

Proof of Contract Stake Formal Specification

Joby J Reuben
Auguth Research Foundation
Bangalore, India
joby@auguth.org

March 13, 2025

Abstract

PoCS ties stake score (S_c) to smart contract execution, ensuring security through verifiable computation. Unlike token-based staking, PoCS rewards actively executed contracts using reputation score (R_c) preventing artificial inflation. Delegation allows contracts to assign validation rights based on reputation, ensuring Sybil resistance. By aligning computation with staking, PoCS strengthens economic security and incentivizes long-term execution. This research and reference-implementation development is funded by the Web3 Foundation Grants Program, Switzerland.

1 Summary

1. **Contracts as Staking Bonds** The Proof of Contract Stake (PoCS) mechanism formalizes smart contracts as staking bonds, denoted as B_c , where a contract C can be delegated or nominated to a validator V for block production. The bond value B_c is a function of the contract's execution history which we denote as stake score S_c , ensuring a quantitative commitment to the network's security and consensus mechanism. Formally,

$$B_c = S_c \tag{1}$$

where S_c is computed based on contract-specific parameters, including execution history and network contribution.

2. **Stake Score (S_c)** The stake score S_c is a dynamic metric representing a contract's contribution to network security. It updates as:

$$S_c^{(t+1)} = \begin{cases} (S_c^{(t)} \cdot R_c^{(t+1)}) + G_c^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_c^{(t)} + G_c^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \tag{2}$$

where:

- $S_c^{(t)}$ is the contract's stake score at time t .
- $R_c^{(t+1)}$ is the updated reputation score, incrementing once per block.
- $G_c^{(t+1)}$ is the gas consumption for execution at $t + 1$.
- $b(t)$ denotes the block index at time t .

A higher S_c results in a larger bond value B_c , affecting validator nomination and reward allocation.

3. **Reputation Factor (R_c)** The reputation score R_c increases only once per block when the contract is executed. It updates as:

$$R_c^{(t+1)} = \begin{cases} R_c^{(t)} + \Delta R_c, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ R_c^{(t)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \quad (3)$$

where:

- $R_c^{(t)}$ is the reputation score at time t .
- ΔR_c is a protocol-defined increment applied only once per block, which is generally a constant or consensus decided integer.
- $b(t)$ denotes the block index at time t .

Since reputation does not increase for multiple executions within the same block, contracts benefit from sustained execution over multiple blocks. A higher R_c enhances the stake score S_c and strengthens a contract's eligibility for validator nomination i.e., delegation and reward distribution.

4. **Minimum Requirements** PoCS enforces minimum participation criteria for both contracts and validators to ensure network stability and security. These requirements include:

- (a) *Minimum Reputation Requirement:* A contract C must satisfy the reputation threshold:

$$R_c \geq R_{\min} \quad (4)$$

where R_{\min} is a network-defined constant, either set by governance consensus or a fixed protocol parameter. Contracts failing to meet this threshold are ineligible for nomination or delegation.

- (b) *Minimum Delegation Requirement for Validators:* A validator V can only begin validation if it has received stake from at least N_{\min} distinct contracts:

$$N_c \geq N_{\min} \quad (5)$$

where N_c represents the number of unique delegated contracts, and N_{\min} is a predefined network constant. This constraint ensures that validators operate with sufficient decentralized support.

These conditions guarantee fair participation, prevent centralization risks, and uphold the integrity of the PoCS mechanism.

For a comprehensive formal analysis of PoCS, refer to its Research Model documentation. This document strictly defines the reference implementation structure along with its computational invariants, ensuring adherence to protocol constraints and correctness conditions.

2 Instantiation

The process of *contract instantiation* varies across different blockchain architectures, depending on their execution models and state management mechanisms. In some implementations, deployment involves uploading a *binary file* \mathcal{B} , which can be of some majorly used choices such as the Ethereum Virtual Machine (EVM) bytecode or WebAssembly (WASM) binary. The latter is often preferred due to its encapsulated runtime model, providing near native performance and security guarantees.

