Orb: Decentralized and Sustainable Oracles

J. Zahnentferner
The Stable Order
zahnentferner@gmail.com

Sarthak Dengre
The Stable Order
dengresarthak2002@gmail.com

Luca D'Angelo
zenGate Global
The Stable Order
ldgaetano@protonmail.com

Louis Quilling
Carnegie Mellon University in Qatar
equillin@andrew.cmu.edu

Abstract

In the context of blockchains, oracles are components that provide data from the external world to the smart contracts. Therefore, they are essential to applications that require such data. For these applications to be truly decentralized, it is crucial that the oracles on which they depend be decentralized as well. This paper presents the Orb Oracle Protocol. Orb Oracles are decentralized in the sense that any oracle token holder can submit values and thus can act as an oracle operator, and are sustainable in the sense that oracle operators are rewarded for submitting values and thus have a source of revenue to cover their operational costs. We prove theorems about the efficiency of updates, delays and sustainability of the rewards, and we formally verify the theorems using the Rocq proof assistant.

Keywords: Blockchain, cryptocurrency, staking, reward distribution

We thank the referees and the community of the Stability Nexus for their feedback and support. We also thank Giselle Reis of Carnegie Mellon University in Qatar, who contributed to discussions of Orb, especially its formalization. This work was supported in part by the Qatar Foundation through Carnegie Mellon University in Qatar's Seed Research Funding program.

1 Introduction

The great promise of *smart contracts* [20, 4] is to allow people to implement programs that automatically perform predefined actions when agreed conditions are satisfied. When smart contracts are deployed on decentralized blockchains, their execution is guaranteed to occur without reliance on a central authority above, or a single intermediary between, the people interested in the execution of the smart contract.

However, some of the most interesting applications of smart contracts require data that is external to the blockchain. For example, crypto-backed stablecoin [24] protocols (e.g. [16, 26, 27]) need to know the price of the underlying reserve asset in relation to the peg asset, in order to know how much of the reserve asset to give to a user who redeems stablecoins; crypto-collateralized stablecoin protocols (e.g. [15]) and lending protocols (e.g. [13]) need to know the value of the user's collateral, in order to know how much the user may borrow and when liquidations should be triggered; rebasing stablecoin protocols (e.g. [12]) need to know the market price of the issued stablecoins, in order to know when a rebasing is needed; prediction market protocols (e.g. [25]) need to know the outcome of an event, in order to be able to transfer funds to the users who correctly predicted the outcome; blockchain-based business process management applications need oracles to feed data about the off-chain processes to the on-chain smart contracts [2].

The components that allow data to be fetched from various sources and submitted to blockchains, to be used by smart contracts, are commonly called *Oracles*. Clearly, for an application to be fully decentralized, the oracles on which it depends need to be decentralized as well.

Despite their criticality, most oracles available nowadays are surprisingly centralized. Furthermore, the centralized entities operating these oracles often rely on venture capital, or grants from the blockchains on which they deploy their oracles, to fund their operations. This raises the concern of long-term sustainability of these oracles, since their future is unclear if their operators run out of capital[1].

Immutable, unstoppable, decentralized applications need a solid platform where every component on which they depend is sustainable and decentralized as well. This paper describes Orb, a decentralized and sustainable oracle protocol, aimed at providing a reliable, resilient, robust and anti-fragile foundation for truly decentralized applications that require real-world data.

The following sections start with an informal and concise description of the protocol in Section 2. Section 3 then discusses the motivations and rationales for the protocol design decisions. Section 4.3 presents a precise mathematical definition of the protocol. These three sections are intended to be read together. Section 5 shows and proves 4 theorems that are

[1]This is not only a hypothetical concern. For example, the oracle of the Milkomeda EVM sidechain of Cardano simply halted, when dcSpark decided to stop operating it due to financial considerations.

enjoyed by the protocol. Section 6 describes and discusses the formalization and verification of the protocol and its theorems in the Rocq theorem prover. Section 7 compares this protocol with related works. Finally, Section 8 concludes the paper and discusses directions for future work.

2 Protocol Description

Orb is a protocol for the creation, operation and use of oracles where the data are temporal streams of values. Examples of temporal streams of values include: price of an asset over time; economic indicators (e.g. inflation indexes, base interest rates, unemployment, ...) over time; weather variables (e.g. temperature, rainfall, ...) in a location over time....

Every Orb oracle is associated with a fungible *oracle token*. Any oracle token holder can deposit oracle tokens into the oracle and become an *oracle operator*. The amount of tokens an operator has deposited is the operator's *stake*. The deposit and withdrawal of tokens, to increase or decrease one's stake, are subject to locking periods. During a deposit, the tokens are considered staked only after a *deposit-locking period* has elapsed. Likewise, an oracle operator can only withdraw tokens, after a *withdrawal-locking period* has elapsed since his/her last operation.

Every oracle operator may submit values to the oracle. The oracle calculates a weighted average aggregate of the latest values submitted by all oracle operators. This weighted average takes into account the elapsed time of each submitted value, subjecting it to an exponential decay, and the stake of the oracle operator who submitted the value.

Anyone can, at any time, fund an oracle with a *reward token* (typically, the underlying blockchain's native cryptocurrency). The balance of the oracle is a *reward pool* from which oracle operators are rewarded for every value submission. The reward for each value submission is calculated in a way that operators with higher stake and who submit values regularly receive higher rewards, since their submitted values contribute more to the average.

Oracle consumers are users or smart contracts who read data from the oracle. By default, anyone may read data from the oracle. The protocol maintains a single blacklist, initially empty. Oracle operators may vote to blacklist (i.e. add to blacklist) or whitelist (i.e. remove from blacklist) oracle consumers, with voting power linearly proportional to their stake. If the weight of blacklist votes exceeds the weight of whitelist votes for an oracle consumer and certain quorum conditions are satisfied, the oracle consumer is disallowed to read data from the oracle.

3 Motivations and Rationales for Protocol Design Choices

Orb Oracles are decentralized in the sense that any oracle token holder can submit values and thus can act as an oracle operator. Thanks to its efficient constant-time weighted average value update rule, an arbitrarily large number of oracle operators can participate, with no performance impact. There are, therefore, no limits to the decentralization.

Furthermore, the exponential decay in the computation of the average ensures that, if any oracle operator becomes inactive, the contribution of its latest value submission to the average eventually and quickly becomes negligible. Considering only the latest value submitted by each oracle operator prevents a single operator from dominating the average and bounds the oracle delay.

Orb oracles as defined here deliberately avoid outlier rejection, since the rejection of a new outlier value that was submitted in reaction to a sharp change in the source value would mean that the oracle would delay to take that sharp change into account. However, variations of the Orb oracle protocol with outlier rejection could be defined and implemented. There is a trade-off between speed and robustness against possibly, but not necessarily, "wrong" outlier values.

Locking periods for depositing tokens to become an oracle operator and for withdrawing tokens to cease to be an oracle operator deter governance attacks whereby a malicious user would briefly buy or borrow tokens just to act as an oracle operator for a short time.

Orb oracles are sustainable in the sense that oracle operators are rewarded for submitting values and thus have a source of revenue to cover their operational costs. The reward formula discourages too frequent submission and prevents draining of the reward pool by a single oracle operator. It also ensures that the reward pool, if it is ever funded at least once, never becomes empty and thus there are always incentives for oracle operators to continue submitting values.

Orb oracles are also resistant to free-riding. Oracle operators are free to set their own terms and conditions for using the oracle and, if an oracle consumer is not satisfying the terms and conditions, a weighted majority of oracle operators can blacklist this consumer. In particular, the terms and conditions may involve requirements to fund the oracle (by requiring that oracle consumers execute the *reward token funding* operation). In this way, free-riding consumers, who use the oracle but do not contribute to its funding, may be blacklisted. This governance-based free-riding resistance further contributes to oracle sustainability.

