Nockchain

A Distributed Market for Verifiable Computation

Logan Allen Justin Murphy

April 30, 2026

Abstract. Nockchain is a Proof-of-Work blockchain whose mining produces zero-knowledge proofs of state transitions rather than discarded hashes. Under Zero-Knowledge Proof-of-Work (ZK-PoW) consensus, miners generate STARK proofs over the NockVM, so the energy that secures the chain also subsidizes a global market for verifiable computation. Applications run off-chain as NockApps and settle on the shared base layer through succinct proofs whose verification cost is effectively constant in computation size — decoupling scalability from consensus without fragmenting composability across separate execution layers. The native $NOCK currency is hard-capped at 2^{32} units and minted exclusively through ZK-PoW, preserving Bitcoin's monetary properties with no premine and no founder allocation. Beyond securing the chain, the same incentive structure points toward a general base-layer market for computation: any workload with economic value and an efficient verifiable certificate — matrix multiplication, AI inference, distributed compute — could in principle be incentivized through additional Proof-of-Useful-Work puzzles.

Keywords: Zero-Knowledge Proof-of-Work (ZK-PoW), Nock ISA, Proof of Useful Work, Verifiable Computation, Smart Contract Platforms

1 Useful Proof-of-Work for Zero-Knowledge Proofs

Zero-knowledge proofs make it possible to attest complex computations with short, efficiently verifiable proofs — a capability that runs from Goldwasser, Micali, and Rackoff [8] through modern SNARKs and STARKs. The economic consequence is that verification becomes portable: the truth about what a computer did can be checked by anyone, anywhere, without re-running the computation or seeing the data. High-value interactions such as payments, compliance checks, markets, and critical infrastructure can therefore settle on *proofs of correct behavior* rather than institutional trust, exposing fraud quickly and increasing security.

The first ZK-native blockchains, like Zcash, used these techniques primarily for private transfers, while early SNARK-based rollups on Ethereum repurposed them for scalability by proving batches of transactions off-chain and verifying them on-chain. In both cases, proofs remained auxiliary: miners and validators still earned rewards for maintaining consensus, not for producing proofs themselves. Nockchain instead makes Zero-Knowledge proofs the *primary* economic object

of the base layer, replacing arbitrary hashing with Zero-Knowledge Proof-of-Work (ZKPoW) over NockVM execution traces.[1] Following the model of Kattis et al. [6], the protocol's monetary incentives directly subsidize the global supply of verifiable computation: anyone who can produce proofs gains a permissionless income stream. The NockVM, chosen for its functional purity and stateless architecture, makes this practical: Nock execution is "arithmetized" cheaply enough that proof generation, not just block-finding, can be the work that secures a chain.

The result is a system that maintains the sound monetary properties of Bitcoin—a PoW-secured, hard-capped asset ($NOCK)—but programmable atop a global market for verifiable computation. It serves as a settlement layer where individuals and institutions can clear trades, coordinate across chains and jurisdictions, and transact globally, privately, and non-custodially by posting proofs instead of exposing raw activity. The protocol also introduces a self-securing merge-mining model: app developers can direct their local proving hardware to simultaneously verify their own programs and secure the global chain. As application demand grows, the network's physical security budget expands in lockstep.[2]

Beyond ZK proofs. Nockchain's economic design is, more abstractly, a protocol for incentivizing distributed computation: miners commit hardware to a verifiable workload, and block rewards subsidize whichever task the protocol designates as the puzzle. ZK-PoW over NockVM execution traces is the first instantiation. Nothing in the consensus design forecloses adding further Proof-of-Useful-Work puzzles for other workloads with concrete economic value — recent work on PoUW for arbitrary matrix multiplication [14] points toward AI training and inference as a natural target, where GPU consumers could re-use their MatMul work to claim block rewards. Earlier systems like Bittensor's subnet model gestured at a multi-game market for computational incentives but inherit substantial coordination complexity. The trajectory points toward a single base-layer market for any verifiable computation with concrete economic value.

Three design pillars guide the architecture and the rest of this paper:

- **Programmable Sound Money.** $NOCK is hard-capped and minted exclusively via Proof-of-Useful-Work, preserving Bitcoin's thermodynamic security guarantees while supporting rich programmability through ZK-Intents and NockApps — users declare desired outcomes, and zero-knowledge proofs verify that off-chain execution satisfied those outcomes.[3]
- **Proof-of-Useful-Work.** Rather than burning energy on arbitrary hash puzzles, mining power subsidizes the generation of Zero-Knowledge proofs, bringing proving capacity to market as a tradable resource.

[1]The current implementation mines a proof of a block-bound ZKPoW puzzle and validates transaction effects separately. The planned ZKPoW design is for the mining proof itself to prove the candidate block transition whose commitment it binds.

[2]The shipped ZK-PoW puzzle binds each proof to a fresh per-block subject derived from `(H_prev, nonce)` (Section "Puzzle Definition"), so a single proving operation today does not simultaneously satisfy block eligibility and a NockApp settlement statement. The merge-mining model described here is the planned puzzle redesign in which application proofs themselves contribute to block eligibility.

[3]The `zkp` lockscript opcode that brings off-chain proofs under base-layer verification is planned for a future protocol upgrade.

- **Sovereignty at the Edge.** NockApps run off-chain with their own execution environments and consistency rules, settling on a shared base layer. Merge-mining means that as application demand grows, the network's security budget expands with it rather than competing for it.

1.1 Architecture at a Glance

At its core, Nockchain is defined by a distinct set of primitives and mechanisms:

- **Notes and Transactions (UTXO-based settlement)**: The system state is composed of **Notes** (UTXO-like objects) and **Transactions** (state transitions) that consume and create notes under programmable spending conditions.
- **ZK Proof-of-Work and Heaviest-Proof Chain Rule (consensus)**: Consensus is maintained through a **ZKPoW puzzle** and the **Heaviest-Proof Chain Rule**, which measure network security in terms of "proofpower"—the aggregate computational capacity dedicated to proving correct execution—rather than raw hashpower.
- **Nock ISA and NockVM (verifiable execution)**: Execution relies on the **Nock ISA**, a minimal, axiomatic instruction set, and the **NockVM**, a verifiable virtual machine optimized for zero-knowledge proofs.
- **Nockchain Intent Script and NockApps (programmability)**: **Nockchain Intent Script**, a scripting language for defining spending conditions, specifies the predicates that govern note consumption. Applications are built as off-chain **NockApps**, typically compiled from high-level languages like Hoon or Jock, which execute arbitrary logic off-chain and settle via proofs on the base layer.

2 ZK-Intents and the Limits of On-Chain Execution

Bitcoin demonstrated that a neutral, decentralized ledger could be secured by Proof-of-Work over a simple UTXO state model, yielding censorship-resistant money with strong assurances about ownership and ordering — but minimal programmability. Ethereum generalized this into a "world computer" by embedding a Turing-complete virtual machine in the consensus layer, but at significant cost: every node must re-execute every computation, every intermediate state is public, and execution is tightly coupled to consensus. In both designs, "what happened" is something each chain must recompute locally rather than a portable proof that any system can cheaply verify.

