Proof of Contract Stake
Formal Specification

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

March 7, 2025

Abstract

Abstract Here

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 stake score S_c, ensuring a quantitative commitment to the network's security and consensus mechanism. Formally,

$$B_c = f(S_c) \tag{1}$$

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:

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

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:

   $$R_c \geq R_{\min} \tag{4}$$

   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:

   $$N_c \geq N_{\min} \tag{5}$$

   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 majorily 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 higher performance and security guarantees.

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

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

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* $\mathcal{H}_{\mathcal{D}} : K \to \Psi$
- *StakeInfo* $\mathcal{H}_{\Sigma} : K \to \Omega$

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:

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

where:

- $\mathcal{D}_\lambda$ = `owner`: The unique identifier of the contract deployer, responsible for authorization of delegation updates.
- $\mathcal{D}_\nu$ = `delegateTo`: The validator controller contract address to which the contract is currently staked.
- $\mathcal{D}_\tau$ = `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:

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

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

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

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 of a staked contract. It is formally defined as an ordered triplet:

$$\Sigma = (\Sigma_\rho, \Sigma_\beta, \Sigma_s) \tag{10}$$

where:

- Σ_ρ = `reputation`: The contract's reputation score $R_c^{(t)}$, derived from recurrent contract invocations over consecutive blocks.
- Σ_β = `recentBlockHeight`: The most recent block height $b(t)$ in which the contract was invoked via an extrinsic.
- Σ_s = `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:

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

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 $\mathcal{H}_{\Sigma}$, respectively. Formally, these mappings are defined as:

$$\mathcal{H}_{\mathcal{D}} : K \to \mathcal{D}, \quad \mathcal{H}_{\Sigma} : K \to \Sigma \tag{12}$$

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 $\mathcal{H}_{\Sigma}$ 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 architecture.

The lookup operations in $\mathcal{H}_{\mathcal{D}}$ and $\mathcal{H}_{\Sigma}$ 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. In comparison, using $\mathcal{H}_{\mathcal{D}}$ and $\mathcal{H}_{\Sigma}$ 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, where a contract C is delegated or nominated to a validator V for block production. The bond value B_c is a function of the contract's stake score S_c:

$$B_c = f(S_c) \tag{13}$$

where $f(\cdot)$ represents the network-defined function governing stake contribution. 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:

$$S_c^{(t+1)} = \begin{cases} (S_c^{(t)} \cdot R_c^{(t+1)}) + G_c^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_c^{(t)} + G_c^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \tag{14}$$

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.

4 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:

$$B_u \neq B_v \quad \forall u, v \in C, \quad B_c \notin \mathbb{F} \tag{15}$$

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.

$$\forall C_i, C_j \in C, \quad S_{C_i} \neq S_{C_j} \text{ unless identical execution history} \tag{16}$$

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.

$$B_c \not\rightarrow B_{c'} \tag{17}$$

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:

$$S_c^{(t+n)} \leq S_c^{(t)} \prod_{k=1}^{n} R_c^{(t+k)} + \sum_{k=1}^{n} G_c^{(t+k)} \tag{18}$$

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.

5 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:

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

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:

$$S_c^{(t+1)} = \begin{cases} S_c^{(t)} \cdot R_c^{(t+1)} + G_c^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_c^{(t)} + G_c^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \tag{20}$$

   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.

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.

$$R_c^{(t+1)} \geq R_c^{(t)} \quad \forall t \tag{21}$$

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

$$R_c^{(t+n)} \leq R_c^{(t)} + n \cdot \Delta R_c \tag{22}$$

  This ensures that contracts cannot grow their reputation arbitrarily within short timeframes.

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

$$\forall t_1, t_2 \text{ such that } b(t_1) = b(t_2), \quad R_c^{(t_1)} = R_c^{(t_2)} \tag{23}$$

  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:

  .

6 Stack-Frame

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

$$C = \{f_1, f_2, \ldots, f_n\}, \quad f_i : I \to O \tag{24}$$

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:

$$\mathcal{F} = (\mathcal{C}, \mathcal{T}, \mathcal{G}) \tag{25}$$

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}.
- $\mathcal{G}$ 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*:

$$\mathcal{F}_0 \to \mathcal{F}_1 \to \mathcal{F}_2 \to \cdots \to \mathcal{F}_k \tag{26}$$

where $\mathcal{F}_0$ is the *transaction originator* (typically an Externally Owned Account (EOA)), and $\mathcal{F}_k$ is the deepest execution frame.

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

$$G_{\text{tx}} = \sum_{i=0}^{k} G_{\mathcal{F}_i} \tag{27}$$

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

$$S_{C_j}^{(t+1)} = \begin{cases} S_{C_j}^{(t)} \cdot R_{C_j}^{(t+1)} + G_{\mathcal{F}_j}^{(t+1)}, & \text{if } b(t+1) \neq b(t) \text{ (new block)} \\ S_{C_j}^{(t)} + G_{\mathcal{F}_j}^{(t+1)}, & \text{if } b(t+1) = b(t) \text{ (same block)} \end{cases} \tag{28}$$

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:

$$\sum_{i=0}^{k} G_{\mathcal{F}_i} = G_{\text{tx}} \tag{29}$$

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.

6.1 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:

$$\text{owner}(C_{\text{new}}) = C_{\text{caller}} \tag{30}$$

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:

$$\Psi(C_{\text{new}}) \leftarrow (C_{\text{caller}}, C_{\text{caller}}, b_t) \tag{31}$$

6.2 Rollbacks

Each stack frame consumes gas, and execution halts under two conditions:

1. *Successful Execution*: $\mathcal{F}_i$ returns execution to its caller.
2. *Gas Exhaustion*: $G_{\text{tx}} = 0$ before completion, causing a forced revert.

If a revert occurs due to gas exhaustion, blockchain VMs enforce *full execution rollback*, denoted as:

$$\forall i, \quad G_{\text{tx}} = 0 \implies \mathcal{S}_{\text{final}} = \mathcal{S}_{\text{initial}} \tag{32}$$

where:

- $\mathcal{S}_{\text{initial}}$ is the storage state before execution.
- $\mathcal{S}_{\text{final}}$ is the storage state after execution.

This implies that all stake score updates must be reverted if gas runs out.
If execution fails due to gas exhaustion:

$$S_c^{(t+1)} \leftarrow S_c^{(t)} \tag{33}$$

This ensures that no partial stake score accumulation occurs during unsuccessful execution.
Formally, let the execution function $\mathcal{E}$ be defined as:

$$\mathcal{E}(\mathcal{F}_i) = \begin{cases} \mathcal{S}_{\text{final}}, & \text{if } G_{\text{tx}} > 0 \text{ and execution completes} \\ \mathcal{S}_{\text{initial}}, & \text{if } G_{\text{tx}} = 0 \text{ (revert condition)} \end{cases} \tag{34}$$

Given the recursive stack execution model, for any execution path:

$$\mathcal{E}(\mathcal{F}_k) \circ \mathcal{E}(\mathcal{F}_{k-1}) \circ \cdots \circ \mathcal{E}(\mathcal{F}_0) \tag{35}$$

gas exhaustion at any level i causes a *cascading revert*, restoring $\mathcal{S}_{\text{initial}}$ for all prior frames.

7 Delegation

When to delegate How to delegate ValidatorInfo Count vs KeyStore Stake Score reset Reputation Holds Delegate At change

8 Destruction

Update to Maps Alive Dead Status using Tagged Unions

9 Migration

Migration only may include code hash where storage shall be same no contract address will change so no issues

10 Validation

Min delegates

11 Staking

Future