Open Store

Decentralized Application Distribution Protocol

Andrei Chupin

July 30, 2025

Contents

Context

The OpenStore project was initiated in 2025.

This first version of the Whitepaper serves as a foundational document, describing the architecture and principles of the project before its public launch on the testnet.

The main purpose of this document is to structure the answers to key questions:

- The motivation and problems the project solves
- The technical implementation and architectural decisions

1. Motivation

Traditional stores for applications and other digital goods are built as closed ecosystems (walled gardens), which creates a number of critical problems, especially when used within the context of Web3 and Self-Custody. These include:

- **Lack of Transparency in Publishing**

  The industry spends vast resources on security: smart contract audits, formal code verification, security mechanisms, and the fault tolerance of blockchains. However, the strength of any security system is determined by its weakest link. In the mobile application ecosystem, there is a massive but often overlooked vulnerability—the publishing process.

  In existing systems, the publishing process is a black box. The user cannot see what specific changes are made to the code and cannot verify the integrity or security of an update. This feature is particularly dangerous for non-custodial wallets: a single trusted developer is all it takes to add a hidden exploit, publish the application, and steal users' private keys—such incidents have already occurred multiple times in the industry.

- **Increased Censorship Susceptibility**

  Any centralized platform is forced to comply with legal and political decisions in the countries where it operates. These decisions are sometimes motivated not so much by security concerns as by geopolitical or ideological factors. As a result, innocent creators and their audiences suffer: their applications can be blocked or removed at the request of regulators, even if they do not violate any technological standards.

- **Human Factor in the Review Process**

  The review of publication requests is often delayed for an indefinite period—from several hours to several weeks. During this time, a reviewer, guided by personal opinions or outdated criteria, can suddenly demand revisions or even prohibit the release of an application, having found an "error" that was present from the very beginning. This unpredictability disrupts developers' plans and undermines trust in the platform.

Open Store offers an alternative that solves a number of problems at the protocol level, designed with a focus on security, fairness, and greater control for developers and users.

1.1 A Protocol for Enhanced Security

The OpenStore protocol lays the foundation for building more secure applications through new architectural capabilities:

- **DAO-Governed Releases:**

  The architecture is compatible with DAO-based governance systems, allowing communities to create fully decentralized and transparent publishing processes for critical assets.

- **Separation of Concerns:**

  While working at one of the most popular non-custodial crypto wallets, for many years I had the right to force-push to the main branch and publish new versions of the application. This situation is trivial for most companies. Potentially, a single person could cause tens or even hundreds of billions of dollars in damage to the industry.

  Solving this problem was one of the main goals in creating Open Store. The protocol allows for a modular architecture, separating applications into independent components with different levels of criticality. This makes it possible to isolate the most sensitive parts (e.g., key management or data signing logic) from elements that require frequent updates (e.g., the user interface).

  The publishing process for critical modules can be made as transparent and secure as possible, for example, through community verification or DAO voting, as mentioned earlier. This would make hidden attacks practically impossible.

- **Local Artifact Validation**

  The protocol supports a mechanism for local verification of published files. To do this, the author publishes various on-chain proofs of ownership, linked to their contract address. The client application, upon installation, queries the on-chain data and verifies the signatures, guaranteeing the authenticity and integrity of the downloaded file.

1.2. Fair Fee Model

- **No Royalty Fees**

  The protocol does not charge a percentage of the authors' revenue.

- **Explicit Network Fees**

  All costs are reduced to explicit blockchain fees for specific operations. The author pays only for the resources they actually use.

- **Cost Optimization via Custom Distribution**

  The protocol does not impose a single method for content delivery. To reduce costs, authors can use their own or third-party distribution systems (CDNs) to deliver appli-

cation files.

1.3. Self-Custody

The protocol ensures that developers retain control over their applications, and users retain control over their access to them.

2. Principles

2.1. Openness

The protocol must be as open as possible:

- **Open Source**

  The source code of the protocol and its main components must be publicly available for audit and review.

- **Open Data**

  All data should be public and, where possible, stored on-chain to ensure full transparency of the ecosystem's state.

- **Open Participation**

  The protocol must be permissionless, allowing participants to act in any possible role.

2.2. Decentralization

Striving to eliminate centralized points of failure and control.

- **No Single Point of Failure**

  The architecture must be resilient and independent of any single entity for its operation or availability.

- **Consensus-Driven Governance**

  Critically important protocol decisions must be made at the DAO level.

- **On-Chain Source of Truth**

  Core metadata and state changes must be recorded on-chain, creating an immutable and verifiable source of truth.

- **Data Availability**

  Data stored on-chain must guarantee its censorship resistance and provide any user with unimpeded access to it.

2.3. Viability and Pragmatism

The protocol must maintain a balance, avoiding extremes.

The protocol aims to store as much data on-chain as possible; however, this creates a problem with storing large files. Many applications reach 100–150 MB and are updated weekly, and in the case of mobile games, we are talking about gigabytes of data. Requiring all these files to be stored and paid for on-chain would make the protocol non-viable.

