A BosOn Infinity - SHA-512 Proof-of-Work Cryptocurrency with Fixed 50 Million Supply

Whitepaper v1.0
Kamil Padula
12.11.2025
4056 Basel
Switzerland

1. Introduction

This document presents a decentralized, peer-to-peer electronic cash and data asset system based on a SHA-512-driven Proof-of-Work (PoW) blockchain with a fixed maximum monetary supply of 50,000,000 coins. A reference ERC-20 bridge to Ethereum has already been implemented and is planned to be gradually rolled out from testnet to mainnet.

The system aims to:

1. Provide a simple, auditable, and predictable monetary policy through a halving-based issuance schedule with a hard cap enforced directly in consensus.
2. Use well-understood cryptographic primitives – Ed25519 digital signatures and SHA-512 hashing – to secure transactions, blocks, and consensus parameters.
3. Support GPU-friendly mining with a bit-mask difficulty rule over a “mix” value produced by miners, combined with a dynamic difficulty retargeting mechanism.
4. Offer a basic but extensible on-chain platform, including:
    a. a straightforward account-based ledger,
    b. a staking / masternode reward layer,
    c. minimal smart contracts (e.g. time-lock contracts),
    d. native NFT support,
    e. a Data→Token protocol for monetizing digital files (PDF, DOC, JPEG, TXT, etc.).

The design follows the general outline of Bitcoin's proof-of-work blockchain but uses a different cryptographic stack and integrates staking incentives, simple contracts, NFTs, and a data monetization layer directly into the base protocol.

2. System Overview

The system is a public, permissionless blockchain. Nodes maintain a common ledger state containing:

- a sequence of blocks linked by cryptographic hashes,
- a global mapping of addresses to account state,
- auxiliary structures for staking validators, deployed contracts, and NFTs.

Participants fall into three main categories:

1. **Full nodes**
    a. Validate blocks and transactions.
    b. Maintain the full chain and global state.
    c. Provide RPC endpoints for wallets, miners, and applications.
2. **Miners**
    a. Receive work templates from full nodes.
    b. Perform Proof-of-Work using a SHA-512-based algorithm.
    c. Submit candidate blocks when a valid solution is found.
3. **Staking / Masternodes**
    a. Lock coins to become validators in the staking set.
    b. Optionally register network endpoints to provide additional services.
    c. Receive a share of block rewards proportional to their stake.

3. Ledger and Account Model

3.1 Units and Decimals

The currency is measured in atomic units. One full coin is defined as:

1 COIN = 10^d units

where d (Decimals) is typically 8, meaning that one coin corresponds to 100,000,000 units. All balances in consensus are integers of atomic units.

3.2 Accounts

The ledger uses a simple account model. Each account is identified by an address and stores:

- **Balance** – the number of atomic units.
- **Nonce** – a monotonically increasing counter used to order transactions from the same address and prevent replay.

The global state can be understood as a mapping:

State: Address → (Balance, Nonce)

3.3 Transactions

A transaction describes a state transition. The core fields are:

- `from` – sender address,
- `to` – recipient address (or contract / NFT / system address),
- `amount` – value to transfer (in units),
- `fee` – transaction fee (in units),
- `nonce` – sender's sequential transaction number,
- `pubkey` – Ed25519 public key corresponding to `from`,
- `signature` – Ed25519 signature over the transaction payload,
- `hash` – SHA-512 hash of the canonical payload.

Additionally, a transaction may include:

- `type` – transaction type, for example:
  - `transfer` (default),
  - `stake`, `unstake`, `mn_register`,
  - `deploy`, `call` for contracts,
  - `nft_mint`, `nft_transfer`, `nft_burn` for NFTs,
  - `data_commit` or similar for Data→Token flows.
- `data` – optional metadata (e.g. contract parameters, NFT metadata hash, Data→Token scoring data) encoded as a structured payload.

