UMA Data Verification Mechanism: Adding Economic Guarantees to Blockchain Oracles

DRAFT — Version 0.2

April 24, 2020

Abstract

Economic guarantees around the cost of corrupting blockchain "oracles" are critical for the development of useful smart contracts, particularly in decentralized finance (DeFi) applications. This paper introduces a simple "Cost of Corruption" vs "Profit from Corruption" framework to analyze the economic security of a blockchain oracle. Using this framework, we present a novel *Data Verification Mechanism* oracle construction that guarantees the economic security of a smart contract and oracle system in a fully decentralized and permissionless blockchain setting.

Contents

1 Introduction

Many use-cases for blockchains and smart contracts require trustless access to off-chain information. Decentralized financial contracts, for example, require accurate price data for valuation, margining and settlement.

The mechanism used to report off-chain information to a blockchain or smart

contract is typically referred to as an oracle. Despite a large body of existing research into oracle system design, current approaches are missing one key feature: an *economic guarantee* around the cost of corrupting an oracle system.

We argue that economic guarantees are critical for the development of useful smart contracts. Since blockchains are pseudo-anonymous, they cannot use most of the identity-based tools typically used to enforce contracts in the "real-world"—like the threat of lawsuits or jail-time. Public blockchains can only rely on economic incentives to influence behavior, meaning that the data an oracle provides to a smart contract can only be believed if it can be shown there is no economic incentive to corrupt the oracle.

In this paper we: i) show how economic guarantees are critical for the development of usable smart contracts, ii) define a framework for constructing economic guarantees within the permissionless blockchain environment, and iii) propose a *Data Verification Mechanism* (DVM) that adds economic guarantees to measure the accuracy of the information an oracle provides.

1.1 Motivation: Why Economic Guarantees Matter

In the "real-world" the enforceability of contracts is an obvious prerequisite to the development of useful legal contracts. Legal agreements are useless if participants cannot trust that the contract will be accurately interpreted and enforced by the rule of law.

The same logic applies to smart contracts on the blockchain: smart contracts are worthless if participants do not believe they will be accurately interpreted and enforced by the blockchain system. For deterministic, on-chain calculations, this is easy: the self-policing and public nature of the blockchain system itself provides guarantees for the accuracy of those computations. But for smart contracts that rely on off-chain information delivered by an oracle, their utility depends on the accuracy of that oracle.

Imagine two counterparties entering into a financial derivative contract where, in the best/worst case, either counterparty could win or lose $1B. Each counterparty has an incentive to bribe, manipulate, or corrupt the system to win $1B. In the traditional legal system, counterparties do not do this for fear that their actions would be considered criminal and land them in jail; the real world has large external costs for corrupting the system. But in the pseudo-anonymous world of blockchains, no such external cost exists—counterparties are free to manipulate the system without consequence, and it is dangerously easy to do so.

UMA's system operates under that assumption that in a permissionless blockchain, *private economic incentives* are the only tool that can be used to change behavior. This is why economic guarantees matter: in order to be able to trust a smart contract that relies on off-chain inputs (via an oracle), the cost to incentivize bad behavior must be measurable. Without an economic bound on

this cost of corruption, it becomes impossible to prove that the smart contract will not be manipulated.

1.2 Defining the Oracle Problem

Oracles can be conceptually decomposed into two functions: a mechanism for *reporting* off-chain data on to the blockchain, and a mechanism for *verifying* the accuracy of the reported information. In most smart contract designs, if the contract participants agree on the data provided by the reporting mechanism, there is no need for verification. The verification mechanism is only required in the event of a dispute. This verification system is the focus of this paper[1].

Formally, the verification mechanism is a decision-making agent[2] that validates the reported data. Since behavior on a permissionless blockchain is only motivated by economic incentives, there exists some bribe that can corrupt the verification mechanism and produce dishonest behavior. We define the smallest possible bribe that can alter the verification system's behavior as the *Cost of Corruption* (CoC). There also exists some maximum amount of profit that can be stolen by corrupting the oracle. We define this maximum profit as the *Profit from Corruption* (PfC). It follows that, in order for oracle agent to remain honest, the cost of corruption must always remain greater than the profit from corruption, i.e.: $CoC > PfC$.

As long as this $CoC > PfC$ inequality holds true, there is no economic incentive for a rational actor to bribe and corrupt the oracle.

