# 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*[5]

[1,2]The University of Edinburgh
[3]Rensselaer Polytechnic Institute
[4]University of California, Los Angeles
[5]Purdue University
[1]michele.ciampi@ed.ac.uk, [2]ishaq@ishaq.pk, [3]magdon@gmail.com, [4]rafail@cs.ucla.edu, [5]vzikas@cs.purdue.edu

**Abstract**

As new and emerging markets, crypto(-currency/-token) markets are susceptible to manipulation and illiquidity. The theory of market economics, offers *market makers* that bear the promise of bootstrapping/stabilizing such markets and boosting their liquidity. In order, however, to achieve these goals, the market maker operator (typically an exchange) is assumed trusted against manipulations. Common attempts to remove/weaken this trust assumption require several on-chain rounds per trade or use expensive MPC machinery, and/or are susceptible to manipulative market-maker operators that perform *informed front-running attacks*—i.e., manipulate the sequence of trades using future trade information. Our work proposes a market-maker-based exchange which is resilient against a wide class of front-running (in particular, reordering attacks). When instantiated with a *monopolistic profit seeking market maker* our system yields a market where the trading price of crypto-tokens converges to a bid-ask spread centered around their true valuation. Importantly, after an initial setup of appropriate smart contracts, the trades are done in an off-chain fashion and smart contracts are invoked asynchronously to the trades. Our methodology yields a highly efficient exchange, where the market maker's compliance is ensured by a combination of a rational market analysis, cryptographic mechanisms, and smart-contract-based collaterals. We have implemented our exchange in Ethereum and showcase its competitive throughput, its performance under attack, and the associate gas costs.

## 1 Introduction

Markets, even mature ones, are susceptible to *price manipulation*, namely (possibly legal) strategies that results in trading away from the commodity's real valuation.[1] Naturally, this is a bigger issue for emerging markets such as cryptocurrency/token exchanges. *Market makers* (in short, MMs) are an market-economics tool to stabilize, reduce manipulation, and increase liquidity. A market maker specifies the way/algorithm by which the exchange-rate (price) is decided,[2] and is a key differentiator of exchanges, whether of tokens, cryptocurrencies, fiat-currency, or stocks.

The most common market maker in the cryptocurrency arena is via an *order book*: Assume the MM allows for traders to buy/sell two assets $t_1$ and $t_2$; the MM receives orders in which a party $P_i$, $i \in \{1, 2\}$,

*Work done in part while the author was at the University of Edinburgh.

[1]See e.g., https://www.cnbc.com/2021/01/27/gamestop-mania-explained-how-the-reddit-retail-trading-crowd-ran-over-wall-street-pros.html.

[2]The term market maker is often used to refer to the algorithm or to the exchange/operator. Here, whenever clear from the context, we will use the term both to refer to the exchange/operator and to the algorithm.

specifies an amount of $t_i$ and a price (in $t_{3-i}$) that the party is willing to trade at. When two parties $P_1$ and $P_2$ have matching orders—i.e., $P_1$ is willing to buy the quantity that $P_2$ offers at a price matching or exceeding $P_2$'s offer—then a trade is facilitated between the two. An order book MM is very simple and intuitive but it is known to result in illiquidity on emerging immature markets (i.e., if no one is willing to buy at the lowest selling price, the market halts). A number of different, more intelligent algorithms e.g., [BP09] have, therefore, been proposed, which in addition to increased liquidity have desirable properties for the market maker (e.g., maximizing its profit) or the market itself (e.g., creating forces that drive the traders towards reporting their real evaluation of the goods thereby stabilizing the market (cf. Section 1.2 for a more detailed discussion).

In addition to the traditional question of which market-makers is most suitable, the decentralisation paradigm has put in the spotlight the question of *trust* to the MM. Traditionally, crypto exchanges have been made mostly through dedicated exchange platforms, which operate similar to a standard money-market exchange broker. These platforms are typically opaque and to boost throughput process trades off-chain and do not report them on the corresponding ledger(s). For example, exchanging an Ethereum [Woo14] token for ETH on an off-chain exchange, e.g., Coinbase, will not result in a transaction posted on the Ethereum blockchain unless. Stated simply, these exchanges maintain their own off-chain private ledger, similar to how banks keep account balances.

More recently, several exchanges started using the blockchain itself towards a more transparent and auditable exchange mechanism. Example vary from the most transparent (but less scalable) solution of running the exchange completely on chain, as a smart contract [Uni18, WB17, Air18, Del18, Ban, IDE18, Kyb18], to running it off-chain by using the blockchain ledger for accountability, via posting (typically NIZK) publicly verifiable succinct proofs of compliance. Despite, however, a number of novel ideas in such systems, *front running* by the market maker remains a sticky point [DGK$^+$20] (unless one employs multi-party computation MPC [BDF21], which is both expensive and requires additional trust assumptions, cf. Section 1.2). In a front running attack, the MM uses the information that it has on the incoming bids to manipulate the price, typically by reordering the different trade requests in a way that increases its own profit.

In this work we propose and analyze a market-maker-based exchange of tokens on a smart-contract-enabled blockchain, e.g., Ethereum tokens, with built-in cryptographic defences agains a wide class of front-running. Combining, in a non-trivial manner, our new defences with ideas from optimistic fair exchange, game theory, and market analysis, we get a market-maker that is resilient to front-running, while driving the market to convergence to a bid-ask spread centered around the assets' true valuation. Our proposed system is scalable and highly efficient, as demonstrated by our experiments, and might have applications beyond block-based crypto-assets, such as on standard fiat-currency or stock exchanges, where front-running is a long-standing unresolved problem.

## 1.1 Our Contributions

The first step towards our MM is what we call a $\Sigma$-*trade protocol*. Intuitively, this is a fair-exchange protocol with the following structure: It consists of an off-chain and on-chain phase. During the off-chain phase (which consists of three messages, similarly the $\Sigma$-protocols from the interactive-proofs literature) the buyer and the seller agree on the amount and the type of assets, and exchange the information required to create a special transaction `trx`. If the off-chain phase is successful, then (only) the seller can post on the blockchain `trx` (during the on-chain phase) and make effective the agreed trade. We note that although one could extract a protocol with the above properties from the fair-exchange literature (see Section 1.2), we consider our abstraction of $\Sigma$-trade protocolan intuitive step towards a modular design and analysis of MMs which could be of independent interest. In particular, it allows us to present the different components of our construction independent of which $\Sigma$-trade protocolis used. For completeness we include a simple $\Sigma$-trade protocolwhich we use in our benchmarks in Section 4.

The second step is to instantiate the seller in the $\Sigma$-trade protocolwith a deterministic market-maker whose execution is verifiable. Informally, verifiability comes from the deterministic nature of the MM: anyone that sees a sequence of trades can check whether or not the prices quoted in this sequence are consistent with

what the (publicly known) algorithm the MM is supposed to run. We take advantage of this verifiability as follows: In a setup-phase, the MM instantiates a smart contract that has its algorithm hardwired, and locks a large amount of an asset owned by the MM (e.g., ETH) as collateral. Upon being called, this contract will observe the sequence of trades posted by the market maker and verify compliance—by simulating the MM's internal state (including the bid-ask spread) and comparing it to the price quoted in the trade sequence. If a discrepancy is observed, the contract "burns" (most of) the collateral. In particular, to improve the efficiency and avoid DDoS attacks, the contract is executed only upon request (i.e., accusation) by a party $P$ and using gas/fees provided by $P$, which are returned to $P$ if the accusation is verified augmented by a generous bonus. This mode ensures that parties have an incentive to invoke the contract if and only the MM misbehaved—as any invalid accusation results in the accuser waisting gas on running the contract (see Section 2.2 for a more detailed discussion).

The above mechanism does not prevent a deviating MM from a front-running attack which first looks at the trade information of the multiple buyers and decides accordingly which trade to consider first. As a third step, we provide a cryptographic mechanism that prevents such attacks by forcing the MM to *verifiably sequentialize its interactions with the buyers*. We view this as the major contribution of the paper as it solves a congenital issue of centralized exchanges/market-makers—where the reordering of transactions is inherrently possible—while maintaining the same transparency of a decentrilzed market maker. In fact, as we show, by instantiating our system with a *monopolistic profit seeking market maker* algorithm [Das05, DMI08] (see Section 1.2), we can ensure that the worst the MM might do without sacrificing its collateral or lowering its expected profit is stop responding (i.e., shutdown the system at its will) but with no trader loses money by this action. We prove this by a novel combination of cryptographic and market-economics arguments, which show that such an MM does gains neither by front-running nor by collaborating with traders to manipulate the price. In fact, as a worst-case fallback guarantee, even if our MM decides, irrationally, to sacrifice its collateral, the honest traders will still not lose money, but in that case they might become susceptible to front-running.

We have implemented and benchmarked our MM. We show that the throughput of our system (see Fig. 9) is extremely competitive, despite the trades being processed sequentially. In particular, we achieve a throughput of over 200 trades/minute when running the off-chain part on a relatively weak, consumer laptop.[3] It is already more than twice the throughput of e.g. P2DEX [BDF21] (an MPC-based off-chain exchange with front-running defenses) in the practical setting, and about $\approx 50\%$ higher than the maximum daily volume of Uniswap (see Section 9.2). We have also compared the performance of FairMM with Uniswap [Uni18], and demonstrate the superiority of our approach in Section Section 9.2.

## 1.2 Comparison with Existing Works

**Market Makers for Crypto-token Markets** New emerging markets, e.g. prediction markets [WZ04] 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 [PS07]. A market maker algorithm typically aims to maximize liquidity in the market (which ensures that there are constantly open trades being executed) and/or to maximize its own profit. The *zero-profit competitive market maker* model [Glo89, Das05, GM85] considers multiple MMs that compete with each other by lowering their marginal profit in order 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 [Glo89, Das05, GM85, DMI08]. In the model of monopolist market-makers, the market is viewed as a process which only involves a single market maker that is trying to maximize its profit and the market liquidity. The Glosten and Milgrom model [GM85] has become a standard in modeling monopolist market makers: The market maker is considered *environment oblivious* (aka zero-knowledge), i.e., is assumed to have no independent knowledge of the market (other than a mild prior over the market value) and only receives information from the trades it executes against informed traders. We adopt the extension by Das [Das05]: The idea of such an MM is that for each pair of assets

[3] A commercial off-chain MM would be usually deployed on high end servers which are far superior than our test machine and could significantly improve the throughput.

$A_1$ and $A_2$, the MM preserves a *bid* price, representing the MM's estimate of the value of $A_1$, and an *ask* price, representing the MM's estimate of the value of $A_2$. (The bid and the ask price need not match in which case there is a *bid-ask spread*.) In this model, every asset has a true price (value) $V$, which is however unknown to the MM. The goal is to design an MM, which facilitates price discovery through the trading of informed traders against the MM. The traders have independent estimates randomly distributed around the true value $V$ and will trade with the MM as long as the price is favorable to them. Every trade allows the MM to learn about the true value. Hence, by learning from the trades, the MM adapts the price (bid and ask) after each trade and as a result, quickly converges to the true market value.

