OPEN SOURCE COIN

TRUST AND SUSTAINABILITY IN OPEN SOURCE COMMUNITIES

VERSION 1.0 – MARCH 2019

ALEXIS SELLIER[†] ELEFTHERIOS DIAKOMICHALIS[†] JAMES HAYDON[†]

Abstract. The success of the open source movement stems from its dynamic culture of peer-to-peer distribution and collaboration. Yet, as open source has grown from free *software ideal* to *industry standard*, finding a means of sustainable financing has become an increasingly pressing concern. Here we propose open source coin (oscoin), a secure peer-to-peer protocol for establishing a sustainable free and open source software economy.

1. Background

> *It is in this light that we must recognize that only a restoration of open-source culture, and all that enables across the full spectrum of open-source possibilities, can allow humanity to harness the distributed intelligence of the collective and create the equivalent of heaven on Earth — in other words, a world that works for all.*
>
> — The Open-Source Everything Manifesto

With the advent of digital scarcity, it became possible to economically incentivize and remunerate network participants simply and transparently, without need of a trusted third-party. The introduction of Bitcoin [1], and later Ethereum [2], led to the proliferation of alternative cryptocurrencies primarily competing on network utility and monetary policy.

If Bitcoin sought to reward network operators for validating transactions and securing the network, projects such as Zcash and Decred extended this concept to incentivizing developers and other value producers in order to further stimulate their respective economies. Despite these experiments, no cryptocurrency has succeeded in creating direct sustainable incentives for developers who either contribute directly to the protocol, or indirectly to the underlying open source infrastructure.

This is one of the many cases of free software projects facing incentivization challenges, a problem that has been extensively discussed in literature [4], and affects the software ecosystem at large.

1.1. Developer incentives and sustainability.

To motivate this work, let us familiarize ourselves with the conditions in which free and open source software[1] is developed. Most of the software we use on a daily basis relies on free, publicly available code. In an increasingly digitized society, free software projects have become the foundation of digital infrastructure underpinning many of our societal goods and services.

With the emergence of software hosting sites like GitHub and community sites like Stack Overflow, open source became the preferred software engineering paradigm, resulting in numerous high quality projects published in the open, available for anyone to use. This phenomenon helped companies reduce lead times and bring products to the market faster, as well as recruit talent from a new pool of technologists educated on the basis of open source software.

Today, many free software projects start with individuals or small groups solving a personally, socially, or technologically relevant problem. Reviewing their motivation for participating in the free software ecosystem, we see: pride in one's work, reputation, learning, responsibility for something they believe in, and being part of a community.

While most open source software projects start for the reasons above, those that gain momentum require significant time and financial resources to maintain. Therein lies the fundamental problem: an important subset of projects started on a volunteer basis are becoming critical public infrastructure. And while some developers find ways to finance their efforts, most struggle to keep up with community requests and maintenance during their free time, often abandoning their projects or burning out under the burden of increasing responsibility. It is this problem, developing a healthy and sustainable means of free software maintenance and financing, that we address in this work.

2. The Oscoin Network

In this paper, we introduce oscoin, a cryptocurrency protocol designed to provide a solution to the open source sustainability problem.

[†]Monadic, {alexis,ele,james}@monadic.xyz.

[1]In this work, we shall use the terms *open source software* and *free software* interchangeably, to mean *software distributed under terms that allow users to run it for any purpose, as well as change and re-distribute its source code.*

Oscoin is a public network of computers participating in a consensus protocol around a shared transaction ledger. This ledger is materialized into a global state $\mathcal{S}$ which contains the canonical registry $\mathcal{R}$ of all open source projects participating in the protocol; a set of accounts $\mathcal{A}$, containing the balances of all currency holders; and a network graph $\mathcal{N}$, of the relevant relations between entities in the network, including software dependencies between registered projects.

The purpose of this network is to secure a digital currency—oscoin—and reward the most valued projects in the network, without the need for intermediaries or central control.

2.1. **The Oscoin Blockchain.** To design a safe, open, "permissionless" currency shared amongst all network participants, we propose oscoin as a *blockchain* protocol, as described by Nakamoto [1] and Wood [2].

Blockchain protocols solve the problem of trust at scale by allowing monetary transactions to be processed without middleman, while remaining censorship-resistant—an essential characteristic of open networks. Our proposed solution introduces a new cryptocurrency, oscoin, designed to be governed by network participants rather than a central entity. We believe this is crucial to the success of oscoin, and blockchain technology offers the most promising solution to this end.

Though the specific choice of the underlying consensus protocol is not critical to oscoin, we believe that one which allows open participation is best for the long term success of the network. Furthermore, it is important that reliable support for light clients be possible, as running a full node is prohibitively expensive for many that could benefit from participating in the oscoin network. As of today, the only family of blockchain protocols with these characteristics are *proof-of-work* protocols.[2] However, we are fully aware of the environmental impact of proof-of-work at scale and propose transitioning to a more energy-efficient protocol as research in this area develops.[3]

2.2. **The Oscoin Treasury.** The treasury system (Figure 1) is designed to continuously rank projects on the network based on their relative importance, and reward them with oscoin. This mechanism is an essential component of the network and fulfills the role of providing continuous incentives to maintainers.