Upon deployment, a contract is assigned a *unique contract address* A_c , which is typically computed as:

$$A_c = H(N_d, H(\mathcal{B})) \quad (6)$$

where $H(\cdot)$ denotes a cryptographic hash function, and N_d represents the deployer's nonce. Alternative instantiation and contract address derivation models exists to cater contract migration, code upgrades, and state inheritance. Some blockchain architectures allow contracts to store only unique storage slots, dynamically fetching execution logic from a *code hash* $H(\mathcal{B})$ rather than embedding a unique-bytecode directly.

However, within the *Proof of Contract Stake (PoCS)* mechanism, staking computations are strictly associated with *contract addresses* A_c , rather than the underlying code hash. This ensures that stake accumulation and delegation remain bound to individual contract-runtimes rather than their instruction logic.

Upon contract instantiation at block height b_t , two essential mappings are initialized:

- *DelegateInfo* $\mathcal{H}_D : K \rightarrow \mathcal{D}$
- *StakeInfo* $\mathcal{H}_\Sigma : K \rightarrow \Sigma$

where K represents the unique contract identifier. The mappings store delegation and staking-related metadata separately to ensure efficient state management.

2.1 Structures

In addition to the pre-existing on-chain state variables stored within a Contract Account, two supplementary data structures, namely **DelegateInfo** and **StakeInfo**, are introduced. These structures encapsulate essential parameters governing stake delegation, validator nomination, and stake aging mechanisms.

2.1.1 DelegateInfo

The **DelegateInfo** structure, denoted as \mathcal{D} , encapsulates delegation-related metadata for a contract account. It is formally represented as an ordered triplet:

$$\mathcal{D} = (\mathcal{D}_\lambda, \mathcal{D}_\nu, \mathcal{D}_\tau) \quad (7)$$

where:

- $\mathcal{D}_\lambda = \text{owner}$: The unique identifier of the contract deployer, responsible for authorization of delegation updates.
- $\mathcal{D}_\nu = \text{delegateTo}$: The validator controller contract address to which the contract is currently staked.
- $\mathcal{D}_\tau = \text{delegateAt}$: The block height $b(t)$ at which the last delegation update occurred, essential for stake age computation.

Upon contract instantiation at initial block height b_t , \mathcal{D} is initialized with **default values**:

$$\mathcal{D} \leftarrow (\mathcal{D}_\lambda, \mathcal{D}_\nu, \mathcal{D}_\tau) = (\text{deployerID}, \text{deployerID}, b_t) \quad (8)$$

Initially, the delegation target \mathcal{D}_ν is set to the deployer's account ID, ensuring that delegation can only be reassigned once the contract accrues sufficient reputation, satisfying:

$$R_c^{(t)} \geq R_{\min} \quad (9)$$

where:

- $R_c^{(t)}$ is the reputation score of the contract at time t .
- R_{\min} is the protocol-defined threshold required for delegation eligibility.

Since block height $b(t)$ is monotonically increasing, delegation updates are constrained by the progression of the blockchain, preventing rapid or arbitrary delegation switching. This mechanism ensures that only reputable contracts participate in validator nomination, reinforcing network security and stability.

2.1.2 StakeInfo

The structure **StakeInfo**, denoted as Σ , encapsulates scarcity-related parameters i.e., execution history as a computed single unit of a staked contract. It is formally defined as an ordered triplet:

$$\Sigma = (\Sigma_\rho, \Sigma_\beta, \Sigma_s) \quad (10)$$

where:

- $\Sigma_\rho = \text{reputation}$: The contract's reputation score $R_c^{(t)}$, derived from recurrent contract invocations over consecutive blocks.

- $\Sigma_\beta = \text{recentBlockHeight}$: The most recent block height $b(t)$ in which the contract was invoked via an extrinsic. Utilized for reputation increment conditionals.
- $\Sigma_s = \text{stake_score}$: The stake score $S_c^{(t)}$ representing the scarcity value of the contract.

