Distributed Blockchain Price Oracle

Léonard Lys, Maria Potop-Butucaru

Distributed Blockchain Price Oracle

Léonard Lys[1,2] and Maria Potop-Butucaru[2]

[1] LIP6, UMR 7606 Sorbonne University - CNRS, 4 place Jussieu 75252 Paris Cedex 05, France
[2] Palo IT, 6 rue de l'Amiral Coligny 75001 Paris, France

Abstract. Blockchain oracles are systems that connect blockchains with the outside world by interfacing with external data providers. They provide decentralized applications with the external information needed for smart contract execution. In this paper, we focus on decentralized price oracles, which are distributed systems that provide exchange rates of digital assets to smart contracts. They are the cornerstone of the safety of some decentralized finance applications such as stable coins or lending protocols. They consist of a network of nodes called oracles that gather information from off-chain sources such as an exchange market's API and feed it to smart contracts. Among the desired properties of a price oracle system are low latency, availability, and low operating cost. Moreover, they should overcome constraints such as having diverse data sources which is known as the freeloading problem or Byzantine failures.
In this paper, we define the distributed price oracle problem and present PoWacle, the first asynchronous decentralized oracle protocol that copes with Byzantine behavior.

Keywords: Blockchain oracle · price oracle · Decentralized finance.

1 Introduction

Decentralized finance (DeFi) is a term that emerged during the past few years to describe financial instruments that do not rely on centralized intermediaries such as brokerages exchanges or banks. In order to implement those instruments, DeFi protocols make use of smart contracts hosted on blockchain systems. Those smart contracts are programs that implement the logic of classical financial instruments. A wide range of applications is already in production, from interest-earning saving accounts to lending protocols to synthetic assets or trading platforms, etc. This industry is quickly gaining in popularity both in terms of the number of users and in market capitalization.

In order to function, a lot of those DeFi protocols make use of what is called blockchain oracles and more specifically blockchain price oracles. A price oracle is a system that provides exchange rates or prices to DeFi protocol's smart contracts. They gather data from off-chain sources, such as an API, and feed it to a smart contract on-chain. For example, a popular DeFi application consists in issuing a number of tokens to a user in exchange for collateral that will be locked in a smart contract, until the user pays back his debt. Obviously, for the

process to be fair, it is necessary to know the current exchange rate between the token issued and the token locked as collateral. This is where prices oracles come into the picture.

Price oracles can be split into two categories, centralized and decentralized. A centralized oracle relies on the observations of a single trusted entity while decentralized oracles gather information from several sources. In this paper, we focus on decentralized ones, as we consider that DeFi protocols should not rely on a single trusted entity.

Although decentralized price oracles have a central role in designing DeFi applications, there is very little academic literature that addresses fault-tolerant decentralized oracles. To the best of our knowledge, the only academic work addressing this problem is [2]. However, several non-academic reports propose ad hoc solutions practical solutions. In the line of non-academic work, the most interesting contributions are the band protocol [1] and DOS network [6].

Band protocol [1] is a public blockchain network that allows users to query off-chain APIs. It is built on top of the Cosmos-SDK and uses the Tendermint consensus engine to reach instant finality. The oracles in the band protocol are selected pseudo-randomly to produce observations that they gathered from off-chain data sources. Much like in a proof-of-stake-based chain, they have tokens staked on the chain and their chances of being elected to produce an observation are proportional to their share of the total stake. When an oracle produces an observation, this observation is published to the bandchain, and a proof is generated. This proof can be later used by smart contracts on other blockchains to verify the existence of the data as well as to decode and retrieve the result stored. This process obviously requires a bridge contract on the targeted blockchain in order to interpret the proof. While Band protocol's approach is interesting, we think that it lakes interoperability concerns. Indeed a bridge contract has to be implemented for each new integration. Our proposal is integrated by design as it leverages the target blockchain's keys.