In [DMI08] the authors show that monopolistic profit seeking MMs, in seeking to make long term profit, can provide more liquidity and faster price discovery than even zero-profit market-makers. This is because MM makes more profit when the true value is inside the bid-ask spread, hence the MM has a strong incentive to quickly converge to the true-value by encouraging more trades (liquidity) even at a small cost. Thereafter, since the MM makes most money when the true value is inside the bid-ask spread, this means that a rational MM will have no incentive to manipulate the price away from the true value and this true value is solely determined by the information brought by the arriving traders. The result is a liquid sequential trading process with fast price discovery in which a MM learns as it trades. For more details on such MM algorithms and convergence/price discover, we refer to [Das05, DMI08]. Most importantly, the automated profit-seeking monopolistic market makers can significantly increase liquidity, speed up price discovery and are rationally incentivized to not manipulate the price (bid ask) away from the true value because that is where they make most money.

**Fair Exchange and Blockchains** There is a large amount of literature on fair exchange dating all the way to the early MPC works [Yao86, GMW87, ASW98, CC00, KL10], which has been reignited with the adoption of blockchains and cryptocurrencies [BK14, KZZ16, Wik18, BDM16, CGGN17, Fuc19]. Due to the relevance of these works to ours, we include a detailed review in Appendix A. 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 settle trades instantly. Therefore, we designed our own fair exchange protocol, in fact a $\Sigma$-trade protocol, that we proved amenable to such composition.

**Decentralized exchanges** Closer to our goals is the work on decentralized exchange—e.g. Uniswap, Kyber, etc. However, the relation to them is mostly limitted to the fact that aim to solve the same problem. First, recall that none of these works solves front-running attacks. Indeed, there is evidence [DGK$^+$20] that such front-running attacks are not only actively launched, they are also very profitable. In fact, it can be argued that no front-running defense is possible with on-chain solutions due to inherently public nature of the mainstream blockchains. Secondly, these exchanges are inherently susceptible to manipulation by the miners. Moreover, since running computations on the blockchain is expensive and market making computations happens on-chain, these transactions induce higher costs. Finally, the trade execution delay is at least as high as the block delay of the blockchain. In contrast, we prevent front running, settle the trades instantly and, in the honest case (no dispute resolution), the only transactions that hit the blockchain are mostly the ones that move funds which are cheaper. Ordering of trades (and therefore pricing) is decided by our protocol off-chain, hence the miners may include the resulting transactions in any order but they will not be able to influence actual ordering of the trades (or more precisely, the exchange rates decided during the off-chain phase of the protocol) .

More recently, Baum *et al.* [BDF21] propose P2DEX, a decentralized cryptocurrency exchange that aims at solving the front running problem. It does so by deploying an off-chain MPC protocol emulating an order-book MM. As discussed above, exchanges based on order books cannot provide liquidity when no such matches are found. Our approach based on market-making algorithms helps mitigating both these problems. More importantly, to prevent front-running P2DEX relies on the security of an MPC run among a set of off-chain servers which adds an extra trust assumption: If all the servers are colluding, then it is easy for the servers to front-run honest traders (without the being detected). Our work avoid such trust assumption by a combination of cryptographic, game-theoretic, and market-economics arguments. Finally, in terms of

performance, P2DEX's throughput of $\approx 100$ orders per minute (in the only realistic setting of WAN) is less than 50% of what we achieve on a consumer grade computer.

# 2 Technical Overview

Before getting in the guts of our MM construction we provide a more detailed overview of the techniques used our MM construction. For simplicity we focus the discussion to the case, where there are just two exchangeable Ethereum tokens/assets $t_1$ and $t_2$; however, our protocol trivially allows to exchange arbitrary such assets and can be used with any smart-contract-enabled blockchain that supports the contracts we describe here.

## 2.1 $\Sigma$-trade protocol

Our $\Sigma$-trade protocol involves a seller that wants to sell assets of type, say $t_2$ and a buyer that has assets $t_1$ that he wishes to exchange with type-$t_2$ assets. (In this work, we consider assets to be Ethereum tokens or ETH.) The buyer has an amount of type-$t_1$ assets deposited in a smart contract $SC_B$, which is set up once and be used for multiple exchanges assuming sufficient funds/assets are held by it.

Our protocol borrows ideas from the blockchain-aided optimistic fair-exchange literature [Wik20a, Wik20b, Her18, ZHL+19, HLY19, Gug20, DH20, HLS19, DEF18, EFS20]. It proceeds in three rounds and operates similar to a one-sided exchange scheme: In the first round, the buyer $B$ sends a message to $S$ expressing that he wants to trade. In the second round, the seller gives the buyer a quoted price (number of $t_1$ assets per $t_2$ assets) at which he is willing to sell, along with the address $\mathtt{pk}_S$. In the third round, if the buyer disagrees with the quote, then he sends a $\perp$; otherwise, the buyer issues a (signed) certificate $c$ of purchase that includes the quantity $y$ of $t_2$ he wished to buy and the quoted price (including the seller's signature from Round 2). In addition to that, $B$ sends his wallet address $\mathtt{pk}_B$ and $SC_B$'s address.

The certificate output to the seller can be used as input of the contract $SC_B$ to transfer the amount of assets of type $t_1$ (corresponding to $y$ and the agreed exchange rate) from the buyer's wallet to its own wallet. More precisely, in order for $SC_B$ to accept $c$ and perform the transfer of the type-$t_1$ assets to the seller's possession, the contract checks that a transfer of the corresponding amount of type-$t_2$ from the (seller's) address indicated in Round 2, to the (buyer's) address indicated in Round 1 has already been processed and settled on the blockchain. It is this property of $SC_B$ that makes the exchange *fair*: only the seller can utilize the certificate, and can only receive its asset if it has already sent the buyer the bought asset. An important feature to note, however, is that the certificate creation occurs entirely off-chain. In that sense, the certificate can be thought of as the digital analogue of a conditional "cashier's check" that can be cashed in by the seller at any point after the protocol terminates but only after having transferred the sold items to the buyer and until the contract $SC_B$ is valid.

## 2.2 Turning a $\Sigma$-trade protocol to a Verifiable MM

We employ a deterministic MM as the seller to make the above $\Sigma$-trade protocol *verifiable*—anyone that sees a sequence of trades can check whether or not the prices quoted in this sequence are consistent with what the algorithm would produce—and combine it with a collateral mechanism to restrict the MM's cheating strategies. The idea is to require the market maker, as part of its setup, to instantiate the following smart contract $SC_{MM1}$: The smart contract has the code of the MM algorithm hardwired and has a large amount of asset owned by the MM (e.g., ETH) locked as collateral; intuitively, $SC_{MM1}$ monitors all the trades made by the market maker, simulates its internal state (including the bid-ask spread) after every such trade, and compares it to price quoted in the trade sequence.[4] If a discrepancy is observed, $SC_{MM1}$ burns the collateral. We note that the above $SC_{MM1}$ is costly in terms of gas; however, we can easily turn it into a contract that is only invoked upon misbehavior by simply requiring the market maker to post the identifiers

[4]Note that the trades are executed on the same blockchain on which $SC_{MM1}$ lives, i.e., $SC_B$ and $SC_{MM1}$ are on the same blockchain.

of the executed transactions on the blockchain every fixed number of rounds (denoted with $\Delta$). In this way, anyone (even a smart contract) can check that the exchange rates used in the transactions that involve the exchange's public key are consistent with what the MM algorithm would output.

Clearly if the collateral in $SC_{MM1}$ is high-enough, the exchange will not post transactions and rates that deviate from what an honest execution of the MM algorithm would yield. Thus, although the trades occur off-chain, just by observing the sequence of trades and comparing it with what the MM algorithm would produce, the traders can verify compliance of the MM with the algorithm.

However, given that now $SC_{MM1}$ needs to be invoked in order to penalize the MM, we need to specify who covers the gas costs required to run it. We first note that an honest MM should never be penalized, neither pay for the gas required to run the contract that checks if MM was proposing the correct prices. On the other hand, we do not want an honest trader to pay the fee required to run $SC_{MM1}$ as this party has nothing to gain by invoking the contract rather than the satisfaction of knowing that the misbehaving MM will lose the collateral. To solve both these problems, we modify $SC_{MM1}$ as follows. Whoever invokes $SC_{MM1}$ needs to cover for the gas costs (let us say $C$), and if $SC_{MM1}$ detects a malicious behaviour of the MM, a generous portion of the collateral, say $10C$, goes to the caller of $SC_{MM1}$ and the remaining part is burned as before. We note a MM is never incentivized to invoke $SC_{MM1}$ if the amount of burn collateral is higher than $10C + \alpha$, where $\alpha$ is the potential gain of a market maker that decides to not follow the market-making algorithm. Moreover, the first party that notices that MM is misbehaving is incentivized in invoking $SC_{MM1}$ to get the reward.

Unfortunately, there are still certain attacks that a cheating MM could perform. We discuss these attacks and how we defend against them in the following two sections.

## 2.3 Trade Reordering

The first attack is reordering. Concretely, if the market maker knows the details (token-type to be spent and amounts) of a number of different pending trades, then he could potentially benefit by processing them in an out-of-order manner. We note that such re-ordering attacks are even present in standard (non-crypto) markets, in particular within high-frequency trading. Here is how we counter this attack in our setting: At his response to the first message of the $\Sigma$-trade protocol, the MM attaches a signed hash of the outcome (certificate) of the previous trade. Thus, this message works both as a quote, and as a "ticket" that puts this trader in the queue for the next trade. This ticket is then also included by the trader on his third round message.

It is not hard to verify that this mechanism prevents reordering attacks: Indeed, the above "ticket"-issuing mechanism forces the MM to sequentialize the trading process, in a way that does not allow it to process new trades before the one with the current ticket is completed—since a next trade would require the MM to provide a hash of the completion certificate—or a timeout occurs. As before, this method makes reordering publicly detectable: Assume that order $A$ from $P_A$ comes before order $B$ from $P_B$; $P_A$ will receive a ticket that includes $H(c_{A-1}, \mathtt{pk}_A)$, where $H$ is a collision-resistant hash-function, e.g., SHA256, $c_{A-1}$ is the previous-trade certificate (i.e., the Round-3 message of the trade that occurred before $P_A$ showed up ), and $\mathtt{pk}_A$ is $A$'s wallet address (that the MM received in the first round of the $\Sigma$-trade protocol). Now for $P_B$ to perform a trade, he needs to be issued a previous-trade certificate as well. However, the tickets and the corresponding trades create a hash-chain that forces the MM to comply with the order they are processed. In particular the market maker has two choices: (1) either complete the trade with $P_A$, get the corresponding certificate $c_A$ and issue to $P_B$ the next ticket $H(c_A, \mathtt{pk}_B)$, or (2) ignore the trade of $P_A$ and hand $P_B$ a ticket $H(c_{A-1}, \mathtt{pk}_B)$.