The funding mechanism (allowing anyone to fund the oracle via the *reward token funding* operation at any time) and the governance mechanism (allowing oracle operators to blacklist oracle consumers that are not adhering to the terms and conditions) were designed with flexibility in mind. For example: an oracle that is funded directly by emissions of a blockchain's native currency, serving as public infrastructure for that blockchain, is possible; so is an oracle that relies on donations to get funded; and so is an oracle that is funded by consumers

of the oracle's data, which are expected to call the funding operation at regular intervals, perhaps in proportion to how frequently they read data or in proportion to their total value locked protected by the oracle. By not prescribing a specific business model for the oracle operators, Orb can cover a wide range of business models. The policies that oracle operators may use to jointly decide whom to blacklist or whitelist are outside the protocol itself.

The token-based oracle operation is flexible enough to encompass a wide range of operational arrangements. For example: a centralized first-party oracle could be set up by keeping all the tokens under the control of the first-party; a permissioned consortium of oracle operators could be set up by distributing oracle tokens to the members of the consortium and making these tokens non-transferable; a fully permissionless oracle could be set up by using an oracle token that can be minted to oracle consumers in proportion to their alignment to the oracle's correct operation. The oracle token could even be made minable in a proof-of-work style, further ensuring permissionlessness and unstoppability.

4 Protocol Specification

Every oracle created via the Orb protocol has immutable parameters, chosen by the oracle creator at the moment of deployment, and state variables, which vary as users execute operations on the oracle after deployment.

An oracle's parameters are a tuple of constants

$$C = (h, q, \Delta_{\text{dep}}, \Delta_{\text{wd}}, \alpha, \tau_w, \tau_r) \tag{1}$$

where:

- $h \in \mathbb{R}_{>0}$ is the half-life constant.
- $q \in \mathbb{R}$ is the quorum constant.
- $\Delta_{\text{dep}} \in \mathbb{R}_{\geq 0}$ is the deposit-locking period.
- $\Delta_{\text{wd}} \in \mathbb{R}_{\geq 0}$ is the withdrawal-locking period.
- $\alpha \in \mathbb{R}$ is a reward factor determining which fraction of the oracle's balance is paid as reward.
- τ_w is an identifier of the oracle token.
- τ_r is an identifier of the reward token.

An oracle's state is a tuple of variables

$$V = (L_\ell, L_f, T_{\text{dep}}, T_{\text{op}}, P_{\text{user}}, W_{\text{user}}, T_{\text{user}}, \bar{p}, \tilde{p}, Q, \mathcal{P}, t_{\text{sub}}, t_{\text{last}}, L_{\text{tot}}, \mathcal{G}) \tag{2}$$

where:

- U is the set of all users[2] (people and smart contracts), which may execute operations on the oracle.
- $L_\ell, L_f : U \to \mathbb{R}_{\geq 0}$ track the locked and unlocked token balances (`lockedTokens` and `unlockedTokens`)[3].
- $T_{\text{op}} : U \to \mathbb{R}_{\geq 0}$ store the timestamps of the last operation of any kind of each user.
- $T_{\text{dep}} : U \to \mathbb{R}_{\geq 0}$ store the last deposit timestamp of each user.
- $T_{\text{user}} : U \to \mathbb{R}_{\geq 0}$ stores the last submission time of each user (`tof`).
- $P_{\text{user}} : U \to \mathbb{R}$ stores each submitter's last reported value (`pof`).
- $W_{\text{user}} : U \to \mathbb{R}_{\geq 0}$ stores the last submission weight[4] of each user (`wof`).
- $\bar{p} \in \mathbb{R}$ is the stored aggregate value (`aggregatedValue`).
- $\tilde{p} \in \mathbb{R}$ is the latest submitted value (`latestValue`).
- $Q \in \mathbb{R}_{\geq 0}$ is the stored aggregate weight (`aggregatedWeight`).
- $\mathcal{P} = \big((t_1, \bar{p}_1, \tilde{p}_1), \ldots, (t_m, \bar{p}_m, \tilde{p}_m)\big)$ is the time-ordered value history[5].
- $t_{\text{sub}} : \mathbb{R}_{\geq 0}$ stores the time of last value submission by any oracle operator.
- $t_{\text{last}} : \mathbb{R}_{\geq 0}$ stores the time of the last call to any oracle operation.
- $L_{\text{tot}} : \mathbb{R}_{\geq 0}$ stores the current total balance of deposited oracle tokens.

[2] U itself is not part of the oracle's state. It is just the universe of all users, which serves as the domain for a few mappings that are part of the oracle's state.

[3] The names between parentheses are the names of the state variables in our reference implementation in `https://github.com/StabilityNexus/OrbOracle-Solidity` and are intended to ease the mapping between the mathematical specification and the reference implementation. Note that the Solidity implementation uses Rationals for the real valued numbers as reals as a built-in type do not exist in Solidity.

[4] This is the weight that the user had when it submitted its last value.

[5] We write $\mathsf{append}(\mathcal{P}, x)$ for appending a triple x to the ordered value history.

- $\mathcal{G} = (\mathcal{B}, V_{\text{black}}, V_{\text{white}}, M_{\text{black}}, M_{\text{white}}, \mathcal{W}_{\text{black}}, \mathcal{W}_{\text{white}})$ is the governance state[6], where:
  - $\mathcal{B} : U \to \{0, 1\}$ is the blacklist indicator,
  - $V_{\text{black}}, V_{\text{white}} : U \to \mathbb{R}_{\geq 0}$ are the accumulated blacklist and whitelist weights,
  - $M_{\text{black}}, M_{\text{white}} : U \to 2^U$ are the sets of targets each voter has supported,
  - $\mathcal{W}_{\text{black}}, \mathcal{W}_{\text{white}} : U \times U \to \mathbb{R}_{\geq 0}$ are the per-target stored weights.

The user balances of oracle tokens and reward tokens outside are denoted by $B_{\text{user}} : U \times \{\tau_w, \tau_r\} \to \mathbb{R}_{\geq 0}$, while the balances of the oracle itself are denoted by $B_{\text{oracle}} : \{\tau_w, \tau_r\} \to \mathbb{R}_{\geq 0}$.

4.1 Auxiliary Definitions

Definition 1 (Exponential Decay Factor)**.** *For $\Delta \geq 0$, the exponential decay factor, used to halve weights every h seconds, is defined as:*

$$\delta(\Delta) = 2^{-\Delta/h} \tag{3}$$

Definition 2 (Time-Dependent Functions)**.** *Given the current state V and time t, the following time-dependent functions are defined:*

- *The* decayed weight *of user u at time t is $W_{\text{decayed}}(u, t) = W_{\text{user}}(u) \cdot \delta(t - T_{\text{user}}(u))$.*
- *The* decayed aggregate weight *at time t is $Q_{\text{decayed}}(t) = Q \cdot \delta(t - t_{\text{sub}})$.*
- *The* decayed aggregate value *at time t is $\bar{p}_{\text{decayed}}(t) = \bar{p} \cdot \delta(t - t_{\text{sub}})$.*
- *The* ideal decayed weighted mean *at time t is*

$$P(t) = \frac{\sum_{x \in U} P_{\text{user}}(x) \cdot W_{\text{decayed}}(x, t)}{\sum_{x \in U} W_{\text{decayed}}(x, t)}. \tag{4}$$

[6]The governance state contains all the state variables related to blacklisting of oracle consumers by oracle operators. Through governance operations of voting to blacklist or whitelist oracle consumers, oracle operators may affect the governance state.