Following this principle, we provide authors with the ability to delete old, unused versions of publications. However, this means validators lose the ability to independently verify the entire chain of blocks.

Is this critical?

- **Atomic Publication.** Unlike traditional blockchains where the entire transaction history is needed to reconstruct the current state (e.g., balances), here each publication is an independent unit. Deleting an old version does not affect the validity of subsequent ones.
- **Dual-Layer Validation.** On-chain validation primarily determines whether content will be displayed in the public catalog (e.g., in search). The final and most important check of authenticity and integrity is performed locally by the client application, just before installation or interaction.
- **Ecosystem Integrity & Spam Prevention.** Without on-chain validation, the catalog could be filled with applications with incorrect signatures or invalid data (accidentally or intentionally). This would lead to mass errors on the user side. Thus, validation serves as a barrier, filtering out invalid and malicious content at an early stage.

The ability to delete old versions is a compromise between full decentralization and economic reality.

2.4. Symmetric Incentives: Advantage & Responsibility

Following Game Theory, I tried to design the protocol so that each actor receives some benefit from using the protocol, while also being assigned a corresponding level of responsibility for their actions.

Actor	Advantage	Responsibility
User	Access to secure applications for critical tasks and a censorship-resistant platform	Exercising due diligence when installing and interacting with new applications

Actor	Advantage	Responsibility
Publisher	No percentage-based fees, resistance to censorship, access to users, ability to create a transparent development, release, and publication process	Ensuring the availability and distribution of their application's data.
Validator/Oracle	Rewards for data validation	Penalties (slashing) for malicious behavior or providing incorrect data.
Owner	Rewards for developing and maintaining the protocol	Long-term development and maintenance of the ecosystem's health.

2.5. Humanity and Responsible Freedom

One of the goals of Web3 is to create freer systems. However, freedom should not mean permissiveness. The protocol must find a balance between openness and preventing obviously destructive use.

To implement this concept, the protocol incorporates the following compromise:

- The platform is not intended for files and applications whose primary function involves universally recognized crimes (see Appendix A. Inhumane List).
- Such materials may be hidden from public catalogs and search results by name and description.
- All content remains permanently accessible when searching by the specific address of a file or application.

Example 1 (Acceptable): A general messenger used for illegal transactions. Consider a regular messaging app—"ChatSphere." Its main features are end-to-end encryption, group chats, file sharing, and voice calls. The developers did not intend for "ChatSphere" to be a tool for criminals but created a universal messenger for any task.

Why this is allowed:

- The core functionality of the application is neutral and suitable for legitimate communication.
- The developers do not advertise "ChatSphere" as a platform for selling drugs or other prohibited goods.

- Blocking “ChatSphere” because of the actions of a few users would punish millions of law-abiding people who use it for work, study, and communication with loved ones.

Thus, when someone uses “ChatSphere” to arrange the purchase of hard drugs, it is considered a misuse of a general-purpose tool, not a reason to ban the application entirely.

Example 2 (Unacceptable): A specialized application for drug trafficking. Now imagine another application—“DrugDirect.” Its design, interface, and functionality are specifically created for buying and selling hard drugs:

- Upon launching, “DrugDirect” opens a catalog with photos, prices, and seller ratings.
- The main screen features buttons like “Heroin,” “Cocaine,” “Secure Delivery,” etc.
- The store description states: “Fast and anonymous deals, no questions asked.”

Why this is prohibited:

- Primary purpose: Everything in “DrugDirect” is exclusively aimed at organizing illegal trade.
- User expectation: The UI and marketing directly promise convenient access to prohibited substances.
- Threat to the ecosystem: Allowing such an application would actively facilitate organized crime and pose a risk to users’ lives and freedom.

For these reasons, “DrugDirect” is hidden from the public catalog and does not appear in search results. It can only be accessed via a direct link or its exact address, which prevents its accidental discovery and stops the platform from actively promoting criminal content.

This mechanism helps protect users from the most harmful content without violating the basic principle of openness.

3. Overview

3.1. Glossary:

References to Glossary sections:

[a] - Actor

[e] - Entity

[u] - UI Application

[c] - Contract

[p] - Process

[n] - Node

[s] - Smart Contract Structure *(added for clarity, as it's used in the text)*

[f] - Fee *(added for clarity, as it's used in the text)*

Data mutability:

(immutable) - Contract data that is set only once.

(mutable) - Contract data that can be changed.

3.1.1. Actor

- **Publisher**

  Publishes an [e]Asset in [u]Open Store Studio.

- **Validator**

  Processes a [s]ValidationRequest, creates blocks, participates in voting.

- **Oracle**

  Synchronizes data from Web2 to Web3.

- **Broadcaster**

  Monitors events in smart contracts, stores them in a structured format, and provides them to users via an API.

- **Owner**

  Participates in the development or governance of Open Store.

- **User**

  The end-user of the [u]Open Store App who interacts with an [e]Asset.