Intents invert the smart-contract model. Users specify *what* outcome they want ("transfer A to B if conditions C are met") and off-chain solvers decide *how* to achieve it, with execution either satisfying the intent in full or not executing at all. Zero-knowledge proofs supercharge this idea: complex logic runs off-chain, and a succinct proof attests that the declared conditions were satisfied. Together, intents and ZK form **ZK-Intents** — unlimited off-chain expressivity, cheap on-chain settlement whose verification cost is effectively constant in computation size, and atomic composability across applications that all settle against the same root of truth. Nockchain realizes this through a two-layer model. **Nockchain Intent Script** is the declarative specification language

for spending conditions. **NockApps** are the off-chain execution environment that processes intents, computes state transitions, and generates the zero-knowledge proofs the base layer verifies.

Nockchain is not the first project to pursue Zero-Knowledge Proof-of-Work, and the precedents inform its design. Aleo initially shipped a pure ZK-PoW consensus, but during its Testnet 2 a single entity dominated the network by exploiting selfish-mining strategies, and the project ultimately transitioned to AleoBFT — a hybrid combining Proof-of-Stake validation with a Proof-of-Work "coinbase puzzle" for proof generation [12]. The lesson is that replacing hash-based puzzles with ZK proofs opens new attack vectors that must be addressed at the protocol-design level. Nockchain's puzzle construction achieves *single-witness hardness* (Section 3.2), the property whose absence underlay those earlier attacks, ensuring that security scales thermodynamically while maintaining decentralization.

3 Nockchain

For consensus analysis, let Δ denote the maximum post-stabilization delay, in seconds, for an honestly propagated block or transaction to reach honest peers via peer-to-peer gossip. Δ is an environmental network bound rather than a consensus constant encoded in blocks. Proof generation is modeled as a miner-local renewal process with nonzero inter-attempt latency drawn from that miner's hardware-and-software latency distribution L_i; the analysis does not treat proof attempts as instantaneous hash trials.

3.1 Nockchain Objects: Notes, Intents, Transactions, Proofs, and Blocks

3.1.1 Notes

The fundamental unit of value in Nockchain is the **note**, which serves as the protocol's analog to Bitcoin's Unspent Transaction Output (UTXO). Unlike account-based systems such as Ethereum, where user balances are stored as mutable state in a global ledger, Nockchain represents ownership as a collection of discrete, immutable notes. This design choice preserves the explicit input-output structure of UTXO-based systems while extending their expressivity through generalized spending conditions.

A note is structurally a versioned record whose four conceptual components—lock, source, assets, and note-data—are bound together by the note's *name*. The name itself encodes the first two:

Note Name. Each note is uniquely identified by a *note name* whose first hash commits to the lock and whose second hash commits to the source. The lock commitment is the Merkle root of the note's spending conditions (described below), while the source commitment cryptographically binds the note to its origin—either the seeds used to derive it in a previous transaction or, in the case of coinbase notes, a commitment to the prior block hash. This two-part identifier ensures that notes are globally unique and their provenance is cryptographically verifiable. Notes also carry a small `version` tag and an `origin-page` (the block height at which the note was created), so the

protocol can evolve the note schema in upgrade-gated phases without losing the ability to reference older notes.

Lock. The lock component specifies the conditions under which the note may be spent. Rather than a single predicate, Nockchain represents locks as Merkle trees of spending conditions, enabling complex, composable authorization logic while maintaining constant-size proofs of spending authority. This structure, known as an *intent tree*, is described in detail in the Intent Script section.

Source. The source field contains a commitment that establishes the note's lineage. For notes created by ordinary transactions, this commitment encodes the seeds from which the note was derived, linking it to its parent transaction. For coinbase notes—those created as block rewards—the source instead commits to the hash of the previous block, anchoring the newly minted value to a specific point in the chain's history.

Assets. The assets field specifies the quantity of value held by the note, denominated in *nicks*. Nockchain's native currency, $NOCK, is subdivided such that one nock equals 2^16 (65,536) nicks. This subdivision provides sufficient granularity for micropayments and precise fee calculations while maintaining human-readable quantities at the nock level. All arithmetic involving value transfers operates on nicks to avoid floating-point imprecision.

Note-Data. Finally, each note may carry a bounded key-value payload called *note-data*: a persistent sorted map from short symbol keys to arbitrary Nock nouns. This field enables notes to store application-specific metadata—token identifiers, timestamps, auxiliary commitments, or even small structured records—without polluting the core monetary structure. Note-data is structurally analogous to Cardano's eUTXO datum but uses Nock's noun model rather than Plutus types. To prevent state bloat, the size of any single output's note-data is capped at a protocol constant `max-size`, measured by the leaf count of its noun-tree representation. When a transaction creates multiple outputs to the same lock, their note-data maps are merged before size accounting and fee calculation, eliminating double-charging for shared payloads.

Nockchain maintains no concept of account balances. A user's "balance" is simply the sum of assets across all notes for which they can satisfy the spending conditions. As in Bitcoin, this gives natural parallelism (disjoint note sets have no ordering dependencies) and avoids persistent account identities, but transparent notes still expose transaction-graph structure unless a higher-level privacy construction hides it.

3.1.2 Nockchain Intent Script

Spending conditions are expressed in **Nockchain Intent Script**, a declarative language for specifying *intents* — formal statements of the outcomes a user requires for a note to be consumed ("spendable if Alice signs, or if Bob reveals a preimage, or if 1000 blocks have passed"). Intent Script is deliberately distinct from the Nock ISA: where Nock is general-purpose and Turing-complete, Intent Script is a small, purpose-built vocabulary of predicates evaluated directly by validators. Each opcode is non-Turing-complete and bounded — a signature check, a hash comparison, a block-height bound — so lock validation terminates in predictable time without gas metering, and each primitive can be audited and formally verified in isolation. Expressivity comes from *composition*: simple opcodes combine through intent trees (below) into sophisticated authorization

logic such as Hash Time-Locked Contracts. Complex application logic that exceeds the script's vocabulary is deferred to NockApps and brought back on-chain through proofs (the planned `zkp` opcode formalizes this bridge).

Opcodes. Intent Script currently supports four fundamental opcodes, each implementing a distinct class of spending condition:

`pkh` (Pay-to-PubKey-Hash). The signature verification opcode commits to a set of N public-key hashes and a threshold m, and is satisfied by *exactly* m valid Schnorr signatures whose pubkeys hash into the committed set. Verification is performed in batch over the Cheetah curve, an elliptic curve defined over a sextic extension of the Goldilocks field. This curve was specifically designed for efficiency in zero-knowledge proofs: its base field structure enables fast native arithmetic and efficient FFT operations within STARK proving systems, making both signature generation and in-circuit verification substantially cheaper than curves like secp256k1. The Cheetah curve provides 128 bits of security.

`hax` (Hashlock). The hashlock opcode commits to a set of Tip5 digests and is satisfied by revealing a preimage—an arbitrary Nock noun, serialized via Nock's `jam` encoding—for each committed digest. Verification reconstructs the noun and checks that hashing it under Tip5 reproduces the committed value. Because preimages are nouns rather than raw bytes, hashlocks can commit to structured data, not just opaque secrets. Hashlocks form the cryptographic foundation of Hash Time-Locked Contracts (HTLCs), enabling trustless atomic swaps across chains and payment channel networks.