3.4 Transaction Validity

A transaction is valid if and only if:

1. The public key is well-formed and of the correct length for Ed25519.
2. The address derived from the public key equals `from`.

3. The Ed25519 signature verifies over the canonical payload.
4. The sender's account has sufficient balance:

Balance_from ≥ amount + fee

5. The nonce is exactly one more than the current account nonce:

nonce_tx = Nonce_from + 1

6. Any type-specific rules (for staking, contracts, NFTs, or Data→Token) are satisfied.

If any condition fails, the transaction is rejected by full nodes and never included in blocks.

4. Cryptography

4.1 Digital Signatures: Ed25519

The system uses Ed25519 for digital signatures:

- Key pairs are generated off-chain.
- The public key is encoded (e.g. in hex) and embedded in each transaction.
- The address is derived from the public key via SHA-512.
- Every transaction includes a signature over the canonical payload containing `from`, `to`, `amount`, `fee`, `nonce`, and `pubkey`.

Only the holder of the private key can produce a valid signature. Nodes independently verify signatures before accepting any transaction into the mempool or a block.

4.2 Hashing: SHA-512

SHA-512 is the core cryptographic hash function for:

- transaction hashes,
- Merkle roots,
- block hashes,
- consensus parameter hashing,
- address derivation,
- and various protocol-level identifiers (e.g. NFTs, Data→Token commitments).

Let `Hash(x)` denote SHA-512 applied to the byte string `x`, yielding a 512-bit digest typically represented as 128 hex characters.

4.2.1 Address Derivation

Given a public key pub, the corresponding address is derived as:

- **Address = first 20 bytes (40 hex characters) of SHA-512(pub)**.

This yields short addresses while retaining the security margin of SHA-512.

4.2.2 Transaction Hashes

The transaction hash (`txid`) is the SHA-512 digest of a canonical serialization of its logical content (excluding `hash` and `signature`). This `txid` is used in block inclusion, Merkle trees, and explorers.

4.2.3 Merkle Tree

For a list of transaction hashes `[h1, h2, …, hn]`, the Merkle root is computed in the standard way:

1. If `n = 0`, the root is SHA-512 of the empty byte string.
2. Otherwise, group hashes in pairs; if n is odd, the last hash is duplicated.
3. For each pair `(a, b)`, compute `SHA-512(a || b)`.
4. Repeat until a single hash remains; this is the Merkle root embedded in the block header.

4.2.4 Block Hash

The block hash is derived by hashing a serialization of key header fields together with the miner-provided PoW mix value, for example:

block_hash = SHA-512(PrevHash || Height || Nonce || Mix || MerkleRoot)

This value uniquely identifies the block and links it to its predecessor.

4.2.5 Consensus Parameter Hash

The network's fundamental consensus parameters (network name, decimals, initial reward, halving interval, maximum supply, target block time, retarget window, maximum difficulty step) are serialized into a canonical string and hashed with SHA-512 to produce a **consensus parameter hash**.

This value is stored in chain state and compared on startup. Nodes that do not agree on this hash reject the existing chain unless explicitly forced into a development mode. This mechanism prevents silent consensus parameter divergence.

5. Proof-of-Work

5.1 Mining Objective

Miners receive a “work package” containing:

- the previous block hash,
- the height of the next block,
- the Merkle root of the chosen transaction set,
- the current difficulty (in bits),
- parameters such as `reads_per_try`, block reward preview, and job expiration time.

From this data, miners construct candidate blocks and search for a valid proof-of-work by varying a nonce and computing a **mix** value and corresponding block hash.

5.2 Difficulty Representation

Difficulty is represented as an integer D (1 ≤ D ≤ 62), interpreted as:

The last D bits of a 64-bit value derived from the mix must be zero.

Concretely, given a 64-bit unsigned integer v extracted from the last 8 bytes of the mix, the condition is:

v & ((1 << D) − 1) = 0

If this condition holds, the proof-of-work for the candidate block is valid at difficulty D.