1.3 Prior Work

Most of the existing work in the oracle space has been focused on reporting—not verification—mechanisms. Reporting focused approaches like Oraclize [1] and TownCrier [2] require trust in the API source for the data. Other projects like ChainLink [3], Witnet [4], Rlay [5], Verity [6], and Tellor [7] use decentralized designs to distribute the reporting mechanisms across multiple nodes for redundancy, and implement a reward system to incentivize node performance. Other approaches like Astraea [8] and Shintaku [9] randomize the selection of the reporter to make collusion harder.

Prediction market systems like Truthcoin [10], Augur [11] and Amoveo [12] have made important contributions to verification mechanism design; these systems all use Schelling Point voting schemes to incentivize truthful voting and have heavily influenced UMA's design.

Each of these existing systems has its own set of trade offs. UMA is building

[1] The verification mechanism can be thought of as a consensus machine for external (off-chain) data.

[2] We are using "agent" in the economic sense: an agent is a decision-maker that could be comprised of an unknown number of individual actors.

on top of these reporting systems by providing a verification mechanism that is complementary to many existing projects: the UMA verification mechanism could be layered on top of another oracle, adding economic guarantees to the accuracy of that oracle's reported data and allowing UMA to function as a verification layer of "last resort."

2 System Overview

The UMA *Data Verification Mechanism* (DVM) verifies off-chain data brought onto the blockchain. In times of uncertainty, smart contracts can call on the DVM to accurately report and verify any publicly verifiable data. For smart contracts that rely on off-chain data, UMA provides the data feed of last resort.

Our goal is to design a verification mechanism where the $CoC > PfC$ inequality holds true. This requires three things:

1. The Cost of Corruption (CoC) for the system can be calculated;
2. The Profit from Corruption (PfC) for the system can be calculated;
3. There exists a working a mechanism to enforce $CoC > PfC$.

At a high level, the system accomplishes these objectives by:

2.0.1 Measuring the CoC with Tradable Voting Rights

UMA's DVM uses a Schelling-Point style voting system to resolve disputes. The system has voters who hold freely-tradable voting tokens. Token holders are paid a reward for voting correctly and penalized otherwise. To manipulate an oracle vote, an attacker must control some minimum threshold of tokens, defined below as y. The market value of this number of tokens represents the cost of corrupting the oracle (CoC).

2.0.2 Measuring PfC via Contract Registration

All contracts using the system must be registered with the DVM. Each registered contract must be able to report the value that could be won or lost if the data inputs to that contract were manipulated. By summing these values across all contracts, the total profit from corruption PfC for the system as a whole can be determined.

2.0.3 Enforcing $CoC > PfC$ via a Variable Fee Policy

If the CoC, as measured by the market price of the freely tradable vote tokens, approaches the PfC, the system levies a fee on all contracts proportional to each

contract's contribution to the *PfC*—larger contracts pay proportionally larger fees. The fee is used to purchase and burn tokens, increasing the token value and therefore increasing the *CoC*. The fee amount can vary depending on how close the market price of the token is to the minimum price needed to secure the system.

2.1 System Parameters

Parameter	Meaning
Contract	A smart contract that relies on data inputs from an oracle to determine the value of economic payouts
Counterparty	The participants in the smart contract using the oracle
Voter	A token holder that submits values for the oracle and is rewarded for doing so
Data request	A contract request to the oracle for a verified price at a specified datetime
Token	Native token for the system; each token represents 1 voting right.
p	Price of the native token
p_{floor}	The minimum token price needed to ensure the system cannot be corrupted
p_{safe}	Desired "safe" price for token so that the oracle is not vulnerable to corruption
s	Total supply of tokens. Each token represents 1 vote.
y	Minimum number of tokens an attacker needs to corrupt the system assuming some token holders do not vote (η)
η	Percentage of total tokens that do not vote during a given voting period
PfC	The total Profit from Corruption
PfC_{contract}	The maximum profit that can be earned from a specific contract if that contract's data inputs are corrupted
F	Total fees charged to the whole system to access the oracle.

F_{contract}	The fee charged to a specific contract.
r	Reward paid to token holders that voted correctly
t	Voting Period

3 Voting System Architecture

Contracts participants seeking a verified data point initiate a `data_request`; DVM voters respond to these requests. Each data request asks for a verified data point at a specified datetime from a predetermined data source. Multiple data requests, potentially spanning multiple data sources, are batched into voting periods. Token holders are compelled to respond accurately to each data request through a system of economic incentives.

