, $p_t(v)$ contains all

[10] We note that by construction the trades encoded in $\mathtt{requests}$ are the same as in $\mathtt{Trades}$.

the information of the MM which now only depends on the historical sequence of trades made $x_1, \dots, x_t$. Given bid and ask prices b, a, and the value V, one can compute the probability of each type of signal, $P[x_t = +1] = 1 - F_\epsilon(a - V)$, $P[x_t = -1] = F_\epsilon(b - V)$. The market makers profit for an ask signal is $a - V$ and for a bid signal is $V - b$. To get the expected profit, we integrate over the possible values of V, b, a,

$$E[\text{profit}] = \underbrace{\int_{-\infty}^{\infty} dv\ p_t(v)(v-b)(F(b-v))}_{\text{bid-side profit}} + \underbrace{\int_{-\infty}^{\infty} dv\ p_t(v)(v-a)(1-F(a-v))}_{\text{ask-side profit}}. \tag{1}$$

The bid-side and ask-side profits are independently controlled by b and a respectively. Hence, to maximize the expected profit, we can independently maximize these terms with respect to a and b respectively. Taking derivatives and setting to zero reproduces a result from [13] that essentially tells the market maker how to set a_t, b_t to maximize expected profit in the next time-step.

Lemma 2. *To maximize the expected profit on the next trade, the market maker sets a_t and b_t to satisfy*

$$b_t = \frac{\int_{-\infty}^{\infty} dv\ p_t(v)(vF'_\epsilon(b_t - v) - F_\epsilon(b_t - v))}{\int_{-\infty}^{\infty} dv\ p_t(v)F'_\epsilon(b_t - v)}$$
$$a_t = \frac{\int_{-\infty}^{\infty} dv\ p_t(v)(vF'_\epsilon(a_t - v) + F_\epsilon(v - a_t))}{\int_{-\infty}^{\infty} dv\ p_t(v)F'_\epsilon(a_t - v)}$$

We denote these optimal bid and ask prices the myopic optimal prices. The approximate version of this myopic optimal bid-ask prices are computed in [13] for the case where the prior $p_0(v)$ and the trader signal F_ϵ are Gaussian. It is also shown in [13] that maximizing aggregated, discounted profit by instead solving the Bellman equation produces higher long-term gain for market maker and simultaneously lowers initial spreads, increasing liquidity - a win win. Our framework is general, and so can use any market maker. We will continue the discussion with the mypoic-greedy market maker because the optimal-market maker is computationally more expensive. Let us state some basic properties of the market maker [13], specifically the market maker's distribution $p_t(v)$, which quantifies how much information the market maker has on the true value V after t trades.

- The expected value of $p_t(v)$ converges to V. That is the market discovers the originally unknown true value of the commodity based on trades with traders who arrive with imperfect information. Empirically, the speed of this convergence is illustrated in [13] and follows the standard $1/t$ convergence for Bayesian updates.
- The market maker uncertainty as captures by the variance of $p_t(v)$ of converges to 0. Thus, not only does the market maker recover the true value V in expectation, but also becomes more and more certain of it. Again, this convergence is standard for Bayesian updates.
- In equilibrium, the market maker spread that produces maximum single step profit monotonically increases with the variance of its distribution, which converges zero. Hence the bid-ask spread converges to a minimum possible for a profit maximizing market maker. The multi-step (non-myopic) optimal market maker produces even lower spreads than a zero-profit competitive market maker in high-volatile uncertain environments. This is because optimal market makers may take early losses (with smaller than myopic bid-ask spreads) to increase the spead of convergence to the true value. This is because the market maker makes maximum long term profit when it trades around the true value V. To see this formally, let $r(b, a, v)$ be the expected profit as a function of the bid, ask and value, denoted respectively by parameters b, a, v. Suppose the market maker does not know the true value V and instead uses W to compute expected profit $r(b, a, W)$ which she maximizes to set prices b_*, a_*,

$$(b_*, a_*) = \underset{a,b}{\text{argmax}}\{r(a, b, W)\}.$$

The actual expected profit, however, is computed with respect to the true value V, because this is where the traders get their signals.

$$\text{true expected profit} = r(a_*, b_*, V) \leq \underset{a,b}{\operatorname{argmax}}\{r(a, b, V)\}.$$

The RHS is the expected profit from setting bid and asks optimally knowing the true value V. That is, a market maker who knows V can always make more expected profit than a market maker who does not. The last bullet above is essentially the intuition behind why an optimal market maker has no incentive to manipulate prices. The maximum profit is made when the market maker knows the true value V. Hence the market maker is incentivized to discover the true value V as quickly as possible. The only information available on the V is through the un-manipulated trader signals x_t.