The effective work factor is approximately 2^D, and network hashrate estimators can derive hashes per second as:

H/s ≈ 2^D / average_block_time

5.3 Mix Value

The **mix** is a 512-bit value (commonly represented as a hex string) produced by the miner's PoW kernel. The protocol only enforces:

- that the mix is included verbatim in the block,
- that the difficulty condition on its derived 64-bit value is satisfied.

The exact internal algorithm used by a miner to produce the mix is not prescribed by consensus. A reference miner can implement a memory-hard routine with a configurable number of global memory reads (`reads_per_try`), using SHA-512 and pseudo-random access patterns that favor general-purpose GPUs and constrain the benefits of simple ASIC designs.

Although miners are free to implement any internal kernel, full nodes strictly validate the header, mix and block according to deterministic consensus rules. Any block whose mix or PoW result does not satisfy these rules is rejected. Changing or "optimizing" mining software cannot bypass the required work or alter consensus; it only affects implementation efficiency, not protocol semantics.

5.4 Difficulty Check

Given the mix as a byte array:

1. Take the last 8 bytes as a big-endian 64-bit integer `v`.
2. Construct the bit mask:
    a. if `0 < D < 64`: `mask = (1 << D) - 1`
    b. if `D ≥ 64`: `mask = 2^64 - 1` (all bits set)
    c. if `D ≤ 0`: `mask = 0`
3. Check that `v & mask = 0`. If this condition fails, the block is rejected.

5.5 Block Validation with PoW

Full nodes, upon receiving a candidate block, verify:

1. Chain continuity (previous hash and height).
2. Header fields (difficulty equals current network difficulty, timestamp in an acceptable range).
3. Merkle root matches the transactions included in the block.
4. Block hash is consistent with header, mix, and Merkle root.
5. Difficulty condition over the `mix` value.

6. All transactions are individually valid and consistent with a simulated local state.
7. Coinbase and fees distribution respect monetary rules and the 50M cap.

Only if all checks succeed is the block appended to the chain.

6. Difficulty Retargeting

To maintain a target average block interval `T` (for example, 15 seconds), the system periodically adjusts difficulty based on recent block timestamps.

6.1 LWMA-Style Retargeting

Let:

- `N` be the retarget window (number of blocks considered),
- `T` be the target block time in seconds,
- `MaxDifficultyStep` be the maximum absolute change per adjustment step.

The system uses a **Linearly Weighted Moving Average (LWMA)**:

1. Consider the last `N` solvetime intervals between blocks (or fewer for young chains).
2. Assign higher weights to more recent intervals.
3. Compute an effective average solvetime (LWMA).
4. Compute a ratio `R = T / LWMA`.
5. Convert `R` into a difficulty delta via a log2-like rule, then clamp `delta` between `–MaxDifficultyStep` and `+MaxDifficultyStep`.
6. Set `D_new = clamp(D_old + delta, 1, 64)`.

This approach stabilizes block times while avoiding abrupt jumps in difficulty and limiting the impact of short-term hashrate fluctuations or timestamp manipulation.

7. Monetary Policy: 50 Million Max Supply

7.1 Denominations and Units

`MaxSupplyCoins` is set to **50,000,000.0** coins. With `Decimals` set to `d`, this corresponds to:

MAX_SUPPLY_UNITS = 50,000,000 × 10^d

All state and emission calculations are performed in these atomic units.

7.2 Block Reward Schedule (Halving)

The base reward per block `R(h)`, in units, is defined by a halving schedule:

- For height `h ≤ 0`, `R(h) = 0`.
- For `h > 0`, `R(h) = R0` shifted right by `floor(h / HalvingInterval)` halvings.

Here, `R0` is the initial block reward in units, and `HalvingInterval` is specified in blocks. After a sufficient number of halvings, the block reward effectively becomes zero.