4.2 Auxiliary State Update Procedures

The following procedures are used by many operations specified in subsection 4.3. The *unlock* procedure is called by any operation that requires knowing the current unlocked oracle token balance of the user. The *recompute* procedure is called whenever users vote to blacklist or whitelist. The *reweight* procedure updates the weights of every blacklist/whitelist vote cast by a user. It is called automatically when users withdraw or on demand by users when they deposit and explicitly synchronize their weights[7].

$$\mathrm{Unlock}(u,t) \coloneqq \begin{cases} \text{if } t \geq T_{\mathrm{dep}}(u) + \Delta_{\mathrm{dep}} \text{ and } L_\ell(u) > 0 \text{ then} \\ \quad L_f(u) \leftarrow L_f(u) + L_\ell(u) \\ \quad L_\ell(u) \leftarrow 0 \\ \text{else} \\ \quad \text{skip} \end{cases} \tag{5}$$

$$\mathrm{Recompute}(x) \coloneqq \begin{cases} \text{if } V_{\mathrm{black}}(x) - V_{\mathrm{white}}(x) > q \cdot \left(L_{\mathrm{tot}} - (V_{\mathrm{black}}(x) + V_{\mathrm{white}}(x))\right) \\ \text{then} \\ \quad \mathcal{B}(x) \leftarrow 1 \\ \text{else} \\ \quad \mathcal{B}(x) \leftarrow 0 \end{cases} \tag{6}$$

$$\mathrm{Reweight}(u) \coloneqq \begin{cases} \forall x \in M_{\mathrm{black}}(u) : \\ \quad V_{\mathrm{black}}(x) \leftarrow V_{\mathrm{black}}(x) - \mathcal{W}_{\mathrm{black}}(x,u) + L_f(u) \\ \quad \mathcal{W}_{\mathrm{black}}(x,u) \leftarrow L_f(u) \\ \quad \mathrm{Recompute}(x) \\ \forall x \in M_{\mathrm{white}}(u) : \\ \quad V_{\mathrm{white}}(x) \leftarrow V_{\mathrm{white}}(x) - \mathcal{W}_{\mathrm{white}}(x,u) + L_f(u) \\ \quad \mathcal{W}_{\mathrm{white}}(x,u) \leftarrow L_f(u) \\ \quad \mathrm{Recompute}(x) \end{cases} \tag{7}$$

[7] Reweight is an expensive procedure. This is the reason why it is called automatically only on withdrawals, when it is necessary, but only on explicit demand by users after deposits, if they wish.

4.3 Protocol Operations

Each operation defined in the following subsections is an action that a user may perform on the oracle. The subsections define how they update the oracle's state, if their stated preconditions are satisfied. Preconditions are checked before executing the state update.

4.3.1 Token Deposit

For $a > 0$, the deposit operation $\delta_{\text{dep}} : U \times \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\delta_{\text{dep}}(u, a, t)(V) \coloneqq \begin{cases} \text{Unlock}(u, t) \\ L_\ell(u) \leftarrow L_\ell(u) + a \\ T_{\text{dep}}(u) \leftarrow t \\ T_{\text{op}}(u) \leftarrow t \\ L_{\text{tot}} \leftarrow L_{\text{tot}} + a \\ t_{\text{last}} \leftarrow t \\ B_{\text{user}}(u, \tau_w) \leftarrow B_{\text{user}}(u, \tau_w) - a \\ B_{\text{oracle}}(\tau_w) \leftarrow B_{\text{oracle}}(\tau_w) + a. \end{cases} \tag{8}$$

4.3.2 Token Withdrawal

For $a > 0$ with $L_f(u) \geq a$ and $T_{\text{op}}(u) + \Delta_{\text{wd}} \leq t$, the withdrawal operation $\chi : U \times \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\chi(u, a, t)(V) \coloneqq \begin{cases} \text{Unlock}(u, t) \\ L_f(u) \leftarrow L_f(u) - a \\ L_{\text{tot}} \leftarrow L_{\text{tot}} - a \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ B_{\text{user}}(u, \tau_w) \leftarrow B_{\text{user}}(u, \tau_w) + a \\ B_{\text{oracle}}(\tau_w) \leftarrow B_{\text{oracle}}(\tau_w) - a. \end{cases} \tag{9}$$

4.3.3 Value Submission

Only token holders with unlocked stake may submit values (`submitValue`). Let $v \in \mathbb{Z}$ denote the new submission and t the current timestamp. Set $w = L_f(u)$, $p_u = P_{\text{user}}(u)$,

$w_u = W_{\text{user}}(u)$, and $t_u = T_{\text{user}}(u)$. Using the decay factor (3), we compute[8] the updated aggregate weight and value as:

$$Q' = \big(Q - w_u \cdot \delta(t_{\text{sub}} - t_u)\big) \cdot \delta(t - t_{\text{sub}}) + w, \tag{10}$$

$$\bar{p}' = \frac{\big(\bar{p} \cdot Q - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)\big) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q'}. \tag{11}$$

The contract pays out:

$$n = \alpha \cdot B_{\text{oracle}}(\tau_r) \cdot \frac{w}{Q'} \cdot \big(1 - \delta(t - t_u)\big). \tag{12}$$

The value submission operation $\phi : U \times \mathbb{Z} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\phi(u, v, t)(V) \coloneqq \begin{cases} \text{Unlock}(u, t) \\ P_{\text{user}}(u) \leftarrow v \\ W_{\text{user}}(u) \leftarrow L_f(u) \\ T_{\text{user}}(u) \leftarrow t \\ T_{\text{op}}(u) \leftarrow t \\ \bar{p} \leftarrow \bar{p}' \\ \tilde{p} \leftarrow v \\ Q \leftarrow Q' \\ t_{\text{sub}} \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \mathcal{P} \leftarrow \mathsf{append}(\mathcal{P}, (t, \bar{p}', v)) \\ B_{\text{user}}(u, \tau_r) \leftarrow B_{\text{user}}(u, \tau_r) + n \\ B_{\text{oracle}}(\tau_r) \leftarrow B_{\text{oracle}}(\tau_r) - n. \end{cases} \tag{13}$$

4.3.4 Value Reading

Anyone who is not blacklisted may read oracle values (`readValue` and `readLatestValue`). This refreshes the liveness timestamp and the timestamp of the user's last operation on the oracle. The reading operation $\rho : U \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\rho(u, t)(V) \coloneqq \begin{cases} t_{\text{last}} \leftarrow t \\ T_{\text{op}}(u) \leftarrow t \end{cases} \tag{14}$$

[8] The equations for Q' and $\bar{p}'$ are key insights of this paper. As proven in Theorem 1, they enable constant-time update of the oracle's aggregated value whenever a new value is submitted by any oracle operator.

4.3.5 Vote to Blacklist

The blacklist vote operation $\beta : U \times U \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\beta(u, x, t)(V) \coloneqq \begin{cases} \text{Unlock}(u, t) \\ V_{\text{black}}(x) \leftarrow V_{\text{black}}(x) - \mathcal{W}_{\text{black}}(x, u) + L_f(u) \\ \mathcal{W}_{\text{black}}(x, u) \leftarrow L_f(u) \\ M_{\text{black}}(u) \leftarrow M_{\text{black}}(u) \cup \{x\} \\ V_{\text{white}}(x) \leftarrow V_{\text{white}}(x) - \mathcal{W}_{\text{white}}(x, u) \\ \mathcal{W}_{\text{white}}(x, u) \leftarrow 0 \\ M_{\text{white}}(u) \leftarrow M_{\text{white}}(u) \setminus \{x\} \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \text{Recompute}(x). \end{cases} \tag{15}$$