`tim` (Timelock). The timelock opcode enforces temporal constraints on when a note may be spent. Each `tim` primitive can carry up to four optional bounds: an absolute minimum and maximum on the spending block height, and a relative minimum and maximum measured from the note's `origin-page` (the block in which the note was created). Absent bounds are vacuously satisfied. Together, these four optional fields express both Bitcoin's CLTV-style "spendable after height X" and CSV-style "spendable N blocks after creation." They also express the upper-bound counterparts — expiring locks — which Bitcoin does not natively provide.

`brn` (Burn). The burn opcode creates provably unspendable notes by specifying a condition that always fails verification.

Future Opcodes. The opcode set will expand as the protocol matures. Planned additions include:

- **`zkp` (Zero-Knowledge Proof Verification):** Validates zero-knowledge proofs of Nock ISA computations, enabling spending conditions that require proof of arbitrary off-chain execution without revealing the computation's inputs.
- **`mrk` (Merkle Path):** Verifies inclusion (or exclusion) within a specified Merkle tree, enabling notes to be conditionally spendable based on membership in off-chain data structures.
- **`cmp` (Comparison):** Performs comparison operations—equality, less-than, greater-than—over values embedded in the transaction or witness, enabling conditional logic based on numerical relationships.

Intent Trees. Intent Trees are Merklized Abstract Syntax Trees (MASTs) of `OR` branches. An intent tree allows a note to specify multiple alternative spending paths, each implemented as a separate lockscript branch. Only one branch must be satisfied to spend the note. When spending, the user reveals only the branch they satisfy, providing a Merkle proof of its inclusion in the tree without exposing the other branches.

This construction offers both privacy—unused spending conditions remain hidden—and efficiency—proof size is logarithmic in the number of branches. A note might specify: "spendable by Alice's signature OR by Bob's signature after 1000 blocks OR by anyone revealing preimage H." The spender satisfies whichever branch applies, revealing only that branch's lockscript.

Intent trees enable sophisticated authorization patterns. A multisignature wallet might combine immediate 3-of-5 spending with a time-delayed 2-of-5 recovery path. An escrow might allow release by mutual agreement (both parties sign), by seller after timeout (timelock), or by arbiter with evidence (hashlock of arbitration decision). Each branch is a composition of opcodes evaluated against the transaction and witness data.

3.1.3 Transactions

A **transaction** in Nockchain is a Nock noun—a binary tree of atoms and cells—that encodes a state transition from one set of notes to another. The choice of Nock as the representation format ensures that transactions are first-class data structures within the protocol's execution environment, enabling uniform treatment by the NockVM.

A **spend** comprises three elements: a *witness* that satisfies the note's lock conditions, *seeds* that deterministically derive the output notes, and a *fee* contribution. The witness varies by lock type —it may be a signature, a preimage, a zero-knowledge proof, or a combination thereof. The seeds are random values that, combined with the transaction structure, produce the output notes in a deterministic but unpredictable manner. The fee is the portion of the input note's value allocated to compensate miners for including the transaction.

The fundamental conservation law of Nockchain transactions is:

$$\sum_{\text{inputs}} \text{assets} = \sum_{\text{outputs}} \text{assets} + \text{fees} \tag{1}$$

Every transaction must satisfy this equation exactly. This conservation rule, enforced at the consensus layer, ensures that $NOCK cannot be created or destroyed except through the controlled issuance of coinbase rewards.

Transaction fees are denominated in *words* — counts of Nock noun-tree nodes — rather than bytes, reflecting Nock's structural sizing rather than a serialization-specific measure. Mirroring Bitcoin's SegWit weight discount, output (seed) words are charged at the full per-word base fee while input (witness) words are charged at one-fourth that rate. The rationale is identical to SegWit's: outputs create new UTXOs that the network must store until they are spent, while witness data is consumed on validation and discarded. The 4× discount on inputs nudges transaction patterns toward consuming UTXOs rather than fragmenting them.

3.1.4 Proofs and Proofpower

A **proof** in Nockchain is a zero-knowledge proof demonstrating that a set of transactions is consistent with the global chain state at a given height. At the architected consensus layer, the mining proof whose hash is tested for difficulty is this block-validity proof, rather than a separate coinbase puzzle. Specifically, the proof attests that:

1. Each transaction in the set is well-formed according to the protocol's syntax rules.
2. All referenced input notes exist in the UTXO set at the specified height.
3. All spending conditions (lockscripts) are satisfied by the provided witnesses.
4. The conservation law holds for each transaction.
5. The resulting UTXO set after applying the transactions is correctly computed.

The zero-knowledge property ensures that the proof can be verified without re-executing the transactions or accessing the witnesses directly.

Proofpower is the Nockchain analog of hashpower in traditional PoW systems. It measures the aggregate computational capacity of the network dedicated to generating valid proofs. Formally, proofpower is the sum of proofs per second produced by all miners across the network. Just as Bitcoin's security derives from the difficulty of out-computing the honest majority's hashpower, Nockchain's security derives from the difficulty of out-proving the honest majority's proofpower.

3.1.5 Blocks and Chain Structure

Nockchain organizes transactions into **blocks**, which serve as the atomic units of consensus. Each block represents a batch of state transitions that the network agrees occurred in a specific order at a specific point in the chain's history.

A block consists of two primary components:

Block Header. The header is a compact summary of the block's contents and position in the chain. It includes a reference to the previous block (the hash of the prior header), a nonce used in the proof-of-work puzzle, cryptographic commitments to the proofs included in the block, and the current difficulty target. The header is designed to be small enough that light clients can download and verify the chain of headers without accessing full block contents.

Transaction Tree. The block body contains a Merklized tree of transactions. This structure enables efficient proofs of transaction inclusion: given a block header, one can prove that a specific transaction belongs to the block by providing a Merkle path from the transaction to the root committed in the header. The tree also supports compact representations of the transaction set.

The canonical chain is selected according to the **heaviest-chain rule**. Rather than counting blocks, Nockchain measures *accumulated proofpower*—the total computational work represented by all proofs in the chain. A chain with fewer blocks but more difficult proofs may be heavier than a chain with more blocks and easier proofs.

This rule, formally expressed as:

$$\text{weight}(\text{chain}) = \sum_{\text{blocks}} \text{difficulty}(\text{block}) \tag{2}$$

ensures that security is measured in terms of the actual computational resources expended. Nodes always prefer the chain with the greatest accumulated proofpower, switching to heavier forks when they appear.

3.2 Zero-Knowledge Proof-of-Work: The Nockchain Puzzle

Nockchain's consensus mechanism replaces the arbitrary hash-based puzzles of classical Proof-of-Work with a structured computational task: generating zero-knowledge proofs over Nock computations. The mining proof is the block-validity proof: the proof whose hash is checked against the target attests to the candidate block transition and is bound to that candidate block's commitment.

3.2.1 From Hashes to ZK Proofs

Replacing hash puzzles with proof generation introduces a critical vulnerability: **amortization**. Hash-based puzzle instances are independent, but proofs are structured, and a naive construction might allow miners to pre-compute portions or reuse intermediate results across instances.

Nockchain addresses this through **single-witness hardness** (formalized below). The puzzle subject is cryptographically derived from the block hash and nonce, so miners cannot share trace-dependent proving work across instances; each puzzle requires fresh per-instance computation.

3.2.2 Puzzle Definition

The ZKPoW puzzle is defined by a deterministic procedure that maps a (candidate block commitment, nonce) pair to a block-validity proof instance, and a verification procedure that checks whether a purported solution is valid.