Another non-academic work is DOS network [6]. The DOS network leverages verifiable random functions, distributed key generation, and threshold signature scheme to implement their Decentralized Oracle Service. The system is made of a network of oracle nodes connected to off-chain data sources. The time is divided into rounds and for each round, a group of nodes is randomly selected to provide the observations. Each member of the group is given a share of a distributed private key. Members of the group exchange messages containing their signed observations until one of them has received enough signatures to generate the group signature. This node will be responsible for publishing the report containing the group's observations and group signature. The smart contract will then verify the signature and execute the payout. The idea proposed by this scheme is interesting however it has a major drawback. The probability that two or more members of the group are able to construct the group signature at the same time is high. This will result in several nodes publishing the same reports simultaneously. Although this can be resolved on-chain by a proxy that only accepts the first response, the cost of the transaction is permanently lost

for the following reporters. Our approach based on proof-of-work can be used in asynchronous settings it allows us to better sample the probability of finding a valid report.

In this paper, we follow the line of research opened by [2]. Differently from their approach, we consider an asynchronous communication model. In [2], Chainlink presents the "Off-chain Reporting Protocol", an oracle system designed, among other goals, to minimize transaction fees. Transaction fees for some blockchains have become quite prohibitive, motivating the need for such systems. The system consists of n Oracles that exchange messages through off-chain communication channels. The Oracles and their respective public keys are listed in a smart contract C. Time is divided into epochs, and a leader taken from the list of oracles is associated with each epoch. Epochs are mapped to leaders through a deterministic function. A simple modulus that can be calculated by anyone in the network. The Oracles make observations (such as price observations), sign them with their private keys, and submit them to the leader. When he received a sufficient amount of observations, the leader builds a report that lists them all, as well as the signatures, and submits it to a transmission protocol. Finally the transmission protocol hands out the report to the smart contract C. A new epoch associated with a new leader starts whenever the oracles think that the current leader does not perform correctly. While this protocol shows good resilience and low transaction fees, they assume a partially synchronous model. Formally, they assume that clocks in the system are not synchronized until a point in time called global stabilization time (GST). Afterward, all correct nodes behave synchronously. Outside those synchronous periods of time, the liveness of the protocol is not ensured. We think that by using proof-of-work for leader election, we could ensure similar properties in a fully asynchronous timing model.

Our contribution In this paper, we formalize the distributed price oracle problem in the context of decentralized finance applications. Furthermore, we propose a protocol and prove that it verifies the specification of the problem in asynchronous communication environments prone to Byzantine failures. The protocol combines a gossip module with a light proof-of-work module and incentives oracles via a simple reputation mechanism to have a correct behavior.

2 Model

We consider a similar model as the one used by the chainlink off-chain reporting protocol [2]. The main difference is the communication model which is partially synchronous while in our system it is asynchronous.

Oracle network. The system consists in a set of n Oracles $P = \{p_1, ..., p_n\}$ that are referred to as nodes. Each oracle node p_i makes time-varying observations over the price of an asset pair. The set of oracles is determined by an oracle smart contract C that records the public keys of the nodes. The owner of the

smart contract has administrative powers which allow him to update the list of oracles. As we are working with time-varying quantities, there is no proper way to evaluate if an observation is correct or not. Thus the protocol only guarantees that the report contains a sufficient number of observations signed by honest oracle nodes.

Nodes exchange messages through peer-to-peer, bi-directional communication channels. Nodes are identified by unique identifiers and authenticated by digital signatures. All communications are authenticated and encrypted, which means any node can authenticate any other node based on the oracle list recorded in the smart contract C. We consider an asynchronous communication model, which means that there is no global clock shared among the nodes. Moreover, we make no assumption over the reliability of the network, meaning that messages can be delayed or lost. Furthermore, messages can be delivered in a different order than the order they were sent. In the following, we assume that the network does not partition.

Failures. We consider that any $f < n/3$ nodes may exhibit Byzantine faults, which means that they may behave arbitrarily and as if controlled by an imaginary adversary. All non-faulty nodes are called honest or correct. We consider that these faults can occur adaptively meaning an adversary can choose and control the faulty nodes on the fly.

Cryptographic primitives. The protocol uses public-key digital signatures. We assume an idealized public-key infrastructure: each process is associated with its public/private key pair that is used to sign messages and verify signatures of other nodes. A message m sent by a process p_i that is properly signed with the private key of p_i is said to be properly authenticated. We denote by m_{σ_i} a message m signed with the private key of a process p_i. In practice, we would use the standard EdDSA and ECDSA schemes for digital signatures. It is assumed that signatures cannot be forged.