4.3.6 Vote to Whitelist

Analogously, the whitelist vote $\omega : U \times U \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\omega(u, x, t)(V) \coloneqq \begin{cases} \text{Unlock}(u, t) \\ V_{\text{white}}(x) \leftarrow V_{\text{white}}(x) - \mathcal{W}_{\text{white}}(x, u) + L_f(u) \\ \mathcal{W}_{\text{white}}(x, u) \leftarrow L_f(u) \\ M_{\text{white}}(u) \leftarrow M_{\text{white}}(u) \cup \{x\} \\ V_{\text{black}}(x) \leftarrow V_{\text{black}}(x) - \mathcal{W}_{\text{black}}(x, u) \\ \mathcal{W}_{\text{black}}(x, u) \leftarrow 0 \\ M_{\text{black}}(u) \leftarrow M_{\text{black}}(u) \setminus \{x\} \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \text{Recompute}(x). \end{cases} \tag{16}$$

4.3.7 Weight Synchronisation

The weight synchronisation operation $\psi : U \times \mathbb{R}_{\geq 0} \to V \to V$ (`updateUserVoteWeights`) updates the state as follows:

$$\psi(u,t)(V) \coloneqq \begin{cases} \mathrm{Unlock}(u,t) \\ \mathrm{Reweight}(u) \\ t_{\mathrm{last}} \leftarrow t. \end{cases} \tag{17}$$

4.3.8 Reward Token Funding

Funding operations advance the liveness timestamp and increase the reward pool balance. The funding operation $\varphi : U \times \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\varphi(u,a,t)(V) \coloneqq \begin{cases} B_{\mathrm{user}}(u,\tau_r) \leftarrow B_{\mathrm{user}}(u,\tau_r) - a \\ B_{\mathrm{oracle}}(\tau_r) \leftarrow B_{\mathrm{oracle}}(\tau_r) + a \\ t_{\mathrm{last}} \leftarrow t. \end{cases} \tag{18}$$

Here, a is the amount of reward tokens transferred to the contract. This action is unrestricted and prepares future submissions to receive rewards.

5 Theorems

Orb oracles enjoy many important properties. The following theorems show some of them. Theorem 1 shows that the constant-time update rule computes the same result as the Ideal Decayed Weighted Mean function (Equation 4). A naive computation of this function would require a sum over all previously submitted values whenever a new value is submitted. Instead, this theorem shows that Orb oracles can efficiently compute a new decayed weighted mean value of an unbounded number of submitted values in constant time.

Theorem 1 (Equivalence of the Ideal Decayed Weight Mean Function and the Constant-Time Update Rule)**.** *The Ideal Decayed Weighted Mean function $P(t)$ and the corresponding constant-time update rule $\bar{p}_k(t)$—that is, $\bar{p}$ after the k-th update—are equal.*

Proof. The equivalence is proven by induction on the number of updates k.
Base case ($k = 1$, so that $t(k) = t(1) = 0$): We are considering a user $u' \in U$ that has submitted a new value v at time 0. There are no previous submissions, so $Q = \bar{p} = 0$ and similarly, $p_u = w_u = 0$ for any $u \in U$. Thus, $Q' = w$ and $\bar{p}_1(0) = \frac{v \cdot w}{w} = v = P(0)$.

Inductive case (Assume by the induction hypothesis that $P(t) = \bar{p}_k(t)$): At the $(k+1)$-st update, there is a user $u \in U$ with unlocked stake that submits a new value v at a time $t(k+1) = t > t_{\text{sub}}$. So,

$$Q' = [Q_k - w_u \cdot \delta(t_{\text{sub}} - t_u)] \cdot \delta(t - t_{\text{sub}}) + w \tag{19}$$

$$= \left[\sum_{x \in U} W_{\text{user}}(x) \cdot \delta(t_{\text{sub}} - t_x) - w_u \cdot \delta(t_{\text{sub}} - t_u)\right] \cdot \delta(t - t_{\text{sub}}) + w \tag{20}$$

$$= \sum_{x \neq u} W_{\text{user}}(x) \cdot \delta(t_{\text{sub}} - t_x) \cdot \delta(t - t_{\text{sub}}) + w \tag{21}$$

$$= \sum_{x \neq u} W_{\text{user}}(x) \cdot \delta(t - t_x) + w \tag{22}$$

$$= \sum_{x \in U} W_{\text{user}}(x) \cdot \delta(t - t_x) \tag{23}$$

$$= \sum_{x \in U} W_{\text{decayed}}(x, t) \tag{24}$$

and

$$\bar{p}_{k+1}(t) = \frac{(\bar{p}_k \cdot Q_k - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q'} \tag{25}$$

$$= \frac{(\sum_{x \in U} p_x \cdot w_x \cdot \delta(t_{\text{sub}} - t_x) - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q'} \tag{26}$$

$$= \frac{\sum_{x \neq u} p_x \cdot w_x \cdot \delta(t_{\text{sub}} - t_x) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q'} \tag{27}$$

$$= \frac{\sum_{x \neq u} p_x \cdot w_x \cdot \delta(t - t_x) + v \cdot w}{Q'} \tag{28}$$

$$= \frac{\sum_{x \in U} p_x \cdot w_x \cdot \delta(t - t_x)}{Q'} \tag{29}$$

$$= \frac{\sum_{x \in U} P_{\text{user}}(x) \cdot W_{\text{decayed}}(x, t)}{\sum_{x \in U} W_{\text{decayed}}(x, t)} \tag{30}$$

$$= P(t) \tag{31}$$

□

The next theorems concern robustness against delays, which are defined as follows.

Definition 3 (Oracle Operator Delay)**.** *Define an oracle operator's delay as:*

$$\Lambda_x(t) = \frac{W_{\text{decayed}}(x, t) \cdot (t - T_{\text{user}}(x))}{\sum_{x \in U} W_{\text{decayed}}(x, t)} \tag{32}$$

Definition 4 (Oracle Delay)**.** *From the definition of the oracle operator delay, define the oracle's delay as:*

$$\Lambda(t) = \sum_{x \in U} \Lambda_x(t) \tag{33}$$

Theorem 2 shows that, if an oracle operator becomes inactive, the delay that it causes to the oracle gradually converges to zero. This is important, given that Orb oracles do not delete values submitted by inactive oracle operators, but only decay them.

Theorem 2 (Inactive Oracle Operator Delay)**.** *If an oracle operator $u \in U$ becomes inactive, the operator delay $\Lambda_u(t)$ caused by this inactivity converges to 0.*

Proof. By definition, the oracle operator delay for $u \in U$ is:

$$\Lambda_u(t) = \frac{W_{\text{decayed}}(u,t) \cdot (t - T_{\text{user}}(u))}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{34}$$

$$= \frac{w_u \cdot \delta(t - t_u) \cdot (t - t_u)}{\sum_{x \in U} w_x \cdot \delta(t - t_x)} \tag{35}$$

$$= \frac{w_u \cdot 2^{\frac{-(t-t_u)}{h}} \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{-(t-t_x)}{h}} + w_u \cdot 2^{\frac{-(t-t_u)}{h}}} \tag{36}$$

$$= \frac{w_u \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{(t-t_u)}{h}} \cdot 2^{\frac{-(t-t_x)}{h}} + w_u} \tag{37}$$

$$= \frac{w_u \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{(t_x-t_u)}{h}} + w_u}. \tag{38}$$

So, if the oracle operator u is inactive, then w_u and t_u remain constant as $t \to \infty$. As long as a subset $Y \subseteq U \setminus \{u\}$ remains active, then $y \in Y$, $t_y \to \infty$ as $t \to \infty$, so that $t_y - t_u \to \infty$. Hence, for each active $y \in Y$, the term $w_y \cdot 2^{\frac{t_y - t_u}{h}}$ in the denominator tends to infinity, while the numerator $w_u \cdot (t - t_u)$ grows only linearly in t. Thus,

$$\lim_{t \to \infty} \Lambda_u(t) = 0 \qquad \square$$

Theorem 3 shows that the oracle delay is never greater than the difference between the current time and the submission time of the latest value submitted by the most delayed oracle operator. This theorem is true thanks to the fact that the constant-time update rule carefully removes the influence of non-latest values in the computed weighted average.