Subject Construction. The puzzle subject is a balanced binary tree with 64 leaves, each leaf containing an atom (a natural number encoded in the Goldilocks field). The construction proceeds as follows:

1. **Seed derivation**: Concatenate the candidate block commitment C_B with the miner's chosen nonce η to form a seed:
$$\text{seed} = C_B \parallel \eta \tag{3}$$
2. **Field element generation**: Apply the Tip5 hash function iteratively to the seed, extracting 64 Goldilocks field elements $\{e_1, e_2, ..., e_{64}\}$. Tip5 is an algebraic hash function optimized for arithmetic circuits, chosen specifically for its efficiency within zero-knowledge proof systems. Each field element is a member of the Goldilocks field $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$, a prime selected for its compatibility with modern ZK proving systems.
3. **Tree assembly**: Construct a balanced binary tree with the 64 field elements as leaves. The tree structure ensures that the subject has a consistent shape across all puzzle instances, simplifying the arithmetization of the verification circuit.

Proof Generation. Given the subject tree and candidate block commitment, the miner generates a STARK proof over the block-validity formula. The formula checks that the committed transaction set is well formed, consumes existing notes, satisfies the revealed locks, conserves value, computes the resulting notes, and binds the resulting transition to the candidate block commitment. The

NockVM executes this formula, records the execution trace, and produces a STARK proof attesting to correctness. Proof size is polylogarithmic in computation size.

Mining Loop. The miner's workflow proceeds as follows:

1. Assemble a candidate block with transactions, coinbase reward, and previous block hash, then compute its block commitment C_B.
2. Select a nonce η and derive the puzzle subject from (C_B, η).
3. Execute the block-validity formula and generate a STARK proof π of the candidate transition.
4. Compute $\text{Tip5}(\pi)$; if below the difficulty target, broadcast the block; otherwise, increment η and repeat.

Unlike hash-based mining where billions of hashes execute per second, proof generation is computationally intensive—each proof requires seconds.

Difficulty Check. Puzzle validation verifies the block-validity proof, computes the Tip5 hash of the proof, and compares that hash against the difficulty target:

$$\text{Tip5}(\pi) < T_{\text{difficulty}} \tag{4}$$

The proof is deterministic given the puzzle instance, but the hash of the proof is pseudorandom. Miners cannot predict which nonces yield proofs meeting the target without generating them, preserving the randomness of the lottery despite deterministic computation.

Single-Witness Hardness. The puzzle's security rests on a simple property: each nonce produces a fresh execution trace, so prior proving work doesn't transfer to the next attempt. This is the property whose absence broke Aleo's pure ZK-PoW — when miners can amortize proving work across nonces, selfish-mining strategies dominate.

Concretely, the protocol absorbs (c, η) — a block-commitment c and a miner-chosen nonce η — into a Tip5 sponge and expands the digest into 64 Goldilocks field elements that form the subject tree. The block-commitment binds all consensus-relevant block fields (parent block id, transaction set, coinbase, timestamp, epoch counter, target, accumulated work, height, and message field), and the nonce reseeds the subject. Changing η changes the subject, the trace, and ultimately the proof.

Formally: producing a valid (c, η, π) triple whose proof hashes below target requires per-instance computation proportional to a fresh execution of the puzzle formula and a fresh STARK proof. The conjecture decomposes into three claims:

- *Trace uniqueness.* The puzzle formula f is fixed at consensus and the subject s is determined by the puzzle instance. Nock's deterministic reduction yields exactly one valid execution trace $\tau(s, f)$, so there is no shortcut at the level of the computation itself.
- *Proof binding.* The STARK over $\tau(s, f)$ is sound under the assumed FRI parameters and Tip5′s use as the Fiat-Shamir oracle. Any accepted π attests to that trace except with negligible probability; distinct prover instantiations can yield syntactically distinct proofs for the same (s, f), but all attest to the same trace.
- *Per-instance dominant cost.* Some prover setup is reusable across instances — the trace domain is fixed-length, and NTT twiddle-factor tables and FRI commitment-scheme para-

meters do not depend on the per-instance trace. Per-attempt cost is dominated by trace construction over a fresh subject, polynomial commitment, FRI low-degree testing, and query openings, none of which admit a published precomputation strategy.

Tip5′s collision-resistance under the deployed 7-round parameters (cf. [12] for the cryptanalysis informing those parameters) anchors instance unpredictability.

Verification. Proof verification requires $O(\log^2 n)$ operations—polylogarithmic in computation size—enabling verification in milliseconds even for proofs over millions of computational steps.

3.2.3 Difficulty, Block Targets, and Issuance

To maintain stable block times despite fluctuations in network proofpower, Nockchain implements a difficulty adjustment algorithm. The protocol has shipped two regimes, separated by a height-gated upgrade.

Legacy Regime: Bitcoin-Style Epoch Retarget. Through blocks 1 through 65,500, Nockchain used a Bitcoin-style epoch retarget: a 2,016-block window with a $4\times$ clamp on adjustment magnitude. The design inherited Bitcoin's known weaknesses — proofpower swings could leave the network running off-target for up to a full epoch, and the boundary blocks of an epoch were exposed to timestamp-placement attacks that could bias the next-epoch target.

Current Regime: ASERT Per-Block Retarget. At a height-gated upgrade (the *Aletheia* upgrade), the difficulty algorithm switches to **aserti3-2d** (Absolutely Scheduled Exponentially Rising Targets), the per-block exponential adjustment first formalized by Toomim and shipped on Bitcoin Cash. From a fixed *anchor* (h_a, T_a, t_a) captured at activation, the target for any block at height h with parent timestamp t is computed as

$$T(h,t) = T_a \cdot 2^{\,(t-t_a-I\cdot(h-h_a))/\tau} \tag{5}$$

where I is the ideal inter-block interval and τ is the half-life. Because the formula reads only the current block's parent chain (rather than a sampled epoch window), the boundary-sampling attack surface is eliminated, and the response to drift is a smooth exponential rather than a clamped ratio.

Aletheia activates at block height **65,500** and bundles three consensus changes onto a single height: the ASERT switch above, the block-time reduction from 600s to **150s**, and a unified replacement for the issuance schedule (described next). The half-life is scaled to **43,200s (12 hours)** so that the ~288-block stability window of BCH's parameters carries over to the new cadence. The activation anchor target is chosen as 2^{291}, slightly conservative against the ~4× target jump implied by the cadence change. ASERT then converges to the 150s target from that starting point.

Issuance. The original issuance plan front-loaded most of the supply into the chain's first thirty years through a sequence of halvings, leaving little long-run mining subsidy. Aletheia replaces it with a single height-dispatched schedule that runs from genesis through the 2^{32}-nock hard cap:

Phase	Reward (nocks/ block)	Block interval	Notes
Eon 1	65,536	600s	Pre-activation bootstrap
Eon 2	32,768	600s	Pre-activation bootstrap
Eon 3	16,384	600s	Bootstrap concludes at block 65,500
Activation era	2,048	150s	First 6 months post-Aletheia
Decay year 1	1,536	150s	
Decay year 2	1,024	150s	
Decay year 3	768	150s	
Decay year 4	512	150s	
Decay year 5	384	150s	
Decay year 6	256	150s	
Decay year 7	192	150s	
Decay year 8	128	150s	
Decay year 9	96	150s	Decay phase ends at block 2,060,500
Floor	64	150s	~67 years, until block 16,144,876
Post-cap	0	—	Hard cap of 2^{32} nocks reached

The activation transition pairs a $4\times$ increase in block rate with an $8\times$ decrease in per-block reward, so per-time issuance falls by exactly 50%. This converts the activation event into a clean per-time halving without imposing the full 50% per-block revenue cliff that traditional halvings inflict on miners. Each subsequent era boundary is a 25% or 33% drop—not a halving—so a miner displaced at any boundary was already in roughly the bottom 25–33% of the fleet by efficiency rather than the bottom 50%. Because 1 nock equals 2^{16} nicks, the total fixed supply is 2^{48} atomic units, distributed entirely through Proof-of-Useful-Work mining with no premine and no founder allocation.