3.1 Data Request Submission Phase

A contract counterparty submits a data request to the DVM. All data requests are batched into a voting period t.

3.2 Voter Commit Phase

Token holders then have one voting period to commit a hash of the data that they believe is accurate.

3.3 Voter Reveal and Reward Phase

At the end of the commit phase, submitted votes are publicly revealed and scored. Each tokenholder's token balance is snapshotted when the first voter reveals their vote. Voters have one full voting period to reveal their commit. At the end of the reveal period, the oracle calculates the verified price and pays rewards to voters with the following logic:

1. If the distribution of votes is highly unimodal (frequency of the mode $>$ 50%), the mode is returned as the verified price. Token holders who voted for the mode are rewarded; all other token holders are penalized.
2. If the frequency of the mode of the votes is $<$ 50%, the median price is returned as the verified price. Token holders who submitted votes between

the 25th and 75th percentile are rewarded; all other token holders are penalized. [3]

The DVM collects the mode, median, total number of votes, and 25th percentile and 75th percentile values. The median and first and third quartile values can be calculated using rolling min-max heap algorithm [15].

3.4 Reward Distribution and Voter Incentives

Rewards are paid to incentivize i) voter participation and ii) accurate voting. UMA's reward system is heavily inspired by Schelling Point style systems popularized by Paul Sztorc, Vitalik Buterin and others [10, 16]. The DVM supports data requests where there is a known "correct" answer in addition to requests where the answer is more ambiguous. For price requests where there is a widely agreed precise value (like the closing price of the S&P 500 index), the system incentivizes voters to report that precise value or risk being penalized for not reporting the mode. For price requests where there is no precise value (i.e. less liquid or OTC traded assets), the system incentivizes voters to vote as close to the median as possible or risk getting penalized for not voting within the 25-75th percentiles.

The size of the reward paid to accurate voters is determined by a system parameter r. This parameter controls redistribution of the tokens: for each vote, accurate voters increase their token ownership by a factor of r at the expense of all other (non-participating or inaccurate) voters. This redistribution policy can be implemented by either a fixed-token supply model (where tokens are taken away from non-participating or inaccurate voters and transferred to accurate voters), or by an inflationary model (where accurate voters are rewarded with new tokens and non-participating or inaccurate voters are diluted).

The r parameter is set as a system parameter that should be adjusted to ensure sufficiently high voter participation. An economic model analyzing the optimal level for r is an area of active research. Future designs could include a system to programmatically adjust r.

4 Economically Securing the Oracle

4.1 Measuring the Profit from Corruption

High value smart contracts that depend on accurate data from an oracle present a lucrative target for attackers. In order to protect a contract with economic guarantees, it is essential to measure the total value that is at risk. We define

[3] This is not reflected in the v1 implementation of the DVM, but will be implemented in the future.

the maximal value that can be extracted if a contract's data inputs were to be corrupted as PfC_{contract}.

For example, assume Alice and Bob want to enter into a total return swap style smart contract on $1mm of gold. Alice wants to go long and Bob wants to short gold. Alice and Bob agree to secure their contract with a 10% margin requirement, or $100k at the start of the trade. If Bob were to control the price feed used to re-margin the contract, Bob could set the price of gold to 0, immediately re-margin the trade, and "steal" Alice's $100k of margin. Alice could do the same thing by setting the price of gold to some large number. Therefore the initial PfC_{contract} for this contract is $100k.

Under a worst-case analysis, the maximum value an attacker could steal from the system as a whole can be calculated by summing PfC_{contract} for all contracts. We define this system-wide number as the *Profit from Corruption* or *PfC*.

4.2 Registering Known Contracts

The *PfC* can be measured if all contracts relying on the oracle are known and accounted for. To support this, the DVM requires all contracts to be constructed using an oracle-approved bytecode template. This template enforces a standard interface that: i) registers each contract upon its initial creation, ii) pays fees to the system as required, and iii) and maintains a publicly accessible function to accurately calculate a contract's PfC_{contract}. The system-wide *PfC* can then be computed by inspecting and summing the PfC_{contract} for all contracts in this registry.

Templates are added, updated, and removed from the system by vote; token holders have an incentive to only approve templates that have been thoroughly vetted to accurately calculate and report their *PfC* values.