Theorem 3 (Optimally Small Oracle Delay)**.** *The oracle delay $\Lambda(t)$ is bounded above by $(t - t^*)$, where $t^* = \min_{x \in U} t_x$.*

Proof. Expanding the definitions of oracle delay and oracle operator's delay, we have that:

$$\Lambda(t) = \sum_{x \in U} \Lambda_x(t) \tag{39}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - T_{\text{user}}(x))}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{40}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{41}$$

$$\leq \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot \max_{x \in U}(t - t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{42}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - \min_{x \in U} t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{43}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - t^*)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{44}$$

$$= \frac{\sum_{x \in U} W_{\text{decayed}}(x,t) \cdot (t - t^*)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{45}$$

$$= (t - t^*) \tag{46}$$

Thus,

$$\Lambda(t) \leq (t - t^*)$$

□

Theorem 4 shows that as long as the oracle has been funded at least once, the balance of the oracle's reward pool will always remain positive, ensuring that there is always a positive reward[9] to incentivize oracle operators to continue submitting values.

Theorem 4 (Sustainable Rewards)**.** *If* $0 \leq \alpha < 1$ *and* $B_{\text{oracle}}(\tau_r) > 0$ *at a given time* t*, then* $B_{\text{oracle}}(\tau_r) > 0$ *for all times* $t' > t$*.*

Proof. By assumption, since $\alpha \in [0, 1)$ and $B_{\text{oracle}}(\tau_r) > 0$ at a given time t, then any

[9] Note that, although the reward pool always remains positive, it can become arbitrarily small. In particular, the reward could become smaller than the blockchain's gas fee to submit values. Users relying on the oracle can ensure that this does not happen by executing the reward token funding operation.

update must by definition occur at a time $t' > t$, where the remaining oracle balance will be:

$$B_{\text{oracle}}(\tau_r) - n(t') = B_{\text{oracle}}(\tau_r) - \alpha \cdot B_{\text{oracle}}(\tau_r) \cdot \frac{w}{Q'} \cdot (1 - \delta(t' - t_u)) \tag{47}$$

$$= B_{\text{oracle}}(\tau_r)(1 - \alpha \cdot \frac{w}{Q'} \cdot (1 - \delta(t' - t_u))) \tag{48}$$

$$> B_{\text{oracle}}(\tau_r)(1 - \alpha) \tag{49}$$

$$> 0 \tag{50}$$

Thus, for all future updates, and therefore all future times, since the remaining balance is positive and becomes the new balance, the new balance is also positive. □

6 Formalization

We formalized[10] the Orb oracle protocol and verified the main theorems of this paper in the Rocq (formerly Coq) theorem prover. Rocq [21] is an interactive proof assistant based on the Calculus of Inductive Constructions, which provides a formal language for definitions, algorithms, and machine-checked proofs. Its tactic language allows proofs to be developed incrementally while checking each step for correctness, providing interactive feedback to the user. We chose Rocq over other theorem provers due to our familiarity with its tactic language and its proven record in formal verification.

The purpose of the formalization was twofold: to obtain stronger guarantees for theorems about Orb, and to make implicit assumptions and proof steps fully explicit. The process of formalization helped uncover places where the informal presentation needed clarification, adjustment, or a more precise statement, or even revealed typos in the preliminary protocol statement.

6.1 Design

Abridged Rocq code snippets are provided to showcase the datatypes and records representing the oracle protocol in the formalization. Rocq names for oracle parameters and variables were chosen to be as close as possible to the protocol specification.

```
Definition timestamp : Type := R.
Definition weight : Type := R.
```

[10]The formalization is available at `https://github.com/StabilityNexus/OrbOracle-Paper/`. The accompanying documentation is structured to align with and reference both the paper and this formalization section. The HTML documentation can be generated using coqdoc, or accessed directly at `https://stabilitynexus.github.io/OrbOracle-Formalization/Oracle.html`. Installation instructions for Rocq/Coq are provided in the GitHub repository.

```
Definition value : Type := R.
Definition balance : Type := R.
Definition history : Type := list (timestamp * value * value).
Module UserMap := FMapList.Make(UserOT).
```

Listing 1: Domains

In any formalization, it is important to balance faithfully capturing the mathematical model with maintaining tractability and convenience in the mechanization. Upon starting the formalization effort, we were immediately confronted by such a concern regarding numbers. Timestamps, weights, and balances are defined as non-negative reals in the protocol in the paper. We considered representing them as non-negative reals in the form of a sigma type [22] that carried both the number and a proof of its non-negativity, in order to keep close with their mathematical meaning. However, this quickly proved impractical because we needed to prove their non-negativity at every occasion the oracle operations modified any weight or balance. While keeping with this approach might be more thorough, our focus was to prove the theorems about the oracle protocol. Instead, we kept timestamps, weights, and balances as real numbers, and enforced their non-negativity through explicit hypotheses and auxiliary lemmas when required in proofs.

We used Rocq's Real number library [23] as it has a rich set of lemmas to manipulate inequalities and algebraic expressions, and access to the `lra`, `lia`, `field_simplify` automated tactics.

```
Parameter h : Rpos.                (* half-life constant *)
Parameter q : R.                   (* quorum constant *)
Parameter Delta_dep : timestamp.   (* deposit locking period *)
Parameter Delta_wd : timestamp.    (* deposit locking period *)
Parameter alpha : R.               (* reward factor *)
Definition User : Type := nat.
Definition UserSet := list User.
Parameter Users : UserSet.
```

Listing 2: Parameters

The oracle parameters, which are global constants, are naturally left as Rocq `Parameter`. Users are defined abstractly and `Users`, the universe of users is defined as a parameter (hence immutable), and is also kept as a list to work with our defined summation over real numbers `sum_list_R`, which is simplest to do as folding addition over a list.

```
Record GovernanceState := {
    B : UserMap.t bool;              (* blacklist indicators *)
    V_black : UserMap.t weight;      (* accumulated blacklist weight *)
    V_white : UserMap.t weight;      (* accumulated whitelist weight *)
    M_black : UserMap.t UserSet;     (* blacklisted targets *)
    M_white : UserMap.t UserSet;     (* whitelisted targets*)
```

```
    W_black : UserPairMap.t weight; (* pairwise blacklist weight *)
    W_white : UserPairMap.t weight; (* pairwise whitelist weight *)
  }.

Record OracleState := {
    L_l     : UserMap.t balance;    (* locked token balances *)
    L_f     : UserMap.t balance;    (* unlocked token balances *)
    T_dep   : UserMap.t timestamp;  (* last deposit timestamp *)
    T_op    : UserMap.t timestamp;  (* last operation timestamp *)
    P_user  : UserMap.t value;      (* last reported value of users *)
    W_user  : UserMap.t weight;     (* last submission weight  *)
    T_user  : UserMap.t timestamp;  (* last submission time  *)
    pbar    : value;                (* stored aggregated value *)
    ptilde  : value;                (* latest submitted value *)
    Q       : weight;               (* stored aggregate weight *)
    Phist   : history;              (* time-ordered value history *)
    t_sub   : timestamp;            (* last value submission time *)
    t_last  : timestamp;            (* last operation timestamp *)
    L_tot   : balance;              (* total deposited tokens *)
    G       : GovernanceState       (* governance state *)
  }.

Record State := {
    V : OracleState;                (* oracle state *)
    B_user_w : UserMap.t balance;   (* external oracle token balance *)
    B_user_r : UserMap.t balance;   (* external reward token balances *)
    B_oracle_w : balance;           (* oracle token balance of oracle *)
    B_oracle_r : balance;           (* reward token balance of oracle *)
  }.
```