Reward Distribution. Pre-activation, every coinbase pays a single output to the block miner. Post-activation, every coinbase is split deterministically: 80% of the emission to the miner and 20% to a consensus-known protocol fund address. Both outputs are subject to the same standard coinbase timelock; the fund's share carries no separate unlock schedule. The split is exact for every reward in the post-activation table because each value is a clean multiple of 5. A future upgrade tied to the next-generation proof-of-useful-work puzzle is expected to revert the chain to a 100%-miner coinbase; until then, every post-activation block pays the 80/20 split unconditionally.

Negative Feedback into Cadence. The schedule is defined in *blocks*, not wall-clock time, so the ASERT difficulty adjustment acts as a negative feedback loop on emission rate as well as block latency. If proofpower rises and blocks arrive faster than the ideal interval, the per-block target tightens, restoring the intended cadence and keeping the long-run issuance trajectory anchored to the table above.

3.2.4 Security Arguments

The security of Nockchain's consensus rests on single-witness hardness and the heaviest-chain rule.

Amortization Resistance. Per-attempt prover work depends on the instance-specific trace: trace construction over a fresh subject, polynomial commitment, FRI low-degree testing, and query openings. Instance-independent setup such as NTT twiddle-factor tables, the fixed trace domain, and FRI commitment-scheme parameters is amortizable as one-time precomputation; the dominant per-nonce cost remains the trace-dependent work. Tip5′s collision-resistance over (block-commitment, nonce) inputs prevents miners from finding distinct instances that collide to the same subject.

Genesis Security and Fair Launch. The Nockchain Genesis Block was published on May 21, 2025. The Nockchain Genesis Block includes a hash of Bitcoin block 897767, proving that it could not have been mined at any point earlier, and that there was no pre-mine.

3.3 Code Execution: Nock ISA, NockVM, and NockApp

Execution in Nockchain rests on a vertically integrated stack: the Nock Instruction Set Architecture (ISA) provides the semantic foundation, the Nock Zero-Knowledge Virtual Machine (NockVM) enables efficient proving over Nock computations, and the NockApp framework supplies the developer-facing abstraction for building complex applications.

3.3.1 The Nock ISA

The Nock ISA is the target language for all proofs in Nockchain. Unlike conventional instruction set architectures designed for silicon execution, Nock is a purely functional combinator calculus optimized for formal reasoning, deterministic evaluation, and efficient arithmetization within zero-knowledge proof systems.

The Noun Data Model. All data in Nock is represented as *nouns*. A noun is defined recursively as either an *atom*—an unsigned integer of arbitrary precision—or a *cell*—an ordered pair of nouns. This definition, while minimal, is sufficient to encode any computable data structure: lists become nested cells terminating in a null atom, trees are cells of cells, and even executable code is itself a noun. The uniformity of representation yields a property known as *homoiconicity*: code and data share identical structure, enabling programs to introspect, generate, and transform other programs as naturally as they manipulate data.

Formally, the set of nouns $\mathcal{N}$ is defined as:

$$\mathcal{N} = \mathbb{N} \cup (\mathcal{N} \times \mathcal{N}) \tag{6}$$

where $\mathbb{N}$ denotes the natural numbers (atoms) and $\times$ denotes the Cartesian product (cells). Every noun is either an atom or a cell of two nouns, with no other cases. This recursion provides a universal data format that requires no type annotations, schema definitions, or serialization protocols—a noun is its own canonical representation.

The Twelve Opcodes. Nock computation is defined by a reduction relation over nouns, specified by exactly twelve opcodes. These opcodes are sufficient to express any computable function

while remaining simple enough that the entire specification fits in approximately 4 kilobytes of formal text.[4] The opcodes fall into three categories:

Structural operations navigate and deconstruct nouns. Opcode 0 (slot) extracts a subtree at a given axis address, implementing arbitrary tree traversal in a single instruction. Opcode 1 (constant) returns its argument unchanged, providing literal values.

Computational operations perform the core work of evaluation. Opcode 2 (evaluate) applies a formula to a subject, implementing function application. Opcode 3 (cell test) determines whether a noun is a cell. Opcode 4 (increment) adds one to an atom. Opcode 5 (equality) tests structural equality of two nouns.

Control flow operations enable conditional execution and composition. Opcode 6 (if-then-else) branches based on a boolean test. Opcode 7 (composition) sequences two computations. Opcode 8 (push) extends the subject with a computed value. Opcode 9 (invoke) retrieves and applies a formula from within the subject—the fundamental mechanism for procedure calls. Opcode 10 (replace) edits the subject by patching a subtree at a given axis with a computed value. Opcode 11 (hint) attaches advisory annotations to a computation without affecting Nock semantics; the runtime recognizes specific hint tags to dispatch *jets*—native implementations of frequently-used operations that preserve Nock semantics while bypassing interpretation overhead. (The NockVM additionally exposes an opcode 12 (`scry`) for IO and effect dispatch, but `scry` is a runtime extension outside the proved trace and does not appear in consensus formulas.)

This minimal opcode set achieves Turing completeness. Loops are expressed via recursion (opcode 9), conditionals via branching (opcode 6), and arbitrary data manipulation via tree addressing (opcode 0) and pair construction (implicit in the evaluation rules). The absence of mutable state, explicit memory management, or side effects ensures that Nock evaluation is purely functional: the result of evaluating a formula against a subject depends only on those two nouns, never on external context.

Solid-State Semantics. Nock evaluation is completely determined by the expression itself — no hidden dependencies on interpreter state, system calls, or environment. Two parties who agree on a subject and formula necessarily agree on the result, with no undefined behavior or implementation-dependent variation. This simplifies formal verification: the Nock specification can be encoded directly in proof assistants such as Coq or Agda, so the consensus rules — expressed as Nock formulas — can be machine-checked to satisfy desired properties rather than merely tested.

3.3.2 The NockVM

The Nock Virtual Machine is the proving engine that transforms Nock computations into succinct, verifiable proofs. Built atop the Nock ISA, the ZKVM enables any Nock computation to be proven correct without revealing the computation's inputs, achieving both verifiability and—where desired—privacy.

[4] Nock 4K specification, *Nock — A Definition*, Urbit Developers Reference, https://developers.urbit.org/reference/nock/definition. The full specification of reduction rules and opcodes occupies roughly 4 KB of plain text.

Architecture and Security Level. The NockVM implements a STARK-based (Scalable Transparent Argument of Knowledge) proving system operating over the Goldilocks field $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$. This prime was selected for its exceptional efficiency in modern proof systems: its special structure enables fast arithmetic using native 64-bit operations, and its size provides approximately 121 bits of security against known attacks on the underlying cryptographic assumptions.