Analogous to \mathcal{D} , the structure Σ is initialized with default values during contract instantiation. These values can only be modified externally via an invocation of the function `delegate()`. Assuming an ordered indexing scheme, the **default values** are:

$$\Sigma_i \leftarrow (\Delta R_c, b_t, S_{\text{init}}) \quad (11)$$

where:

- ΔR_c is a protocol-defined increment applied only once per block, which is generally a constant or a consensus-decided integer.
- b_t represents the block height at contract instantiation.
- S_{init} is the protocol-defined initial stake score, a consensus-decided constant.

Setting the initial stake score S_{init} as a network-decided constant provides flexibility in adjusting staking parameters based on network conditions. This ensures an optimal balance between security, decentralization, and economic stability. The network can dynamically adjust S_{init} in response to staking inflation, validator performance, or governance policies. For instance, a congested blockchain might decrease S_{init} to limit unnecessary contract deployments, while a growing ecosystem may increase it to incentivize adoption.

2.2 Considerations

The structures `DelegateInfo` and `StakeInfo` need not be embedded within the primary contract storage but can instead be modeled as external state mappings, denoted as $\mathcal{H}_{\mathcal{D}}$ and \mathcal{H}_{Σ} , respectively. Formally, these mappings are defined as:

$$\mathcal{H}_{\mathcal{D}} : K \rightarrow \mathcal{D}, \quad \mathcal{H}_{\Sigma} : K \rightarrow \Sigma \quad (12)$$

where K represents the unique contract identifier. These mappings enable dynamic queries for delegation and stake-related operations without imposing rigid storage constraints within the contract itself.

The selection of an appropriate storage paradigm depends on the blockchain execution environment—whether a monolithic system like Geth (Ethereum) or a modular blockchain framework such as Substrate. External state mappings $\mathcal{H}_{\mathcal{D}}$ and \mathcal{H}_{Σ} ensure backward compatibility and prevent hard forks by decoupling delegation and stake-related metadata from the contract’s core storage.

This design facilitates seamless upgrades without requiring modifications to the contract’s fundamental data structures.

The lookup operations in $\mathcal{H}_{\mathcal{D}}$ and \mathcal{H}_{Σ} depend on the underlying storage structure. In standard blockchain implementations, hash maps typically exhibit an average-case complexity of $O(1)$ for lookups and updates, assuming an ideal hash function with minimal collisions. However, in the worst case (e.g., due to hash collisions or poor distribution), performance may degrade to $O(n)$.

In contrast, if delegation and stake information were stored within additional contract-bound structs, accessing them would require traversing storage slots or executing nested calls, often resulting in polynomial-time retrieval complexities, denoted as $O(p(n))$, where $p(n)$ represents the degree of polynomial growth based on the number of stored contract instances. This is particularly inefficient in environments with large-scale contract participation, as the retrieval time increases significantly with network size.

By leveraging external hash maps, the proposed model optimally balances storage efficiency and retrieval time. While direct struct-based storage simplifies contract design, it imposes higher computational costs during execution with tightly coupled changes. In comparison, using $\mathcal{H}_{\mathcal{D}}$ and \mathcal{H}_{Σ} allows efficient indexing while avoiding unnecessary state bloat, ensuring scalable stake management and delegation processes.

3 Invocations

Every contract invocation dynamically updates its stake score S_c in accordance with the Proof of Contract Stake (PoCS) framework. The update mechanism influences validator selection, security commitments, and staking bond formation.

PoCS formalizes smart contracts as staking bonds, denoted as B_c or explicitly known as stake score S_c , where a contract C is delegated or nominated to a validator V for block production. The stake score dynamically evolves based on execution behavior, ensuring a quantitative metric for network security participation.

The stake score S_c follows a *block-dependent update rule*, expressed as:

$$S_c^{(t+1)} = \begin{cases} (S_c^{(t)} \cdot R_c^{(t+1)}) + G_c^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_c^{(t)} + G_c^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \quad (13)$$

where:

- $S_c^{(t)}$ is the contract’s stake score at time t .
- $R_c^{(t+1)}$ is the currently available *reputation score*.
- $G_c^{(t+1)}$ is the *gas consumption* of execution at $t + 1$.
- $b(t)$ is the block index at time t .