Proof of Theorem 3 We now prove the main theorem 3, which is that after stating, the bid and ask prices, a rational market maker will not deviate from these prices, i.e. manipulate them, after learning of a trader intending to trade (say) with a buy.

Proof. Suppose the trader wishes to trade, buying from market maker at the ask-price (the argument is analogous if the trader wishes to sell at the bid). The best the market maker can do is to try to manipulate the price after having already posted bid and ask prices. The goal is to make more expected profit given this additional knowledge that the trader wishes to buy. So let us consider what price the market maker should charge to make *maximum* expected profit given this additional knowledge. The trader has announced an intention to buy, and has the option to reject any final trade offered. The trader will buy provided $V + \epsilon \geq a_t$. Hence the market maker needs to set a_t to maximize the expected ask-side profit. Since the expected profit breaks into two independent terms in (1), maximizing only the ask side profit corresponds exactly to setting the ask to mazimize the expected profit, which is exactly the prescription in Lemma 2 This means that the bid-ask prices set by the market maker are exactly those that maximize expected profit, and hence there is no incentive for a rational market maker to deviate from these prices after learning that a trader wishes to buy.

Proof of Lemma 1 Lemma 1 states that it is suboptimal for the market maker in our setting to ignore trades without knowledge of other trades. The proof follows analogously to Theorem 3 by using the fact that by ignoring real trades, the market maker will be trading at an inferior price away from the most current estimate of the value V. Hence, by excluding from its learning/price-discovery process these real trades (and or their associated profits), expected profits are lower and convergence to V is slower. This in turn produces lower long-term profit, because, as we already mentioned, the market maker makes most profit by trading around the true value V.

E Implementation and Evaluation

In this section, we describe implementation and evaluation of our system. We begin by describing implementation of the smart-contracts, followed by the implementation of the applications for seller and buyer (to run $\Pi^{\texttt{trade}}$, see Fig. 5). Experiment setup and analysis of evaluation results is presented in Section 6.

E.1 Smart-Contracts

We wrote our smart-contracts using Solidity and used TruffleSuite[11] for development cycle management. We also reused useful abstractions, e.g. access control, from the OpenZeppelin[12] framework. Concretely for exchangeable assets, we used Ether and ERC20 tokens.

[11] https://www.trufflesuite.com/
[12] https://www.openzeppelin.com/

Due to the way ERC20 tokens work—the token owner needs to call *transfer* on the token smart-contract—the functionality of the buyer smart-contract $\texttt{SC}_i$ (see Fig. 1) is split into two smart-contracts: the 1) *Seller-Contract* and the 2) *BuyerContract*. The *SellerContract* is about 25 lines of code. A transaction is initiated upon a call to *execute* method by the seller. Internally, it calls *BuyerContract*, which verifies buyer's signature and pays the seller. Upon getting paid, *SellerContract* pays corresponding tokens to the buyer. The entire transaction is executed atomically. The gas cost of *execute* method is $\approx 33K$. The *BuyerContract* is implemented in about 50 lines of Solidity code. Deploying the contract locks an amount till lock expiry. Its method, *claimExpiry*, claims the remaining funds after lock expires. The expensive method here is *execute* which costs $\approx 67K$. A trivial extension to this contract is a functionality to renew lock time/amount for continued trading. We do not discuss the gas costs of pessimistic case here, namely the verification contract. This is in line with other works in this area e.g.[31, 47]. Besides, the reward for a valid complaint to verification contract far outweighs the gas costs, thus the gas cost is not important.