Oracle smart contract and report. The goal of the system is to produce reports containing a sufficient number of signed observations from the oracle nodes when a client requests them. Differently from Breidenbach et. al in [2], the reports are submitted to the oracle smart contract C by some node during a proof-of-work inspired cryptographic race. The smart contract C corresponds to a single asset pair (e.g. BTC/USD). When submitted a report, the oracle smart contract C verifies the signatures of each observation as well as the proof-of-work. If they are valid, the oracle smart contract updates its variable $lastPrice$ to the median of observation values contained in the report. Using the median value among more than $2f$ observations guarantees that the reported value is plausible in the sense that malicious nodes cannot move the value outside the range of observations produced by honest nodes. The value $lastPrice$ can then be consumed by the requesting client or by any other user. The requesting client pays a fee for each new report he requests, which will be distributed equally among the observant nodes.

3 Decentralized price Oracle problem

In this section, we will define the decentralized oracle problem and review major threats and constraints that must be taken into account when designing a decentralized oracle system.

The blockchain Oracle problem is well known in the ecosystem and the grey literature. It is also described in [4] by Cardelli et al. The oracle problem describes the ambivalence between blockchain systems that are supposed to be immutable through decentralization and oracles that, by definition, input outside world data that cannot be verified by the blockchain itself. A blockchain is a self-contained environment with its own validation rules and consensus mechanism. This isolation is what makes blockchain transactions safe and immutable. However, when data is inputted from off-chain data sources, the said data cannot be verified by the blockchain itself. A piece of software, in this case an oracle, is required to guarantee the veracity of the inputted data. As the safety of a system is limited by its weakest element, in a system where the execution of a smart contract depends on the data provided by an oracle, the oracle may be a single point of failure. This oracle problem has already led to several exploits. In 2019, an oracle reported a price a thousand times higher than the actual price [5], which led to a one billion U.S. dollars loss. Funds have then been recovered but it shows how crucial is it to have reliable oracles. To be best of our knowledge there is no formal definition of the blockchain price oracle problem.

The price oracle smart contract can be seen as a particular single-writer multi-reader shared register. The variable of this particular shared register, $lastPrice$, can be read by any client of the system. This variable can be modified only by the smart contract C, and the modification is triggered each time clients invoke $requestNewPrice()$. We propose below a definition of the blockchain price oracle problem in terms of liveness, integrity and uniformity.

Definition 1 (Decentralized blockchain price oracle) A decentralized blockchain price oracle should satisfy the following properties :

- Δ**-Liveness**: There exist a $\Delta > 0$ such that if a client invokes a price request to the smart contract C at time $t > 0$ then a corresponding report r will be retrieved from C within $t + \Delta$ time.
- **Observation integrity**: If a report with v is declared final by C, then v is the observation of a correct oracle or in the range of the observations of the two correct oracles in the system.
- **Uniformity**: If two clients, c_1 and c_2 read the oracle $lastPrice$ at time $t > 0$ then the same price report will be retrieved by both of them.

The price oracle smart contract can be seen as a shared register. The variable of this particular shared register, $lastPrice$, can be read by any client of the system. This variable can be modified only by the smart contract and the modification is triggered each time clients invoke price requests.

Designing distributed price oracles is not an easy task. In the following, we discuss several difficulties.

Freeloading attacks The freeloading attack, also known as mirroring is formally described in [7]. It refers to the technique employed by malicious oracles, that instead of obtaining data from their data source, replicate the data fetched by another oracle. Since oracle systems are often comprised of a reputation system, a "lazy" oracle could simply copy the values provided by a trusted oracle instead of making the effort of fetching the data itself. By doing so, the lazy oracle maximizes its chances of receiving its payout and avoids the cost of requesting data from sources that might charge per-query fees. As explained in [9], this freeloading attack weakens the security of the system by reducing the diversity of data sources and also disincentivizes oracles from responding quickly: Responding slowly and freeloading is a cheaper strategy.

Majority attacks Much like in blockchain systems, oracle systems may be threatened by majority attacks. If an entity controls a majority of oracles in a network, it may be able to manipulate the data to provide prices that diverge from the real observable data.