3.1 Stake Score

In PoCS, the *stake score* can either be recorded as an independent state variable or be directly coupled with a existing *staking module* to optimize storage efficiency and minimize computational overhead. The latter reduces redundant storage overhead, as each stake score update would otherwise require separate state storage operations.

The typical *bit-width* requirements for on-chain storage are:

- *Stake Score*: 64 to 128 bits, based on execution volume i.e., computational units.
- *Reputation Score*: Fixed at 32 bits.

The PoCS framework necessitates a distinct staking module, as traditional cryptocurrency-focused staking mechanisms fail to accommodate its *non-fungible* and *non-transferable* stake score properties. Unlike native token-based staking, where staking assets are fungible and transferrable:

$$B_u \neq B_v \quad \forall u, v \in C, \quad B_c \notin \mathbb{F} \quad (14)$$

This condition states that the staking bond B_u of a contract C_u is distinct from the staking bond B_v of another contract C_v . That is, for any two contracts $u, v \in C$, their respective staking bonds are unique and non-interchangeable. Unlike traditional token-based staking mechanisms, where participants stake identical fungible tokens, PoCS enforces a one-to-one mapping between contracts and their staking bonds.

$$\forall C_i, C_j \in C, \quad S_{C_i} \neq S_{C_j} \text{ unless identical execution history} \quad (15)$$

This condition states that the stake score S_c is uniquely tied to each contract C . Two different contracts C_i and C_j will generally have different stake scores, even if they perform similar functions. The only exception is when two contracts have an identical execution history—meaning they have been invoked in precisely the same manner, with the same gas consumption, over the same sequence of blocks.

Even if contracts C_i and C_j possess similar execution patterns, their respective reputations R_{C_i} and R_{C_j} may differ due to variations in their historical interactions, delegation status, or accumulated validation contributions. Furthermore, since computed stake score types exhibit high entropy (e.g., 128-bits), the probability of two distinct contracts converging to the exact same stake score remains negligibly small, ensuring uniqueness and preventing fraudulent replication.

$$B_c \not\rightarrow B_{c'} \quad (16)$$

This expression asserts that a contract's staking bond B_c is **non-transferable** to another contract $B_{c'}$. Unlike token-based staking, where a validator can move

staked tokens freely, PoCS enforces **strict contract-bound staking**—the stake score earned by a contract remains permanently attached to it.

Since reputation R_c increments at most once per block, the upper bound on the stake score’s growth over n blocks is given by:

$$S_c^{(t+n)} \leq \left[\left(S_c^{(t)} \prod_{k=1}^n R_c^{(t+k)} \right) + \sum_{k=1}^n G_c^{(t+k)} \right] \quad (17)$$

where:

- $S_c^{(t+n)}$ is the stake score after n blocks.
- $S_c^{(t)}$ is the initial stake score at time t .
- $R_c^{(t+k)}$ is the contract’s **reputation** at block $t + k$.
- $G_c^{(t+k)}$ is the **gas consumption** at block $t + k$, contributing directly to stake growth.

The historical stake score $S_c(t)$ evolves dynamically, influenced by two primary factors: *reputation scaling* and *gas expenditure*. Reputation scaling introduces a *multiplicative effect*, where the stake score is progressively scaled by the contract’s reputation coefficient R_c , ensuring that long-term reliable execution strengthens its staking power. Concurrently, *gas expenditure contributes additively*, meaning that each contract invocation increases the stake score by an amount proportional to the gas consumed during execution. This dual mechanism ensures that both sustained activity and computational resource usage play a crucial role in determining the contract’s staking influence.

3.2 Reputation

The reputation factor R_c is a dynamic metric that quantifies the historical execution frequency of a contract. It plays a crucial role in determining a contract’s *stake score* S_c and its eligibility for validator nomination, delegation, and reward distribution in the Proof of Contract Stake (PoCS) model.