3.1.2. Node

- Validator
- Oracle
- Daemon (Broadcaster)
- API (Broadcaster)
- Statistic (Broadcaster)
- Blockchain (BSC, opBSC, Greenfield)

3.1.3. Entity

- **Asset**

  Any type of published resource (application, game, book, etc.).

- **Assetlink**

  A mechanism that links a domain (website) to an application. It includes certificates and data about which applications are authorized to link to this domain for verification purposes (see **https://developer.android.com/training/app-links/verify-android-applinks**).

- **Asset Artifact**

  A file representing a specific, unique version of an [e]Asset. It has an on-chain representation as a [s]BuildInfo and is stored in a storage space owned by the [a]Publisher.

- **Asset Endpoint**

  The application owner's website. This domain is visible to the [a]User and is used to make an installation decision.

  It must contain the path **$ENDPOINT/.well-known/assetlinks.json** listing all **Certificate SHA-256 Fingerprints** used to sign the application

  (see **Android Assetlinks**).

- **3.1.4. Greenfield**

  A blockchain for file storage from Binance.

- **Bucket**

A namespace in Greenfield storage for storing files. Similar to AWS S3.

- **Ownership Version**

  A version of the ownership data for an [e]Asset. It is stored in the [c]AppOwnerPluginV1 plugin and has an on-chain representation as a [s]OwnerInfo. It is automatically incremented when the [s]OwnerInfo is updated.

- **Cross Chain**

  A bridge that enables interaction between the BSC, opBSC, and Greenfield blockchains.

- **ProofOfCertificateOwnership**

  A digital signature confirming that the [a]Publisher owns a certificate. It is generated by signing a data string in the format 'APP_ADDRESS::SHA_256_CERT_FINGERPRINT' with the private key corresponding to that certificate.

- **Distribution Link**

  A link to a server for file distribution. [a]Publishers can use such links to optimize data transfer costs (e.g., via a CDN).

- **Validation Block**

  A Protobuf-formatted object containing block metadata and information about request validations. It is passed in the calldata field of a BSC transaction. A reference to the transaction is stored in the [s]BlockRef.

- **Proposal Window**

  The time frame within which a [e]Validator must perform [p]Block Proposing.

  Set by the [a]Owner in the [c]OpenStore contract.

3.1.5. UI

- **Open Store Studio**

  A website for creating, managing, and publishing [e]Assets.

- **Open Store App**

  An application for finding, installing, and updating [e]Assets.

3.1.6. Contract

- **OpenStore**

The main contract that implements the consensus system, stores published [e]Asset versions, their validation statuses, and validator data.

- **AssetlinkOracleV1**

  The main contract for storing requests and results of [p]Ownership Verification.

- **Plugins**

  Contracts designed to extend the base functionality of other contracts.

- **DevFactory**

  A factory contract for creating a DevAccount.

- **DevAccount** — a representation of a [a]Publisher in the system.
  - **DevAccount**

    Stores the publisher's username and connected plugins.

  - **AppsPluginV1**

    Stores applications belonging to the publisher.

  - **GreenfieldPluginV1**

    A contract for [e]Cross-Chain interaction with a [e]Bucket in [e]Greenfield storage.

- **App** — a representation of an [e]Asset of type "Application".
  - **App**

    Stores general information about the application.

  - **BuildsPluginV1**

    Stores application versions and links to on-chain files.

  - **OwnershipPluginV1**

    Stores information about application ownership and proofs of that ownership.

  - **DistributionPluginV1**

    Stores Web2/Web3 links for file distribution (downloading).

3.1.7. Smart Contract Structures

- BuildInfo - representation of an [e]Asset Artifact
  - versionCode - int64

  - versionName - string
  - referenceId - bytes
  - protocolId - uint16
  - checksum - bytes
- OwnershipInfo - ownership data for an [e]Asset
  - endpoint - string
  - proofs
    - sha256CertificateFingerprint - bytes32
    - proofOfOwnership = signature(appAddress::sha256CertificateFingerprint) - bytes
- BlockRef - metadata for a [e]Validation Block
  - id - uint256
  - fromRequestId - uint256
  - toRequestId - uint256
  - result - uint256
  - objectId - bytes
  - protocolId - int32
  - objectHash - bytes32
  - blockMask - uint8
  - createdBy - address
- ValidationRequest - abstract data for [e]Asset validation
  - id - uint256
  - type - uint32
  - target - Address
  - data - bytes

3.1.8. Process

- **Ownership Verification**

  The process of comparing data from the [c]AppOwnerPluginV1 plugin with the information published at ENDPOINT/.well-known/assetlinks.json to confirm the link between the application and the domain.

- **Artifact Validation**

  The process of validating (checking) a new version of an [e]Asset.

- **Block Proposing**

  The process in which a [a]Validator proposes a [e]Validation Block for consideration.

- **Block Voting**

  The process in which a [a]Validator votes on an existing [e]Validation Block during [p]Block Proposing or creates a new one, initiating [p]Block Discussing.