2.2.1. *Overview.*

- Maintainers and contributors collaborate on software projects registered within the network.
- Maintainers merge contributions and synchronize their project state with the ledger, which includes per-project dependency information and contribution metadata (§4.4.2).
- Every κ blocks the osrank of each project is calculated (§3) based on the metadata above.
- A reward in oscoin is derived from each project's osrank and distributed via the project's associated smart contract (§5).
- Based on the project's smart contract specification, a share of the reward flows to contributors and maintainers of the project.
- These rewards in oscoin follow a fixed vesting schedule, after which they can be transfered out of the project.

Each block, the protocol is allowed to mint a certain amount B_r of oscoin as part of the "block reward" or *coinbase*. These coins are allocated through the treasury system to two distinct groups: network operators ($\mathcal{M}$), and open source projects ($\mathcal{P}$).

The key question we must address, is what share of B_r is distributed to each individual project in the network. For this, we leverage the oscoin network graph $\mathcal{N}$ (Section 3.4), which maps value flows between projects and contributors and assigns a weight $\omega(x)$ to each. This weight or osrank, represents the relative importance of entities within the network.

2.2.2. *Algorithm.* Let t be the total amount of oscoin available for distribution to n projects every epoch κ. The amount t_P distributed to a given project P over κ is a function ψ of its osrank $\omega(P)$, which awards higher ranking projects more oscoin. We describe this *reward function* as

$$\psi(t, \omega(P)) \to t_P.$$

To prevent trivial Sybil attacks, the payment function takes into account a minimum threshold for $\omega(P)$ under which a project does not receive funds. Rewards can be further equalized by adjusting ψ to compress or expand the reward range.

2.2.3. *Vesting.* To align incentives, oscoin received as a reward follows a fixed *vesting schedule*. Practically, this means that full ownership of the rewarded balance is not immediate, but acquired over time. The intention of such a system is to discourage opportunistic behavior and give oscoin holders an interest in the network's long-term future. Once the reward is fully vested, it may be transfered out of the account with no restrictions.

[2]Proof-of-work blocks can be verified cheaply and securely on most mobile devices available today, as demonstrated by Bitcoin. In contrast, proof-of-stake protocols, a common alternative, rely on stakeholder signatures, which require access to account balances for validation. Thus block data must be downloaded and verified or account balances must be queried repeatedly from third-parties, neither of which is feasible in constrained environments.

[3]One promising candidate is the *Proof-of-Storage* family of protocols outlined in Filecoin [3].

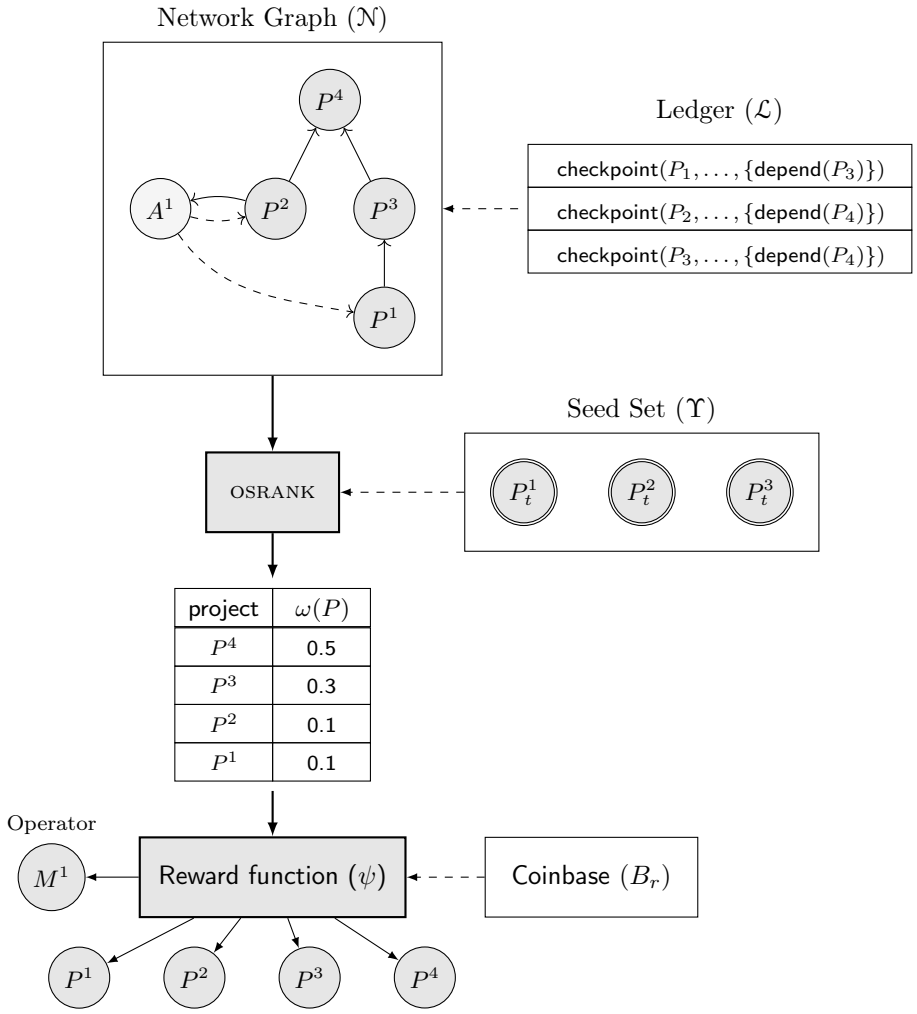

FIGURE 1. The OSCOIN Treasury System

At scale, the OSCOIN treasury has the potential to solve many of the issues with open source monetization and sustainability. Projects can receive income in the form of OSCOIN while licensing their work as free software. This continuous algorithmic funding moves away from current transactional models—which require open source projects to change the way they work, re-license, or setup a business—to a new model, aligned with the motivations that led to the success of the free software movement.[4]