Contracts that do not use known and approved bytecode templates cannot access the oracle's data or initiate data requests. Data requests from an unknown contracts are denied, potentially freezing the contract and preventing its settlement. Methods to avoid registration, otherwise known as *tax evasion*, are discussed in §8.1.

4.3 Measuring the Cost of Corruption

The total supply of tokens is defined by S. We define η as the expected percentage of token holders that do not participate in the voting process. To control the oracle, an attacker must control at least half of all participating tokens. We define the minimum number of tokens as attacker needs as y:

$$y > \frac{(1-\eta)}{2} S \tag{1}$$

Given a cost p for a vote token, the cost to control the system is $p \cdot y$. We define p_{floor} as the minimum vote token cost where it is unprofitable to corrupt the system. Therefore, the minimum *Cost of Corruption*, or CoC, is:

$$CoC = p_{\text{floor}} \cdot y \tag{2}$$

4.4 Calculating the p_{safe} Target

To eliminate any economic incentive to corrupt the oracle, we need the *Cost of Corruption* to exceed the *Profit from Corruption*. Mathematically this means:

$$\begin{aligned} CoC &> PfC \\ p_{\text{floor}} \cdot y &> PfC \\ p_{\text{floor}} \cdot \frac{(1-\eta)}{2} S &> PfC \end{aligned} \tag{3}$$

Intuitively, we can increase the left-hand side of the inequality by minimizing η (non-participating voters). As previously discussed, the parameter r does this by rewarding accurate voting.

The S and PfC parameters are observed or given. This leaves p_{floor} as the remaining variable we can affect to maintain the inequality. Rearranging:

$$p_{\text{floor}} > \frac{2(PfC)}{(1-\eta)S} \tag{4}$$

In order to secure the oracle, we need to maintain the vote token price p_{floor} above this threshold. To maintain some resiliency in the system, we target a desired "safe" price target p_{safe} such that $p_{\text{safe}} > p_{\text{floor}}$.

Targeting a $p_{\text{safe}} >> p_{\text{floor}}$ would secure the system from shocks or volatility in the price of the vote token; however this will ultimately increase the fees charged to undesirable levels, likely leading to decreased usage of the system. It is in the best interest of the vote token holders to target a p_{safe} that maintains the security of the system while minimizing the cost to users. We hypothesize that a p_{safe} target of $1.5p_{\text{floor}}$ would be optimal in most contexts; a more rigorous model to support this hypothesis is forthcoming.

5 Maintaining the p_{safe} Target

It is in the collective best interest of contracts using the oracle to maintain the safe price target p_{safe}: this price gives contract counterparties an economic guarantee that their contracts cannot be manipulated via oracle inputs.

UMA maintains this target by initiating programmatic, repeated, token buybacks if the oracle token price drops below p_{safe}. All purchased tokens are burned, reducing token supply to increase the market price p. The funds needed to conduct these buybacks are raised by collecting fees from contracts using the DVM system.

5.1 Calculating Buyback Size

The buyback system requires logic to compute the quantity of vote tokens that should be purchased at any given period. We can use the math describing and automated market maker (AMM) to calculate the size of a purchase needed to achieve the desired change in the token price p. This logic can be used regardless of how the buyback is actually conducted.

Several AMM designs exist like LMSR [13] and $x * y = k$ market makers, which have recently been popularized by Uniswap [14]. Regardless of the AMM design chosen, the process used to calculate the buyback is straight forward: (i) measure the current price p;[4] (ii) if the current price p is below the target price p_{safe}, use the math of the AMM to calculate how many tokens need to be purchased to move the price back to p_{safe} under a given set of AMM parameters.[5]

5.2 Collecting Fees and Conducting the Buyback

Conducting a buyback requires that a total fee F is levied and collected across all system contracts to pay for the purchase of the tokens. Individual contracts are required to pay their pro rata share, F_{contract}, as measured by the contract's PfC_{contract} relative to the system-wide PfC:

$$F_{\text{contract}} = \frac{F \cdot PfC_{\text{contract}}}{PfC} \tag{5}$$

[4] To solve the circular oracle problem (the fact that this oracle requires an oracle for the token price), the system can add a `data_request` for the current price of the token to the previous voting round. This means a verified price (that is one voting cycle old) is always available.