Price slippage Price slippage refers to the fact that the price returned by an oracle may differ from the actual price. This may be intentional or unintentional but the result is the same. Price slippage may be the consequence of delays generated by the transaction validation time. Indeed real-world prices evolve continuously while the events on a blockchain are discrete. The state of the chain only changes when a new block is added.

Data source manipulation The source which the data is gathered from might also be a vector of attack. Indeed, if the data source or the communication channel between the oracle and the data source can be altered, the resulting observation will ultimately be altered too.

4 PoWacle Protocol Overview

In this section, we propose first a high-level overview of the PoWacle protocol then propose the protocol pseudo-code.

The goal of the protocol is to publish reports containing price observations from the oracles when a client makes a request. The oracle nodes are connected to external data sources such as a market's APIs where they find their price data. When a client requests a price, the oracles will exchange messages containing hash-signed observations of the asset's price. It is very important to note that the content of the observations exchanged is not readable by the other oracles, as they are hashed observations. This is to avoid the freeloading problem. Each oracle node listens for incoming hash-signed observation messages and much like in a proof-of-work-based blockchain builds a pool of messages in their local memory. The difference with a proof-of-work-based blockchain is that instead of transactions, the pool contains hash-signed price observations, and instead of producing a block of transactions, the goal is to produce a report containing

the said observations. To select the oracle that will be responsible for proposing and publishing the report, a proof-of-work protocol is applied. Once they have received hash-signed observations from a sufficient number of nodes, i.e. more than $2f + 1$, the oracles start the report mining process. They try to build a report proposal whose hash value is inferior to some target difficulty number. When a node finds a report proposal whose hash value is inferior to the target difficulty number, he broadcasts the report proposal to the network. On receiving the report proposal, the other nodes return the readable pre-image of their hash-signed observations. For each received observation, the proposer verifies that the observation matches its hash. Once he has collected at least $2f+1$ clear observations, he submits the report proposal to the oracle smart contract along with the proof-of-work. The oracle smart contract verifies the proof-of-work, the match between observations and their hashes, and the signatures. Once all verifications are done, the smart contract calculates the median of the received observations and updates the price of the asset. The price can then be consumed by the client. The smart contract calculates the payout and updates the reputation of the oracle nodes.

The incentive mechanism is designed such that each oracle that produced an observation will be paid equally. Thereby, the report proposer, the one that won the proof-of-work and published the report to the chain, has not more incentives than the other oracles. This should help to limit the computing power that the oracles will put in the network, and thus the operating cost of the system. In the meantime, the oracles are still encouraged to find valid reports regularly, as this is the way they get paid. The goal is to have an incentive equilibrium between producing reports regularly and having a low operating cost in terms of computing power.

Let us unfold the main steps of the protocol:

1. The client requests a new price to the smart contract C and pays a fee.
2. The oracles pick up the request and gather price data from their data sources. They create their observations, hash and sign them. They broadcast it to the oracle network through gossiping.
3. On receiving the hashed signed observations, the nodes verify the signatures. If they are correct, they add the observation to their local memory. Once they have received a sufficient number of hashed signed observations, i.e., above the $2f+1$, the oracles sequentially rearrange their report and a nonce until they find a report proposal whose hash value is below a target difficulty number.
4. The first oracle to find a valid report proposal broadcast it to the oracle network via gossiping.
5. On receiving a valid report proposal, the oracles whose observations were contained in the report return the readable pre-images of their hashed signed observations to the candidate leader.
6. Once he has collected enough pre-images, the proposer verifies that they match their hashes. If they do match, he submits the report to the smart contract C.

7. On receiving a report proposal, the smart contract C verifies the signatures of each individual observation as well as the matching between clear observation and their hash. If everything matches, the smart contract verifies the proof of work. It also verifies that there are at least $2f + 1$ clear observations in the report. Once all checks are passed, the smart contract updates the price, making it available to the client. The report becomes final.
8. The smart contract C calculates the payouts and updates the reputations.

5 Protocol detailed description

We provide a detailed description of the protocols using an event-based notation (e.g. [3] Chap. 1).