In the first case, if the MM tries to post the two trades in an out-of-order fashion to manipulate the price, this will be observable as the hash-chain will appear inconsistant (unless a collision is found on $H(\cdot)$). Hence, similar to how we turned the $\Sigma$-trade protocol into a verifiable protocol, we can force the MM to comply with a consistent order by having him create, once and for all in a setup phase, a smart contract $SC_{MM2}$ with a large locked collateral, which checks that the posted transactions (in fact their tickets and certificates) form an actual hash chain; if deviation is observed $SC_{MM2}$ destroys the collateral. (As in the

case of $SC_{MM1}$ to save on gas we can make the contract opt-in, and allow users that observe a discrepancy to trigger it).

The second case is somewhat trickier as it relates to robustness: The MM might decide to ignore a trade and move on to the next one. We discuss this attack in the following section, as well as our mechanism to mitigate it. In a nutshell, we can employ a profit-maximizing market-maker to ensure that it is not rational to launch such an attack. We stress that this last part below is the only mitigation which depends on the rational properties of the actual MM algorithm. In particular, the compliance and non-reordering defences discussed above restrict/deter the front-running attack surface for any exchange as long as the collateral is high enough. To our knowledge, even just the above level of front-running defence is a drastic improvement over the state of the art which simply allows any type of front-running.

### 2.4 Private Manipulation and Aborting/Erasures

We have argued that neither the market maker nor a trader can provoke an unfair outcome (i.e., receive the other's asset without transferring their own), and that the market maker cannot perform an *informed*—i.e., based on their order size—reordering attack against trades from honest traders, i.e., has to process trades sequentially and in the order they are queued. However, there are still two types of attacks that are not covered by the above: (1) Disregarding/ignoring a trade before posting any following trade (i.e., one that would depend on it), and (2) injecting trades created by the MM—i.e., trading with himself or, equivalently, inject trades in collusion with a malicious trader.

Notwithstanding, we show in Section 7 that a rational (profit-seeking) oblivious MM in our setting cannot increase his expected profit by performing any of these attacks. Informally, the reason is that as discussed in the previous section, the MM makes maximum expected profit when it trades around the real value of the goods *at the bid/ask prices implied by the MM-algorithm.* Since the only information available for the market maker to learn this value is based on the traders trading around their informative signals [Das05], a rational market-maker cannot gain by injecting uninformed (self) trades. Further, MM will not ignore real trades because its sequence of actions (bid-ask prices) are monitored/dictated to be compliant by the $SC_{MM2}$, so by ignoring real trades, the MM will simply be trading at an inferior price away from the most current estimate of the value $V$. That is, 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 turns produces lower long-term profit, because, as we already discussed, MM makes most profit by trading around the true value $V$.

Note that, to our knowledge, prior to our work, the only proposed alternative to protecting against the first attack (disregarding a trade) was by means of an expensive on-chain mechanism, where the cheated side complains to some smart-contract, the MM is given the chance to counter, and if he does not he is penalized. This mechanism, however, is clearly unacceptable in a fast-trading market-makers situation, where the trades would have to either have long enough time-outs (several seconds in the Ethereum case) for this on-chain dispute-and-resolution process, or the MM would have to adjust the bid-ask spread independently of such disputes which creates a separate attack surface. Thus, to our knowledge, this is the first attempt to solve this problem by using principles from the economic markets theory and can be of independent interest.

## 3 Preliminaries and Notation

We use "=" to check equality of two different elements (i.e. $a = b$ 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 $a \xleftarrow{\$} A$, 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 $\lambda > \lambda_0 : \nu(\lambda) < 1/p(\lambda)$.

## 3.1 Signatures

**Definition 1** (Signature scheme [Can03])**.** *A triple of* PPT *algorithms* $(\mathtt{Gen}, \mathtt{Sign}, \mathtt{Ver})$ *is called a* signature scheme *if it satisfies the following properties.*

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

**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* $\nu$*, 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* $\mathtt{Sign}(s, \cdot)$.

## 3.2 Blockhain and Smart-Contracts

Ethereum is arguably the most popular blockchain for smart-contracts. The Ethereum 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*.

**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. [GKL15, PSs17, BMTZ17]. We assume familiarity with this notion and refer to B.2 for more detail.

## 3.3 Security framework

The Universal Composability (UC) framework introduced by Canetti in [Can01] is a security model capturing the security of a protocol $\Pi$ under the concurrent execution of arbitrary other protocols. All those other protocols and processes not related to the protocol $\Pi$ go through an environment $\mathcal{Z}$. The environment has the power to decide the input that the parties should use to run the $\Pi$, and to see the output of these parties. In this framework there is also an adversary $\mathcal{A}$ for the protocol $\Pi$ 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 $\Pi$. 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 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 $\Pi$ 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 $\Pi$, 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 [Can01] (in fact, its synchronous version from [KMTZ13, KZZ16, BMTZ17].) Nonetheless to make the presentation more accessible to a non-UC expert we often use the convention and language similar to [Can00]. 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 [KMTZ13] it is easy to project our statement to (synchronous) UC.

# 4 $\Sigma$-Trade Protocols

In this section we introduce the notion of $\Sigma$-trade protocols. A $\Sigma$-trade protocol is a protocol for the fair exchange of tokens that lives on a blockchain $E$.[5] More precisely, a $\Sigma$-trade protocol consists of three rounds of off-chain interaction between a buyer and a seller that are used to 1) agree on the exchange rate of the tokens and 2) generate the information required to create the on-chain transactions that will move the tokens from the buyer to the seller (and vice-versa) according to the agreed price. One important property of a $\Sigma$-trade protocol is that the seller has the power to decide whether to make the trade effective (by posting an appropriate transaction on the blockchain $E$) or not. More precisely, there is a bounded amount of time $T$ decided by the buyer, before which the seller has to decide whether or not to make the trade effective. This asymmetry between the buyer and the sender will become important when the $\Sigma$-trade protocol is used by a market-maker that acts as the seller. Indeed, we need the maker-maker to know exactly what kind of trades will or will not be effective almost immediately (independently from the performance of the underlying blockchain). More details on this are provided in the next section. We now provide a more formal abstraction of a $\Sigma$-trade protocol.

A $\Sigma$-trade protocol $\Pi$ is an interactive protocol run by a seller $S$ and potentially many buyers $B_1, \ldots, B_m$, where $m$ might be unknown to the the seller $S$. We assume that there are only two tokens $t_1$ and $t_2$, and that each buyer wants to buy tokens of type $t_2$ in exchange of tokens of type $t_1$. Moreover, each buyer $B_i$ has an upper bound of type $t_1$ tokens that can spend which we denote with $\mathtt{z}_i$.

The exact amount of $t_2$ tokens the buyer wants to buy can be decided adaptively in the last round of interaction. A $\Sigma$-trade protocol $\Pi$ consists of the following steps:

1. Each buyer $B_i$ creates a smart contract $\mathtt{SC}_i$ on $E$ that locks $\mathtt{z}_i$ tokens of type $t_1$ (more details on $\mathtt{SC}_i$ are provided later).

2. $B_i$ and $S$ exchange three off-chain messages, where $B_i$ speaks first. In the first message $B_i$ sends a trading request to the seller $S$. We note that in this phase $B_i$ does not disclose anything about the trade he would like to do.

3. In the second round $S$ proposes the exchange rate for the tokens he owns, let us call this information `askedPrice`.

4. Let $y$ be the quantity of tokens of type $t_2$ that the buyer wants to buy that is such that $y \cdot \mathtt{askedPrice} \leq \mathtt{z}_i$. If $B_i$ agrees with the exchange rate indicated in `askedPrice`, then $B_i$ sends a *certificate* $c$. $c$ can

[5] We can think of $E$ as the Ethereum blockchain.

$\mathtt{SC}_i$

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

Figure 1: Smart contract $\mathtt{SC}_i$ for the case where $B_i$ wants to buy $\check{T}$ for $\Xi$. 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 $\Xi$ from $B_i$

be used by $S$ to invoke $\mathtt{SC}_i$ and withdraw $x = y \cdot \mathtt{askedPrice}$ tokens of type $t_1$ from the account of $B_i$. However, $\mathtt{SC}_i$ will move the $x$ tokens from $B_i$'s account only under the condition that $S$ has moved to $B_i$'s account $y$ tokens of type $t_2$. At a very high level, $\mathtt{SC}_i$ ensures atomic transactions pre-agreed by the seller and the buyer, but it can be triggered only by the seller.

We will argue that any abstraction of $\Sigma$-trade protocol can be used in combination with our protocol that prevents the reordering of transactions, and that guarantees that `askedPrice` is computed according to a publicly known market-maker algorithm. For completeness, we now show an example of how to instantiate a $\Sigma$-trade protocol for the case where the buyers wants to buy tokens $\check{T}$ for ethereums $\Xi$.

## 4.1 Selling tokens for ethers

For a buyer $B_i$ we denote with $(\mathtt{sk}_i^C, \mathtt{pk}_i^C)$ the pair of signing-verification keys associated to the account $C \in \{E, \check{T}\}$, where $E$ represents Ethereum and $\check{T}$ a token that lives in Ethereum. We also denote with $\Xi$ the Ethereum currency. Analogously, for the seller $S$ we denote with $(\mathtt{sk}_S^C, \mathtt{pk}_S^C)$ the pair of signing-verification keys associated to the account $C \in \{E, \check{T}\}$.

In Fig. 1 and Fig. 2 we provide the formal description of the smart-contract and our protocol $\Pi$ respectively, and provide here the intuitions about how they work. The smart contract $\mathtt{SC}_i$ locks for $T_i$ rounds $\mathtt{z}_i\Xi$ and manages a list of transaction identifiers. Upon receiving an input $(x, y, \mathtt{ID})$ that has been authenticated by both the buyer and the seller, $\mathtt{SC}_i$ moves $x\Xi$ to the account of the seller under the conditions that 1) a transaction `trx` that moves $y\check{T}$ from the seller's account to the buyer's account has been made, 2) `trx` contains the identifier $\mathtt{ID}$ in its payload and 3) $\mathtt{ID}$ does not appear in the list of transaction identifiers. In addition, when such conditions verify, $\mathtt{SC}_i$ stores $\mathtt{ID}$ in the list of identifiers. This is required to prevent that the same transaction `trx` is used to withdraw money from $\mathtt{SC}_i$ multiple times. We note that the same contract $\mathtt{SC}_i$ can be used for multiple trades if $\mathtt{z}_i$ is big enough. We are now ready to describe how our protocol works. The buyer sends a trade request to the seller, which replies with the exchange rate between $\Xi$ and $\check{T}$ that we denote with `askedPrice`. If the buyer agrees with `askedPrice`, then he generates an identifier $\mathtt{ID}$, converts $y$ in $\Xi$ using `askedPrice` thus obtaining $x$, and signs $x||y||\mathtt{ID}$. Then the buyer sends the signed values (with their signature) to the buyer. The seller, in order to withdraw $x\Xi$ from $\mathtt{SC}_i$ needs to 1) post a transaction `trx` that pays $y\check{T}$ into the account of the buyer, where `trx` contains in its payload the identifier $\mathtt{ID}$ and 2) sign $x||y||\mathtt{ID}$, and use the resulting signature, and the signature received from the buyer to run the contract $\mathtt{SC}_i$. We note that the seller could post `trx` and contextually sends it to the buyer. Therefore, the buyer can be sure that the trade will eventually occur, as if the seller does not post `trx` within a certain time-frame, the buyer will do that (by broadcasting `trx`).