The ZKVM achieves *transparency*—proofs can be verified without a trusted setup ceremony, unlike earlier pairing-based systems such as Groth16. This transparency is essential for a permissionless blockchain: there is no toxic waste to protect, no multi-party computation ceremony to coordinate, and no trusted party whose compromise could forge proofs. Anyone can verify proofs using only publicly available information.

Efficiency Relative to Generic ZKVMs. Generic RISC-V ZKVMs must encode the full complexity of a general-purpose ISA: dozens of opcodes, multiple register files, byte-addressable memory with alignment constraints, privilege levels, and interrupt handling. Each feature adds constraints to the proving circuit, so a single RISC-V instruction may require hundreds or thousands of constraints to encode correctly. The NockVM, by contrast, proves execution of a language designed explicitly for efficient arithmetization — twelve opcodes, uniform noun representation, and no memory addressing modes — yielding trace and gate counts similar to custom-ISA circuits and roughly 10× more efficient than RISC-V [19].

Because consensus verification itself is expressed in Nock and proven via the ZKVM, the cost of participating in consensus scales with Nock's efficiency rather than a generic ISA's overhead. The entire consensus logic — note validation, transaction processing, block construction — is representable within the Goldilocks field and provable with practical resource requirements.

The Proving Pipeline. When a miner constructs a block, the NockVM evaluates the consensus formula against the current state plus transactions, recording the full execution trace. That trace is arithmetized into an algebraic intermediate representation (AIR) of polynomial constraints satisfied if and only if the trace is valid Nock execution. The prover commits to the trace polynomials in a Merkle tree and demonstrates that the AIR constraints hold at random points using the FRI (Fast Reed-Solomon Interactive Oracle Proof of Proximity) protocol for low-degree testing. Fiat-Shamir then compiles the resulting interactive protocol into a non-interactive proof. The resulting proof is polylogarithmic in computation size — $O(\log^2 n)$ for n trace steps — so verification cost is effectively independent of computation complexity.

3.3.3 NockApp Runtime and Kernel Model

The NockVM Runtime. NockApps execute on top of the *NockVM*, a Rust runtime that interprets Nock formulas and emits the execution traces from which proofs are produced. The runtime applies *jets* — native implementations of common operations such as cryptographic primitives, big-integer arithmetic, and common data-structure manipulations — that produce exactly the same results as the Nock code they replace but execute orders of magnitude faster. Persistent NockApp state is held in a log-structured merge tree, journaled before commitment for crash safety. A

gRPC interface handles networking, so the same wire protocol carries inter-app communication, Nockchain node interaction, and external client requests.

The Kernel Model. Each NockApp consists of a *kernel*: a Nock state machine compiled from a high-level language, typically Hoon or Jock. The kernel runs on the NockVM via a thin Rust interface (`nockapp`) that mediates pokes, peeks, and effect handling. *Peeks* are read-only queries against the kernel's state tree, answered immediately without consensus or proof generation, and used by clients for balance lookups, transaction history, and similar reads. *Pokes* deliver an event noun to the kernel, which processes it and potentially updates its state; pokes that affect shared state (notably those that create or consume notes on-chain) trigger the runtime to generate a proof and submit it to the network for inclusion in a block. This peek/poke split cleanly separates read and write paths, enabling cached reads, batched writes, and parallel independent pokes.

Because every NockApp — and Nockchain itself — exposes the same gRPC peek/poke interface, NockApps compose uniformly: locally on a single machine, remotely across the network, or with the chain itself, with no special-case code at any boundary. From an application's perspective, Nockchain is simply another service that answers peeks and accepts pokes.

3.3.4 On-Chain versus Off-Chain Execution

In Ethereum and similar platforms, contract code runs on-chain as part of block validation: every node re-executes the contract against current state, metered by gas, and throughput is bounded by the slowest honest validator while inputs and intermediate state are visible to all of them. Scalability under that model requires either accepting these limits or moving execution to Layer 2 systems that fragment liquidity and composability.

Nockchain inverts the relationship. Computation occurs off-chain in NockApps, where developers have access to arbitrary resources and no gas constraints; the results — note consumptions, creations, and state transitions — are then proven via the NockVM, and validators verify the proofs without re-execution. Verification cost is effectively independent of computation size, so a proof attesting to millions of operations costs roughly the same as one attesting to a hundred. Zero-knowledge can hide inputs while proving output correctness, so trades, votes, or model inferences settle on the chain without exposing the underlying data. And because only the proof format is standardized, NockApps choose their own execution environment — local machines, cloud servers, or specialized hardware — rather than conforming to a global VM.

3.3.5 Protocol Evolution and Upgrade Gates

Nockchain's consensus rules are not static. Consensus-affecting changes are introduced through numbered, versioned upgrade specifications that activate at fixed block heights, so every node can deterministically compute which rule set applies to which range of history. Notes, spends, and lockscript witnesses are all versioned at the type level: a lock-merkle-proof, for example, is dispatched on its `version` tag, with older shapes retained so that historical proofs continue to decode and validate.

Three upgrade gates are particularly relevant to the description above:

- **V1 phase.** Introduced version-1 notes and a redesigned spend format that, in the spirit of Bitcoin's SegWit, separates the witness from the spend's structural body. The witness is no longer hashed into the transaction identifier, which both shrinks the consensus-relevant view of a transaction and removes a class of malleability risks.
- **Bythos.** Tightened the lock-Merkle-proof commitment to include the proof axis (so witness hashes commit to *which* branch of the lock tree was satisfied, not just the satisfying primitive) and introduced the 4:1 input-vs-output fee asymmetry described earlier.
- **Aletheia** (block height $65,500$). Bundles three consensus changes onto a single hard fork: the switch from Bitcoin-style epoch retarget to ASERT per-block adjustment, the reduction of the ideal block interval from 600s to 150s, and the replacement of the original 16-halving issuance curve with the unified table described above (including the 80/20 miner-and-protocol-fund coinbase split).

Each upgrade is a separate specification with a defined activation height. Together, they give Nockchain a disciplined evolution path that preserves verifiability across rule changes.

4 Applications

NockApps anchor their state transitions to Nockchain's verifiable global root, so complex off-chain activity — markets, infrastructure control, identity systems, private applications — reduces to succinct claims about what is true at the base layer. This section sketches concrete categories enabled by that pattern.

4.1 Protocol-Driven Finance

Financial protocols today rely on discretionary intermediaries — exchanges, custodians, clearing-houses — that users must trust to follow their published rules. NockApps replace discretion with verifiable code: financial logic runs off-chain and commits net state changes to the base layer as batched note transitions plus a proof.

- **Exchanges.** Lock user assets into NockApp-controlled notes, match trades off-chain, and settle as batched note transitions. Conservation laws hold and no party's balance changes outside the published rules.
- **Lending and credit protocols.** Prove that collateral ratios are respected and liquidations follow policy. Stable-value systems, structured notes, and insurance pools likewise commit to their economics in code, with Nockchain as the canonical record of issued claims and settled obligations.
- **Cross-venue settlement.** Because every venue settles the same notes against the same root under the same rules, an asset cannot be double-spent across venues — arbitrage becomes a race to submit valid proofs.