[5] For example, assume a token supply of 100 units with a current price p of \$10 and a desired price p_{safe} of \$11. We can use the math of an $x * y = k$ AMM where 20% of the token supply has been committed to the liquidity pool to calculate the total tokens that need to be purchased such that $p = p_{\text{safe}}$: $100 * 20\% * \sqrt{\$10/\$11} = 0.93$ tokens. Therefore, paying p_{safe} for these 0.93 tokens implies that the total fees to be collected $\tau = \$11(0.93) = \$10.23$.

Many fee collection implementations are possible. One design could require contracts to pay their fee on a regular basis or face a penalty. Another design could employ a system of third party "tax collectors" who are rewarded for monitoring and enforcing fee payments.

Similarly, there are many potential mechanisms to conduct buybacks. One trust-minimized approach could task a third party foundation with conducting the buyback. Another approach could use an on-chain open auction process to purchase tokens until the market price reaches the target price p_{safe}. A third approach could use a Uniswap style AMM to conduct buybacks automatically.

The logic for the buyback process is summarized below:

Each voting period:

1. At the start of the voting period, calculate the desired target price p_{safe} by observing the system-wide *Profit from Corruption*
2. Measure the correct price p of the token
3. If the current price p is less than the target price p_{safe}, calculate the F fees that need to be collected to conduct the buyback
4. Collect the necessary fees F
5. At the end of the voting period, conduct the buyback

5.3 Analysis of Buyback Incentives

Because the logic for the buyback calculation and fee collection is immutably written into the system, there is certainty that the buybacks will occur. This creates an incentive for third party arbitrageurs to independently maintain the p_{safe} target and front run any potential buyback: if a third party can purchase tokens cheaper than p_{safe} , they will be able to profitably sell those tokens into the buyback mechanism.

The credibility of this mechanism could be questioned by the market if the price p is allowed to trade below p_{safe} for a prolonged period of time. In this situation, contract participants may fear that very high fees will be levied to push p back above the p_{safe} level; this may cause participants to close their contracts and leave the system. The risk is eliminated by conducting buybacks with enough frequency to prevent p from trading too far from it's target. Alternatively, the system can also be designed with a "rainy day" fund that collects modest fees when buybacks are not actually needed; this fund could then be used to (fully or partially) fund buybacks when required, helping to bolster the credibility of the buyback mechanism.

6 Expected Fees for Using the DVM

The DVM system is designed to levy the lowest fees possible while maintaining the $CoC > PfC$ economic guarantee. As such, the system is not rent-seeking—it is designed to minimize the fees required while maintaining the p_{safe} price target for the token.

The voting token has a fundamental value because fees are ultimately levied on contract participants to compensate the voters for their work in securing the system. It follows that the current value of the token should reflect the present discounted value of all future fees expected to be levied on contracts. More formally, the market capitalization of the voting token today, τ, (as measured by price times supply, or $p_\tau S_\tau$) should equal the summation of all expected future buyback fees F_t collected on PfC_t discounted back to today at a discount rate r:

$$p_\tau S_\tau = \mathbb{E}_\tau \left[\sum_{t=\tau}^{\infty} \frac{1}{(1+r)^{t-\tau}} F_t \right] \tag{6}$$

Since any fee F_t is a function of the total value secured by the system at the time t, the market's expectations for all future fees are a reflection of the market's expectation for growth in the total value secured by the system.

This is a useful result. It implies that if the market is expecting growth in the DVM's usage, that expectation should be reflected in a higher token price today. This means that the token price p can be maintained above the required p_{safe} level *without initially levying any fees at all.* In other words, the $CoC > PfC$ inequality can be maintained purely on expectations of growth.

This is no free lunch; fees will ultimately be levied at a future steady state when there is no expected growth in the usage of the system. However, this structure has the useful property of charging zero or de minimis fees in the early days of the system when growth expectations are high. This should help encourage early adopters to use the system.

At a future steady state, when growth in the usage of the system slows or stops, voters will need to be compensated for both their voting work and their cost of capital (the cost to buy and hold their voting tokens). Suppose an annualized return of 5% is sufficient to compensate vote token holders at this steady state. This means a fee of 5% of the system wide PfC will be collected from contracts and paid to vote token holders (annually). It is important to note, however, that the margin locked in the system is far smaller than the total notional of the underlying financial contracts. We expect most contracts to be designed so that their total margin at risk (aka their PfC_{contract}) is a small fraction of the total notional secured—for example, a \$100 notional contract might be secured with \$10 of margin, implying a PfC_{contract} of \$10 on \$100 of notional risk. This means the 5% annualized fee on PfC_{contract} becomes a

0.50% fee on the contract's total notional—inline or below current transaction costs on most existing centralized systems. Furthermore, as blockchain speeds increase and scale, faster and cheaper transactions will allow for more frequent remargining of contracts, allowing for more capital efficient contract designs with lower *PfC* requirements per unit of risk. This will drive steady-state fees to even lower rates.