Π

**$B_i$'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 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 $\Xi$. Send `trade-request` to $S$

$S$. Let `askedPrice` be the price at which $S$ is willing to sell $\check{T}$ for $\Xi$ (i.e., $1\check{T}$ = `askedPrice`$\Xi$). Send `askedPrice` to $B_i$.

$B_i$. Upon receiving `askedPrice` from $S$, if `askedPrice` represents a good price (w.r.t. the strategy of $B_i$) then do the following steps, otherwise send `NO-TRADE` to $S$.

- Compute $\mathtt{ID} \leftarrow \mathtt{ID} + 1$ and $x \leftarrow y \cdot \mathtt{askedPrice}$ and $\sigma_1 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}_i^{\Xi}, x||y||\mathtt{pk}_S^{\Xi}||\mathtt{ID})$
- Send $x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}_i^{\check{T}}, \mathtt{pk}_i^{\Xi}$ to $S$.

$S$. Upon receiving $(x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}_i^{\check{T}}, \mathtt{pk}_i^{\Xi})$ from $B_i$ do the following steps.

- If $B_i$ has created a contract according to Fig. 1 then continue, otherwise stop interacting with $B_i$.
- If $\mathtt{Ver}(\mathtt{pk}_i^{\Xi}, \sigma_1, x||y||\mathtt{pk}_S^{\Xi}||\mathtt{ID}) = 1$ and $x = y \cdot \mathtt{askedPrice}$ then continue with the following steps, ignore the message of $B_i$ otherwise.
- Compute $\sigma_2 \xleftarrow{\$} \mathtt{Sign}(\mathtt{sk}_S^{\Xi}, x||y||\mathtt{pk}_i^{\Xi}||\mathtt{ID})$.
- Post a transaction `trx` with the identifier 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 $(x, y, \mathtt{ID}, \sigma_1, \sigma_2)$.

Figure 2: Π, $B_i$ wants to sell $\Xi$ for $\check{T}$.

# 5 (Fair) Ordering of Transactions

In this section we describe the trade functionality $\mathcal{F}^{\mathtt{trade}}$ relying on the UC framework. The trade functionality describes the only ways in which the market maker can reorder the trades to, for example, manipulate the market in his favour. For simplicity, we assume that there are only two assets: $\Xi$ and $\check{T}$. We denote with $\mathtt{price}^{\check{T}\to\Xi}$ (and $\mathtt{price}^{\Xi\to\check{T}}$) the price at which MM sells $\check{T}$ ($\Xi$) in exchange for $\Xi$ ($\check{T}$). We assume that the information that describes the trade of a party $P_i$ is encoded in $\mathtt{trade}_i$. That is, $\mathtt{trade}_i$ describes the type and the amount of assets, the prices of the assets, the type of the trade (sell or buy) and might contain an additional payload. We also assume that all the parties share the procedure `MMalgorithm` (the MM algorithm), which on input of a trade and the current prices, outputs the updated prices (the new values for $\mathtt{price}^{\check{T}\to\Xi}$ and $\mathtt{price}^{\Xi\to\check{T}}$). At a high level, $\mathcal{F}^{\mathtt{trade}}$ works as follows. Upon receiving a request from a trader $P_i$, $\mathcal{F}^{\mathtt{trade}}$ sends the prices of the assets to $P_i$, and signal to MM that $P_i$ wants to trade. If $P_i$ agrees with these prices, he sends the information regarding the trade $\mathtt{trade}_i$ to $\mathcal{F}^{\mathtt{trade}}$. Upon receiving $\mathtt{trade}_i$, $\mathcal{F}^{\mathtt{trade}}$ forwards $\mathtt{trade}_i$ to MM which has only two choices: 1) to decide not to trade with $P_i$ by sending a command `NO-TRADE` to $\mathcal{F}^{\mathtt{trade}}$, or 2) to accept to trade with $P_i$. If MM does any other action before doing one these two (e.g., MM starts trading with a party other than $P_i$), $\mathcal{F}^{\mathtt{trade}}$ would allow that, but it would also set a special flag `abort` to 1. This means that if the traders query $\mathcal{F}^{\mathtt{trade}}$ with the command `getTrades` (to get the list of trades accepted by MM), $\mathcal{F}^{\mathtt{trade}}$ would return a special symbol $\perp$, to denote that MM has misbehaved. A corrupted MM can also decide to set the output of $\mathcal{F}^{\mathtt{trade}}$ always to $\perp$. This capture the fact that MM can decide to stop working at his will. Moreover, MM can decide to add any trade of a corrupted party to the list of trades using the command `setAdvTrade`, but this can be done only after that MM has concluded the trading phase with any honest traders, as specified above.

$\mathcal{F}^{\mathtt{trade}}$ is parametrized by $\Delta$, which denotes the maximum number of rounds per *epoch*. In each *epoch* MM

should allow the other parties to see the entire list of trades. MM can make the list of trade accessible via a special command setOutput. In the case where MM does not send such a command at least every $\Delta$ rounds, then $\mathcal{F}^{\texttt{trade}}$ would return $\perp$ to any honest party that requests to see the full list of trades (thus noticing that MM is not responsive). This mechanism capture the fact that MM might trade with multiple traders without making the list of trade public for at most $\Delta$ round.

We stress that $\mathcal{F}^{\texttt{trade}}$ 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 anyway consistent with the prices that $\mathcal{F}^{\texttt{trade}}$ sent to the traders. We observe that the market maker still has the power to decide with what parties he wants to trade first, however, this decision has to be taken 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 $\mathcal{F}^{\texttt{trade}}$ does not allow any real exchange of assets between the parties and the market maker. However, as we will argue in the next section, if the output of $\mathcal{F}^{\texttt{trade}}$ is posted on a blockchain and the trades are defined properly (according to the language of the blockchain), then the MM can use the trades to trigger events on the blockchain that move the assets between the market maker and the parties 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 procedure 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 checks that the prices involved in each trade are computed accordingly to the market-making algorithm (we refer to Fig. 4 for the formal description of these procedures). In Fig. 3 we propose the formal description of $\mathcal{F}^{\texttt{trade}}$.

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

We assume that the parties have access to a bulletin board BB and have a public key for a signature scheme. We also assume that all the parties know the MM's public key and share the knowledge of the same procedure MMalgorithm. At a very high level, our protocol that realizes $\mathcal{F}^{\texttt{trade}}$ works as follow. MM maintains a hash chain that starts with a special value that we denote with $h_{\texttt{start}}$ (it can be the all-zero string) that all the traders know. Any time that MM receives a request from a trader $P_i$, he adds to the hash chain the public key of $P_i$, signs the head of the hash chain (let us say $h_i$), the public key of $P_i$ and the current price of the assets. 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 it is the case then $P_i$ defines the trade $\texttt{trade}_i$, signs it thus obtaining $\sigma_i$, and sends $(\texttt{trade}_i, \sigma_i)$ to MM (the signature $\sigma_i$ 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 this is the case then MM adds to the hash chain $\texttt{trade}_i$, adds $\texttt{trade}_i$ with its signature $\sigma_i$ to a list requests, run MMalgorithm on input $\texttt{trade}_i$ and the current prices to get the new prices, and waits to receive another request from a new trader. In every epoch (one epoch lasts a most $\Delta$ rounds) we require MM to publish on the bulletin board the head of the hash chain $h$ and the list requests, all authenticated with his signing key. (If MM that does not post such authenticated information within $\Delta$ rounds then all the traders will understand this as an abort and output $\perp$). Each honest party that has access to the BB now does the following: 1) checks that each trade in requests is either NO-TRADE or a correctly signed trade; 2) checks that all the prices used to construct a trade in requests have been computed according to MMalgorithm and that the hash chain that starts at $h_{\texttt{start}}$ and finishes at $h$ can be constructed using the trades in requests (this can be done by recomputing the hash chain using the information contained in requests); 3) checks if the hash value $h_i$ (received as part

**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 $\tau$, $\texttt{timer}$ (with $\texttt{timer}$ initialized to $-\tau$), the starting prices $\texttt{SP}^{\check{T}\to\Xi}$ and $\texttt{SP}^{\Xi\to\check{T}}$ at which $\texttt{MM}$ is willing to sell (and respectively buy) $\check{T}$ for $\Xi$, respectively, the integers $\Delta$, $R$, and the epoch index $\texttt{e}$. The functionality initializes $\texttt{price}^{\Xi\to\check{T}} \leftarrow \texttt{SP}^{\Xi\to\check{T}}$, $\texttt{price}^{\check{T}\to\Xi} \leftarrow \texttt{SP}^{\check{T}\to\Xi}$, $R \leftarrow \Delta$, $\texttt{e} \leftarrow 0$.
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 $(\texttt{request}, \texttt{sid})$ from a party $P_i$
  - If $\texttt{MM}$ is corrupted then send $(\texttt{new-request}, P_i)$ to $\mathcal{A}$ else
  - If $T_{\texttt{now}} - \texttt{timer} > \tau$ then[a] send $(\texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ to $P_i$, send $(\texttt{new-request}, P_i)$ to $\texttt{MM}$, set $\texttt{activep} \leftarrow P_i$ and $\texttt{timer} \leftarrow T_{\texttt{now}}$ else ignore the command.
- Upon receiving $(\texttt{setPrice}, \texttt{sid}, P_i, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ from a corrupted $\texttt{MM}$ then send $(\texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ 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 $(\texttt{ok}, \texttt{sid}, \texttt{trade}_i)$ 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 $\texttt{checkTrade}(\texttt{trade}_i, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi}) = 1$ 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 $(\texttt{setTrade}, \texttt{sid}, P_i, y)$ 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 $(\texttt{setAdvTrade}, \texttt{sid}, P_i, \texttt{trade})$ 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 $(\texttt{getTrades}, \texttt{sid})$ 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 $\texttt{checkPrices}(\texttt{SP}^{\check{T}\to\Xi}, \texttt{SP}^{\Xi\to\check{T}}, \texttt{Trades}) = 1$ 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 $\texttt{Trades}^\star$) and set $R \leftarrow T_{\texttt{now}} + \Delta$, $\texttt{e} \leftarrow \texttt{e} + 1$, $\texttt{output} = 0$.
  - else return $\perp$.

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

Figure 3: The trades functionality $\mathcal{F}^{\texttt{trade}}$.

**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 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 $\texttt{flag} \leftarrow 1$.

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 $\texttt{flag} \leftarrow 0$.

compute $p_1, p_2 \leftarrow \texttt{MMalgorithm}(p_1, p_2, \texttt{trade})$.