The reputation score R_c follows a strictly *block-based increment rule*, ensuring that a contract’s activity is rewarded based on sustained execution across multiple blocks rather than repeated calls within the same block. Formally, its evolution is governed by:

$$R_c^{(t+1)} = \begin{cases} R_c^{(t)} + \Delta R_c, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ R_c^{(t)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \quad (18)$$

where:

- $R_c^{(t)}$ is the contract’s reputation score at time t .
- ΔR_c is a *protocol-defined increment* applied at most once per block. This increment is generally a *constant* or a consensus-decided integer, ensuring controlled growth.

- $b(t)$ denotes the block index at time t .

Some Implications of Reputation Constraints involve:

1. *Prevention of Exploitative Execution*: Since R_c does not increase for multiple invocations within the same block, contracts cannot artificially inflate their reputation by self-invoking or being spammed with calls within a single block.
2. *Long-Term Execution Incentive*: Contracts benefit from consistent execution over multiple blocks, fostering sustained utility rather than short-term burst execution. A higher R_c correlates with higher stake score (S_c) growth, making the contract a stronger candidate for validator delegation.
3. *Influence on Stake Score Computation*: The stake score S_c is dependent on the reputation score, particularly in cases where contract execution occurs in non-consecutive blocks:

$$S_c^{(t+1)} = \begin{cases} S_c^{(t)} \cdot R_c^{(t+1)} + G_c^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_c^{(t)} + G_c^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \quad (19)$$

where $G_c^{(t)}$ represents the gas expenditure of the contract execution. This equation ensures that contracts executing persistently over time gain higher stake scores, reinforcing their importance in the network's security and consensus mechanisms.

4. *Decentralized Block Production and Spam Resistance*: Block production is handled by different validators at varying intervals, each with discretion over transaction inclusion. Since validators operate independently, it is computationally infeasible to coordinate 2/3 of the validators for spam attacks. This decentralized decision-making process makes it significantly harder to exploit stake score or reputation computations through coordinated spam transactions.

The reputation system in PoCS adheres to the following invariants:

- *Monotonicity*: Reputation never decreases, ensuring that once a contract builds execution history, it retains its standing.

$$R_c^{(t+1)} \geq R_c^{(t)} \quad \forall t \quad (20)$$

- *Upper Bound on Reputation Growth*: Given that R_c increments at most once per block, the maximum possible reputation at time $t + n$ is:

$$R_c^{(t+n)} \leq R_c^{(t)} + n \cdot \Delta R_c \quad (21)$$

This ensures that contracts cannot grow their reputation arbitrarily within short timeframes, where each individual unit of time in the chain corresponds to the block time.

- *Block-Dependence*: The function $f : T \rightarrow R_c$, mapping time t to reputation score, is discrete and block-indexed, meaning that:

$$\forall t_1, t_2 \text{ such that } b(t_1) = b(t_2), \quad R_c^{(t_1)} = R_c^{(t_2)} \quad (22)$$

ensuring no intra-block variations in reputation, where:

- t_1, t_2 are distinct timestamps occurring within the same block.
- $b(t)$ represents the block index corresponding to time t .
- If t_1 and t_2 belong to the same block, then the reputation score remains unchanged:

4 Stack-Frame

In a blockchain execution model, smart contracts serve as isolated computational units, executing within an instance of a *Virtual Machine (VM)* environment. Formally, each contract C consists of a set of externally invokable functions:

$$C = \{f_1, f_2, \dots, f_n\}, \quad f_i : I \rightarrow O \quad (23)$$

where f_i denotes an invokable function mapping an input set I to an output set O . While contract execution is self-contained, modern blockchains support *cross-contract execution*, enabling a contract C_i to invoke another contract C_j through a *delegate call* mechanism.

When a contract C invokes another contract C' , a new *stack frame* \mathcal{F} is created. An execution stack may consist of nested frames. Formally we assume a frame \mathcal{F} to be:

$$\mathcal{F} = (\mathcal{C}, \mathcal{T}, \mathcal{G}) \quad (24)$$

where:

- $\mathcal{C} = (C_{\text{caller}}, C_{\text{callee}})$ represents the caller-callee contract pair.
- \mathcal{T} denotes the execution target function within C_{callee} along with its arguments.
- \mathcal{G} represents the gas allocated for execution within this stack frame.