Listing 3: State records

The formalization separates the protocol state into an `OracleState` and an outer `State`. The `OracleState` contains the oracle-local quantities needed for update and delay computations: user balances internal to the oracle, per-user submission data, aggregate quantities such as Q and $\bar{p}$, timestamps, history, and governance state. The outer `State` extends this with external token balances, including user balances outside the oracle and the oracle's own reward-token and oracle-token balances. This separation is largely arbitrary and was made to mirror the separation of the external balances, oracle state and governance state in Section 3.

```
Inductive Operation : Type :=
| Deposit (u : User) (a : balance) (t : timestamp)
| Withdrawal (u : User) (a : balance) (t : timestamp)
| Submission (u : User) (v : value) (t : timestamp)
| Reading (u : User) (t : timestamp)
| VoteBlacklist (u : User) (x : User) (t : timestamp)
| VoteWhitelist (u : User) (x : User) (t : timestamp)
| WeightSync (u : User) (t : timestamp)
```

```
| RewardFunding (u : User) (a : balance) (t : timestamp)
| NoneOp (t : timestamp) .

Definition Run : Type := nat → Operation.

Fixpoint exec_prefix (run : Run) (n : nat) (St : State) : State :=
  match n with
  | O ⇒ St
  | S n' ⇒ exec_op (run n') (exec_prefix run n' St)
  end.
```

Listing 4: Operations and infinite traces

Operations are modeled as a datatype where each constructor corresponds to a protocol action such as deposits, withdrawals, submissions, governance votes, and reward funding. This representation makes the transitions explicit and allows case analysis over operations to be carried out directly. System executions are then modeled as infinite traces called runs, represented as functions from natural numbers to operations. Since the natural numbers are infinite, the `run` encodes a sequence of infinite operations indexed by the natural number.

The state at a given step is obtained by executing the prefix of the run up to that index from an initial state, using a recursively defined execution function. This allows for induction over the length of the prefixes. In particular, we can exploit this to show that a property holds initially and is preserved by each transition (casing over it), showing that the property holds for the entire run. This method of proof will be used in the proofs of Theorems 1, 2 and 4.

6.2 Formalization of the Theorems

The formalization of the main theorems varied in difficulty. Some proofs followed the informal arguments closely, while others required additional auxiliary lemmas, more explicit hypotheses, or a different proof structure in order to be carried out mechanically. In several cases, formalization helped identify assumptions that were implicit in the paper but needed to be stated clearly in the mechanized development.

6.2.1 Theorem 1

The formal statement of Theorem 1 is stated as follows:

```
Definition mean_eq_curr_pbar (st : OracleState) : Prop :=
  0 < Q st →
   P_ st (t_sub st) = pbar st.

Theorem P_equiv_pbar:
  forall (run : Run) (n : nat),
    (forall m u v t,
```

```
      run m = Submission u v t →
      In u Users) →
mean_eq_curr_pbar (V (state_at run n)).
```

Listing 5: Theorem 1 statement

The formalized proof of Theorem 1 proceeds by first strengthening the theorem statement into a raw state invariant rather than proving the final equality directly. `Q` coincides with the denominator of $P(t)$, and `p_bar * Q` coincides with the numerator of $P(t)$. More explicitly, the invariant asserts that

$$Q = \sum_{x \in U} W_{\text{user}}(x)\, \delta\big(t_{\text{sub}} - T_{\text{user}}(x)\big)$$

and

$$\bar{p}\, Q = \sum_{x \in U} P_{\text{user}}(x)\, W_{\text{user}}(x)\, \delta\big(t_{\text{sub}} - T_{\text{user}}(x)\big).$$

This reflects the two main algebraic identities in the paper proof: first, that the updated denominator Q' matches the sum of decayed weights, and second, that the updated aggregate value $\bar{p}'$ is the corresponding decayed weighted average. The submission case is then proved at this stronger level following the paper's algebraic steps, and then auxiliary lemmas connect the updated sums to the state at the time of a submission operation. Once this is established, the theorem-facing statement is recovered by a bridge lemma, `mean_raw_eq_implies_mean_eq_curr_pbar`, which shows that the ideal mean at the current submission time equals the stored $\bar{p}$.

A lesson from the formalization concerns the structure of the induction. The paper proof is phrased as an induction on the number of updates k, focusing directly on the next submission and the updated quantities Q' and $\bar{p}'$. In the mechanization, however, we worked over an infinite trace of operations (`run`), and therefore had to reason about every possible operation that may occur. Instead of only handling the update step, we proved a general step lemma over runs, where the substantial case is submission and all other operations are routine cases that show that other operations do not affect the values relevant to $P(t)$.

We assumed that every user making a submission belongs to `Users`, and that `Users` behaves like a set, in the sense that it contains no duplicates. As explained in the design section, we kept `Users` as a list so it could work with our defined summation over real numbers. This assumption is technical and does not change the meaning of the original theorem. This is relevant because one of the algebraic steps of the proof isolates a term from a sum over `Users`.

6.2.2 Theorem 2

The formal statement of Theorem 2 is stated as follows:

```
Theorem inactive_oracle_operator_delay :
  forall (run : Run) (u : User),
    In u Users →
    inactive_along_run run u →
    (forall z n,
        getR z (T_user (V (state_at run n))) <=
        getR z (T_user (V (state_at run (S n))))) →
    (forall i x,
        0 <= getR x (W_user (V (state_at run i)))) →
    (forall i, exists u0,
        In u0 Users ∧ getR u0 (W_user (V (state_at run i))) > 0) →
    (exists y,
        In y Users ∧
        y <> u ∧
        (forall T : R,
             exists n, is_submission_by y (op_at run n) ∧
                       T <= getR y (T_user (V (state_at run n)))) ∧
        (forall n : nat,
             getR y (T_user (V (state_at run n))) <=
             t_last (V (state_at run n))) ∧
        (exists K : R, 0 <= K ∧
             forall n : nat, t_last (V (state_at run n))
            <= getR y (T_user (V (state_at run n))) + K) ∧
        (exists wy_min : R,
            0 < wy_min ∧
            exists Ny : nat,
              forall n : nat, (n >= Ny)%nat →
                wy_min <= getR y (W_user (V (state_at run n))))) →
      tends_to_0_seq (Lambda_run run u).
```

Listing 6: Theorem 2 statement

Theorem 2 is one of the most substantial parts of the formalization, primarily because it required reasoning about asymptotic behavior. To support this, we introduced the notion of an infinite trace (a `run`), and defined convergence over runs as limits over sequences indexed by the execution step number (`tends_to_0_seq`). The proof itself follows the high-level structure of the paper: we first rewrite the oracle delay into a form where asymptotic behavior is visible, then use inactivity to show that the inactive operator's timestamp and weight are eventually constant (by casing on operations and showing that they do not affect it, similar to Theorem 1's casing). This reduces the problem to bounding a ratio where the numerator grows at most linearly in n and the denominator contains a sum of exponentially growing terms.

Notably, proving this lemma required additional effort, as it involves establishing a fully quantified inequality (`exp2_dominates_linear`) showing that exponential growth eventually dominates linear growth. We also considered using Coquelicot [3], an external

real analysis library extending Rocq's standard Reals library, to formalize the limit argument. However, in our setting, time is not a primitive real parameter but is derived from discrete execution steps, so the relevant notion of convergence is that of real-valued sequences. While Coquelicot provides a comprehensive development of real analysis — including limits of sequences — it introduces a substantial amount of additional infrastructure that was not directly needed for this proof. Instead, we opted for a lightweight, self-contained definition of convergence.