Composability follows from shared settlement: a user can trade on one NockApp, post proceeds as collateral on another, and hedge on a third — atomically, backed by one proof and one update to the global root. Zero-knowledge proofs and intents thus act as a *ZK transport layer*

for settlement, routing a high-level intent to whichever venue (on-chain DEX, off-chain dark pool, even a traditional custodian) offers the best liquidity.

4.2 Critical Infrastructure and Public Systems

Critical infrastructure — elections, power grids, industrial control, supply chains — demands high assurance and strong accountability for actions that are hard or impossible to reverse. A verifiable global root can serve as an *immutable control and audit layer* for these systems.

- **Voting.** A NockApp posts a commitment to all cast ballots and a proof that the published tally follows from those ballots, without revealing individual votes.
- **Grid and industrial control.** SCADA-like systems accept command streams only with proofs that they satisfy pre-agreed safety predicates, anchoring resulting state commitments on-chain. Post-incident analysis becomes a matter of re-checking proofs rather than trusting logs.
- **Supply chains.** Each transfer of custody or transformation of goods is a note transition annotated in note-data and periodically proven to respect origin, certification, and segregation rules. The chain does not store full logistics data — it gives all parties a cryptographically enforced common story of how an object got here.

4.3 Compliance and Continuous Assurance

Compliance reduces to demonstrating that a complex system of people, processes, and software stays within prescribed bounds. On Nockchain, this becomes the problem of proving that system state always lies inside a policy-defined set and anchoring those proofs to a public root.

- **Solvency proofs.** "All claims are over-collateralized by at least X%" — without revealing customer-level data.
- **Access-control proofs.** "Every privileged action came from an authorized configuration."
- **Reporting proofs.** "These aggregate numbers were computed from underlying records in a policy-compliant way."

Regulators and counterparties verify these against state roots rather than re-running entire systems. NockApps can be packaged as turnkey "compliance as code" modules, so smaller organizations inherit high-assurance behavior without building bespoke infrastructure — and the root of truth becomes a shared backbone for cross-firm workflows like trade settlement, escrow releases, and supply-chain handoffs that are simultaneously private and universally verifiable.

4.4 Identity, Reputation, and Access Control

Identity on Nockchain is not a static profile but a collection of verifiable claims anchored to the root of truth. Credentials — KYC attestations, membership proofs, employment status, certifications — are issued as notes or commitments managed by identity NockApps. Users never reveal the credentials directly; instead, they generate proofs about them.

- **Selective disclosure and predicate proofs.** "I am over 18," "I am a resident of jurisdiction X," "I belong to group Y" — without revealing name, address, or document numbers.

- **Reputation over time.** Earn a reputation note for successful participation, then later prove "I have at least R reputation" to access credit, governance, or higher throughput — without linking actions across contexts.
- **Cross-application interop.** Because every claim reduces to a fact about notes under the same root, one credential satisfies many checks: better market terms, compliance checks, governance participation — each verified by a different NockApp against a single universe of credentials.

4.5 Private and Confidential Applications

Many high-value applications require strong privacy by default: private finance, competitive bidding, medical and enterprise data, ML-powered services with proprietary models. Nockchain supports them through a simple pattern — execute in private, prove in public, settle against a global root.

- **Private payments.** Hide transfer graphs while proving conservation of value and adherence to AML-style constraints.
- **Sealed-bid auctions.** Prove that the winner and clearing price follow from all submitted bids without revealing the losing ones.
- **ML inference.** Prove that a decision followed from a committed model and input without exposing either.

In each case the chain sees only note updates plus a proof; the semantics of those updates live entirely inside the NockApp and its circuits. Users and institutions can mix and match privacy-preserving services — private credit markets, confidential trading venues, data cooperatives — without fragmenting trust, since all of them ultimately settle against the same verifiable root.

4.6 Toward a Market for Verifiable Computation

The applications above all share a pattern: a NockApp executes off-chain, generates a proof, and settles on the base layer. Proofpower is already a tradable resource — miners supply it, applications consume it. Looking forward, the same machinery could host additional Proof-of-Useful-Work puzzles whose underlying computation has direct economic value beyond consensus.

The concrete near-term direction is **matrix multiplication**: recent work [14] constructs a PoUW for arbitrary MatMul with $1 + o(1)$ overhead over naïve matmul, so a future puzzle could simultaneously secure a chain and provide a verifiable settlement layer for AI training and inference workloads. Datacenters and GPU consumers would re-use existing matrix-multiplication work to claim block rewards rather than paying separately for compute and security. Bittensor's subnet model anticipated a multi-game incentive market for computation but pays for that flexibility in coordination complexity; Nockchain's path is to keep consensus simple and let the puzzle space expand cleanly over time. The destination, in either case, is a base-layer market where compute is a first-class commodity — minted through Proof-of-Useful-Work, traded as proofs, and consumed by any application that needs a verifiable certificate of computation.

5 Conclusion

Nockchain is a deliberate synthesis of three architectural insights from fifteen years of blockchain research. Sound money requires thermodynamic objectivity — Proof-of-Work grounds digital scarcity in physical reality in a way no purely virtual staking mechanism can. Verifiable computation decouples execution from consensus — zero-knowledge proofs let validators confirm correctness without re-executing. Sovereignty at the edge preserves composability without centralization — applications execute off-chain on their own terms but settle atomically on a shared base layer. Where Ethereum has evolved toward Proof-of-Stake with modular L2 rollups, Nockchain returns to Proof-of-Work while transforming "useless" hashing into useful proof generation. Where modern chains fragment into dozens of independent sequencers and bridges, Nockchain provides a single settlement layer — $NOCK minted by Zero-Knowledge Proof-of-Work — coordinating a diverse ecosystem of NockApps without trust in intermediate operators.

This design imposes explicit tradeoffs. The base layer is deliberately conservative: block intervals are tuned for proof-generation latency rather than maximum throughput; ZKPoW mining requires substantially more computational effort than traditional hashing; and the Nock ISA's minimalism demands careful engineering to compete with mature VMs like the EVM or WASM. These constraints are intentional, reflecting a bet that the ecosystem's most pressing problems are not throughput ceilings — which off-chain execution addresses — but fragmented liquidity, trusted intermediaries, and the gradual erosion of decentralization under economic pressure toward efficient but centralized infrastructure. Real challenges remain. Proof generation is still expensive, though ongoing research into specialized circuits, recursive composition, and hardware acceleration promises one to two orders of magnitude of improvement. The Nock ISA lacks mature tooling that established platforms enjoy — a gap the Jock language and NockApp framework aim to close. And fair-launch mining faces the same centralization pressures — economies of scale, geographic concentration, hardware specialization — that have shaped Bitcoin's mining landscape.

Several areas invite experimentation:

- **Hardware acceleration** for ZKPoW mining — FPGAs, ASICs, and specialized proving circuits — will determine mining's economic viability and decentralization profile.
- **NockApp ecosystems** will test whether the off-chain execution model supports the diversity of applications that users demand.
- **Bridges and interoperability** will be essential for bootstrapping liquidity.
- **Proving-system improvements** — recursion, aggregation, batching — translate directly into higher throughput.
- **Formal verification** of core protocol components and critical NockApps will provide stronger correctness guarantees than testing alone.
- **New Proof-of-Useful-Work puzzles** could extend mining beyond ZK-PoW: recent work on PoUW for arbitrary matrix multiplication [14] suggests one path — AI training and inference workloads — and the broader trajectory is toward a base-layer market for any verifiable computation with concrete economic value.