7 DVM and Smart Contract Interaction

To help illustrate the interaction between a contract and the DVM, we walk through the example trade introduced in §4:

Assume Alice and Bob want to enter into a total return swap style smart contract on $1mm of gold. Alice wants to go long and Bob wants to short gold. Alice and Bob agree to secure their contract with a 10% margin requirement, or at least $100k at the start of the trade. Failure to maintain this margin requirement results in a default and loss of any remaining margin. (For more details on the design of this contract, see §3.3 of *UMA: A Decentralized Financial Contracts Platform* [17]).

7.1 Creating the Contract and Collecting Fees

Alice and Bob initially create their contract using a bytecode template that is known to the DVM. This template implements an interface to report an accurate *PfC* for the total-return contract Alice and Bob are looking to enter. In this contract design, PfC_{contract} is the largest total margin held on either side of the contract. For example, suppose Alice (the long) had $150k of margin in her side of the trade. Bob—or an attacker conspiring with Bob—could steal that $150k by setting the data inputs of the contract to a price where Bob (the short) would be immediately owed all of Alice's margin. If Alice has $150k of margin and Bob has $130k of margin securing their trade, the PfC_{contract} would be the larger value, or $150k.

When Alice and Bob first create their contract, the contract is registered with the DVM as described in §4.2. This allows the DVM to add the contract's PfC_{contract} to the system wide *PfC* number used to calculate the p_{safe} token value.

If the market price p drops below the p_{safe} target, fees can be calculated proportional to the contract's *PfC* and collected support the p_{safe} price, as described in §5. The bytecode template defines a standard interface to support this fee calculation and collection.

7.2 Unhappy Path: Disputing Contract Inputs

If there is a dispute in the price data used to value their contract, Alice and Bob query the DVM. Typically, the contract would terminate when the dispute is initiated, margin withdrawals would be locked until the result is returned, and a `data_request` for the price of gold at the time of the dispute would be submitted to the DVM for inclusion in the next voting period. Voters would be asked to commit their votes for the price of gold at the requested timestamp; voters would later reveal their submissions and the DVM would calculate and return the mode/median result back to the contract.

To prevent Alice or Bob from initiating a meritless dispute, the contract that Alice and Bob entered can be designed to levy a fee or penalty to any party that inaccurately calls the DVM. This "cost" for meritless data requests further helps to prevent DoS and spam attacks on the DVM.

7.3 Happy Path: Remargining and Settling the Contract without the DVM

In the normal course of their trade, Alice and Bob can remargin their contract themselves. They can simply propose and use their own price inputs—provided they agree. Alice and Bob do not need to use any oracle or 3rd party so long as they can agree on the data used to value their contract.

At first glance, this seems surprising—why should the DVM exist at all? The fiat legal system provides a useful analogy: in the fiat world, counterparties commonly enter into private contracts without any intention of litigating those contracts. The vast majority of fiat contracts are honored without any 3rd party intervention. For the minority where counterparties cannot agree on the contract's terms, a legal system exists to resolve those disputes. It is the mere existence of the legal system that keeps most participants honest.

With Alice and Bob's smart contract, the DVM system performs a similar function. Alice and Bob only need to use the DVM if they fail to agree on the data inputs to their contract. In the optimistic case, Alice and Bob's contract could progress through its entire lifecycle and settle without any calls to any oracle or 3rd party. It is the very existence of the DVM that make this optimistic path the expected outcome.

Although the contract, following the happy path, can settle without the DVM system, we expect most implementations to use the DVM to calculate the final settlement of any trade. In this context, Alice and Bob's contract at expiry would call the DVM for a final `data_request` for the value of gold at the time of expiry. The DVM would process this vote, and the result would be used to calculate the final contract payout.

8 Attack Vectors

The cost of corruption framework is useful methodology for thinking through all potential attacks. Any corruption of an honest oracle means that the $CoC > PfC$ condition has been broken. This can occur if either the CoC or PfC numbers are incorrectly measured, or if the mechanism to enforce $CoC > PfC$ is broken. Below we analyze some potential attack vectors.