Return flag

[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.

Figure 4: Auxiliary procedures

of the ticket) is part of the hash chain that starts at $h_{\texttt{start}}$ and ends at $h$.

We observe that anyone can check if the first and the second conditions hold (even a party that did not interact with MM at all). Hence, if either the first or the second condition does not hold, then all the honest traders output a special symbol $\perp$ (to claim that MM has misbehaved). The third condition instead can be checked only by a trader that received a ticket. However, if a trader detects that the third condition does not hold, then he can post his ticket (which we recall has been authenticated by MM) on the bulletin board, in such a way that all the parties that have access to the BB can detect that MM misbehaved and output $\perp$. Intuitively, our protocol realizes $\mathcal{F}^{\texttt{trade}}$ because once that MM has send a ticket to a trader, he has also committed to a set of trades. Hence, as long as MM cannot generate collisions for the hash function, he cannot include new trades in the hash chain depending on the amount of assets that the new traders wants to sell/buy. We denote our protocol with $\Pi^{\texttt{trade}}$ and provides its formal description in Fig. 5. In the protocol MM maintains $h \leftarrow 0^\lambda$, an initially empty list `requests` and four integers $R$, $\tau$ and $\Delta$. $\Delta$ represents the maximum number of rounds after which MM has to post the trades on the BB, $\tau$ represents the upper bound on the time (i.e., number or rounds) that a party has to reply to MM (this is to avoid DoS attack) and $R$ is initialized to $\Delta$. Let also $\texttt{SP}^{\check{T}\to\Xi}$ and $\texttt{SP}^{\Xi\to\check{T}}$ be the starting price at which MM is willing to sell (and respectively buy) $\check{T}$ for $\Xi$. MM maintains an integer that we call *epoch index* and denote with $\texttt{e}$ (each epoch lasts at most $\Delta$ rounds). 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 $\texttt{e}_i$ (initialized to 0).

To simplify the description of the protocol, we have described the procedures used to check whether or not MM is misbehaving in Fig. 4. A brief overview on what these procedures do follows:

- `verifyPrices` takes as input a list of trades and checks that the prices involved in each trade are computed accordingly to the market-making algorithm.
- `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 on the BB by MM. It checks whether the ticket information appears in the hash chain and is consistent with the list of trades.
- `checkBB` looks on the BB for valid tickets (a ticked is valid if is signed by MM), and for each valid ticket runs the procedure `verification`. If `verification` outputs 0 (hence one of the ticket posted by a trader proves that MM was trying to cheat) the procedure outputs 0.

To not overburden the notation, the parties will just use an hash function instead of querying the random oracle functionality (RO) $\mathcal{F}_{RO}$. Moreover, whenever a party $P$ wants to post a value $x$, or look for a value $x$ on the bulletin board we will just write, respectively *$P$ posts $x$ on the BB* and *$P$ looks for a value $x$ on the BB*.

In Appendix C.1 we formally prove the following theorem.

**Theorem 1.** *If there exists a signature scheme (accordingly to Definition 1) then $\Pi^{\texttt{trade}}$ realizes $\mathcal{F}^{\texttt{trade}}$ in the $(\mathcal{F}_{RO}, \texttt{BB})$-hybrid model.*

# 6 Combining $\mathcal{F}^{\texttt{trade}}$ with $\Sigma$-Exchange Protocols.

We observe that if in the realization of $\mathcal{F}^{\texttt{trade}}$ we replace the BB with a blockchain which supports smart contracts, then we could have also a smart contract acting as a party registered to $\mathcal{F}^{\texttt{trade}}$ that in every round queries $\mathcal{F}^{\texttt{trade}}$ with the command `getTrades`. We can program this smart contract in such a way that if the output of $\mathcal{F}^{\texttt{trade}}$ is $\perp$ then the MM is penalized. In our final protocol the traders and MM will be running a $\Sigma$-trade protocol $\Pi$ and in parallel invoke $\mathcal{F}^{\texttt{trade}}$ using as input the same information (prices, quantity and the type of the trades) used in the execution of $\Pi$. Once that the output of $\mathcal{F}^{\texttt{trade}}$ is generated, we can rely on a smart contract to check that the trades are consistent with the transactions generated by $\Pi$. If this is

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

1) $P_i$: **creation of a request.** Upon receiving $(\texttt{request}, \texttt{sid})$, 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 $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\Xi\rightarrow\check{T}}, \texttt{price}^{\check{T}\rightarrow\Xi}, \texttt{pk}_i)$ to $P_i$ and ignore any request that comes from any party $P_j \neq P_i$ for $\tau$ rounds.
3) $P_i$: **finalizing the request.** Upon receiving $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\Xi\rightarrow\check{T}}, \texttt{price}^{\check{T}\rightarrow\Xi}, \texttt{pk}_i)$ 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 $\texttt{trade}.p_1 = \texttt{price}^{\Xi\rightarrow\check{T}}$ and $\texttt{trade}.p_2 = \texttt{price}^{\check{T}\rightarrow\Xi}$ 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 $\tau$ rounds such that $\texttt{Ver}(\texttt{pk}_i, \sigma_i, h||\texttt{trade})) = 1$ and $\texttt{checkTrade}(\texttt{trade}_i, \texttt{price}^{\Xi\rightarrow\check{T}}, \texttt{price}^{\check{T}\rightarrow\Xi}) = 1$ 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 $\texttt{MMalgorithm}(\texttt{trade}, \texttt{price}^{\Xi\rightarrow\check{T}}, \texttt{price}^{\check{T}\rightarrow\Xi})$ thus obtaining $\texttt{price}^{\Xi\rightarrow\check{T}'}, \texttt{price}^{\check{T}\rightarrow\Xi'}$ and set $\texttt{price}^{\Xi\rightarrow\check{T}} \leftarrow \texttt{price}^{\Xi\rightarrow\check{T}'}, \texttt{price}^{\check{T}\rightarrow\Xi} \leftarrow \texttt{price}^{\check{T}\rightarrow\Xi'}$.

   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) $\texttt{MM}$: **flush of the trades on 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 $R \leftarrow R + \Delta$, update the epoch number $\texttt{e} \leftarrow \texttt{e} + 1$ and reinitialize $\texttt{requests}$.
6) $P_i$: **checking the honest behavior of** $\texttt{MM}$. 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 $\Delta$ 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 $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\Xi\rightarrow\check{T}}, \texttt{price}^{\check{T}\rightarrow\Xi}, \texttt{pk}_i)$ 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 $(\texttt{getTrades}, \texttt{sid})$, 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 $\perp$.

Figure 5: Our protocol that realizes $\mathcal{F}^{\texttt{trade}}$.

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, will inspect the output of $\mathcal{F}^{\mathtt{trade}}$ and if the output is $\perp$ then $\mathtt{SC}_{\mathtt{penalize}}$ will burn the collateral of MM. Otherwise $\mathtt{SC}_{\mathtt{penalize}}$ checks that the transactions in the list received from $\mathcal{F}^{\mathtt{trade}}$ are consistent with the transactions generated by MM on the ethereum blockchain with respect to the MM's wallet addresses $(\mathtt{pk}^{\Xi}_{\mathtt{MM}}, \mathtt{pk}^{\check{T}}_{\mathtt{MM}})$. If they are not consistent, $\mathtt{SC}_{\mathtt{penalize}}$ would (also in this case) burn the MM's collateral. We note that this contract might be 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 can detect that by looking at the output of $\mathcal{F}^{\mathtt{trade}}$ and consequentially decide to invoke $\mathtt{SC}_{\mathtt{penalize}}$. As argued in the introductory part of the paper, we can incentivize the honest parties to invoke $\mathtt{SC}_{\mathtt{penalize}}$ (and cover the gas cost) in the case when MM is misbehaving by transferring a small portion of the locked collateral to the calling party before burning the rest of it.

To complete the description of our protocol we need to introduce yet another smart contract that we denote with $\mathtt{SC}_{\mathtt{account}}$ and describe in Fig. 6. This contract is also created by MM, and it checks if the transactions that pay the MM's account exceed a certain value $Y$. If this is the case, then the contract would block any additional payment towards MM. This contract limits the amount of commodities that MM can trade, and the reason for doing that is to cope with a market-maker that is willing to be penalized (by for example reordering the trades thus causing $\mathcal{F}^{\mathtt{trade}}$ to output $\perp$) because the profit MM would gain by misbehaving overcomes the collateral locked in $\mathtt{SC}_{\mathtt{penalize}}$.

We recall that no malicious (even irrational) MM could steal money from the traders, and that the worst that MM can do in our protocol is to front-run the traders (by letting $\mathcal{F}^{\mathtt{trade}}$ output $\perp$) or avoid posting transactions that allow the settling of the trades. Hence, we just need to assume that the collateral locked in $\mathtt{SC}_{\mathtt{penalize}}$ could never be gained by MM through one of the mentioned actions. We denote the upper bound on the commodity that MM can trade with $Y$. If we set $Y$ to be smaller than the collateral of $\mathtt{SC}_{\mathtt{penalize}}$ (that we denote with $z$) then it is never convenient for a rational MM to be penalized by means of $\mathtt{SC}_{\mathtt{penalize}}$.

We call our protocol $\Pi^{\mathtt{full}}$, and for sake of completeness we propose its formal description in Fig. 7. We describe the case where there are only traders that want to buy $\check{T}$ for $\Xi$, and assume that $Y$ is big enough, and that MM has enough funds to satisfy all the requests. This protocol simply combines together the functionality $\mathcal{F}^{\mathtt{trade}}$ and the $\Sigma$-trade protocol of Section 4.1. We also propose a description of $\mathtt{SC}_{\mathtt{penalize}}$ in Fig. 8. We observe that we can assume that the smart-contract $\mathtt{SC}_{\mathtt{penalize}}$ acts like a party registered to $\mathcal{F}^{\mathtt{trade}}$ would act only because to realize $\mathcal{F}^{\mathtt{trade}}$ we replace the bulletin board with the blockchain that hosts the contract $\mathtt{SC}_{\mathtt{penalize}}$. Let $T$ be the number of round within the contract $\mathtt{SC}_{\mathtt{penalize}}$ can be invoked, then 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 MM is burned;*
2. *the $\mathcal{F}^{\mathtt{trade}}$ is not $\perp$ but there is not a perfect correspondence between the trades contained in the output of $\mathcal{F}^{\mathtt{trade}}$ and the transactions that appear on the blockchian $E$ with respect to MM's public keys. Moreover, the collateral locked in $\mathtt{SC}_{\mathtt{penalize}}$ is burned;*
3. *the $\mathcal{F}^{\mathtt{trade}}$ is not $\perp$, there is a perfect correspondence between the trades contained in the output of $\mathcal{F}^{\mathtt{trade}}$ and the transactions that appear on the blockchian $E$ with respect to MM's public keys. Moreover, all the collateral remains locked in $\mathtt{SC}_{\mathtt{penalize}}$ for $T$ rounds.*

If we set the smart contracts with the right parameters, and assume that the aim of the market-maker is to maximize the amount of $\Xi$, we can argue that the first and the second case listed in Theorem 2 happen with negligible probability. More precisely, let $\alpha$ be the gas cost required to run $\mathtt{SC}_{\mathtt{penalize}}$ with the input `detected`, and `reward` be reward that could be given to a party calling $\mathtt{SC}_{\mathtt{penalize}}$, let $z$ be the locked collateral in $\mathtt{SC}_{\mathtt{penalize}}$ and let $Y$ be the maximum amount of $\Xi$ that MM can have on the account with address $\mathtt{pk}^{\Xi}_{\mathtt{MM}}$. In this case, intuitively, if there is at least one honest party $P_i$, $\mathtt{reward} > \alpha$, and $z > Y$ then a market-maker that wants to maximize its profit would not let either the first or the second case of Theorem 2 to happen.