3. OSRANK

Since one of the fundamental mechanisms of OSCOIN is to distribute a portion of the block reward (B_r) to high-value projects on the network, the protocol must reach consensus on which projects are "high value". To choose which projects receive OSCOIN, and in which proportions, the network uses OSRANK, which is a variant of the well-know PageRank algorithm [5], applied to the graph of relationships between projects and contributions on the network. The output of the OSRANK algorithm is a score $\omega(P)$ associated to each project P.

3.1. **Intuition.** While the importance of an open source software package is highly subjective, valuable information about a project's relative importance can be surfaced by studying the structure of open source software through dependencies and contributions.

Similarly to how Google pioneered a large-scale search engine around PageRank that took advantage of the hypertextual structure of the Web, the graph of open source software dependencies and contributions offers an important resource representing trust and relative value in the ecosystem.

3.2. **Rationale.** While it may be clear that value is being produced by a system as a whole, it is often unclear how much value particular subsystems are contributing. Furthermore, subsystems have inherently differing capacities to transform value into revenue: subsystems at the boundaries with other domains are at an advantage, even though they may

[4]See *The Cathedral and the Bazaar*, by Eric S. Raymond

(transitively) derive much of their value from other subsystems. This imbalance in value flows in and out of subsystems is detrimental to the system as a whole.

Thankfully, human activity does not happen in a void[5]: be it knowledge work, content/media creation, financial activity, etc., entities interact with one another in meaningful ways that are highly influenced by the relative value each entity has with respect to the system as a whole. Crucially, during this activity, a trail of *artefacts* (hypermedia links, contracts, transactions, dependencies etc.) is produced; concrete manifestations of the relationships between the various entities.

The basic insight of PageRank and similar metrics, is that by stochastic simulation of meaningful activity traces of system agents, producing trails of artefacts consistent with observations, one can learn about the value flows between subsystems, and hence approximate the underlying value distribution. In the case of PageRank one simulates a human searching for high-relevance data over the Internet, following a chain of links. A sophisticated algorithm would use as many heuristics as possible for choosing the next link to follow (as a real human would), e.g. evaluating the relevance of the link-text. If computational resources are scarce, the crudest algorithm simply picks the next link at random, and this is what produces the classic PageRank formula which has proven so successful on the Web.

3.3. **Notation.** In this section we will use the following conventions: A *graph* refers to a finite directed graph. For such a graph G the vertex set is denoted $V(G)$ and the edge-set is denoted $E(G) \subseteq V^2$. An edge $e = (x, y)$ of G is denoted $x \xrightarrow{e} y$. Given a subset X of the vertices of G, $G[X]$ denotes the *induced subgraph* on X, that is, the graph with X as the set of vertices and as edges the set $\{x \xrightarrow{e} y \mid e \in E(G), \{x, y\} \subseteq X\}$. A *walk* of length n in a graph G is a morphism of graphs $L_n \to G$ where L_n is the n-th linear-graph: $V(L_n) = \{i \in \mathbb{N} \mid i \leq n\}$, $E(L_n) = \{(i, i+1) \in \mathbb{N}^2 \mid i < n\}$.

3.4. **Ranking entities in OSCOIN.** The artefacts related to OSS development which are captured on the OSCOIN ledger are:

- Maintenance of a project,
- Dependencies between projects,
- Signed code contributions to projects.

We summarize these interactions with a weighted directed graph G (Figure 2), the weights corresponding to the relative importance of the edges, set as hyperparameters to the algorithm. The weights on all edges outgoing from a vertex sum to 1.

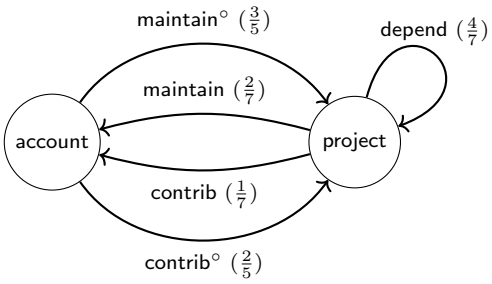

FIGURE 2. Schema of entity relations in open source software development, with example weights.

Note that some of the relationships are bidirectional, but each direction is weighted uniquely. For example both contrib and contrib° pertain to the same artefact: a contribution of code from an account to a project. The edge contrib represents the statement *the contributor is valuable to the project.* The edge contrib° going in the opposite direction represents the statement *the project is valuable to the contributor*; the rationale for this less obvious flow is that developers are unlikely to invest time contributing to projects they do not find useful.