The protocol consists of an oracle smart contract hosted on a blockchain presented in Algorithm 1 and an oracle network. The nodes in the oracle network react to events triggered by the oracle smart contract C. The oracle nodes can at any time read the state and the variables of C. Protocol instances that run on the same oracle node (instances of Algorithm 2,3 and 4) also communicates trough events. In the following, Algorithm 1 is the oracle smart contract C, Algorithm 2, executed by every node is a daemon that reacts to events triggered by C. Algorithm 3 is the observation gossip protocol and Algorithm 4 is the report mining protocol.

5.1 The Smart Contract (C)

The protocol Algorithm 1 is orchestrated by an oracle smart contract C hosted on a blockchain. Each oracle smart contract represents a price feed for a single pair of assets, for example, USD/BTC for bitcoin versus U.S. dollars. The oracle smart contract C consists of four primitives; identity, proof-of-work, incentive, and reputation. We will review individually each primitive in this section.

Identity. The oracle smart contract is responsible for storing the identity of the oracles nodes. It maintains a list of oracle nodes and their public keys. The set of oracles is managed by an owner with administrative power. How the administrator curates the list of authorized oracles is behind the scope of this document. The oracle list is used to verify signatures.

Proof-of-work. The proof-of-work primitive is responsible for verifying the proof-of-work that is submitted along with each report. It verifies the match between hash-signed and clear observation, verifies that there are more than $2f$ clear observations, requests the identity primitive to verify the signatures, and finally verifies the proof-of-work. To do so, it checks that the submitted report header hash value is below the target difficulty number. If so, the report is final and the contract updates the last price value, making it available to the client.

The proof-of-work primitive is also responsible for adjusting the difficulty target number. To do so it uses a system similar to Bitcoin's difficulty adjustment

[8]. For each new request, the smart contract C records the time difference between the moment the request was made by the client, and the moment the corresponding report was submitted. Then for every new report, the contract calculates the average report generation time over the last hundred reports. It then adjusts the target difficulty number to have a stable average report generation time. To do so, it multiplies the current difficulty target number by a ratio between observed and desired report generation time. Target report generation time is specified by the owner with administrative power. What value should be chosen is out of the scope of this document and would require further analysis.

Incentive. The incentive primitive calculates and broadcasts the payouts to the oracles. For each final report, the contract C pays an equal share of the total fee to each oracle that produced an observation that was included in the report.

As introduced earlier, the incentive system must be designed to create an incentive equilibrium between producing reports regularly and having a low operating cost in terms of computing power. Thus, the payout received by the report publisher will be equal to the payouts received by the other oracles that produced an observation contained in the report. The only difference is that the report producer gets his transaction fees refunded.

In order to reduce transaction fees, the payout is not automatically transferred at each report. Instead, the smart contract C records the total payout each oracle is eligible to, and they can cash out on request.

Reputation. The reputation system is primarily used by the oracles to prioritize the observation they will include in their report proposal. Indeed, an adversarial oracle could choose not to send back his clear observation at stage 7, thus slowing down protocol execution. To minimize this risk, each oracle is associated with a reputation number corresponding to the number of his observations that have been included in past valid reports. When an oracle tries to find a valid report proposal, it is in his interest to prioritize observations from nodes that have a high reputation number. When a submitted report becomes final, the smart contract increases by one the reputation number of each oracle that produced an observation contained in the said report.

5.2 The oracle network

In this section, we detail the algorithms executed by nodes in the oracle network. It consists of three algorithms. Algorithm 2 is a daemon that listens for event triggered by C. Algorithm 3 is instantiated by every p_i for each new request by the daemon. Its role is to propagate and collect observations among the network. Algorithm 4 is the report mining protocol. Its role is to build a report proposal out of the delivered observation and eventually submit this report to the oracle smart contract C. The oracle smart contract C maintains a list of oracles, their public keys, and their reputation. The client makes his request to the smart contract C. The instances of Algorithm 3 and Algorithm 4 can read the state of