- **Block Discussing**

  A process that begins when at least two competing [e]Validation Blocks are proposed for a vote.

- **Block Finalization**

  The process in which the data of the winning [e]Validation Block is recorded in the state of [c]OpenStore, and the winning [a]Validators are rewarded while the losing ones are penalized (slashed).

3.1.9. Fees

- **Network Fee**

  Gas Fee - The standard fee for executing a transaction.

  Cross Chain Fee - The fee for sending a message from BSC to Greenfield.

- **Greenfield Fee** - (see **Greenfield Billing**)

  Download Quote Fee - A weekly prepayment for a certain volume (in GB) that users can download from your [e]Bucket.

  Storage Fee - A weekly prepayment for storing your files.

- **Oracle Fee**

  A fee paid by a [a]Publisher to an [a]Oracle for conducting [p]Ownership Verification.

  The fee amount is set by the [a]Owner in [c]AssetlinkOracleV1.

- **Validation Fee**

  A fee paid by a [a]Publisher to a [a]Validator for processing a [e]Validation Request.

  The amount is set by the [a]Owner in the [c]OpenStore contract.

- **Proposal Stake**

  A stake that a [a]Validator provides for [p]Block Proposing. If another [a]Validator challenges the block during [p]Block Discussion, the first one will lose their stake, and the reward is divided among the winners (the initiator of the [p]Block Discussion and those who voted for it).

- **Vote Stake**

  A stake that a [a]Validator provides for [p]Block Voting. If the [a]Validator loses the vote during a [p]Block Discussion, all [a]Validators who voted for the losing [e]Validation Block will lose their stake, and the reward is divided among the winners (the initiator of the [p]Block Discussion and those who voted for it).

- **Proposal Penalty**

  A penalty for a [a]Validator if they do not perform [p]Block Proposing within the [e]Proposal Window after the previous block's proposal was completed, provided there is at least one [e]Validation Request in the queue.

  The amount is set by the [a]Owner in the [c]OpenStore contract.

- **Inactivity Penalty**

  A penalty for a [n]Validator if it does not participate in network operations and fails to deregister within X blocks.

  The amount is set by the [a]Owner in the [c]OpenStore contract.

- **Minimum Stake**

  The minimum balance a validator must have in [c]OpenStore to register on the network. It is calculated by the formula:

$$\text{Vote Stake} \times (\text{MAX\_CONCURRENT\_VOTINGS} - 2) + \text{Proposal Stake} \times 2$$

  MAX_CONCURRENT_VOTINGS — the maximum number of concurrently active [p]Block Votings.

3.2. Workflow

TL;DR

Below is a simplified model of the interaction between the main actors and the protocol. In reality, some steps may be performed within a single atomic operation.

1. A Publisher creates and configures a Publisher Account contract.
2. The Publisher creates a new Asset (Application) contract.
3. The Publisher specifies the Ownership Info for the new Asset (website domain, certificate hashes, and proofs of ownership for those certificates).
4. The Publisher sends the Ownership Info for Ownership Verification.
5. The Oracle performs Ownership Verification.

6. The Publisher uploads an Asset Artifact (the file for the new Asset version).
7. (Optional) The Publisher can add custom distribution links for the Asset Artifact (e.g., CDN).
8. The Publisher sends the new Asset Artifact for Artifact Validation.
9. A Validator performs Artifact Validation, checking the structure and signatures of the Asset Artifact.
10. The Publisher publishes the validated Asset Artifact for public access.
11. The Daemon synchronizes the necessary data from the Blockchain to a DB.
12. The API provides data to users from the DB in a structured format.
13. The User interacts with the Asset (Asset Artifact) through an application.

3.2.1. Creating a DevAccount Contract

1. The [a]Publisher creates a new [c]DevAccount in [u]Open Store Studio via [c]DevFactory, specifying:
    1. Name (immutable) - a unique username
    2. File Storage (mutable) - storage for [e]Asset Artifacts
        1. Currently, the only available file storage is [e]Greenfield.
2. Upon creation, two plugins are connected to the [c]DevAccount: [c]DevGreenfieldPluginV1 and [c]DevAccountAppsPluginV1.
3. As part of connecting [c]DevGreenfieldPluginV1:
    1. A minimum amount of BNB is sent to [e]Greenfield to top up the balance and pay for storage, using [e]Cross Chain.
    2. A [e]Bucket is created in [e]Greenfield, using [e]Cross Chain.

3.2.2. Managing DevAccount

The [a]Publisher, using [u]Open Store Studio, can change parameters such as:

1. Bucket Read Quote - the amount of data in GB available for download from the [e]Bucket. If the limit is exceeded, [n]Greenfield may provide data at a minimal speed OR completely block data distribution (a [e]Distribution Link can be used for optimization).
2. Bucket Balance - payment for file storage and distribution is made in [e]Greenfield BNB; the balance can be topped up directly from a BSC BNB wallet using [e]Cross Chain.
3. Invisibility - a parameter that allows the [a]Publisher to hide an application in [c]OpenStore. After activating this option, the [e]Asset will be unsearchable in the [u]Open Store App until this function is turned off.