A sequence of nested contract calls constructs a *LIFO (Last-In, First-Out)* execution stack, where given nested contract calls, the execution *may follow*:

$$\mathcal{F}_0 \rightarrow \mathcal{F}_1 \rightarrow \mathcal{F}_2 \rightarrow \dots \rightarrow \mathcal{F}_k \quad (25)$$

where \mathcal{F}_0 is the *transaction originator* (typically an Externally Owned Account (EOA)), and \mathcal{F}_k is the deepest execution frame. Each frame resolves when its top frame returns the execution result, as blockchains are single-threaded to

ensure deterministic state transition and consistent distribution across nodes, unless operating under a sharded execution model.

Each stack frame consumes a gas amount $G_{\mathcal{F}_i}$, and the total execution gas for a transaction is:

$$G_{\text{tx}} = \sum_{i=0}^k G_{\mathcal{F}_i} \quad (26)$$

Since PoCS defines *stake score* S_c as a function of *gas expenditure*, we formally establish, for each individual stack frame \mathcal{F}_j :

$$S_{C_j}^{(t+1)} = \begin{cases} (S_{C_j}^{(t)} \cdot R_{C_j}^{(t+1)}) + G_{\mathcal{F}_j}^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_{C_j}^{(t)} + G_{\mathcal{F}_j}^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \quad (27)$$

where:

- $S_{C_j}^{(t)}$ is the stake score of contract C_j at time t .
- $R_{C_j}^{(t+1)}$ is the reputation score, which increments at most once per block.
- $G_{\mathcal{F}_j}^{(t+1)}$ is the gas spent by the contract in execution.
- $b(t)$ represents the block index at time t .
- C_j is the callee contract, which accumulates the stake score based on its gas expenditure per execution frame.

Regardless of whether the caller is an *EOA* or another contract, the stake score is bound to the *callee* C_j , ensuring that execution costs reflect computational contributions.

For PoCS correctness, the system must ensure:

$$\sum_{i=0}^k G_{\mathcal{F}_i} = G_{\text{tx}} \quad (28)$$

i.e., the total gas allocated across all stack frames must match the transaction-level gas usage, preventing incorrect stake score accumulation.

This ensures fair stake score attribution, maintaining consistency in PoCS computations across nested contract executions.

4.1 Delegate Instantiation

Contract deployment is analogous to asymmetric-pair signing, where an execution binary or bytecode is uploaded and instantiated on-chain. However, beyond externally owned accounts (EOAs) deploying contracts, it is also possible for a contract to deploy another contract.

In such cases, the instantiating contract, rather than the original transaction signer, assumes the role of the *responsible caller*. This mirrors stack-frame execution semantics, where the caller initiates execution but does not retain responsibility for subsequent computations.

During contract deployment within a transaction signed by an externally owned account E , if a contract C_{caller} deploys another contract C_{new} , the ownership and execution responsibility shift to:

$$\text{owner}(C_{\text{new}}) = C_{\text{caller}} \quad (29)$$

Thus, C_{caller} becomes the effective **responsible caller** in the current execution frame. The transaction signer S may have initiated the process, but the on-chain instantiation logic ensures that ownership adheres to the deploying contract. Upon deployment, PoCS ensures that the **DelegateInfo** structure of C_{new} is initialized to its default values as:

$$\mathcal{D}(C_{\text{new}}) \leftarrow (C_{\text{caller}}, C_{\text{caller}}, b_t) \quad (30)$$

4.2 Rollbacks

Todo (PoCS v0.2) - Out of Gas, Partial Executions , etc

5 Delegation

Delegation in PoCS is the process of assigning a contract to a validator address. A validator in this system may either be an externally owned account (EOA) or a smart contract. In the case of a smart contract validator, it manages its delegated contracts trustlessly, handling reward distribution and governance autonomously.

Let C be a contract seeking delegation and V be a validator.

$$\mathcal{D}_\nu(C) \rightarrow V \quad (31)$$

where C satisfies a set of eligibility conditions, primarily the *minimum reputation* requirement.