7.3 Hard Cap Enforcement

The chain tracks a cumulative value `TotalMinted` (in units) and enforces:

TotalMinted ≤ MAX_SUPPLY_UNITS

For each new block at height `h`:

- compute the nominal reward `R(h)`,
- compute `remaining = MAX_SUPPLY_UNITS − TotalMinted`,
- set `coinbase = min(R(h), remaining)`.

If `remaining` is zero or negative, the coinbase reward is zero and no additional coins are created, even if the halving schedule is not yet exhausted. This guarantees that the total supply never exceeds 50 million coins.

7.4 Fees

Transaction fees are aggregated per block as the sum of `fee` over all included transactions. Fees are always paid in addition to the block reward and are not subject

to halving. They serve as the long-term incentive for miners once the coinbase reward diminishes.

8. Incentive Layer: Mining, Staking, and Masternodes

8.1 Miner Reward

For each valid block, the miner identified in the header receives a share of:

coinbase + total_fees

In the reference design, the payout is split as follows:

- **80%** of `(coinbase + total_fees)` goes to the block miner,
- **20%** is distributed among active staking validators / masternodes.

This split is implemented in consensus logic and ensures that both hash-power providers and long-term stakers are rewarded.

8.2 Staking and Validator Set

The system maintains a staking state composed of entries for each validator, including:

- `Owner` – the account that provided the stake,
- `Amount` – staked units,
- `SinceHeight` – block height when stake was activated,
- `Active` – whether the stake currently participates in rewards,
- optional `NodeID` and `P2PAddr` – identifying the node's network endpoint.

Stake is created and destroyed via dedicated transaction types:

- `stake` – locks an amount (plus fee) and adds it to the staking pool,
- `unstake` – releases staked amount back into the user's balance and marks the validator as inactive.

The total staked amount is tracked as `TotalStaked`.

8.3 Reward Distribution to Stakers

The masternode share `mnShare` of each block is defined as approximately 20% of the block's total payout:

mnShare = floor((coinbase + total_fees) / 5)

The miner share is the remainder:

minerShare = (coinbase + total_fees) − mnShare

If `TotalStaked > 0` and `mnShare > 0`, each active validator with stake `Si` receives:

reward_i = floor(mnShare × Si / TotalStaked)

This aligns validator incentives with long-term network security while preserving PoW as the primary block production mechanism.

9. Smart Contracts and Programmability

The protocol includes a minimal contract mechanism to support simple yet useful on-chain logic without introducing a complex general-purpose virtual machine.

9.1 Contract Model

Contracts are defined by:

- a unique contract address,
- an owner address,
- a `code` identifier (e.g. `TIME_LOCK`, `ESCROW`, `NFT_COLLECTION`),
- a key-value `storage` (e.g. JSON-like fields such as `unlock_height`, `beneficiary`, `metadata_uri`).

Contracts are deployed and invoked via transaction types:

- `deploy` – creates a new contract with initial storage and funds,
- `call` – executes a contract's logic with the calling transaction as input.

9.2 Example: TIME_LOCK

A `TIME_LOCK` contract contains at least:

- `unlock_height` – the block height at which funds become spendable,
- `beneficiary` – the address that will receive the locked funds.

Execution rule:

- Before `unlock_height`, attempts to transfer value from the contract fail.
- At or after `unlock_height`, the contract allows locked funds to be transferred to `beneficiary`.

This enables use cases such as vesting, delayed payments, and “vault” accounts.

9.3 Extensible Contract Templates

The system is designed to support additional templates, including:

- Multi-signature contracts (requiring multiple signatures for spending),
- Escrow contracts for marketplace scenarios,
- Contracts managing NFT collections and Data→Token reward logic.

These templates extend functionality while keeping the execution environment simple and auditable.

10. Data→Token Protocol: Monetizing Digital Files

The Data→Token module enables users to convert certain types of digital files into coins in a privacy-preserving way. Supported file types may include PDF, DOC, DOCX, TXT, JPEG, PNG, and others.