3.2.3. Creating an App

1. The [a]Publisher creates a new [c]App via [c]DevAccountAppsPluginV1 in [u]Open Store Studio, specifying:
    1. PackageName (immutable) - a unique text identifier
    2. Name (mutable)
    3. Description (mutable)
    4. ProtocolId (mutable) - app metadata storage
    5. PlatformId (immutable) - type of OS platform (e.g., Android, iOS, etc.)
    6. CategoryId (mutable) - type of [e]Asset category (Books, Tools, Sport, etc.)
2. Upon creation, the [c]App connects to three base plugins: [c]AppOwnerPluginV1, [c]AppBuildsPluginV1, and [c]AppDistributionPluginV1.

3.2.4. Ownership Verification for an Android App

1. The [a]Publisher, using [u]Open Store Studio, fills out the ownership form ([s]OwnershipInfo):
    1. [e]Asset Endpoint (mutable)
    2. Proofs of ownership of the Android application's signing certificates:
        1. **Certificate SHA-256 Fingerprint** (mutable) - e.g., F8:F9:21:DA:21:05:21:A2:21:BC:21:9A:21:81:21:E0:21:AB:21:2D:93:EA:53:41:7A:45:81:98:F8:ED:5D:80)
        2. **[e]ProofOfCertificateOwnership** (mutable)
2. The [a]Publisher saves the [s]OwnershipInfo in [c]AppOwnerPluginV1.
3. The [a]Publisher sends a [p]Ownership Verification request to [c]AssetlinksOracle.
    1. [p]Ownership Verification requires payment of the [f]Oracle Fee.
    2. On success, the [a]Publisher will be able to send [e]Asset Artifacts of the verified [c]App for [p]Artifact Validation in [c]OpenStore.
    3. On failure, the [a]Publisher loses the ability to send new [e]Asset Artifacts for [p]Artifact Validation (if the last version of [s]OwnershipInfo had been verified); all previously published [e]Asset Artifacts will remain available.
4. When the [s]OwnershipInfo in [c]AppOwnerPluginV1 is changed, the [e]Ownership Version is incremented, after which [p]Ownership Verification must be passed again.
    1. The number of times the same [e]Ownership Version can be sent for [p]Ownership Verification is unlimited.
5. Ideally, a [c]App undergoes [p]Ownership Verification only once; the result of this check is reused in subsequent checks.

3.2.5. The Role of the Oracle

1. Currently, the [a]Oracle is centralized and owned by the [a]Owner.
2. The [a]Oracle is necessary so that a [a]Validator can retrieve all necessary data directly from the [n]Blockchain during [p]Artifact Validation. Otherwise, a huge number of opportunities for data manipulation arise, which in turn leads to security problems for the entire protocol.
3. The functions of the [a]Oracle could be assigned to [a]Validators and local verification by the [u]Open Store App, but in this implementation, [a]Validators do not guarantee 100% reliable verification, and the [u]Open Store App may have problems accessing the [e]Asset Endpoint.
4. The [a]Oracle will be decentralized in future releases, once all other system components have been stabilized.

3.2.6. Processing an Ownership Verification Request for an Android App

1. The [a]Oracle receives the request as a [n]Blockchain event.
2. The [a]Oracle attempts to fetch a JSON file using the Asset Endpoint **$ENDPOINT/.well-known/assetlinks.json** (see **https://developer.android.com/training/app-links/verify-android-applinks**).
    1. If the [e]Asset Endpoint is unavailable, [p]Ownership Verification fails with an **error**.
3. In the fetched JSON, the [a]Oracle tries to find an [e]Assetlink corresponding to the [c]App (e.g., for Android, “namespace”: “android_app” and “package_name”: “org.openstore.example.android”).
    1. If the [e]Assetlink is not found, [p]Ownership Verification fails with an **error**.
4. The [a]Oracle checks all ‘sha256_cert_fingerprints’ from the found [e]Assetlink against those specified in [c]AppOwnerPluginV1.
    1. If any ‘sha256_cert_fingerprints’ is missing from [c]AppOwnerPluginV1, [p]Ownership Verification fails with an **error**.
    2. If the AppOwnerPluginV1 contract contains all ‘sha256_cert_fingerprints’, [p]Ownership Verification completes **successfully**.
5. The [a]Oracle receives the [f]Oracle Fee as a reward.

3.2.7. Artifact Validation for an Asset Artifact in OpenStore

1. The [a]Publisher, using [u]Open Store Studio, specifies:
    1. The [e]Asset Artifact to be validated.
        1. If necessary, the [a]Publisher uploads a new file using [u]Open Store Studio.