8.1 Tax Evasion and Parasitic Usage

To avoid paying any potential fees, an attacker could set up a contract that settles based on the outcome of the DVM, but is not listed in UMA's registry (see section 4.2). These "tax evaders", or *parasitic contracts* as Augur [11] calls them, are damaging to the UMA system because they undermine the ability to accurately measure the PfC, breaking the economic guarantees of the system. Below is a description of an example attack and solution, although they are not implemented in v1 of the UMA DVM.

A typical attack works as follows: Alice and Bob decide to enter into a $100 contract. Instead of registering a single $100 contract with the oracle, they register a $1 contract and create a second $99 parasitic contract that settles to whatever payouts are observed in the $1 contract. The parasitic contract free-rides off the system, potentially distorting the system-wide PfC.

Defense against parasitic usage is an area of ongoing and active research. Ideally, oracle results are inaccessible to any contract outside of the contract that submitted the request. However, most blockchain designs, like Ethereum, expose information that theoretically allows private contract data to be verified on-chain via a reconstruction of the chain's block headers. This makes guaranteeing the privacy of on-chain data impossible.[6]

A more promising approach involves "fuzzing" contract payouts to make it impossible for parasitic contracts to accurately reverse engineer any sort of useful result. In this approach, voters are asked to vote on final contract payouts rather than prices. Ahead of the vote, third parties are allowed to submit deposits that will be paid to a given contract participant without publicly revealing whom the deposit will be paid to. Instead, each third party privately communicates to each voter where they intend their deposit to go using an interactive zero-knowledge proof.[7] Voters are then asked to consider this secret deposit when submitting their votes for the final contract payouts.

An example is helpful: suppose Alice and Bob's valid contract settles with

[6] This could potentially be solved using a zero-knowledge or privacy preserving smart contract platform like AZTEC [18] or Zether [19].

[7] The prover can prove that they know a salt such that destination + salt results in the hash without revealing that salt to anyone. This proof would not be transferable to anyone other than the verifier(s) because of its interactive nature. See [20] for details.

$1 of total margin to pay out. Assume that without any "fuzzing" this contract would pay out $0.50 to each of Alice and Bob. Now suppose Charlie, a friend of Alice, makes an on-chain deposit of $1 to the contract, increasing the total margin pool to $2. Charlie submits a hash of the destination where he wants his $1 to go (to Alice) along with a secret salt. Charlie agrees to stay online and conduct an interactive zero knowledge proof with each of the voters to prove that he intends for his $1 deposit to go to Alice. All voters consider this when submitting their votes for the contract payout: voters report that $1.50 should go to Alice ($0.50 from the original contract and $1 from Charlie's deposit), and that $0.50 should go to Bob.

By "fuzzing" the contract, the DVM result is only useful to the original contract. Any parasitic contract looking at the final payouts of Alice and Bob's valid contract would pay out 75% of the contract value to Alice (because Alice got $1.50 of $2), when the correct answer is to pay out 50% of the contract value to Alice. This breaks the parasitic contract.

Interestingly, this approach creates a strong incentive for participants to *not* write parasitic contracts based solely on the *threat* of fuzzing. Suppose Alice and Bob did create a parasitic contract that read from the results of their valid contract. Alice has a strong incentive for Charlie (her friend) to make a large secret deposit to the valid contract so that she gets an unfair share of the parasitic contract payout. Bob knows this, so he has a strong incentive to have his friend Dave make an even larger deposit to increase his share of the parasitic payout. This process continues recursively, driving up the costs of creating a parasitic contract to the point where writing a parasitic contract becomes uneconomical.

8.2 Bribes and Collusion

Any voting system on a pseudo-anonymous blockchain is vulnerable to bribes and collusion. For example, a malicious actor with unlimited capital could bribe token holders to commit their votes in a particular way, or a group of 51% of token holders could collude to throw a vote.

The system is resilient to these types of attacks under the assumption that the token price p will drop significantly when an attack is detected. The reasoning behind this price drop is that the system—and therefore the token—is only valuable if the DVM is accurate and incorruptible. If the system is then shown to be corrupted, the token should have no value.