Algorithm 1 Oracle smart contract

state
  $lastPrice \leftarrow \perp$: last valid reported price
  $reports \leftarrow [\perp]$: table of valid reports
  $oracles \leftarrow [\perp]^n$: list of oracles' public keys and their reputation
  $targetDifficulty \leftarrow 0$: current difficulty target number
  $tergetReportTime \leftarrow 0$: Target report generation time

function $requestNewPrice()$
  $reports[requestID].requestSubmitted \leftarrow time.now$
  Emit event $newRequest()$

function $verifyProofOfWork(reportHeader)$
  return $hash(reportHeader) \leq targetDifficulty$

function $verifySignatures(hashSignObs)$
  return $\forall\, h_{\sigma_i} \in hashSignObs | verify_i(h_{\sigma_i})|$

function $verifyHashes(hashSignObs, observe)$
  return $\forall o_i \in observe | hash(hashSignObs[i] = hash(o_i))|$

function $submitReport(requestID, [reportHeader, hashSignObs, observe])$
  if $verifyProofOfWork(reportHeader) \wedge verifySignatures(hashSignObs) \wedge verifyHashes(hashSignObs, observe) \wedge observe.length \geq 2f + 1$ **then**
    $reports[requestID] \leftarrow [reportHeader, observe, hashSignObs]$
    $reports[requestID].requestFulfilled \leftarrow time.now$
    $lastPrice \leftarrow median(observe)$
    $\forall o_i \in observe | oracles[i].reputation \leftarrow oracles[i].reputation + 1|$
    $adjustDifficulty()$
    Emit event $finalReport(requestID)$
  end if

function $adjustDifficulty()$
  $l = reports.length$
  $sum \leftarrow \sum_{i=l-100}^{l} reports[i].requestFulfilled - reports[i].requestSubmitted$
  $averageReportTime \leftarrow sum/100$ ▷ Calculate the average report generation time over the last 100 reports
  $targetDifficulty \leftarrow targetDifficulty * \frac{averageReportTime}{targetReportTime}$ ▷ Adjust the difficulty by a factor of the ratio between observed and desired report generation time

function $registerOracle(publicKey)$ ▷ Function restricted to administrator
  $oracles.append([publicKey, 0])$ ▷ Register new oracle and set reputation to 0

function $setTargetReportTime(time)$ ▷ Function restricted to administrator
  $targetReportTime \leftarrow time$

C at any time. They can access the list of oracles, their public keys, reputation, current request-id, and current target difficulty number. Those public variables are used implicitly in the pseudo-code presented here. We denote by $sign_i(m)$ the function that signs the message m with the private key of process p_i producing the signed message m_{σ_i}. Similarly $verify_i(m_{\sigma_i})$ verifies the signature of signed message m_{σ_i} with the public key of process p_i.

The daemon. Algorithm 2 is a deamon that is executed continuously by every process. This deamon awaits for *newRequest* events from C to instantiate the observation gossip protocol presented in Algorithm 3. It is also responsible for stopping instances of Algorithm 3 and Algorithm 4 when a $finalReport$ event is emitted by C.

Algorithm 2 Oracle daemon executed continuously by every $p_j \in P$

Upon event $newRequest(requestID)$ from C **do**
  initialize instance ($requestID$) of observation gossip protocol

Upon event $finalReport(requestID)$ from C **do**
  abort instance gossip protocol ($requestID$)
  abort instance report mining ($requestID$)

The observation gossip protocol. Algorithm 3 is the observation gossip protocol. Its goal is for the nodes to propagate observations messages among the network. It is instantiated by Algorithm 2 upon new request event emitted by C. Oracle nodes gather data from sources, hash and sign their observation, and broadcast it to every $p_i \in P$. Every oracle p_i maintains a list $hashSignObs$ of hashed-signed observations delivered by any $p_j \in P$. When a node p_i has received at least $2f + 1$ hashed-signed observations, he starts the report mining protocol presented in Algorithm 4. For the sake of simplicity and readability, we separated Algorithm 3 and Algorithm 4. In practice, those algorithms will be executed in parallel on a single machine and thus share the same local memory. This means that when Algorithm 3 starts a report mining instance for request $requestID$, the list of hashed observations $hashSignObs$ can still be updated by the observation gossip protocol. The report gossip protocol continues execution even if the report mining process has started. It is in the interest of the nodes to include as many observations as possible in their report proposal to minimize the chances of an adversarial oracle blocking execution. This will be further developed in the paragraph about the report mining process. Yet it must be noted that the abort condition of the observation gossip process is a $finalReport$ event from C.