We list the costs in Table 1. Note that the cost of executing one trade is the sum of the costs of *execute* methods of the *SellerContract* and the *BuyerContract*. While, we have also included the USD cost, this is not a good metric due to variations in USD/ETH exchange rate and average gas price. Gas cost is the only meaningful metric to compare complexity of different smart-contracts. Nevertheless, we provide USD costs here to be consistent with the previous works.

We note that we did, slightly, deviate from the specification of Fig. 1 in our implementation. Specifically, by initiating the transaction in *SellerContract* and restricting the calls to seller only, we saved one signature verification cost i.e. $\approx 30K$ in gas. There may be other, more aggressive, optimizations possible.

E.2 Seller and Buyer Applications

The smart-contracts described above are only used in the last step of the $\Pi^{\texttt{trade}}$ protocol. To run the $\Pi^{\texttt{trade}}$ protocol itself, we implemented the parties—seller and buyer—as nodejs[13] applications. Towards this, we used nodejs version 12.18.2. The source code was written in typescript[14]. The seller acts as a WebSockets server and is implemented with ≈ 800 lines of code. The buyer is a WebSockets client and is ≈ 400 lines of code. We used μWebSocket.js[15] for WebSockets server (i.e. seller) for its excellent performance.

F Uniswap

Uniswap is the largest (in terms of market cap) decentralized exchange implemented through a collection of smart contracts on Ethereum. At a high level, it consists a number of *pools* of assets (pairs of tokens). *Liquidity Providers (LPs)* add liquidity to the system by depositing their tokens into pools. For their service, they are given shares in the system (proportionate to their deposits). The *traders* interact with the pools to buy/sell tokens of their interest. These tokens follow the ERC20 standards. The current version *Uniswap v3* was launched recently, it primarily incorporates more fine-grained control for liquidity providers. We compare our work with *Uniswap v2*. This is because v2 (because of its maturity) presents more favorable numbers for Uniswap e.g. its highest daily volume is more than three times that of v3.

At a lower level, Uniswap contracts are divided into two catagories 1) *Core* contracts implement fundamental functionality and 2) *Periphery* contracts facilitate interaction with the system. In the core, there are a number of *Pair* contracts that encapsulate the functionality of a market maker for a pair of tokens. The *Factory* contract ensures that only one *Pair* contract is created per unique pair of tokens. To interact with the pairs, the *Library* contracts of *periphery* provides convenient access to data and pricing, while the *Router* contract enables trading tokens (even accross multiple pairs). The current version of router contract is *V2Router02*. We observed over a million recent transactions[16] to V2Router02 and averaged the gas cost

Table 3. Uniswap Gas Costs (relavant routines only, minimum cost)

Contract	**Method**	**Gas**
Factory	createPair	2,512,920
	setFeeTo	43,360
	setFeeToSetter	28,294
Pair	burn	85,206
	mint	103,871
	skim	48,051
	swap	61,446
	sync	52,012
V2Router02∗	addLiquidity	179,836
	addLiquidityETH	208,694
	removeLiquidity	136,412
	removeLiquidityETH	153,809
	removeLiquidityETHSFTT	307,229
	removeLiquidityETHWithPermitSFTT	327,705
	removeLiquidityWithPermit	183,505
	removeLiquidityETHWithPermit	168,408
	swapExactTokensForTokens	163,596
	swapExactTokensForTokensSFTT	248,100
	swapTokensForExactTokens	159,212
	swapExactETHForTokens	131,562
	swapExactETHForTokensSFTT	137,987
	swapTokensForExactETH	132,490
	swapExactTokensForETH	123,552
	swapExactTokensForETHSFTT	194,171
	swapETHForExactTokens	137,421

SFTT SupportingFeeOnTransferTokens

∗ Gas costs for V2Router02's methods are average of one million transactions sent to it in block interval 12,162,664 to 12,231,464.

of the invoked methods. These are listed in Table 3. The methods prefixed with *swap* perform various types of trades and are, therefore, relevant for comparison with our system.

[13] https://www.nodejs.org
[14] https://www.typescriptlang.org
[15] https://www.github.com/uNetworking/uWebSockets.js
[16] block 12,162,664 to block 12,231,464