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

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

Figure 6: The contract does not allow MM to gain more than $Y\Xi$,

We finally observe that multiple malicious traders could harm the throughput of the system by never completing the protocol (by not even sending the NO-TRADE message). In a previous version of this work, each trader had to send in the first round of the $\Sigma$-trade protocol his identity (which includes the public key used for the creation of the contract $\mathtt{SC}_i$). In such a setting, it is easy for the market maker to distinguish between *bad traders* (i.e., traders that often do not complete the execution of the protocol) from *good traders* (i.e., trader that always completes the protocol honestly). Indeed, in this case the MM can ignore all the requests that come from traders that have not created the contract $\mathtt{SC}_i$ correctly or the requests of traders that in past have not completed correctly an execution of the protocol. In this version of the paper, we do not disclose the public keys of the traders in the first message of the $\Sigma$-trade protocol to keep hidden all the information about the trade (i.e., the first round sent from the trader to the MM does disclose the upper bound of the amount of commodity which might be inferred by looking at $\mathtt{SC}_i$). However, to keep resilience against malicious traders that never complete the protocol we can simply rely the standard *Know Your Client* (or *Know Your Customer*) which require the traders to register to the market maker and later identify themself for every new trading operation.

# 7 Monopolist Profit Seeking MM

The Glosten and Milgrom model [GM85] has become a standard model of a zero knowledge market-maker who trades against an informed trader. We adopt the extension by Das [Das05], which is also the model used in Das and Magdon-Ismail [DMI08]. 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 $\epsilon$ 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 $\epsilon$ by $F_\epsilon$, 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 the information of the MM which now only depends on the historical sequence of trades made $x_1, \ldots, x_t$. 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 $V$,

$\Pi^{\texttt{full}}$

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

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

$P_i$. Before interacting with MM check that MM has created a smart-contracts $\texttt{SC}_{\texttt{penalize}}, \texttt{SC}_{\texttt{account}}$ prescribed in Figs. 6 and 8. Let $x$ be the amount of $\Xi$ that $P_i$ wants exchange for $\check{T}$. Send $(\texttt{request}, \texttt{sid})$ to $\mathcal{F}^{\texttt{trade}}$. Upon receiving $(\texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ from $\mathcal{F}^{\texttt{trade}}$, if the prices are not good accordingly to the $P_i$'s strategy, then send $(\texttt{NO-TRADE}, \texttt{sid})$ to $\mathcal{F}^{\texttt{trade}}$, else do the following

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

$P_i$. On each round send $(\texttt{getTrade}, \texttt{sid})$ to $\mathcal{F}^{\texttt{trade}}$. If $\mathcal{F}^{\texttt{trade}}$ replies with $\perp$ then invoke $\texttt{SC}_{\texttt{penalize}}$ with the input $\texttt{detected}_1$, else let $(\texttt{Trades}, \texttt{e}_i)$ be the output of $\mathcal{F}^{\texttt{trade}}$ and do the following.

- If $T_i$ rounds have passed (i.e., $\texttt{SC}_i$ cannot be invoked anymore), $\texttt{trade}_i$ belongs to Trades and $\texttt{SC}_i$ has never stored in usedIDs the identifier ID then invoke $\texttt{SC}_{\texttt{penalize}}$ with the input $\texttt{detected}_1$.[b]
- If $\texttt{SC}_i$ has stored the identifier ID in usedIDs but Trades does not contain any entry $\texttt{trade}_i$ such that $(x, y, \texttt{ID}, \sigma_1, \texttt{pk}^{\check{T}}_{i}, \texttt{pk}^{\Xi}_{i}) = \texttt{trade}_i$ and $\texttt{Ver}(\texttt{pk}^{\Xi}_{i}, \sigma_1, x||y||\texttt{pk}^{\Xi}_{\texttt{MM}}||\texttt{ID}) = 1$ then invoke $\texttt{SC}_{\texttt{penalize}}$ with the input $(\texttt{detected}_2, \texttt{SC}_i, \texttt{ID}, \sigma_1, \texttt{pk}^{\Xi}_{i})$.

MM. Upon receiving $(\texttt{new-request}, P_i)$ from $\mathcal{F}^{\texttt{trade}}$ send $(\texttt{setPrice}, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ to $\mathcal{F}^{\texttt{trade}}$.

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

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

MM. Every $\Delta$ rounds MM sends setOutput to $\mathcal{F}^{\texttt{trade}}$.

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

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

Figure 7: Full market-maker protocol with identifiable (and punishable) misbehaviour.

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

**State:** $z\Xi$ and $\mathtt{reward}\Xi$ locked for time $T$, the public key $\mathtt{pk}_{\mathtt{MM}}^{\Xi}$
**Checking that MM is advancing:**

- Act as a party $P$ registered to $\mathcal{F}^{\mathtt{trade}}$ which receives no input.

**Upon receiving the input $\mathtt{detected}_1$ from a party $P_i$ with public key $\mathtt{pk}_i^{\Xi}$:**

- Send ($\mathtt{getTrades}$) to $\mathcal{F}^{\mathtt{trade}}$.
- Upon receiving $\mathtt{output}$ from $\mathcal{F}^{\mathtt{trade}}$, if $\mathtt{output} = \perp$ then
  - move $\mathtt{reward}\Xi$ to $\mathtt{pk}_i^{\Xi}$ and destroy $z\Xi$.
- Else, parse $\mathtt{output}$ as $(\mathtt{Trades}, \mathtt{e})$ and check that each trade in $\mathtt{Trades}$ corresponds to a transaction on the ethereum chain. If that is the case then do nothing, else move $\mathtt{reward}\Xi$ to $\mathtt{pk}_i^{\Xi}$ and destroy the $z\Xi$.

**Upon receiving the input $(\mathtt{detected}_2, \mathtt{SC}_i, \mathtt{ID}, \sigma_1, \mathtt{pk}_i^{\Xi})$ from a party $P_i$ with public key $\mathtt{pk}_i^{\Xi}$:**

- If $\mathtt{SC}_i$ is constructed accordingly to Fig. 1 and it has stored the identifier $\mathtt{usedIDs}$ but $\mathtt{Trades}$ does not contain any entry $\mathtt{trade}_i$ such that $(x, y, \mathtt{ID}, \sigma_1, \mathtt{pk}_i^{\tilde{T}}, \mathtt{pk}_i^{\Xi}) = \mathtt{trade}_i$ and $\mathtt{Ver}(\mathtt{pk}_i^{\Xi}, \sigma_1, x||y||\mathtt{pk}_{\mathtt{MM}}^{\Xi}||\mathtt{ID}) = 1$ for some $x, y \in \{0,1\}^{\lambda}$ then move $\mathtt{reward}\Xi$ to $\mathtt{pk}_i^{\Xi}$ and destroy $z\Xi$

Figure 8: Smart contract $\mathsf{SC}_{\mathtt{penalize}}$ that penalize $\mathtt{MM}$ in the case where $\mathcal{F}^{\mathtt{trade}}$ outputs $\perp$. The gas fee required to run the contract on the input $\mathtt{detected}$ is payed by the contract's caller. We assume that this fee is strictly less than $\mathtt{reward}$ (e.g., $\mathtt{reward}$ is 10 times the gas fee).

$b, a,$

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

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 a result from [DMI08] that essentially tells the market maker how to set $a_t, b_t$ to maximize expected profit in the next time-step.

**Lemma 1.** *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 [DMI08] for the case where the prior $p_0(v)$ and the trader signal $F_\epsilon$ are Gaussian. It is also shown in [DMI08] 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 [DMI08], 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 [DMI08] 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 $b_*, a_*$,

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

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

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

  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 $x_t$. We now prove the main theorem, 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 (we refer to Appendix C.2 for the proof).

**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. 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 $V$.

**Lemma 2.** *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.*

# 8 Implementation and Evaluation

In this section, we describe implementation and evaluation of our framework. We begin by describing implementation of the smart-contracts, followed by the implementation of the applications for seller and buyer (to run the $\Pi^{\texttt{trade}}$ protocol of Fig. 5). We then describe the experiments setup and, finally, conclude the section with a discussion of the results of these experiments.

## 8.1 Smart-Contracts

We wrote our smart-contracts using Solidity version 0.6.12. This is the most widely supported version at the time of this writing. We used TruffleSuite[6] for development, testing and deployment. We also reused useful abstractions, e.g. signature verification, access control, etc, from the OpenZeppelin[7] framework. Concretely, for exchangeable assets we used Ether and *ERC20* tokens.

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) had to be split into two smart-contracts: the 1) *SellerContract* 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 have not discussed the verification contract's cost here. This contract is invoked in pessimistic case i.e. when market maker cheats, and the reward for a valid complaint far outweighs the gas costs. For buyer and seller contracts, 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, note that 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.

[6] https://www.trufflesuite.com/

[7] https://www.openzeppelin.com/

Table 1: Gas Costs of Seller and Buyer Contracts.

| **Methods** | | **Gas** | **USD†** |
|---|---|---|---|
| Contract | Method | 74gwei/gas∗ | 2,158.23 usd/eth∗ |
| BuyerContract | claimExpiry | 31,619 | 5.05 |
| | execute | 67,984 | 10.86 |
| SellerContract | execute | 33,456 | 5.34 |
| **Deployments** | | | |
| BuyerContract | | 1,082,529 | 172.89 |
| SellerContract | | 836,341 | 133.57 |

∗ Prices taken from `https://coinmarketcap.com/` on 2021-04-11.

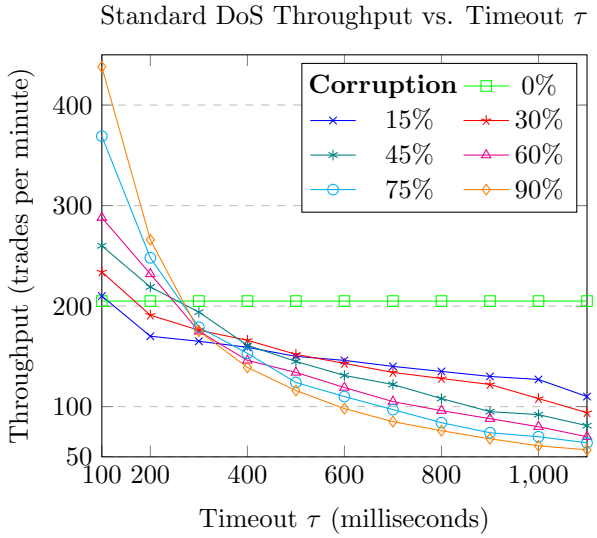


Figure 9: Standard DoS Attack Throughput (at 0% to 90% corruption thresholds). Values are average of 5 runs. Note that x-axis starts at 100milliseconds; this is the typical round-trip time and any $\tau < 100$ may cause timeouts for honest players.