10.1 High-Level Concept

Instead of storing raw data on-chain, the system records only:

- a cryptographic hash of the file,
- metadata describing the file type and characteristics,
- a scoring value reflecting the estimated quality or usefulness of the data,
- the owner address eligible for rewards.

Based on these inputs, a Data→Token contract can issue coins from a dedicated reward pool.

10.2 Architecture

The architecture separates concerns into:

1. **Data Scoring Engine (off-chain or distributed)**

a. Receives a file from the user (or a hash plus out-of-band verification data).
b. Computes:
    i. a file hash via SHA-512,
    ii. metadata (file type, size, optional domain-specific features),
    iii. a quality score (for example, 0–100).

2. **On-Chain Data→Token Contract**
    a. Accepts submissions containing:
        i. file hash,
        ii. score and metadata,
        iii. sender address.
    b. Enforces rate limits and basic rules (e.g. non-duplication of hashes).
    c. Rewards the sender with a number of coins derived from the score, within defined limits.
3. **Privacy Preservation**
    a. Raw documents are never stored on-chain.
    b. Only hashes and derived metadata are stored, supporting proof-of-existence, auditing, and reward computation without exposing the content itself.

10.3 Anti-Spam and Anti-Sybil Protections

To mitigate abuse, the Data→Token system can use:

- submission rate limits per address,
- penalties or no rewards for invalid or low-quality submissions,
- optional deposits that are refunded for legitimate submissions,
- multiple independent scoring nodes whose reputations are tracked over time.

10.4 AI-Oriented Tokenized Representations

For AI and ML workloads, raw files are often inefficient as training or inference inputs. The Data→Token framework is designed to support highly compressed, tokenized representations of text and images, where a large file (e.g. a 10 MB photo) can be converted into a compact semantic representation (e.g. 0.1 MB) without losing essential information for model training or evaluation.

Such tokenized forms can be stored, transmitted and reused significantly faster and cheaper, while remaining cryptographically linked to the original via hashes and on-chain commitments. This makes the system particularly suitable as a long-term, AI-native data backbone.

11. NFT Support: On-Chain Digital Asset Identity

The system includes native support for non-fungible tokens (NFTs) to represent unique digital assets and rights on-chain.

11.1 NFT Representation

Each NFT can be represented by:

- a unique `token_id`,
- an owner address,
- a contract or collection address (grouping related NFTs),
- a metadata URI or hash (e.g. pointing to off-chain JSON metadata),
- optional royalty or creator fields.

Ownership of NFTs is tracked in the same account-based global state, or via specialized contract storage.

11.2 NFT Operations

Standard NFT operations include:

- `nft_mint` – creation of a new NFT under a collection or contract,
- `nft_transfer` – transfer of ownership from one address to another,
- `nft_burn` – destruction of an NFT.

These may be implemented as specific transaction types or as calls to NFT contract templates.

11.3 Integration with Data→Token

NFTs can be linked to Data→Token submissions in several ways:

- Minting an NFT whose metadata references the hash of a submitted document,
- Using NFTs as proof-of-ownership for high-value or curated data sets,
- Attaching royalties or future revenue sharing mechanisms to NFTs that represent data assets.

This combination allows the system to act both as a currency and as an infrastructure for data provenance and digital content identity.

12. Network, Communication, and Reference Client

12.1 P2P Network

Nodes discover each other via UDP-based local broadcast and maintain a dynamic set of peers. Blocks are propagated using HTTP-based P2P endpoints or equivalent transports:

- When a new block is mined or received, the node validates it, appends it to the chain if valid, and gossips it to peers.
- Peers reciprocate by forwarding valid blocks, forming a mesh network.

12.2 RPC Interface

Full nodes expose HTTP RPC endpoints for:

- retrieving chain statistics and state (height, difficulty, network hashrate, total minted),
- creating and submitting transactions,
- retrieving account balances,
- requesting mining work (`getWork`) and submitting PoW solutions (`submitWork`),
- accessing Data→Token and NFT-related operations.