Another aspect clarified during formalization is the notion of inactivity and the role of an active user. The paper proof of Theorem 2 mentions inactivity and reasons about an "active set" of users contributing to the denominator. In the formalization, we define inactivity explicitly as eventual absence of submissions along a run, and prove that this implies that the operator's timestamp and weight become constant beyond some point. For the argument about the denominator, we work with a single user y that submits unboundedly often, rather than an arbitrary active subset $Y \subseteq U \setminus \{u\}$. This is sufficient because the denominator is a sum of nonnegative terms, so the presence of one term that grows without bound already ensures the entire denominator grows without bound. Also, note that `Users` is a fixed finite parameter, with no notion of users joining or leaving during execution. As a result, if submissions continue indefinitely, at least one user must submit infinitely often. This allows us to work with a single indefinitely-active user rather than the "active set" described in the paper proof.

We made several assumptions that were implicit in the informal proof. These include nonnegativity of weights, the existence of at least one user with positive weight at each step, monotonicity of user timestamps, and a relationship between user timestamps and the global time parameter (`t_last`). Making these conditions explicit helps ensure that the denominator remains well-defined and grows sufficiently fast to dominate the numerator.

6.2.3 Theorem 3

The formal statement of Theorem 3 is stated as follows:

```
Theorem optimally_small_oracle_delay :
  forall (t : timestamp) (st : OracleState),
    (forall u, In u Users → 0 <= getR u (W_user st)) →
    (exists u0, In u0 Users ∧ 0 < getR u0 (W_user st)) →
    Lambda t st <= t - tstar st.
```

Listing 7: Theorem 3 statement

Theorem 3 is relatively straightforward as it only required expanding and manipulating the definitions of oracle and oracle operator delay and did not have to deal with infinite traces — for this reason, we formalized this theorem first. The formal proof of Theorem 3 follows the same high-level idea as the informal proof: the total oracle delay is expressed as a sum of per-user contributions, and each contribution is bounded above by the maximal delay

determined by the oldest retained submission time. However, Rocq's Reals library did not come with definitions and lemmas for summations, so we defined a summation of a function over a list of real numbers `sum_list_R` and proved identities such as factoring constants in order to manipulate the expression as needed.

We assume that all user weights are nonnegative (as defined in the protocol), and at least one user has positive stored weight, so that the denominator is strictly positive and we avoid division by zero, which is also required to use Rocq's field rewriting lemmas.

6.2.4 Theorem 4

The formal statement of Theorem 4 is stated as follows.

```
Theorem sustainable_rewards :
    forall (run : Run) (n0 : nat),
    (0 <= alpha < 1) →
    (forall u st, wu u st * decay (t_sub st - tu u st) <= Q st) →
    (forall run k u a t,
        op_at run k = RewardFunding u a t →
    0 <= a) →
    (forall run k u v t',
        op_at run k = Submission u v t' →
        0 < t' - tu u (V (state_at run k))) →
    (forall run k u v t',
        op_at run k = Submission u v t' →
        0 < Q' u t' (Unlock (V (state_at run k)) u t')) →
    B_oracle_r (state_at run n0) > 0 →
    forall n : nat,
    (n >= n0)%nat →
    B_oracle_r (state_at run n) > 0.
```

Listing 8: Theorem 4 statement

The formal proof of Theorem 4 follows the same high-level idea as the informal proof but requires making explicit several implicit assumptions and reasoning steps. We first show that positivity of the oracle balance is preserved by a single transition of the run by performing a case analysis on the operation at that step. For submission operations, we algebraically rewrite the balance update into a multiplicative form and prove that the multiplicative factor is at most 1 ensuring the balance is positive, as done in the paper proof, but in doing so we also had to prove the implicit bounds $0 \leq \frac{w}{Q'} \leq 1$ and $0 \leq 1 - \delta(t' - t_u) \leq 1$. For reward funding and other operations, we show the balance either increases or remains unchanged. Similar to Theorem 1 and 2, reasoning on the run required considering all possible operations, which was something that was implicit to the paper proof.

The mechanized proof required explicitly ensuring that all quantities in the reward expression are well-defined and bounded: in particular, we assume that the denominator Q' is

strictly positive to avoid division by zero, that submission timestamps are strictly increasing to guarantee decay terms lie in $(0, 1)$, and that the aggregate quantity Q bounds individual decayed contributions.

6.3 General Remarks

A consistent theme in the formalization is reasoning over runs. Instead of arguing directly about expressions (besides in Theorem 3), we reason about states obtained from an infinite trace. This requires a systematic case analysis over operations: for each step of the run, we must show how the operation affects (or does not affect) the quantities of interest. This meant that properties like eventual constant weight under inactivity in Theorem 2 and preservation of positive balance in Theorem 4 were all established by proving that they hold initially and are preserved by each possible operation, with certain cases typically carrying the main argument and all other operations shown not to affect relevant values.

Another recurring aspect of the formalization is that many assumptions implicit in the informal proofs had to be made explicit. These include nonnegativity of weights, existence of a user with positive weight, monotonicity of timestamps, and relationships between user timestamps and the global time parameter. In the paper, these conditions are either built into the definitions or used implicitly when manipulating expressions. In the mechanized setting, however, they must be stated as hypotheses to ensure that expressions are well-defined (e.g., denominators are nonzero) and that inequalities can be applied. Making these assumptions explicit clarifies the exact conditions under which the results hold.

The formalization highlights a distinction between defining the structure of the state and specifying its expected behavior. While the state and parameters are defined concretely (e.g., balances, timestamps, and maps), many desirable properties — such as nonnegativity of balances, totality of maps, or consistency conditions like submission times being bounded by the last operation time — are not enforced directly by these definitions. One possible approach is to collect such properties into a global well-formedness predicate and prove that it is preserved along runs. Instead, we chose to state the required invariants locally within each theorem. This keeps the assumptions explicit and closely aligned with the needs of each proof, while avoiding bundling them into a single predicate whose components may not all be relevant in every argument.

Overall, the formalization verifies the main results while making explicit many assumptions and reasoning steps that are implicit in the informal proofs. While the core results in the paper have been formalized and verified, further refinements of the active development remain possible. In particular, some auxiliary assumptions could be weakened, and additional invariants suggested by the protocol design could be formalized in future work.

7 Related Work

There are many thorough and extensive surveys of oracles (e.g. [14, 17], where the second one also cites additional surveys), that contain taxonomies and comparisons of oracles.

Among the various dimensions along which oracles may be classified and grouped, the following are relevant in Orb's context:

- **on-chain or off-chain:** some oracles, like Orb, push data on-chain, writing it to a smart contract, whereas others provide signed timestamped data off-chain to be pulled by applications that need it. The latter avoid the transaction costs of frequently writing on-chain and, precisely due to such costs, it is the only viable option nowadays for arbitrary data that is rarely used. However, for data that is time-sensitive, such as price, the latter has a serious drawback. Applications depending on such oracles need to choose how old the pulled data may be, in order to be acceptable by the application's smart contract. If the chosen maximum age is too short and the oracle becomes unavailable, users may be unable to interact with the smart contract, because all the available signed data would be considered too old and would be rejected by the application's smart contract. On the other hand, if the chosen maximum age is too long, malicious users may fetch values off-chain at multiple points in time, all within the maximum age, and choose the value that is better for them when interacting with the contract, instead of using the latest value. In other words, users have a free option to delay the oracle up to the maximum age for their advantage and possibly to the detriment of the contract itself or other users of the contract. Orb positions itself firmly in the on-chain camp, prioritizing the availability of a single current value on-chain.

- **(de)centralization:** some oracles (e.g. [1]) are intentionally centralized and, for some applications, that wish to know precisely from which first-party they are obtaining data and wish to trust exactly that first-party, this may even be desirable. Another advantage of centralized oracles is that the delay is small. As soon as the single entity submits a new value, it immediately becomes the new value that can be read from the oracle. In contrast, in a decentralized oracle where multiple entities submit values, the aggregated value will often be an average of values submitted at different points in time and thus there will be an unavoidable delay. Even more decentralized oracles (e.g. [9, 5, 8]) often have mechanisms that reward oracle operators with more reputation for submitting the "right" value or punish them for submitting the "wrong" value. But this naturally raises the question: who is deciding whether the submitted values are right or wrong? Many decentralized oracles (e.g. [9, 5]) allow holders of a token to act as oracle operators. But usually, it is a single token, and there is no guarantee that the interests of the holders of this token are aligned with the interests of the users of the oracle. Orb generalizes this idea by making the oracle token a parameter that is set

when oracle contracts are deployed. This allows users to create and operate oracles with their desired degree of (de)centralization and possibly with oracle tokens that are economically aligned with their applications.