    2. The necessity of performing [p]Artifact Validation (see 3.2.15).
2. The [a]Publisher sends a [e]Validation Request to [c]OpenStore.
    1. [p]Artifact Validation requires payment of the [f]Validation Fee.
    2. In case of an error, the [a]Publisher can re-verify the [e]Asset Artifact.
    3. On success, the validated [e]Asset Artifact is marked as valid in [c]OpenStore and can be published.
    4. Each validated [e]Asset Artifact must have a version number greater than the last one that was successfully validated.

3.2.8. Validator Registration

1. The [a]Validator starts a [n]Validator node, specifying the necessary operating parameters.
2. The [a]Validator deposits funds into their balance in [c]OpenStore.
    1. The [a]Validator has the right to withdraw funds from [c]OpenStore.
3. The [n]Validator registers itself in [c]OpenStore.
    1. Registration requires having a balance of at least the [f]Minimum Stake.
4. The [n]Validator attempts to assign itself to validate block N.
    1. **IMPORTANT!** If the next available block is more than X blocks ahead of the currently validating block (parameter set by the [a]Owner in [c]OpenStore), the [n]Validator must pay:

$$\text{Voting Amount} = \frac{\text{Total Stake}}{\text{Validator Stake}} - 1$$

    2. **EXAMPLE!**
        1. Total Stake = 150
        2. Validator Stake = 15
        3. (Total Stake / Validator Stake) - 1 = 9
        4. In this case, the Validator's Stake is 1/10 of the total, which means they will be responsible for every 10th block.
    3. To ensure a [a]Validator has participated in block creation, it must vote on other blocks, thereby replenishing its Voting Amount.
        1. 1 Voting Amount is awarded for each vote.
        2. All Voting Amount calculations are performed with a precision of 10^9, or 1 gwei.
    4. If the [n]Validator does not have enough Voting Amount, it continues to function, performing [p]Artifact Validation and voting on other blocks.
    5. The [n]Validator must temporarily lock the [f]Proposal Stake amount in the con-

tract.

5. The [n]Validator must deregister when it stops operating.
    1. Forced deregistration is possible, in which case the [n]Validator is fined the [f]Inactive Penalty amount, which is given as a reward to the [n]Validator that executed the deregistration.

3.2.9. Processing Artifact Validation for an Android Asset Artifact

1. The [a]Validator receives an event from the [n]Blockchain.
2. Using the data from the event, the [a]Validator downloads the APK ([e]Asset Artifact).
3. The [a]Validator parses the APK's metadata (see **APK Signing V2**).
4. The [a]Validator checks the APK's metadata:
    1. The APK must have a valid structure and signature.
    2. The VersionCode, PackageName, and APK Checksum must match those specified in the [e]Asset Artifact and the [c]App.
    3. All SHA256 Certificate Fingerprints must be listed in [c]AppOwnerPluginV1.
    4. All [e]ProofOfCertificateOwnership must be valid; the signature is verified by manually reconstructing the signed data and using the public key from the certificate in the APK.
5. The [a]Validator saves the verification result until [p]Block Proposal or [p]Block Voting.
6. After the block is created, all verification results are saved in [c]OpenStore.

3.2.10. Block Formation and Proposal by a Validator

1. Block proposals occur incrementally.
    1. **EXAMPLE!** Block #5 cannot be proposed for a vote if block #4 has not yet been proposed.
2. The [a]Validator waits for its turn in the queue.
3. The [a]Validator forms a [e]Validation Block from the available [p]Artifact Validation results.
    1. The [a]Validator must propose the block within the [e]Proposal Window, or it will be penalized with the [f]Proposal Penalty, and any other [a]Validator will gain the right to perform [p]Block Proposing for that block, receiving the [f]Proposal Penalty as a reward.
4. The [a]Validator sends the Validation Block in binary ProtoBuf format to the Binance Smart Chain as calldata, waiting for the transaction to be executed.
5. The [a]Validator proposes the block (a reference to the data structure) for a vote in [c]OpenStore.

3.2.11. Voting for a Block by a Validator

1. The [a]Validator must form its own [e]Validation Block for all requests included in the original [e]Validation Block.
    1. If the formed block differs from the original, the [a]Validator can start a [p]Block Discussing and propose its version of the block.
        1. To propose an alternative version of the block, the [f]Proposal Stake amount must be locked in the contract.
        2. If the proposed block loses the vote, the [f]Proposal Stake will be distributed among the winners; otherwise, the [f]Proposal Stake is returned to the [a]Validator.
    2. If the formed block matches the original, the [a]Validator votes for it.
        1. To vote, the [f]Vote Stake amount must be locked in the contract.
        2. If the block loses the vote, the [f]Vote Stake will be distributed among the winners; otherwise, the [f]Vote Stake is returned to the [a]Validator.
2. Notes:
    1. At any given time, within a single block, a [a]Validator can only perform one of the following functions:
        1. [p]Block Proposing
        2. [p]Block Discussing
        3. [p]Block Voting

3.2.12. Block Finalization by a Validator

1. Finalization, like proposal, occurs incrementally.
    1. **EXAMPLE!** Block #5 cannot be finalized if block #4 has not yet been finalized.
2. A [a]Validator initiates block finalization in [c]OpenStore.
    1. Block finalization occurs if:
        1. the block has gathered enough votes so that the remaining votes cannot change the outcome;
        2. the voting time has expired.
3. After finalization, the data from the block is saved in [c]OpenStore.
4. [a]Validators who performed [p]Block Voting and [p]Block Discussing for the losing [e]Validation Block lose their [f]Proposal Stake and [f]Vote Stake; this amount is distributed among the [a]Validators who won the vote.