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.

## 8.2 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[8] applications. Towards this, we used nodejs version 12.18.2. The source code was written in typescript[9]. The seller acts as a WebSockets

[8]https://www.nodejs.org
[9]https://www.typescriptlang.org

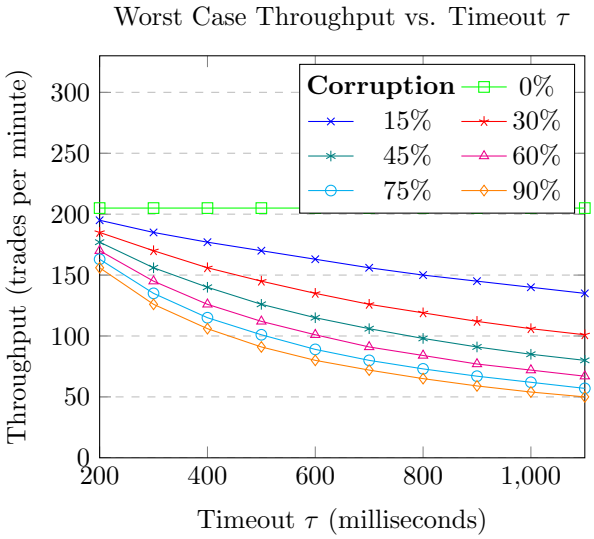


Figure 10: Worst-case Throughput (at 0% to 90% corruption thresholds). Note that x-axis starts at 200milliseconds; this is because a malicious player needs a budget of at least round-trip time (100milliseconds in our case) to respond without risking timing out.

server and is implemented with $\approx 800$ lines of code. The buyer is a WebSockets client and is about 400 lines of code. We used uWebSocket.js[10] for WebSockets server (i.e. seller) for its excellent performance.

## 8.3 Experiment Setup

We performed several experiments to measure throughput of the system. These were run 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 $P_i$ speaks first and sends its public keys to the seller `MM`. Then the seller speaks and 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 $\approx 350$ms to prepare this message. Lastly, we observed that the typical round trip time from buyer $\rightarrow$ seller $\rightarrow$ buyer is less than 100ms (or less than 0.1 seconds).

Our goal was to observe the system's throughput in the following adversarial scenarios. The first one is the *Standard DoS attack*. Here, a malicious buyer floods the system with ticket requests and then stops responding. His goal is to slow 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 $\tau$, the buyer will assume a `NO-TRADE` response from this buyer, execute the `NO-TRADE` scenario, and move to the next buyer. Concretely we ran experiments with $n = 300$ buyers. We repeated the experiment 5 times and report the averages of the measurements. Note that relatively few repetitions of the experiments are not a concern because of low variance of the measured values.

[10] https://www.github.com/uNetworking/uWebSockets.js

The other attack scenario is what we call *Worst-case Throughput attack*. In this scenario the setting remains the same as above except one difference; the malicious buyer now waits until just before the timeout and then responds with a `trade` response. This strategy is more affective at slowing down the system than the standard DoS attack. The reason why it is the case is discussed in Section 8.4).

## 8.4 Analysis

The results of the experiments for *Standard DoS attack* are summarized in Fig. 9, and for *Worst-case Throughput attack*, in Fig. 10.

Before discussing the results, let us briefly look at the theoretical upper bound on the system's throughput due to the Ethereum blockchain limits. The *block gas limit* at the time of this writing is $\approx 12M$ gas. Posting a trade in the implementation of Fig. 1 costs $\approx 101K$ gas. This means that, the maximum throughput $\Pi^{\texttt{trade}}$ can achieve on Ethereum blockchain is $\approx 119$ transactions per block or $\approx 475$ trades per minute (considering an average block generation time is 15 seconds). In reality, this upper bound is much lower because there are many other users/applications competing to put their transactions in the block.

Now, looking at the throughput measurements in Standard DoS attack in Fig. 9 we observe that, with no corruptions, our system can execute a little over 200 trades per minute. Keeping in mind the discussion above, this is an excellent throughput. Recall also, that this result was obtained on a consumer laptop which is far inferior in power compared to the high-end server(s) typically used for such tasks. Therefore, despite sequentialization of trades in our design, the throughput could be much higher.

Interestingly, at low values of $\tau$, the throughput of the system goes up with the number of corruptions. This is not an anomaly. Recall that if a malicious trader does not respond within the timeout $\tau$, the seller assumes a `NO-TRADE` response and executes it. Recall also, that executing the `trade` scenario takes $\approx 350$ms. In comparison, `NO-TRADE` scenario costs almost nothing. Therefore the `NO-TRADE` scenario, with a timeout $\tau$ ¡ the (running time) cost of a `trade` scenario, costs less than the `trade` scenario. This means that, with some corruptions, some trades are cheaper to execute compared to when there are no corruptions (all honest players trigger the `trade` scenario). Therefore, more trades go through, and the system's throughput goes up. This effect disappears as soon as the value of the timeout $\tau$ goes near and above the (running time) cost of the `trade` scenario. While setting a low timeout $\tau$ 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) or it will start causing timeouts for honest players.

Now, 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 is not only better overall, It also removes the advantage (discussed above) the seller would have had at low values of $\tau$ in the Standard DoS attack. Concretely, a malicious seller would wait 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 $\texttt{delayBudget} = \tau - \texttt{RoundTripTime}$. The negative effect of such attack is seen in Fig. 10. The throughput has gone down for all values of $\tau$. 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. We call this attack *Worst-case Throughput attack* because this is the most an adversary can slow down the system.

Drawing from the observations above, we can conclude that the choice for value of $\tau$ should always be the typical round-trip time (perhaps with some noise). This way, we suffer no loss in throughput even against a determined adversary who wants to pay (via `trade` responses) to slow down the system, and in Standard DoS attack, we gain in throughput. Finally, consider that in real life some honest sellers may also respond with `NO-TRADE` e.g., if the prices are not favorable. Therefore, in our setup, the value of 205 trades per minute at $\tau = 100$ should be considered the lower bound on throughput.

# 9 Comparison with Uniswap

In this section we compare our work with Uniswap[Uni18], the DEX with highest market cap on Ethereum at the time of this writing. We provide an overview of Uniswap in Section 9.1 and compare it with this work in Section 9.2.

## 9.1 Uniswap

Uniswap is a 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 is *Uniswap v2*. A newer version *Uniswap v3*, primarily to facilitate more fine-grained control for liquidity providers, is planned for mid-2021.

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[11] to V2Router02 and averaged the gas cost of the invoked methods. These are listed in Table 2. The methods prefixed with *swap* perform various types of trades and are, therefore, relevant for comparison with our system.

## 9.2 Comparison

Recall that our system uses a market making algorithm in a black box mannner i.e. any market making algorithm could be plugged into it. Therefore, we focus our comparison on the parts unrelated to marking making itself. A summary of the comparison is presented in table Table 3.

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

Second, trade execution in our work is bounded by the round trip time of the network, it takes 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) get picked up in about 30 seconds. We can safely say that a trade in Uniswap takes at least 15 seconds (half of etherscan's current estimate). This is much larger than about 0.3 seconds in our system. Moreover, note that that not all trades get mined in the very next block, a large number of them end up waiting for a few blocks before getting picked up, increasing the trade execution delay. Our design is more amenable to a fast trading environment.

Third, in Uniswap and similar systems, miners are free to order trades the way they prefer. This gives them an opportunity for profit by e.g. including favorable trades first. On the contrary, trades in our construction are ordered on first come first served basis and this order is fixed before the corresponding transactions are broadcast to the blockchain, nullyfing miners' influence on trading.

Moreover, precisely because of the above mentioned miners' influence, traders on Uniswap (and similar systems) 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 trader to get their trade executed sooner. Such trading behavior induces the, so called, *gas price auction*s attack. Gas price auctions needlessly raise transaction cost for the traders, and any other players who may need to get a transaction posted sooner.

[11] block 12,162,664 to block 12,231,464

Table 2: 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.

Table 3: Comparison Summary

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

$*$ Based on average cost of one million trade transactions observed from block 12,162,664 to 12,231,464. A trade transaction is a call to any of the swap methods of the V2Router02 contract.

† Corresponding transaction can be seen at: `https://etherscan.io/tx/0xa87b492f2945d2a99ca1f8e2d9530599c040f00c3257f989f9c2822e20b2ed5e`). This is the most expensive transaction we observed in our million transaction dataset. There may be more expensive transactions outside this interval.

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

Transactions in our system are merely moving the funds around and may be mined in any order. Hence traders have no incentive to pay higher than usual gas price, keeping transaction costs low for everyone.

Fourth, gas cost for a trade in Uniswap is variable. In our observation of over a million transactions, the average gas cost is $141K$. Depending on the trade, it can be much higher e.g. gas cost for the transaction 0xa87b492f2945d2a99ca1f8e2d9530599c040f00c3257f989f9c2822e20b2ed5e is over $1,316K$. We note 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 $\approx 340$ trades per minute. This is less than our upper bound of 475. Concretely, highest daily volume[12] on Uniswap has been $\approx 199K$ transactions. On average, this means about 138 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 happens sequentially—on first come first served basis—negatively affects our throughput. Despite this, we achieve at least 200 trades per minute (much higher than the highest volume of the arguably biggest DEX—Uniswap—at the time of this writing). We stress that this throughput was achieved on a mid-range consumer machine. With dedicate server running on a high speed network, this can be significantly mitigated and, therefore, we do not see it as a major problem in practice.

# References


[Air18] AirSwap. Airswap, 2018.

[ASW98] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures (extended abstract). In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 591–606. Springer, Heidelberg, May / June 1998.


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

[Ban] Bancor. Bancor network.

[BDF21] Carsten Baum, Bernardo David, and Tore Frederiksen. P2dex: Privacy-preserving decentralized cryptocurrency exchange. Cryptology ePrint Archive, Report 2021/283, 2021. https://eprint.iacr.org/2021/283.

[BDM16] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 261–280. Springer, Heidelberg, September 2016.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

[BP09] Henry Berg and Todd Proebsting. Hanson's automated market maker. *Journal of Prediction Markets*, 3:45–59, 01 2009.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can03] Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. http://eprint.iacr.org/2003/239.

[CC00] Christian Cachin and Jan Camenisch. Optimistic fair secure computation. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 93–111. Springer, Heidelberg, August 2000.

[CGGN17] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 229–243. ACM Press, October / November 2017.

[CGJ+17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 719–728. ACM Press, October / November 2017.