- **rewards for oracle operators:** many oracles (e.g. [11]) fail to consider a sustainable business model for oracle operators and continue to exist mainly due to altruism. Or, even when they do consider it in theory [9, 5], in practice, they may actually operate as public goods and free infrastructure. It is unclear how they get funded and for how long they will be able to continue operating. In our previous work [18, 19], we designed an oracle protocol where oracle consumers had to pay for every reading of data. However, we later realized that how often an application reads data from the oracle is not necessarily a good estimate for how valuable the oracle is for that application and for how much that application should be paying to fund the oracle. Even in cases when an application is currently not reading data from the oracle, perhaps because there is no significant price movement, it is still important for that application to keep funding the oracle operation since having correct and up-to-date values protects the funds under the application's management. A more flexible approach was needed. We took inspiration from Ergo's Oracle V2 [7, 10], which is being used by Gluon Gold [16] and DexyGold [6]. This oracle makes it possible for the applications (e.g. Gluon Gold, DexyGold) to fund the oracle operator and lets them decide how to do it. However, Ergo's Oracle V2 has no protection against free-riding: applications may read data from it without providing any funding. In the Orb Oracle Protocol, the free-riding problem is solved by allowing the oracle operators to jointly blacklist free-riders.

8 Conclusion and Future Work

The Orb Oracle Protocol allows users to deploy and operate new oracles with a high degree of flexibility regarding decentralization and economic incentives. With a carefully designed value aggregation formula, we showed that the aggregated value can be updated in constant-time, independently of the number of active oracle operators (Theorem 1) and that the effect of inactive operators on the aggregated value quickly becomes negligible (Theorem 2). The practical implication of this result is that the degree of decentralization that can be achieved becomes unbounded. We also showed additional theorems related to optimality of the oracle delay (Theorem 3) and sustainability of the rewards (Theorem 4). We have formally verified the proofs of these theorems.

Besides our formal specification, our reference implementation in Solidity is ready. Our next step is to launch a platform for deploying and operating oracles based on this reference implementation. One direction for future work that we would like to explore is the

efficient composition of oracles, so that oracles for multiple currency pairs could be operated simultaneously with fewer on-chain value submission transactions.

References

[1] API3. API3: Decentralized APIs for Web 3.0. https://drive.google.com/file/d/1b8QsGPCJJC1pQOcg83-knD1IAOIgCtZZ/view.

[2] Davide Basile, Valerio Goretti, Claudio Di Ciccio, and Sabrina Kirrane. Enhancing blockchain-based processes with decentralized oracles. In José González Enríquez, Søren Debois, Peter Fettke, Pierluigi Plebani, Inge van de Weerd, and Ingo Weber, editors, *Business Process Management: Blockchain and Robotic Process Automation Forum*, pages 102–118, Cham, 2021. Springer International Publishing.

[3] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[4] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. White paper, 2014. https://ethereum.org/en/whitepaper/.

[5] Chainlink. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks, 2021. https://research.chain.link/whitepaper-v2.pdf.

[6] Alexander Chepurnoy and Luca D'Angelo. Dexy: Simple stablecoin design based on algorithmic central bank. Accepted paper at First Stability Workshop, 2025. https://workshop.stability.nexus/.

[7] Ergo Developers. Ergo: A resilient platform for contractual money. White paper, 2019. https://ergoplatform.org/docs/whitepaper.pdf.

[8] Jingya Dong, Chunhe Song, Yong Sun, and Tao Zhang. DAON: A decentralized autonomous oracle network to provide secure data for smart contracts. *IEEE Transactions on Information Forensics and Security*, 18:5920–5935, 2023.

[9] Steve Ellis, Ari Juels, and Sergey Nazarov. ChainLink: A decentralized oracle network, 2017. https://research.chain.link/whitepaper-v1.pdf.

[10] Ergo Platform. Oracle Core v2.0. GitHub repository and documentation, 2026. https://github.com/ergoplatform/oracle-core.

[11] Robert Kornacki. Oracle pools. Emurgo Research technical overview, June 2020. https://github.com/Emurgo/Emurgo-Research/blob/master/oracles/Oracle-Pools.md.

[12] Evan Kuo, Brandon Iles, and Manny Rincon Cruz. Ampleforth: A new synthetic commodity, 2019. https://resources.cryptocompare.com/asset-management/534/1693235714553.pdf.

[13] Robert Leshner and Geoffrey Hayes. Compound: The money market protocol, 2019. https://compound.finance/documents/Compound.Whitepaper.pdf.

[14] Bowen Liu, Pawel Szalachowski, and Jianying Zhou. A first look into defi oracles, 2021. https://arxiv.org/abs/2005.04377.

[15] MakerDAO. The Maker protocol white paper, 2020. https://makerdao.com/en/whitepaper/.

[16] Bruno Woltzenlogel Paleo, Luca D'Angelo, Mohammad Shaheer, and Giselle Reis. Gluon w: A cryptocurrency stabilization protocol. Cryptology ePrint Archive, Paper 2025/1372, 2025.

[17] Amirmohammad Pasdar, Young Choon Lee, and Zhongli Dong. Connect api with blockchain: A survey on blockchain oracle implementation. *ACM Comput. Surv.*, 55(10), February 2023.

[18] Mohammad Shaheer. Design and formalization of blockchain oracles. Undergraduate Thesis, 2023. `https://kilthub.cmu.edu/articles/thesis/Design_and_Formalization_of_Blockchain_Oracles/23467967`.

[19] Mohammad Shaheer, Giselle Reis, Bruno Woltzenlogel Paleo, and Joachim Zahnentferner. Formalization of blockchain oracles in coq. In *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts*, page 161.

[20] Nick Szabo. Smart contracts: Building blocks for digital markets. Extropy: The Journal of Transhumanist Thought, 1996. Available at `https://web.archive.org/web/20151222144315/http://szabo.best.vwh.net/smart_contracts_2.html`.

[21] The Coq Development Team. The coq proof assistant, September 2023.

[22] The Coq Development Team. Coq.Init.Specif: Basic specifications, 2023. `https://rocq-prover.org/doc/v8.18/stdlib/Coq.Init.Specif.html`.

[23] The Rocq Development Team. *Rocq Standard Library: Module `Stdlib.Reals.Reals`*. Inria Rocquencourt, 2025. Provides the real number axiomatization and core analysis theorems on the type `R`.

[24] Bruno Woltzenlogel Paleo. *Stablecoin*, pages 1–5. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[25] Zahnentferner, Anjali Jha, and Raj Jitendra Shah. Fate Protocol: Perpetual prediction pools, 2025. `https://workshop.stability.nexus`.

[26] Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed pegged algorithmic stablecoin. Cryptology ePrint Archive, Report 2021/1069, 2021. `https://eprint.iacr.org/2021/1069`.

[27] Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed pegged autonomous stablecoin. In *IEEE International Conference on Blockchain and Cryptocurrency*, 2023. `https://icbc2023.ieee-icbc.org/program/icbc-2023-final-program`.

A License

[11] `https://creativecommons.org/licenses/by-nc-nd/4.0/`

uses, contact the authors first.

B Official Implementations and Deployments of Orb

`https://github.com/StabilityNexus`

C Official Orb Channels

- `https://x.com/StabilityNexus`
- `https://discord.gg/7jS9qJNjJv`

Received