While the graph of dependencies between projects is the most straightforward indication that a project derives value from another, adding edges corresponding to contribution and maintenance serves two purposes:

- The graph of dependencies is directed: it flows from user-facing applications to core libraries. This biases the score greatly towards basic libraries and development tools. Incorporating contributions brings flows in the opposite direction that indicate valued user-facing applications. For example, a fully-featured open source text editor is of high value to open source developers, yet it may never become an explicit dependency of another piece of software. However, developers contributing to this text editor are also likely to contribute to packages upon which the text editor depends, and this allows the algorithm to flow back up from libraries to user-facing packages.
- While the dependency graph alone is unlikely to adequately bridge differing domains, developers often contribute to neighboring ecosystem and to tooling such that these niches can interconnect. Thus a graph incorporating dependencies as well as contributions may be sufficiently connected to reveal the latent value of open source projects.

We propose to record these artefacts (dependencies, contributions, etc.) as transactions in a blockchain, so that one may verifiably construct—and reach consensus on—a finite directed graph $\mathcal{N}$ over G (i.e. in

[5]Space exploration will be addressed in a subsequent paper.

the slice category Graph/G), representing the OSCOIN *network graph*.

In turn, this allows global value scores to be assigned to entities according to a specific stochastic process simulating *random walks*, that is, activity traces in $\mathcal{N}$ with sampling probabilities taken from G. For example: a contributor decides to work on a high value project, contributes a changeset, and in so doing adds a dependency to another project, which imports a bug from an upstream package. The contributor therefore switches to fixing said bug in the dependency, etc. Much like a simulation of a human searching for information by clicking through links, this simulated activity informs us on the value flows between subsystems within the wider OSS ecosystem.

This score, which we call OSRANK, is used to distribute a portion of the block reward to projects as a function of their value to the OSCOIN network.

3.5. **Definition.** To implement OSRANK, we use the following Monte Carlo-based algorithm: for each vertex on the network, R random walks are performed starting at that vertex. To select the next vertex, first an edge type is decided by sampling edges according to the weights in G. The next step is specific to the type of edge:

- In the cases of depend and $\mathsf{maintain}$ edges, all vertices are equiprobable.
- In the cases of edges $x \xrightarrow{\mathsf{maintain}^\circ} p$ and $x \xrightarrow{\mathsf{contrib}^\circ} p$ outgoing from a contributor x, a project p is chosen with probability $\frac{p_x}{N_x}$, where p_x is the number of contributions x has contributed to p, and N_x is the total number of contributions by x.
- In the case of edges $p \xrightarrow{\mathsf{contrib}} x$, the contributors x are chosen with probability $\frac{p_x}{N}$, where N is the total number of contributions p has received from all contributors.

At each step, the walk might terminate with probability $1 - \epsilon_{\mathsf{project}}$ at a project vertex, and $1 - \epsilon_{\mathsf{account}}$ at an account vertex, (here $\epsilon_{\mathsf{project}}, \epsilon_{\mathsf{account}} \in (0, 1)$ are the *damping factors*). This produces a set W of nR random walks, where n is the total number of vertices. The OSRANK for a project x is then:

$$\omega(x) = \frac{W_x \epsilon_{\mathsf{project}}}{nR}$$

where W_x is the number of times all the walks in W visit x, that is:

$$W_x = \sum_{w \in W} |w[\{x\}]|.$$

Similarly, the OSRANK of an account a is:

$$\omega(a) = \frac{W_a \epsilon_{\mathsf{account}}}{nR}.$$

3.6. **Sybil networks.** In order to maintain the integrity of the OSCOIN network graph, illegitimate ecosystems must not be allowed to perpetuate, that is, fraudulent project and contribution structures which have been instantiated for the purpose of exploiting OSRANK must be penalized by the algorithm. Such *Sybil attacks* on PageRank have been examined in detail in previous work [6].

To address this we propose a modification of the PageRank algorithm based on the ideas behind TrustRank [7], which involves two distinct phases:

- In the first phase, all random walks begin at a vertex in the initialization seed set Υ, a subset of all project and account vertices selected for this purpose. The scores obtained are biased towards interactions with the seed set, but they are not used directly by the compensation mechanism. Instead a threshold t is used to determine the legitimacy of entities: any entity falling below this threshold is not considered for the next phase. The output of this phase is a subgraph $\mathcal{N}_t$ of the network graph.
- In the second phase, the algorithm is run on the subgraph $\mathcal{N}_t$ in the same fashion, except without a seed set. The resulting output of the second phase is the OSRANK.

Care must be taken to ensure the seed set is large and well distributed enough such that walks beginning from those vertices will reach all legitimate projects within the network. There are many ways to maintain such a seed set. However, this will be elaborated in future work.

3.7. **Information reliability and OSRANK.** An important factor to consider when implementing OSRANK is the reliablility of information on the ledger, namely dependencies, from which the network graph $\mathcal{N}$ is materialized.

In order for OSRANK to be a good proxy for value in the network, the information on the ledger needs to be accurate. While the proposed algorithm can tolerate a certain error margin, we have to rely on a greater part of the stated "facts" about dependencies and contributions to be correct and up to date. As it stands, there exist intrinsic social incentives for projects and maintainers to keep dependency information up to date – namely, dependency information is public and visible to everyone, and projects have a reputation to withhold towards their dependents. In addition, projects carry the risk of being forked, given the financial incentives associated with OSRANK, and therefore a certain amount of trust from the surrounding ecosystem is important.