The report mining protocol Algorithm 4 presents the report mining process. Much like in a proof-of-work-based blockchain the principle is for the oracles to

Algorithm 3 Observation gossip protocol instance $requestID$ (executed by every oracle p_i)

state
$observe \leftarrow [\perp]^n$: table of observations received in OBSERVATION messages
$hashSignObs \leftarrow [\perp]^n$: table of hashed signed observations received in HASHED-OBSERVATION messages

Upon initialization **do**
$v \leftarrow$ gather price value from data source
$observe[i] \leftarrow [time.now, assetPair, v]$
$hashSignObs[i] \leftarrow sign_i(hash(observe[i]))$
send message[HASHED-OBSERVATION, $hashSignObs[i]$] to all $p_j \in P$

Upon receiving message [HASHED-OBSERVATION, h_{σ_j}] from p_j **do**
if $verify_j(h_{\sigma_j})$ **then**
$hashSignObs[j] \leftarrow h_{\sigma_j}$
end if

Upon $|\{p_j \in P | hashSignObs[j] \neq \perp\}| = 2f + 1$ **do**
initialize instance report mining ($requestID$)

Upon receiving message [REPORT-PROPOSAL,$hashSignObs'$,$reportHeader$] from p_l **do**
if $hashSignObs[i] \in hashSignObs'$ **then**
if $\{\forall\ h_{\sigma_j} \in hashSignObs' | verify_j(h_{\sigma_j})|\} \wedge hash(reportHeader) \leq targetDifficulty \wedge hashSignObs'.length \geq 2f + 1$ **then**
send message[OBSERVATION, $observe[i]$] to p_l
end if
end if

rearrange the content of the report until the hash value of the report header is below a target difficulty number. Each report maintains a pool $hashSignObs$ of n pending observations that have not been yet included in a report. An observation has three attributes, the target asset pair, the observed price, and a timestamp. To this observation correspond an observation header that contains the public key of the oracle, the hash of the observation, and the oracle's signature. When oracle nodes try to find a valid report, they don't hash the full report. Instead, much like in a proof-of-work-based blockchain, they only hash the header of the report. This header consists of a timestamp a nonce and most importantly the observations hash. The observations hash would correspond in a proof-of-work blockchain to the Merkle root. But because we don't need simple payment verification, building a Merkle tree out of the list of observations is unnecessary. Hashing the concatenated list of observations is sufficient in our case.

The way nodes rearrange their hash-signed observation list to find a valid report proposal is not explicitly developed in Algorithm 4. Indeed, similarly to proof-of-work-based mining, it is the responsibility of the nodes to find a strategy

that maximizes their rewards and reduces their costs. However, we would advise that they rearrange their list according to the reputation of the other nodes. Indeed, the only way for an adversarial oracle to slow down or block protocol execution is not to respond with an OBSERVATION message on receiving a REPORT-PROPOSAL message. In order not to include dishonest parties in their report proposal, oracles should prioritize observations from oracles that have a good reputation. It is to be noted that to be considered valid by the smart contract C, a report does not need to have a clear observation for every hash-signed observation contained in the *hashSignObs* list. Indeed, the only requirement is that the report contains at least $2f + 1$ clear observations.

Algorithm 4 Report mining algorithm instance $requestID$ executed by every p_i that has delivered more than $2f + 1$ hash-signed observation

state
  $reportHeader \leftarrow \perp$

Upon initialization **do**
  loop
    $nonce \leftarrow 0$
    $hashSignObs' \leftarrow prioritize(hashSignObs)$ ▷ Create a prioritized copy of the hash-signed observation list
    while $reportHeader = \perp \vee nonce \neq 2^{32}$ **do**
      if $hash(time.now, nonce, hash(hashSignObs')) \leq targetDifficulty$ **then**
        $reportHeader \leftarrow [time.now, nonce, hash(hashSignObs')]$
        send message[REPORT-PROPOSAL, $hashSignObs'$, $reportHeader$] to all $p_j \in P$
        break
      end if
      $nonce \leftarrow nonce + 1$
    end while
  end loop

Upon receiving message [OBSERVATION, o_j] from p_j **do**
  if $hash(o_j) = hashSignObs[j]$ **then**
    $observe[j] \leftarrow o_j$
  end if

Upon $observe.length \geq 2f + 1$
  $C.submitReport(requestID, [reportHeader, hashSignObs', observe]$

6 Analysis

In this section, we prove that the PoWacle protocol satisfies the properties of the oracle problem definition as presented in Section 3.