[Cur20] Curve. Curve, 2020.

[Das05] Sanmay Das. A learning market-maker in the glosten–milgrom model. *Quantitative Finance*, 5(2):169–180, 2005.

[DEF18] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. FairSwap: How to fairly exchange digital goods. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 967–984. ACM Press, October 2018.

[Del18] Ether Delta. Etherdelta, 2018.

[DGK+20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy*, pages 910–927. IEEE Computer Society Press, May 2020.

[DH20] Apoorvaa Deshpande and Maurice Herlihy. Privacy-preserving cross-chain atomic swaps. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *FC 2020 Workshops*, volume 12063 of *LNCS*, pages 540–549. Springer, Heidelberg, February 2020.

[DMI08] Sanmay Das and Malik Magdon-Ismail. Adapting to a market shock: Optimal sequential market-making. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 361–368, December 2008.

[DW15] 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*, page 3–18, Berlin, Heidelberg, 2015. Springer-Verlag.

[EFS20] Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. OptiSwap: Fast optimistic fair exchange. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIACCS 20*, pages 543–557. ACM Press, October 2020.

[Fuc19] Georg Fuchsbauer. WI is not enough: Zero-knowledge contingent (service) payments revisited. Cryptology ePrint Archive, Report 2019/964, 2019. `https://eprint.iacr.org/2019/964`.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[Glo89] Lawrence R. Glosten. Insider trading, liquidity, and the role of the monopolist specialist. *The Journal of Business*, 62(2):211–235, 1989.

[GM85] Lawrence R. Glosten and Paul R. Milgrom. Bid, ask and transaction prices in a specialist market with heterogeneously informed traders. *Journal of Financial Economics*, 14(1):71 – 100, 1985.

[GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 473–489. ACM Press, October / November 2017.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[Gug20] Joël Gugger. Bitcoin-monero cross-chain atomic swap. Cryptology ePrint Archive, Report 2020/1126, 2020. `https://eprint.iacr.org/2020/1126`.

[Her18] Maurice Herlihy. Atomic cross-chain swaps. In Calvin Newport and Idit Keidar, editors, *37th ACM PODC*, pages 245–254. ACM, July 2018.

[HLG20] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. The arwen trading protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 156–173, Cham, 2020. Springer International Publishing.

[HLS19] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain deals and adversarial commerce. *Proc. VLDB Endow.*, 13(2):100–113, October 2019.

[HLY19] Runchao Han, Haoyu Lin, and Jiangshan Yu. On the optionality and fairness of atomic swaps. Cryptology ePrint Archive, Report 2019/896, 2019. https://eprint.iacr.org/2019/896.

[IDE18] IDEX. Idex, 2018.

[KL10] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. In Josef Pieprzyk, editor, *CT-RSA 2010*, volume 5985 of *LNCS*, pages 252–267. Springer, Heidelberg, March 2010.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[Kyb18] Kyber. Kyber, 2018.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.

[Net18] Raiden Network. What is raiden network?, 2018.

[PD16] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[PS07] D. Pennock and R. Sami. Computational aspects of prediction markets. In *Algorithmic Game Theory*. Cambridge University Press, 2007.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 643–673. Springer, Heidelberg, April / May 2017.

[Uni18] Uniswap. Uniswap exchange protocol, 2018.

[WB17] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain, 2017.

[Wik18] Bitcoin Wiki. Zero knowledge contingent payment, 2018.

[Wik20a] Bitcoin Wiki. Atomic swap, 2020.

[Wik20b] Bitcoin Wiki. Hash time locked contracts, 2020.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[WZ04] Justin Wolfers and Eric Zitzewitz. Prediction markets. *Journal of Economic Perspectives*, 18(2):107–126, June 2004.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[ZHL+19] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. XCLAIM: Trustless, interoperable, cryptocurrency-backed assets. In *2019 IEEE Symposium on Security and Privacy*, pages 193–210. IEEE Computer Society Press, May 2019.

# A Works on Fair Exchange

Several early works [Yao86, GMW87, ASW98, CC00, KL10] 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 [BK14, KZZ16]. An interesting use-case of fair exchange is to enable sale/purchase of assets with cryptocurrency [Wik18, BDM16, CGGN17, Fuc19]. 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 [DEF18] provided a more efficient solution in terms of off-chain costs. OptiSwap [EFS20] further improved over [DEF18] 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 [Uni18], 0x [WB17], AirSwap [Air18], EtherDelta [Del18], Bancor[Ban], Idex [IDE18], Kyber [Kyb18], Curve [Cur20], 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) [Wik20a, Wik20b, Her18, ZHL+19, HLY19, Gug20, DH20]. 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 [HLS19]. HTLCs technique has also been used to workaround the scalability problem of blockchains [DW15, GM17, HLG20, Net18, PD16].

# B Additional Preliminaries

## B.1 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 $\lambda$) and returned to the party (and the mapping $(x, \rho)$ is internally stored). If $x$ has been queried before, the corresponding $\rho$ is returned.

## B.2 Bulletin Board

The bulletin board BB allows the following queries (all of which can be emulated in modern blockchains; cf. [CGJ+17] for a formal description):

- `getCounter`. The bulletin board returns the current value of the counter $t$: $t \leftarrow \texttt{BB}(\texttt{getCounter})$.
- `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 $t$: $t \leftarrow \texttt{BB}(\texttt{post}, x)$.[13]
- `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, $x \leftarrow \texttt{BB}(\texttt{getContent}, t)$.

[13] In [CGJ+17] 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.

**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. [GKL15, PSs17, BMTZ17]. We assume familiarity with this notion and refer to B.2 for more detail.

# C Proofs

## C.1 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 $(\mathtt{new\text{-}request}, P_i)$ 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 $\mathtt{Ver}(\mathtt{pk}_{\mathtt{MM}}, \sigma, h||\mathtt{pk}_i||\mathtt{e}_i) = 0$ then ignore the message received from $\mathtt{MM}^\star$, continue as follows otherwise.
  - Add $\mathtt{ticket}_1$ to the list $\mathtt{Tickets}$ and send $(\mathtt{setPrice}, \mathtt{sid}, P_i, \mathtt{price}^{\Xi\to\check{T}}, \mathtt{price}^{\check{T}\to\Xi})$ 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[14] that starts from $h_{\mathtt{temp}}$ and finishes in $h$ do the following

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

      else send $(\mathtt{setAdvTrade}, \mathtt{sid}, P_j, \mathtt{trade}_j)$ 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 $(y, \sigma)$ to $\mathtt{MM}$.
- Upon receiving any command, if no message $(h', \sigma', \mathtt{requests}, \sigma^\star, \mathtt{e})$ is posted on the BB within $\Delta$ 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 $\mathtt{notAbort} \leftarrow \mathtt{notAbort} \wedge \mathtt{verification}(h^\star, h', h, \mathtt{pk}_i, \mathtt{requests})$

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

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

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

Set $h^\star \leftarrow h'$ and send $(\texttt{setOutput})$ to $\mathcal{F}^{\texttt{trade}}$, $\texttt{e} = \texttt{e} + 1$.

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

1. $\mathcal{F}^{\texttt{trade}}$ aborts because the simulator sees (with respect to a party $P_i$) in the hash chain a value $\texttt{trade}_i$ 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 $\texttt{pk}_i$.
2. $\mathcal{F}^{\texttt{trade}}$ 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 $\texttt{MM}$ is honest and an arbitrary set of traders can be corrupted. For simplicity, we consider only the case where MM and exactly one party $P_k$ is honest. The proof can be easily generalized to the case where there is more than one honest trader. Formally, $\mathcal{S}_P$ acts as follows.

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

- Upon receiving $(\texttt{request}, \texttt{pk}_i)$ from a corrupted party $P_i$ send $(\texttt{request}, \texttt{sid}, P_i)$ to $\mathcal{F}^{\texttt{trade}}$.
- Upon receiving $(\texttt{setPrice}, \texttt{sid}, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ from $\mathcal{F}^{\texttt{trade}}$ do the following.
  - Compute $h' \leftarrow \mathcal{H}(h||\texttt{pk}_i)$ and set $h \leftarrow h'$.
  - Compute $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h, \texttt{pk}_i||\texttt{e})$.
  - Send $\texttt{ticket}_1 := (h, \sigma, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi}, \texttt{pk}_i)$ to $P_i$.
- Upon receiving $(\texttt{trade}, \sigma_i)$ from a corrupted $P_i$, if $\texttt{Ver}(\texttt{pk}_i, \sigma_i, h||\texttt{trade}_i) = 0$ then send $(\texttt{ok}, \texttt{sid}, \texttt{NO-TRADE})$ to $\mathcal{F}^{\texttt{trade}}$ on the behalf of $P_i$, else send $(\texttt{ok}, \texttt{sid}, \texttt{trade}_i)$ to $\mathcal{F}^{\texttt{trade}}$.
- If $\mathcal{F}^{\texttt{trade}}$ replies with $\texttt{ok}$ then do the following
  - Compute $h' \leftarrow \mathcal{H}(h||\texttt{trade})$ and set $h \leftarrow h'$.
  - Compute $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h)$.
  - Set $\texttt{requests}[k] \leftarrow (\texttt{pk}_i, \texttt{trade}, \sigma_i)$.
  - Run $\texttt{MMalgorithm}(\texttt{trade}, \texttt{price}^{\Xi\to\check{T}}, \texttt{price}^{\check{T}\to\Xi})$ thus obtaining $\texttt{price}^{\Xi\to\check{T}'}, \texttt{price}^{\check{T}\to\Xi'}$ and set $\texttt{price}^{\Xi\to\check{T}} \leftarrow \texttt{price}^{\Xi\to\check{T}'}, \texttt{price}^{\check{T}\to\Xi} \leftarrow \texttt{price}^{\check{T}\to\Xi'}$.

  If $\mathcal{F}^{\texttt{trade}}$ replies with $\texttt{ko}$ then do the following
  - Compute $h' \leftarrow \mathcal{H}(h||\texttt{NO-TRADE})$ and set $h \leftarrow h'$.
  - Set $\texttt{requests}[k] \leftarrow (\texttt{pk}_i, \texttt{NO-TRADE}, 0^\lambda)$.
  - Compute $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h)$.

- Upon receiving (getTrades) forward it to and $\mathcal{F}^{\texttt{trade}}$. If $(\texttt{Trades}, \texttt{e})$ is received from $\mathcal{F}^{\texttt{trade}}$ then post $(h, \sigma, \texttt{Trades}, \sigma^\star, \texttt{e})$ to the BB, where $\sigma \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, h)$ and $\sigma^\star \leftarrow \texttt{Sign}(\texttt{sk}_{\texttt{MM}}, \texttt{Trades})$, else ignore the command[15].

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

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

□

We can argue that due to the unforgeability of the signature scheme the first case cannot occur. The second case instead cannot occur due to the security of the BB.

## C.2 Proof of Theorem 3

*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 1 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. □

[15] We note that by construction the trades encoded in requests are the same as in Trades.