In the long term, however, these incentives might not be enough to stop dishonest parties from taking advantage of the system. We briefly discuss potential solutions in §7.

3.8. **Implementation.** There exist several options for ensuring the OSRANK computation does not become prohibitively expensive for network operators to compute.

- *Incremental Monte Carlo.* An incremental algorithm is used so that OSRANK values may be updated as the network graph evolves. Instead of computing OSRANK from scratch, we rely on the fact that only a small percentage of vertices or edges are added or removed from one calculation to the next. Therefore, most of the random walks performed in the previous calculation remain valid in the updated graph. For example if only a single dependency $p_1 \xrightarrow{\mathsf{depend}} p_2$ has been added since the last calculation, then a random walk that was generated on the old graph is only invalidated in the updated graph if it passes through p_1.

  In practice the operators of the network cache the set of random walks from the previous calculation, updating the set by removing invalid walks and replacing them with new ones. Details of such an incremental PageRank algorithm can be found in [8].

  Since all the computations must be deterministic, the walks are not truly random. Rather, they must use a built-in pseudo-random number generator[6] to seed the random walks.
- *Long epoch.* In this scenario payouts according to OSRANK are only made infrequently, by choosing a large value for the reward epoch κ, e.g. monthly. In this case abusers might modify network structures of several projects in anticipation of the payout block. To mitigate this, edges are weighted by their lifetime in the last epoch. For example, if a dependency is added n blocks before the reward block, then it only has a weight proportional to n/κ.

Storing OSRANK on the ledger is fundamental to the OSCOIN "monetary policy", as explained in §2.2. Furthermore, OSRANK and $\mathcal{N}$ will be available to smart contracts (§5) so that projects can script custom behaviors using this data.

4. The Oscoin Ledger

The OSCOIN network is composed of a set of nodes, or *replicas*, which execute a distributed protocol. Together, these nodes form a *Replicated State Machine* with a set of states $\mathcal{S}_0, \ldots, \mathcal{S}_n$, a transition function Ξ, a set of input blocks $B_0, \ldots, B_n$ or "chain" where B_0 is the "genesis block", and a set of outputs.

The OSCOIN ledger, formally $\mathcal{L}$, is the ordered set of all transactions agreed upon by the network. The ledger can be updated by submitting a valid transaction to the network. Transactions operate on the current state $\mathcal{S}$ of the ledger, materialized from all valid transactions.

The transactions available on the ledger are described in this section, as well as how they affect the state $\mathcal{S}$.

4.1. **Supply.** The supply of OSCOIN is subject to an increase $B_r > 0$ carried out by the protocol every epoch, following a fixed supply curve. This ensures a continuous flow of OSCOIN into the treasury (Section 2.2), to reward work in the network.

4.2. **Accounts.** Currency (OSCOIN) is held in *accounts* which can be unlocked by the signature of the account holder. Accounts have addresses which are used to send and receive OSCOIN. This model follows the design of Ethereum [2] rather than Bitcoin's *utxo* model [1].

Accounts are created by transferring OSCOIN to them (Section 4.3), and are removed from the ledger when their balance reaches zero. The set of all accounts is known as $\mathcal{A}$.

The account balance A_b of an account A is held in the state $\mathcal{S}$, and can be accessed with the account address A_a. Formally, $\mathcal{S}(A_a) \to A_b$.

Accounts also have an associated *smart contract* (Section 5), formally A_{contract} which is used to validate and process transfers of OSCOIN to and from the account.

4.3. **Transferring OSCOIN.** Each account in $\mathcal{A}$ is associated with an OSCOIN balance:

$$\mathsf{bal}(a) \in \mathbb{N}_{\geq 1} \mid a \in \mathcal{A}.$$

Transfers to and from accounts are performed with

$$\mathsf{transfer}(x, y, n) \mid x, y \in \mathcal{A}, n \in \mathbb{N}_{\geq 1}$$

where x is the account from which funds are withdrawn, y is the deposit address, and n is the amount of OSCOIN to transfer. The transaction is valid so long as $\mathsf{bal}(x) \geq n$ holds, and the contracts associated to x and y authorize the transfer (Section 5).

4.4. **Projects.** A project P is a tuple:

$$P = \langle P_a, P_h, P_s \rangle$$

where P_a is the project's address and unique identifier, P_h is the project's current hash and P_s is the canonical project source URL.

The project address, an alphanumerical string, is used to uniquely identify the project when transacting on the ledger. Each project has a special account associated with its address called the *project fund*. The project hash is a digest of the project's source code at the time it is entered in the ledger. Finally, the project URL is there for convenience, as a means to fetch the source code. It should be noted that the source code retrieved from P_s must always hash to P_h, otherwise the project is considered invalid.

[6] Designing a suitable random number generator in the permissionless setting is an area of active research. Solutions exist which have been described in recent literature, and are beyond the scope of this paper.