Rate limiting and IP bans are applied to mitigate denial-of-service attacks. An API key can be required for sensitive endpoints such as `submitWork` and `tx/submit`, especially in smaller networks and early phases.

12.3 Reference Implementation

A reference implementation of the protocol is written in Go and provides:

- a full node with disk-backed state (separate chain files for mainnet and testnet),
- an embedded HTTP server for RPC, statistics, and basic explorer functionality,
- automatic peer discovery via UDP broadcast,
- consensus logic for block and transaction validation and reward computation,
- a minimal smart-contract engine (including `TIME_LOCK` and room for further templates),
- basic structures for staking, NFT management, and Data→Token integration.

Miners may use external GPU kernels (for example, OpenCL-based) that interact with the node via `getWork` and `submitWork`. The consensus rules depend only on the resulting mix and block validity, not on the internal structure of the mining algorithm.

13. Security Considerations

13.1 51% Attacks

As in other PoW systems, an attacker controlling a majority of the network's effective hashrate can:

- build a private chain faster than the public chain,
- perform double-spend attacks by revealing a longer private chain that invalidates previously accepted blocks.

Mitigations are mainly economic and operational:

- distributing hash-power across many independent miners,
- using more confirmations for high-value transactions,
- monitoring for unusual difficulty and timestamp patterns.

13.2 Timestamp Manipulation

Difficulty adjustment uses a window of recent blocks and a weighted average of solving times. Constraints on acceptable timestamps (for example, limiting how far into the future a block timestamp may be relative to local wall-clock time) reduce the impact of timestamp manipulation.

13.3 Cryptographic Assumptions

Security relies on the hardness of:

- forging Ed25519 signatures without the private key,
- producing collisions or preimages in SHA-512.

If these primitives were ever seriously weakened, the protocol could require a hard fork or migration to stronger primitives.

13.4 Staking Layer Risks

The staking layer is an incentive mechanism, not the primary consensus. However:

- centralization of stake could create governance or perception issues,
- poorly tuned parameters might unfairly favor a small number of large validators.

The current design distributes a fixed portion of each block to stakers in proportion to stake, encouraging participation without giving stakers direct authority over PoW validity.

13.5 Data→Token and NFT Risks

The Data→Token and NFT layers introduce additional vectors:

- spam submissions of low-value or harmful data,
- attempts to game scoring algorithms,
- abuse of NFTs for fraudulent representations.

These risks are mitigated by off-chain scoring, reputation systems, rate limiting, transparent algorithms, and, where appropriate, application-level moderation.

14. Roadmap

The roadmap below is indicative and expected to evolve with community input and real-world feedback.

Phase 1 — Public Testnet and Core PoW Layer

Objectives:

- Launch a public testnet with the core protocol:
  - SHA-512 PoW with bitmask difficulty,
  - halving-based emission with a 50M hard cap,
  - staking / masternode reward layer,
  - minimal `TIME_LOCK` contract.
- Release:
  - a reference Go node,
  - a GPU miner (OpenCL) using `getWork` / `submitWork`,
  - RPC endpoints and a lightweight block explorer.

Outcome:

A stable testnet with block production, transaction flow, and staking behavior visible to the community.

Phase 2 — Smart Contracts v1 (Template System)

Objectives:

- Extend the minimal contract engine into Smart Contracts v1, including templates for:
  - time-lock contracts (vesting, delayed payments),
  - escrow contracts (basic marketplace scenarios),
  - multi-signature contracts.
- Define the execution model, error handling, and fee consumption for `deploy` and `call` operations.
- Provide simple SDK tools and documentation for contract developers.

Outcome:

Contract templates available and tested on the testnet, forming the foundation for Data→Token and NFT logic.