3.2.13. Terminal States of a Validation Request

1. Within the protocol, a block is stored as two entities:

    1. [e]Validation Block - the full version, as a Protobuf object.
    2. [s]BlockRef - a truncated version, as a struct.
2. The [s]BlockRef, being a truncated version of the block, does not store all information about the [e]Validation Request verification results, only its terminal state.
3. In the [s]BlockRef, the terminal state is stored in the uint256 result field and can have 4 values:
    1. 2 bits for each [e]Validation Request.
    2. Up to 128 [e]Validation Requests per block.
4. Types of terminal states:
    1. 0b00 - UNAVAILABLE
    2. 0b01 - SUCCESS
    3. 0b10 - NONE, not used, reserved
    4. 0b11 - ERROR

3.2.14. Handling the UNAVAILABLE State

The UNAVAILABLE status is necessary to keep the protocol running in case one or more [n]Blockchains go down during validation.

Logic for handling UNAVAILABLE:

1. A [a]Validator during [p]Block Proposing can declare ANY [e]Validation Request as UNAVAILABLE; in this case, the [f]Validation Fee is not returned to the [a]User, nor does the [a]Validator receive it as a reward—it remains in the protocol's balance.
    1. This behavior is necessary from a game theory perspective to prevent any party from using it to attack the protocol.
2. A [a]Validator during [p]Block Discussing must treat UNAVAILABLE statuses in the original [e]Validation Block as not having significant differences.
    1. A significant difference is a differing status that is NOT UNAVAILABLE.
        1. UNAVAILABLE == SUCCESS/NONE/ERROR - not a significant difference.
        2. SUCCESS ≠ NONE - a significant difference.
3. A [a]Validator during [p]Block Voting must also treat UNAVAILABLE statuses in the original [e]Validation Block as not having significant differences.
4. A [a]Validator during [p]Block Voting can attach an unavailabilityMask of type uint128 if the original block has no significant differences.
    1. 1 bit for each [e]Validation Request.

2. If a majority of [a]Validators submit an unavailabilityMask with UNAVAILABLE statuses for requests that were marked with non-UNAVAILABLE statuses in the original block, their status changes to UNAVAILABLE.
3. In this case, the [e]Validation Block and [s]BlockRef will differ; here, the source of truth is the final (adjusted) [s]BlockRef and [c]OpenStore. The [e]Validation Block merely reflects the block on which the voting was based.

5. If the unavailabilityMask changes a state that, in turn, affects a [p]Block Discussing block such that the significant differences with the original block disappear, the [f]Proposal Stake and [f]Vote Stake are returned to their respective [a]Validators.

   EXAMPLES!

   1. Suppose a [s]BlockRef has 2 results with statuses SUCCESS and UNAVAILABLE.
      1. If a [a]Validator's own check yields results SUCCESS and SUCCESS, it should consider the original block as equal and initiate [p]Block Voting.
      2. If a [a]Validator's own check yields results ERROR and SUCCESS, it should consider the original block as different and initiate [p]Block Discussing with statuses ERROR and SUCCESS.
   2. Suppose a [s]BlockRef has 2 results with statuses SUCCESS and SUCCESS.
      1. If a [a]Validator's own check yields results SUCCESS and UNAVAILABLE, it votes for the original block, attaching an unavailabilityMask with the value 0b0000…0010 (second request is UNAVAILABLE).
   3. Suppose a [s]BlockRef has 2 results with statuses SUCCESS and SUCCESS.
      1. This [s]BlockRef has a [p]Block Discussing with statuses SUCCESS and ERROR.
      2. The first block with statuses SUCCESS and SUCCESS won the vote.
      3. But a majority of [a]Validators provided an unavailabilityMask, so the second status was changed to UNAVAILABLE.
      4. The final block now has results SUCCESS and UNAVAILABLE.
      5. In this case, the second block with statuses SUCCESS and ERROR loses its significant difference.
      6. The [a]Validators who proposed and voted for the second block with statuses SUCCESS and ERROR will not be penalized and will get their Stake back.

3.2.15. Publishing an Asset Artifact in OpenStore

There are 3 ways to publish a [s]BuildInfo in [c]OpenStore:

1. **Publication** - In this case, the [e]Asset Artifact is marked as published but NOT validated; it will only be available when searching by the [e]Asset's address.