4.4.1. *Registration and maintenance.* Projects must be registered on the ledger before they can participate in the network. This can be done with the

$$\mathsf{register}(P_a, P_s)$$

transaction, signed by a key k_1 which must be used when updating the project at a later time. A transaction is valid so long as the address P_a isn't already in use, and P_s is a valid URL.

On successful execution, the transaction locks a small amount of OSCOIN from the account associated with k_1. This helps to ensure abandoned projects do not clutter the ledger. Once executed, the project is instantiated to $P = \langle P_a, P_h, P_s \rangle$ with $P_h = \varnothing$.

The project P's current *key set* P_K is $\{k_1\}$. We call this set of keys the *maintainers* of the project. These are the keys which effectively "own" the project and have administrative rights over it. P_K can be extended with:

$$\mathsf{addkey}(P_a, k)$$

where k is a valid key not present in P_K. Keys can also be removed, with:

$$\mathsf{removekey}(P_a, k)$$

where k is a key present in P_K. When a project is no longer active, it can be unregistered from the ledger with

$$\mathsf{unregister}(P_a).$$

Execution of this transaction returns the registration deposit to the account associated with the registration key (k_1 in the example above). The above transactions must all be signed by a key present in P_K.

Projects registered on the ledger can be retrieved from the registry $\mathcal{R}$ by using the project address, formally: $\mathcal{R}(P_a) \to P$.

4.4.2. *Checkpointing.* Any project in active development will see its source code change regularly. This means the project hash P_h will quickly become out of sync with the project's latest state and will need to be updated. This is done via the

$$\mathsf{checkpoint}(P_a, P_{h'}, P_{s'}, C^*, D^*)$$

transaction, where $P_{h'}$ is the new project hash, $P_{s'}$ is the URL to retrieve the source code from, C^* is a hash-linked-list of *contributions*, and D^* is a list of *dependency updates*.

Contributions are tuples:

$$\langle C_{\mathsf{prev}}, C_{\mathsf{commit}}, C_{\mathsf{author}}, C_{\mathsf{signoff}} \rangle$$

where:

- C_{prev} is the hash of the previous contribution, or $\varnothing$ if this is the first contribution to the whole project. Note that C^*'s first item must be linked to the last contribution in the project's *previous* checkpoint such that no gaps between contributions exist.
- C_{commit} is the hash of the corresponding commit,
- C_{author} is the author of the contribution,
- C_{sig} is the author's signature of C_{commit}.
- C_{signoff} is the *signoff key*, which must be $\in P_K$.

A contribution must be signed by the signoff key. This signals the contribution has been reviewed and verified by the maintainer. The checkpoint itself must also be signed by a key in P_K.

Because all changes to a project's source code are described in checkpoints, it is possible to reconstruct a full hash-linked list of contributions for the entire project. When cross-referenced with the project's repository, this constitutes a complete historical record of who authored what code, and which maintainer signed off on the contribution. This ensures the project history is auditable and tamperproof, while providing fundamental information to OSRANK. Note that only contribution *metadata* is stored on-chain.

Conceptually, a project P depends on another project P' if it is an "input" to P in some way: P references P' or parts of P' in its source code, or P' is a build/test dependency. For example, if a project used the purely functional package manager *Nix* [9], then the dependencies declared on OSCOIN should map one-to-one with the Nix dependencies.

The dependency update list D^* is a list of *dependency updates*, which are one of

$$\mathsf{depend}(P'_a, n) \quad \text{or} \quad \mathsf{undepend}(P'_a, n)$$

which refer to the nth checkpoint of a project P' (0-indexed from the first checkpoint). The depend update adds a new dependency while the undepend update removes a dependency. The updates are processed in order with depend only being valid if it adds a dependency that the project does not already have and undepend only being valid for current dependencies. The checkpoint is invalid if the update list contains duplicates.

As a project maintainer, adding a dependency signals a variety of things depending of the nature of the project:

- They have verified that P indeed depends on this specific version of P'.
- That P' is suitable as a dependency for P, e.g. if P has very high security requirements, that P' fulfills these.

Since contributions to a project carry additional weight—potentially increasing a project's OSRANK—there is an incentive for maintainers to checkpoint their projects regularly. Similarly, adding dependencies may increase connectivity in the network graph, which may in turn indirectly improve a project's OSRANK.

4.5. **Smart contracts.** Finally, the ledger enables certain transactions to be mediated by smart contracts. We dedicate the following section to this topic, showing how projects can automate fund distribution as well as specify access control policies.

5. Smart Contracts

In order to ensure that funds distributed to projects by the oscoin treasury and those received as donations are spent responsibly and transparently, the oscoin protocol sends them to a special type of account: the *project fund*, which is administered by an updatable *smart contract*. In addition, the protocol requires that transfers from the project fund only be issued from within the fund's smart contract. When a contributor decides to contribute code to a project, they may inspect the contract currently in place, and under what conditions it may be updated, to understand the project's remuneration model, and whether or not they would be rewarded for their work.

5.1. **Definition.** A smart contract is a set of functions, or *handlers*, invoked as part of regular transaction processing. Certain transactions, such as transfer make use of smart contracts to extend their behavior and allow projects to configure and automate aspects of their finances, ownership model, or governance.