Phase 3 — Data→Token: Monetization of Digital Files

Objectives:

- Design and deploy the Data→Token framework that allows users to submit file hashes plus scoring data and receive rewards.
- Implement a Data Scoring Engine (off-chain or distributed) with clear, versioned scoring rules.
- Implement an on-chain Data→Token contract that:
  - tracks submissions,
  - enforces rate limiting and duplication rules,
  - rewards users from a designated pool according to scoring results.
- Preserve privacy by keeping raw files off-chain and committing only hashes and minimal metadata.

Outcome:

A working Data→Token pipeline on the testnet: file → scoring → hash submission → on-chain reward.

Phase 4 — NFT Support and Integration with Data→Token

Objectives:

- Design and implement a native NFT standard on the blockchain.
- Provide contract templates and/or system calls for NFT mint, transfer, and burn.
- Integrate NFTs with the Data→Token system so that high-value or curated data submissions can optionally be represented by NFTs (proof-of-existence, ownership, or licensing).
- Extend the explorer and wallet tools to display NFTs and their relationship to Data→Token entries.

Outcome:

Native NFTs deployed and integrated; creators and data owners can tokenize their assets and link them to Data→Token rewards.

Phase 5 — Deep Decentralization

Objectives:

- Encourage a wide distribution of full nodes and miners through documentation, tooling, and incentives.
- Gradually decentralize the Data Scoring Engine by allowing multiple independent scoring providers with transparent algorithms.
- Introduce a governance process (off-chain or on-chain) for:
    - protocol upgrades,
    - changes to Data→Token scoring rules,
    - adjustments to staking or reward parameters.

Outcome:

A more resilient network with less reliance on any single entity for core services or scoring logic.

Phase 6 — ERC-20 and BTC Compatibility and Conversion Tools

Objectives:

- Finalize, harden and audit the already implemented ERC-20 bridge, and explore additional bridging mechanisms to connect the system to other major external networks:
    - wrapped assets representing BTC or ERC-20 tokens on this chain,
    - preliminary research and, if feasible, implementation of hash time-locked contract (HTLC) based atomic swaps.
- Build user-friendly conversion tools:
    - deposit ERC-20 or BTC to receive wrapped tokens on this chain,
    - burn wrapped tokens to redeem ERC-20 or BTC back on the original chain.
- Provide APIs and reference integrations for wallets and exchanges.

Outcome:

Initial interoperability with Bitcoin and EVM ecosystems, enabling liquidity inflow and cross-chain use cases.

Phase 7 — Mainnet Launch and Ecosystem Growth

Objectives:

- Freeze consensus parameters and launch the mainnet.
- Migrate stable features (Smart Contracts v1, Data→Token, NFT support) from testnet to mainnet.
- Release:
    - a full-featured block explorer,
    - a cross-platform GUI wallet,
    - updated mining tools and developer SDKs.
- Support the growth of an ecosystem of applications:
    - payment and remittance use cases,
    - data marketplaces and NFT-based identity,
    - cross-chain bridges and DeFi integrations where appropriate.

Outcome:

A production-ready blockchain with real-world utility in payments, data monetization, digital identity, and cross-chain value flows.

15. Conclusion

This whitepaper specifies a cryptocurrency and data asset system that combines:

- a SHA-512-based Proof-of-Work blockchain,
- a fixed maximum supply of 50 million coins enforced at consensus level,
- a predictable halving-based emission schedule,
- a hybrid incentive model rewarding both miners and stakers,
- a minimal yet extensible smart contract layer,
- native NFT support,
- and a Data→Token protocol for monetizing digital files and documents in a privacy-conscious way.

By relying on mature cryptographic primitives (Ed25519 and SHA-512) and a conservative, Bitcoin-inspired monetary policy, the system is designed to be understandable, auditable, and relatively straightforward to implement and run. At the same time, extensibility through contracts, NFTs, Data→Token, and cross-chain bridges allows the ecosystem to evolve as real-world usage and community needs grow.

The reference implementation and this document together provide a foundation for developers, miners, node operators, and users to evaluate, implement, and build on top of the protocol.