Lemma 1. *The PoWacle protocol satisfies the liveness property of a decentralized blockchain price oracle.*

Proof. Consider a client request made to C at time t. Every correct node is triggered by this event and broadcasts its observation to every $p_j \in P$. Recall that correct oracles resend every message until the destination acknowledges it. Even if an adversary coalition can choose faulty nodes on the fly, they can not control more than $f < n/3$ process. Thus every correct oracle should eventually deliver a hash-signed observation from every correct node. This also holds for the process where nodes send clear observations to the proposer. Consequently, every correct node should be able to start the mining process. Let $d > 0$ be the difficulty target number. Due to the uniformity and non-locality properties of hash functions (outputs should be uniformly distributed), for each trial, there is a non-zero chance of finding a report whose hash value h is less or equal than the difficulty d. Consequently, there is a finite Δ within which an oracle should find and submit a valid report to C. Because the observations contained in the report are from correct nodes, C should declare the report final. Thus the protocol satisfies the Δ-Liveness property of a decentralized blockchain price oracle.

Lemma 2. *The PoWacle protocol satisfies the observation integrity property of a decentralized blockchain price oracle.*

Proof. Consider a protocol execution where a report with value v has been declared final by C. To be considered final by C, a report must contain at least $2f+1$ observations that have been individually signed by each oracle. The value v is the median of those observations. Since there are at most f faulty nodes, out of the $2f+1$ observations, more than half have been produced by a correct node. Trivially, the median of those observations is either the one of a correct node or the one of a faulty node but in the interval between a larger and a smaller value provided by honest nodes. Thus, the protocol satisfies the observation integrity property of a decentralized blockchain price oracle.

Lemma 3. *The PoWacle protocol satisfies the uniformity property of a decentralized blockchain price oracle.*

Proof. Recall that the value $lastPrice$ of C can only be updated by the smart contract itself when $requestNewPrice$ is invoked by a client. Thus trivially, if two clients c_1 and c_2 read this value at some time $t > 0$, the value they will read corresponds to the same report.

7 Conclusions

Price oracles are at the core of various applications in Decentralized Finance. It should be noted that due to security attacks these oracles are difficult to design in a distributed fashion. In this paper, we propose and prove correct the first distributed price oracle designed for asynchronous Byzantine prone environments.

Price oracles have some similarities with classical distributed shared registers however, the presence of smart contracts (pieces of code) that automatically execute on an underlying blockchain make their particularity. In future works, we would like to investigate how price oracles can benefit from the existing distributed system literature. In the same vein, we will like to investigate new distributed abstractions that encapsulate the blockchain technology specificity.

References

1. Bandchain: Band protocol system overview, https://docs.bandchain.org/whitepaper/system-overview.html
2. Breidenbach, L., Cachin, C., Coventry, A., Juels, A., Miller, A.: Chainlink off-chain reporting protocol. URl: https://blog. chain. link/off-chain-reporting-live-on-mainnet (2021)
3. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to reliable and secure distributed programming. Springer Science & Business Media (2011)
4. Caldarelli, G., Ellul, J.: The blockchain oracle problem in decentralized finance—a multivocal approach. Applied Sciences **11**(16), 7572 (2021)
5. Connell, J.: Sophisticated trading bot exploits synthetix oracle, funds recovered (7 2019), https://cointelegraph.com/news/sophisticated-trading-bot-exploits-synthetix-oracle-funds-recovered
6. DOS: A decentralized oracle service boosting blockchain usability with off-chain data & verifiable computing power (2011), https://s3.amazonaws.com/whitepaper.dos/DOS+Network+Technical+Whitepaper.pdf
7. Murimi, R.M., Wang, G.G.: On elastic incentives for blockchain oracles. Journal of Database Management (JDM) **32**(1), 1–26 (2021)
8. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review p. 21260 (2008)
9. Paradigm: Chainlink: Detailed review on the project (3 2019), https://medium.com/paradigm-fund/chainlink-detailed-review-on-the-project-9dbd5e050974