Using the assumption that p drops to zero if the system is successful manipulated, we can show there is no bribe an attacker can profitably pay token holders. If token holders believe their token value will drop to zero if the system is corrupted, each token holder will demand a bribe greater than their current token price p. Since an attacker needs to bribe y tokens to successfully manipulate a vote, the attacker will need to pay more than $p_{\text{floor}}(y) = CoC$ in

bribes. However, since the maximum profit an attacker can earn is the entire PfC amount, and since $CoC > PfC$, the attack would be unprofitable.

The assumption that p drops to zero can be loosened by setting the target price p_{safe} sufficiently far above the floor. If, for example, p_{safe} is set to twice the floor, bribing the system would still be unprofitable under the assumption that p drops even by half (and not to zero).

9 Future Work

This *draft* paper presents a highly parameterized system. Although some initial modeling has generated parameters that achieve the objectives of the system, namely incentivizing voter participation while maintaining low fees charged to contracts, more robust models for these parameters are needed.

Additional work is needed to better define the protocol specification beyond the core concepts articulated here. Research on governance procedures, mechanisms to adjust core system parameters, and strategies for initial token distribution are also needed.

Acknowledgments

The authors would like to thank Aleks Larsen, Alex Evans, Chris Burniske, Chris Tonetti, Dan Robinson, Daniel Que, Jill Carlson, Teemu Paivinen, as well as many other anonymous reviewers for their helpful feedback, comments, and suggestions.

References

[1] Provable (formerly Oraclize) Website. `provable.xyz`

[2] Zhang F, Cecchetti E, Croman K, Juels A, Shi E. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security 2016 Oct 24 (pp. 270-282). ACM.

[3] Ellis S, Juels A, Nazarov S. ChainLink: A Decentralized Oracle Network. 2017. `link.smartcontract.com/whitepaper`

[4] de Pedro AS, Levi D, Cuende LI. Witnet: A Decentralized Oracle Network Protocol. 2017. `witnet.io/static/witnet-whitepaper.pdf`

[5] Hirn M. Rlay: A Decentralized Information Network. 2018. `rlay-project.github.io/rlay.com/rlay-whitepaper.pdf`

[6] Mikeln M, Perovic L. Verity: Platform for Decentralized Real-World Data Feeds. 2018. verity.network/whitepaper.pdf

[7] Tellor: A Decentralized Oracle White Paper. 2019. tellor.io

[8] Adler J, Berryhill R, Veneris A, Poulos Z, Veira N, Kastania A. Astraea: A decentralized blockchain oracle. In 2018 IEEE Smart Data (SmartData) 2018 Jul 30 (pp. 1145-1152). IEEE. arxiv.org/pdf/1808.00528.pdf

[9] Kamiya R. Shintaku: An End-to-End-Decentralized General-Purpose Blockchain Oracle System. 2019. gitlab.com/shintaku-group/paper/raw/master/shintaku.pdf

[10] Sztorc P. Truthcoin: Peer-to-Peer Oracle System and Prediction Marketplace. 2015. bitcoinhivemind.com/papers/truthcoin-whitepaper.pdf

[11] Peterson J, Krug J, Zoltu M, Williams AK, Alexander S. Augur: a decentralized oracle and prediction market platform. 2018 Jul 12. augur.net/whitepaper.pdf

[12] Amoveo White Paper. 2018. github.com/zack-bitcoin/amoveo/blob/master/docs/white_paper.md

[13] Hanson R. Logarithmic markets coring rules for modular combinatorial information aggregation. The Journal of Prediction Markets. 2012 Dec 13;1(1):3-15.

[14] Uniswap White Paper. 2019. docs.uniswap.io

[15] Atkinson MD, Sack JR, Santoro N, Strothotte T. Min-max heaps and generalized priority queues. Communications of the ACM. 1986 Oct 1;29(10):996-1000.

[16] Buterin V. SchellingCoin: A Minimal-Trust Universal Data Feed. 2014. blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/

[17] UMA: A Decentralized Financial Contract Platform. 2018 Dec 3. github.com/UMAprotocol/whitepaper

[18] Williamson, Z. The AZTEC Protocol. 2018 Dec 4. github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf

[19] Bünz B, Agrawal S, Zamani M, Boneh D. Zether: Towards Privacy in a Smart Contract World. IACR Cryptology ePrint Archive. 2019;2019:191.

[20] Goldreich O, Micali S, Wigderson A. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. Journal of the ACM (JACM). 1991 Jul 1;38(3):690-728.