Each contract C maintains a reputation score R_C , which is an incremental function based on recurrent calls in consecutive blocks, as defined in previous sections. For a contract C to delegate to a validator V , it must satisfy:

$$R_C \geq R_{\min} \quad (32)$$

where R_{\min} is a network-defined threshold. This threshold can be dynamic and can periodically be updated to prevent manipulation through mass contract deployment.

The network can update R_{\min} based on system load and validator participation:

$$R_{\min}^{(t+1)} = \begin{cases} R_{\min}^{(t)} + \Delta R, & \text{if } N_{\text{delegates}} > \theta_{\text{high}} \\ R_{\min}^{(t)} - \Delta R, & \text{if } N_{\text{delegates}} < \theta_{\text{low}} \\ R_{\min}^{(t)}, & \text{otherwise} \end{cases} \quad (33)$$

where:

- θ_{high} and θ_{low} are predefined thresholds for network congestion and underutilization, respectively.
- $N_{\text{delegates}}$ is the total number of delegated contracts.
- ΔR is the adjustment factor.

This ensures that as network activity increases, delegation criteria become more stringent, and vice versa.

Validators must maintain a required number of delegated contracts N_V to participate in validation:

$$N_V \geq N_{\min} \quad (34)$$

where N_{\min} is a network-defined constant. This prevents validators from self-delegating a large number of low-reputation contracts to gain an unfair advantage.

When a contract C is instantiated, as previously discussed, the deployer address is automatically assigned as both the `owner` and the `delegate_toto` field within the `DelegateInfo` mapping:

$$\mathcal{D}(C) = (\text{owner}(C), \text{delegate_to}(C), b_t) \quad (35)$$

where:

- `owner(C)` represents the deploying account.
- `delegate_to(C)` initially points to the owner.
- b_t is the block index of contract instantiation.

Given this initialization, a contract should not accumulate stake score (S_C) if the `owner` and `delegate_toto` fields remain the same:

$$\text{owner}(C) = \text{delegate_to}(C) \implies \Delta S_C = 0 \quad (36)$$

Rationale: Stake score is intended to measure computational contribution validated through delegation. If the owner and delegate are identical, there is no external validation or trustless assurance of execution, meaning self-delegation does not align with the core principles of PoCS.

To ensure trustless validation and correct incentive distribution, a contract must explicitly delegate itself to another address V (typically a validator contract) by invoking a special `delegate()` function. The delegation process is governed by the following constraints:

1. *Minimum Reputation Threshold:* A contract must achieve a minimum reputation R_{\min} before it can delegate:

$$R_C \geq R_{\min} \quad (37)$$

Since reputation (R_C) is non-decreasing (as per its invariants), once a contract attains delegation eligibility, it remains eligible unless R_{\min} is updated by the network.

2. *Owner Authorization:* Only the owner of the contract can authorize delegation:

$$\sigma_{\text{tx}} = \text{owner}(C) \Rightarrow \text{valid delegation} \quad (38)$$

where σ_{tx} is the transaction signer.

If the owner is an externally owned account (EOA), signature verification follows standard cryptographic validation:

$$\text{Verify}(\sigma_{\text{tx}}, \text{owner}(C), H(\mathcal{T})) = \text{true} \quad (39)$$

where $H(\mathcal{T})$ is the transaction hash, and Verify is the digital signature verification function.

However, if $\text{owner}(C)$ is itself a contract, signature verification is undefined, as smart contracts lack private keys for signing. Instead, delegation must be authorized via an on-chain call:

$$\mathcal{E}(\mathcal{F}_{\text{caller}}) = C_{\text{owner}} \Rightarrow \text{valid delegation} \quad (40)$$

where \mathcal{E} represents the execution environment, and $\mathcal{F}_{\text{caller}}$ is the active execution frame. This ensures that delegation is validated by the blockchain VM rather than relying on explicit cryptographic signatures.

3. *Delegate Update:* The **DelegateInfo** mapping is updated upon successful delegation:

$$\text{delegate_to}(C) \leftarrow V \quad (41)$$

where V is the validator address.