2. **Separate [p]Artifact Validation and Publication** - You can first complete [p]Artifact Validation, in which case the [e]Asset Artifact will be marked as validated and NOT published, after which it can be published at any convenient time.
3. **Combined [p]Artifact Validation and Publication** - When submitting for [p]Artifact Validation, there is an option to add auto-publication, and in this case:
    1. Upon successful [p]Artifact Validation in [c]OpenStore, the published [e]Asset Artifact will be automatically marked as validated and published.
    2. If [p]Artifact Validation fails with an error, the [e]Asset Artifact is not marked in any way, meaning it will be NOT validated and NOT published.

3.2.16. Configuring Asset Distribution

1. Normally, distribution of an [e]Asset Artifact occurs directly from [e]Greenfield.
2. If necessary, a [a]Publisher can specify one or more [e]Distribution Links using their own servers.
3. When specifying a [e]Distribution Link, the [a]Publisher can use parameters to generalize the links:
    1. ${VERSION_CODE} - the version number specified in the [e]Asset Artifact (e.g., 132).
    2. ${VERSION_NAME} - the version name of the [e]Asset Artifact (e.g., 1.0.2-beta01).
    3. ${REF_ID} - the link identifier of the [e]Asset Artifact in [e]Greenfield or another protocol (e.g., 0xFFAAWW).

3.2.17. Asset Availability

1. In most cases, a [a]User receives data from the [n]API, which is synchronized by a [n]Daemon from BSC, opBSC, Greenfield, or another blockchain.
2. The availability of an [e]Asset to users via the [n]API may be restricted if:
    1. The [e]Asset Artifact has not passed [p]Artifact Validation.
    2. The [a]Publisher has changed the visibility settings of the [e]Asset in [u]Open Store Studio.
    3. The primary functionality of the [e]Asset falls under one or more categories prohibited on the platform (see Appendix A).
3. It should be noted that any [e]Asset will always be available when searching by address, as the data is retrieved directly from the blockchain; in this case, the [u]Open Store App acts as an Explorer.
    1. Except in cases where the [a]Publisher has changed the visibility settings in [c]OpenStore; in this case, visibility is restricted by the owner at the contract level.

3.2.18. Installing an Asset Artifact via the Open Store App

1. In the [u]Open Store App, there are 3 ways to find an application:
    1. **Catalog** ([n]API) - [e]Assets that have passed [p]Ownership Verification and [p]Artifact Validation.
    2. **Search by name** ([n]API) - [e]Assets that have passed [p]Ownership Verification and [p]Artifact Validation.
    3. **Search by address** ([n]Blockchain) - [e]Assets published in [c]OpenStore; [p]Ownership Verification and [p]Artifact Validation are optional, as all data is fetched directly from the blockchain.
2. Data for installation:
    1. General information and the latest version of the [e]Asset are taken from:
        1. The [n]API if navigating from the catalog or searching by name.
        2. The [n]Blockchain if searching by address.
    2. The download link for the latest version of the [e]Asset Artifact is always obtained strictly from the [n]Blockchain.
3. Before installation, the [u]Open Store App performs a preliminary validation of the [e]Asset Artifact, which includes:
    1. Comparing the actual file checksum with the checksum from the [s]BuildInfo structure.
    2. Comparing the file's sha256CertificateFingerprint with the sha256CertificateFingerprint from the [s]OwnershipInfo.
    3. Verifying the authenticity of the [e]ProofOfCertificateOwnership from the [c]OwnershipInfo using the PubKey from the file itself.

3.2.19. Updating an Asset via the Open Store App

1. New versions of an [e]Asset Artifact are installed manually; the user can allow automatic updates if desired.
2. [c]OpenStore may contain multiple applications with the same identifier (packageName), in which case the [c]App address will serve as the distinct identifier.

Appendix A.

Inhumane List

If the primary function of an application or file is one of the items listed below, it will be hidden from public search and will only be accessible when searching by its address.

Sale of Prohibited Goods and Substances

- Hard drugs
- Counterfeit documents and currency
- Stolen goods and property

Cybercrime and Malicious Activity

- Malicious and spyware software (Malware)
- Hacking services and tools
- Carding and theft of financial credentials

Financial Crimes and Fraud

- Money laundering
- Fraud and scams
- Sale of stolen data

Violence, Extremism, and Exploitation

- Child Sexual Abuse Material (CSAM)
- Human trafficking and exploitation
- Contract killings
- Terrorism

Violation of Privacy and Confidentiality

- Sale of personal information (doxing)
- Illegal surveillance and eavesdropping

Appendix B.

Publisher components:

- Asset Contracts
  - DevAccount
  - App
- Store Contracts
  - Oracle
  - OpenStore
- UI
  - Open Store Studio
- Tools
  - Proof Generator CLI
  - Publishing CLI

Validator/Oracle components:

- Store Contracts
  - Oracle
  - OpenStore
- Node
  - Oracle
  - Validator

Broadcaster components:

- Node
  - API
  - Sync Daemon
  - Stat API (Optional)

User components:

- Open Store App