5.2. **Receiving oscoin.** When a project receives oscoin, the smart contract associated with that project's fund is invoked. The specific handler called in the smart contract depends on the *sender*: for oscoin received from the treasury system, the receiveReward handler is called, while for transfers received from any other source, the receiveTransfer handler is called. This allows projects to handle funds received by the treasury differently than funds received as donations.

5.2.1. *Rewards from the treasury.* The receiveReward handler is invoked every κ blocks for projects getting a reward. The function is called with three arguments: p is a dictionary containing the project data, $r \in \mathbb{N}_{\geq 1}$ represents the amount of oscoin being awarded to the project this epoch and $k \in \mathbb{N}$ is the current epoch. The function must return a *distribution*, a set of tuples assigning an amount of oscoin to a set of accounts. The function is valid as long as it doesn't distribute more than r oscoin in total. Any oscoin that isn't distributed is effectively *burned*.

When a project is first registered, receiveReward returns the empty distribution, that is, the entirety of the reward r is burned:

handler receiveReward(p, r, k)
  return $\varnothing$

Once the project maintainers have decided on a policy, they may issue a transaction to update this handler. For example, they may decide that all contributors should get an equal share of future rewards, with the remainder deposited into the project fund:

handler receiveReward(p, r, k)
  $n \leftarrow |p.\text{contributors}|$
  $q \leftarrow r \mathbin{/\!\!/} n$
  $m \leftarrow r \mod n$
  $x \leftarrow \{(c.\text{addr}, q) \mid c \leftarrow p.\text{contributors}\}$
  return $x \cup \{(p.\text{fund}, m)\}$

This makes it very easy for a potential contributor to see whether or not they would be rewarded for their contributions to a project, and in what proportion.

5.2.2. *Transfers and donations.* When a transfer of oscoin is received from a source other than the treasury, the receiveTransfer handler is invoked. This function takes the same three arguments as receiveReward, in addition to a fourth argument s which represents the sender or "source" account of the transfer. The function is expected to return a distribution, just like receiveReward. Initially, this handler is set to simply transfer the balance to the project fund:

handler receiveTransfer(p, r, k, s)
  return $\{(p.\text{fund}, r)\}$

Like the receiveReward example, it is possible to include arbitrary logic in this handler. For example, since donations to the project would invoke this handler, a project might want to distribute a percentage of each donation to its contributors, and the rest to its maintainers.

5.3. **Contract handler updates.** Updates to the project smart contracts are issued using the

$$\mathsf{updatecontract}(P_a, h, c, \nu, v)$$

transaction, which must be signed by a project maintainer, where h is the *handler* to be updated, c is the code for the new handler, v is a set of *votes*: a collection of signatures the maintainer has collected for the update, and ν is a *nonce* to prevent voting replay attacks.

Since updating a contract may affect existing contributors by, for example, changing the share of rewards they receive for their work; the rules according to which handlers may be updated are also stored in the project contract, as the updateContract handler. The conditions under which an updatecontract transaction is considered valid are specified in this handler.

updateContract is a function which takes three arguments: p, the project data, h, the name of the handler being updated, and v, the list of public keys which signed votes for the contract update transaction. The function must return a boolean value, stating whether the update is valid ($\top$) or not ($\bot$). The default code for this handler is:

handler updateContract(p, h, v)
  return $\{o.\text{addr} \mid o \leftarrow p.\text{maintainers}\} \subseteq v$

which specifies that all the current maintainers must sign any contract update.

In order to specify that contract updates must be accepted by a majority of all *contributors*, this handler could be updated to:

handler updateContract(p, h, v)
  $h \leftarrow |p.\text{contributors}| \mathbin{/\!\!/} 2$

$v' \leftarrow \{c.\text{addr} \mid c \leftarrow p.\text{contributors}\} \cap v$
return $|v'| > 1 + h$

If the handler returns $\top$, the contract update transaction is considered valid, and the handler h specified in the transaction is updated with the new code c.

5.4. **Ad-hoc spending.** While most distribution of OSCOIN can be automated by the RECEIVEREWARD and RECEIVETRANSFER handlers, it will also be necessary to occasionally transfer OSCOIN manually from the project fund to another account. For example, in a scenario where a project receives a large donation, the maintainers may want to hold some funds in reserve for future organizational expenditures.

To spend project funds in an *ad-hoc* manner, the transfer transaction is used with an additional set of parameters:

$$\mathsf{transfer}(P_a, a, n, \nu, v) \mid n \in \mathbb{N}_{\geq 1},$$

where P_a is the project fund account to spend from, a is the account to transfer the funds to, n is the amount of OSCOIN, and the remaining two parameters are a set of votes.

The transaction's validity is determined by the SENDTRANSFER handler, which works in a similar way to UPDATECONTRACT. For example, the function might specify that the project maintainers may spend a small amount x of OSCOIN per month without seeking community agreement, while large sums require over half of the contributors and donors to sign off:

handler SENDTRANSFER(p, n, a, v)
  if spentThisEpoch$(p) + n > x$ **then**
    $h \leftarrow |p.\text{contributors} \cup p.\text{donors}| \mathbin{/\!\!/} 2$
    $c' \leftarrow \{c.\text{addr} \mid c \leftarrow p.\text{contributors}\}$
    $d' \leftarrow \{d.\text{addr} \mid d \leftarrow p.\text{donors}\}$
    $v' \leftarrow v \cap c' \cap d'$
    return $|v'| > 1 + h$
  else
    return $\top$

Leveraged the right way, smart contracts are a powerful tool the community has, which can:

(1) Provide transparency and trust within projects, by making reward distribution explicit and automatic.
(2) Allow potential contributors to easily assess a project before contributing.
(3) Allow projects and their flow of OSCOIN to be transfered over to a new maintainer with low risk.
(4) Offer a certain stability to existing members and collaborators, by making changes to contracts difficult without community buy-in.