Nockchain makes a single bet: that mining and zero-knowledge proof generation can be the same work, so the energy securing the chain also produces the verifiable computation it settles. If that bet holds, the monetary properties of Bitcoin, the verifiable computation of modern ZK research, and the composability of Ethereum collapse into a single coherent architecture. The design has shipped; the code, the network, and the community will determine whether the bet pays out.

6 Notes and Further Reading

6.1 Classic References

1. **Satoshi Nakamoto (2008).** *Bitcoin: A Peer-to-Peer Electronic Cash System.* The foundational whitepaper establishing Proof-of-Work consensus, the UTXO model, and decentralized digital currency without trusted intermediaries.
2. **Vitalik Buterin (2013).** *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* The whitepaper introducing the concept of a Turing-complete blockchain as a "world computer," establishing the foundation for programmable blockchain platforms.
3. **Ralph Merkle (1987).** *A Digital Signature Based on a Conventional Encryption Function.* The original description of Merkle trees, which enable efficient membership proofs and are fundamental to SPV clients and modern blockchain data structures.
4. **David Chaum (1983).** *Blind Signatures for Untraceable Payments.* Early work on privacy-preserving digital payments using cryptographic techniques, a precursor to modern cryptocurrency privacy protocols.
5. **Leslie Lamport, Robert Shostak, Marshall Pease (1982).** *The Byzantine Generals Problem.* Foundational work in distributed consensus, establishing the theoretical framework for achieving agreement in the presence of malicious actors.

6.2 Zero-Knowledge Proofs and Proof-of-Work Research

6. **Kattis, Bonneau, Miller, Clark (2019).** *Proof of Necessary Work: Succinct State Verification with Fairness Guarantees.* Theoretical framework for Proof-of-Useful-Work, proposing consensus mechanisms where mining directly contributes to network functionality.
7. **Eli Ben-Sasson et al. (2018).** *Scalable, transparent, and post-quantum secure computational integrity.* The STARK (Scalable Transparent ARgument of Knowledge) paper establishing transparent zero-knowledge proofs without trusted setup, foundational to Nockchain's proof system.
8. **Shafi Goldwasser, Silvio Micali, Charles Rackoff (1985).** *The Knowledge Complexity of Interactive Proof Systems.* The original zero-knowledge proof paper, establishing the theoretical foundations for proving computational correctness without revealing witnesses.

9. **Andrew Miller, LaViola, Katz (2014).** *Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin.* Analysis of Nakamoto consensus as a mechanism for anonymous leader election in permissionless networks.
10. **Robin Salen, Vijaykumar Singh, Vladimir Soukharev (2022).** *Security Analysis of Elliptic Curves over Sextic Extension of Small Prime Fields.* Cryptology ePrint Archive, Paper 2022/277. Security analysis of the Cheetah curve, an elliptic curve defined over a sextic extension of the Goldilocks field, designed for efficient zero-knowledge proof operations.
11. **Alan Szepieniec, Alexander Lemmens, Jan Ferdinand Sauer, Bobbin Threadbare, Al-Kindi (2023).** *The Tip5 Hash Function for Recursive STARKs.* Cryptology ePrint Archive, Paper 2023/107. Design and analysis of an algebraic hash function optimized for arithmetization in recursive STARK proving systems over the Goldilocks field.
12. **Fukang Liu, Katharina Koschatko, Lorenzo Grassi, Hailun Yan, Shiyao Chen, Subhadeep Banik, Willi Meier (2024).** *Opening the Blackbox: Collision Attacks on Round-Reduced Tip5, Tip4, Tip4′ and Monolith.* Cryptology ePrint Archive, Paper 2024/1900. Cryptanalysis demonstrating collision attacks on round-reduced variants of Tip5, informing the decision to use 7 rounds for enhanced security margins.
13. **Kattis, Klatt, Quirk, Allen (2025).** *A Framework for Compiling Custom Languages as Efficiently Verifiable Virtual Machines.* Cryptology ePrint Archive, Paper 2025/1110. Formalizes the Nock ISA as a ZKVM target and reports trace and gate counts for NockVM relative to custom-ISA and RISC-V ZKVM designs.
14. **Ilan Komargodski, Omri Weinstein (2025).** *Proofs of Useful Work from Arbitrary Matrix Multiplication.* Cryptology ePrint Archive, Paper 2025/685. Constructs a Proof-of-Useful-Work (cuPoW) for arbitrary matrix multiplication with $1 + o(1)$ multiplicative overhead over naïve MatMul, enabling miners to re-use their MatMul work for AI training and inference while securing a chain.

6.3 Nockchain Protocol Documentation

The following documentation pages are maintained by Zorp, the company that developed the initial Nockchain protocol.

15. **The Nock Ecosystem.** Overview of the Nock programming language, its origins in the Urbit operating system, and its application as a minimal instruction set for verifiable computation.
16. **What is Nockchain?** Introduction to Nockchain's core concepts, design philosophy, and relationship to existing blockchain systems.
17. **Architecture.** Detailed specification of Nockchain's consensus layer, including the ZKPoW puzzle, block structure, transaction model, and proof verification.
18. **Transactions.** Technical documentation of the note model, intent-based lockscripts, transaction structure, and spending conditions.

19. **Zero-Knowledge Proofs.** Explanation of how ZK proofs are generated, verified, and integrated into the consensus mechanism, including proof composition and batching.
20. **What is $NOCK?** Documentation of Nockchain's native currency, including monetary policy, issuance schedule, and economic properties.
21. **What is NockApp?** Overview of the off-chain application framework, including execution model, proof generation, and settlement on the base layer.
22. **What is Nock ISA?** Technical specification of the Nock instruction set architecture, including operational semantics and examples.
23. **Features.** Summary of Nockchain's distinctive features, including intent-based transactions, note model, and verifiable computation.

6.4 Zorp Research and Essays

The following materials are Zorp-authored or Zorp-published discussions of Nockchain's design, economics, and roadmap.

24. **The Nockchain ZK-PoW Puzzle.** Detailed explanation of the zero-knowledge proof-of-work puzzle, including difficulty adjustment, proofpower measurement, and mining economics.
25. **Making a New PoW Chain: The Hard Parts.** Candid discussion of the challenges in launching a new Proof-of-Work blockchain, including network bootstrapping, mining decentralization, and protocol stability.
26. **NockApp Framework: Developer Alpha.** Introduction to the NockApp development framework, including the Jock programming language, the `nockapp` kernel interface, the NockVM runtime, and developer tooling.
27. **The Nockchain Consensus Model.** Deep dive into the consensus mechanism, including the heaviest-proof-chain rule, block propagation, and finality guarantees.

6.5 Implementation Resources

28. **Nockchain Protocol Monorepo.** The reference implementation of the Nockchain protocol in Rust, including the consensus engine, transaction validation, proof verification, and network layer.
29. **Wallet Implementation.** Reference wallet software for managing $NOCK, creating transactions, and interacting with NockApps (repository links available through documentation portal).
30. **Miner Implementation.** Open-source reference miner for ZKPoW mining, including GPU acceleration and pool mining support (repository links available through documentation portal).
31. **NockApp SDK.** Development toolkit for building NockApps, including the Jock compiler, proof generation libraries, and testing infrastructure (repository links available through documentation portal).