4. *Validator Delegate Count:* The network is required to maintain a mapping **ValidateInfo**, which tracks the number of contracts delegated to each validator:

$$\text{ValidateInfo} : V \rightarrow \mathbb{N} \quad (42)$$

where:

- V is a validator address.
- \mathbb{N} represents the count of contracts delegated to V .

The number of contracts delegated to a validator is updated as follows:

- If the contract was previously delegated to another validator V' , decrement its delegate count:

$$V'(\mathbb{N}) \leftarrow V'(\mathbb{N}) - 1, \quad \text{if } \text{delegate_to}(C) \neq \text{owner}(C) \quad (43)$$

- Increment the new validator's delegate count:

$$V(\mathbb{N}) \leftarrow V(\mathbb{N}) + 1 \quad (44)$$

Once delegation is complete, the stake score of contract C is updated after each invocation. The condition for stake score accumulation is:

$$(\text{delegate_to}(C) \neq \text{owner}(C)) \wedge (R_c \geq R_{\min}) \implies \Delta S_C = G_C \quad (45)$$

where G_C is the gas consumed by the contract execution.

Validator Address Considerations: PoCS does not enforce that only contract addresses can serve as validators. However, practical implementation of validation logic may favor contract-based validators for trustless execution and reward distribution. In general:

- Delegating to a smart contract validator ensures rewards are distributed trustlessly to delegating contracts.
- Delegating to an externally owned account (EOA) relies on the validator to manually distribute rewards, with its consent only.
- Validator contracts can be designed such that their **owners** sign and produce blocks on behalf of the validator contract.

6 Migration

Contract migration refers to updating the execution logic of a smart contract while preserving its on-chain state. This is achieved by replacing the contract's *code hash* without altering its *storage layout* or *contract address*.

Formally, let a contract C at address α be defined by:

$$C_\alpha = (\mathcal{H}, \mathcal{S}) \quad (46)$$

where:

- \mathcal{H} represents the *code hash* of the contract's execution logic.
- \mathcal{S} is the *persistent storage* associated with C_α .

Migration modifies \mathcal{H} while preserving \mathcal{S} , ensuring that:

$$C'_\alpha = (\mathcal{H}', \mathcal{S}) \quad (47)$$

Since the contract address α remains unchanged, all external interactions and on-chain references to C_α remain valid, eliminating compatibility concerns.

As stake score S_c is derived from gas expenditure and contract execution, migration must ensure that:

$$S'_C = S_C \quad (48)$$

This guarantees that contracts do not reset their stake score upon migration, preventing potential exploitation where execution history is erased. While migration does not affect prior stake accumulation, new execution costs may differ. However, PoCS continues to attribute gas usage fairly under the new code logic.

7 Validation

7.1 Minimum Delegates

In PoCS, block validation requires a structured approach to ensure network integrity while preventing abuse through counterfeit contract deployments. Two key factors govern validation eligibility: **minimum delegates** and **minimum reputation**.

The network enforces a fixed constant for *minimum delegates*, similar to the reputation factor. For a contract to update its delegate, it must satisfy the *minimum reputation threshold*. Likewise, for a validator to participate in validation, they must have a minimum number of delegated contracts.

Formally, let D_{\min} be the *minimum delegates* required per validator.

A validator V is eligible to validate only if:

$$|C_V| \geq D_{\min} \quad (49)$$

where $|C_V|$ denotes the number of contracts delegating to V .

A potential concern is whether an entity can deploy a large number of contracts, delegate them to a validator address, and gain validation privileges. To mitigate this the network can enforce periodic updates to R_{\min} to adapt to ecosystem growth and prevent counterfeit delegate contracts creation.

By dynamically adjusting reputation requirements and enforcing delegation thresholds, sybil-like attacks can be prevented.

7.2 Block Production

Todo (PoCS v0.2) - Validator Election, etc

8 Dynamic Constants

Todo (PoCS v0.2) - Network constants vote

9 Destruction

Todo (PoCS v0.2) - Contract Destruction