6. APPLICATIONS

OSCOIN as a platform has the potential to enable novel applications for software communities. In comparison to a general purpose virtual machine like Ethereum, OSCOIN provides primitives designed around open source software collaboration workflows and enables developers to combine them in flexible ways.

While it is difficult to envision all the potential ways OSCOIN might be used, we believe the following categories of applications can benefit from OSCOIN as a platform.

6.1. **Governance and collective decision making.** We believe developers could leverage OSCOIN's ledger for collective decision making through the use of smart contracts. These contracts can take advantage of the network graph in order to align interests between all project participants and incentivize further participation. In addition, smart contracts can be used within organizations to coordinate around other contentious decisions, using a diverse set of decision making tools.

6.2. **Trust minimization for software development processes.** With contribution histories on chain, new software development processes that minimize trust between network participants can emerge.

Rather than relying on the weak trust model of existing centralized hosting providers, OSCOIN can be used to create powerful continuous integration pipelines that use cryptography to ensure that every commit can be trusted.

This category of applications is particularly relevant to high-assurance software such aerospace applications, biomedical firmware, or cryptocurrencies, including OSCOIN.

6.3. **Incentivization.** The third category of applications we foresee is the one related with developer incentives. This might include applications that expose paid work within the OSCOIN network, describe and enforce service level agreements between projects and their dependencies, or completely re-imagine crowdfunding applications such as Patreon, taking advantage of digital money and collectibles.

6.4. **DAOs.** Finally, combining all of the above with the OSCOIN treasury (§2.2), the open source community will have the ability to create decentralized autonomous organizations (DAOs) that receive continuous funding in OSCOIN. These funds could be allocated to both internal and external network participants for their contributions to the project, while facilitating direct engagement in decision making processes that are entirely transparent, in the true spirit of the free and open source software movement.

7. Future Work

In our attempt to formulate a clear and concise vision for the oscoin protocol, we deliberately left certain questions unanswered. We will briefly state these here and attempt to address them in the context of future work:

- As posed in Section 3, the question of information reliability is crucial. We are actively researching oracle-based solutions to this problem which could provide economic incentives for users in the network to flag suspicious projects, as well as further disincentive projects to cheat, by allowing the protocol to slash their unvested (§2.2.3) reward balance when dishonest behavior is detected.
- In Section 3 we propose a two-phase osrank algorithm for ranking projects, where the first phase uses a seed set Υ of vertices, aimed at addressing the issue around Sybil attacks. However, the question of *how* this set is chosen and updated remains an open research question.
- As discussed in Section 2, we are committed to exploring alternatives to proof-of-work for securing the chain.
- A large part of the work on osrank is experimentation and tuning of the various parameters to the algorithm, e.g. edge weights, damping factor, thresholds etc. that are ongoing.
- As it stands, osrank is a great indicator of relative value for software libraries, but doesn't provide as meaningful a metric for user-facing applications and software used exclusively in proprietary settings. While the latter class of software projects have more options when it comes to monetization, we are looking at ways in which osrank could evolve to serve them, and cover a greater range of free software projects.

Acknowledgements

We thank Sam Hart for his clear thinking, feedback and contributions to some of the core ideas presented in this work. We thank Aaron Levin for his contributions and great discussions early on in the process of conceiving oscoin. Finally, we thank the team at Monadic, as well as our peers for their support and feedback which has been invaluable in finalizing this work.

References

[1] Nakamoto, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System. May 2009

[2] Wood, Gavin. Ethereum: A Secure Decentralised Generalised Transaction Ledger. December 2018

[3] Protocol Labs. Filecoin: A Decentralized Storage Network. July 2017

[4] Eghbal, Nadia. Roads and Bridges. The Unseen labor behind our digital infrastructure. July 2016.

[5] Brin, S.; Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine (PDF). *Computer Networks and ISDN Systems.* 30: 107–117.

[6] Cheng, A. and Friedman, E. 2006. Manipulability of PageRank under Sybil strategies. In First Workshop on the Economics of Networked Systems (NetEcon06).

[7] Z. Gyöngyi, H. Garcia-Molina, J. Pedersen: Combating Web Spam with TrustRank

[8] Bahmani, Bahman and Chowdhury, Abdur and Goel, Ashish. Fast Incremental and Personalized PageRank. Proc. VLDB Endow. December 2010.

[9] Dolstra, E., de Jonge, M. and Visser, E. Nix: A Safe and Policy-Free System for Software Deployment. In Damon, L. (Ed.), 18th Large Installation System Administration Conference (LISA '04), pages 79–92, Atlanta, Georgia, USA. USENIX, November 2004.