# Transformer-Based Proof-of-Work: Aligning Voting Power with LLM Computation

Tamaz Gadaev, David Liberman, Anastasia Matveeva, Gleb Morgachev, Egor Shulgin[*]

**Abstract**

We propose a novel transformer-based proof-of-work mechanism for decentralized AI networks hosting Large Language Models (LLMs). The system addresses the challenge of accurately assessing participants'computational resources by requiring them to perform inference on randomly initialized transformers, thereby aligning voting power with the actual computational tasks participants will execute during LLM operations. We provide a formal specification of the protocol and analyze its security properties, ensuring honest participation in the decentralized network.

## Contents

## 1 Transformer-Based Proof-of-Work

For non-technical readers, Subsection 1.1 provides sufficient conceptual understanding. Subsections 1.2 and 1.3, along with the appendices, formalize the technical procedure. Section 2.1 examines security properties, potential attack vectors, and defense mechanisms. Readers seeking only the conceptual overview may proceed directly from Subsection 1.1 to the conclusion.

### 1.1 High-Level Summary

The proposed Proof-of-Work (PoW) mechanism is designed to leverage the computational characteristics of neural network architectures, effectively aligning voting power with the actual workload of the node, hosting a Large Language Model (LLM) in a decentralized AI network. This approach creates a system where participants are incentivized to optimize their hardware and software specifically for LLM computations.

[*]Gonka AI Team, Authors are listed in alphabetical order.

**Core Mechanism.** The PoW algorithm is structured to closely resemble the computational capacity required for LLM inference and training. Each node executes a parametrized function comprising computational blocks typical to LLM architecture—specifically a Transformer neural network. The computational capacity of a node is demonstrated through the number of vectors found during a "Sprint" that satisfy a specific mathematical condition. A proof of computational capacity is then sent to the network and validated by other nodes. Voting power in the network is directly proportional to the computational capacity demonstrated during the Sprint, ensuring fair representation based on actual contributed resources.

**Sprint Procedure Overview.** At the start of each Sprint cycle, a sprint seed $\text{seed}_S$ is generated based on the latest blockchain state. This seed initializes the shared parameters of a Transformer-based model with approximately 5.5 billion parameters. Each node generates a unique node seed $\text{seed}_N$ from its public key. During the Sprint, participants iterate over nonce values. Each nonce, combined with $\text{seed}_N$ and $\text{seed}_S$, produces input sequences for the Transformer. The last vector of the output sequence is normalized to unit length and then permuted based on the nonce, $\text{seed}_N$ and $\text{seed}_S$. Participants must find permuted output vectors that are close enough (in Euclidean distance) to a common target vector $V_T$ generated deterministically from the sprint seed and normalized to unit length. Vectors satisfying this condition are termed *Appropriate Vectors*. The number of Appropriate Vectors found within the sprint duration indicates a node's computational capacity.

**Validation Overview.** A proof of computational capacity consists of a set of nonce values. Validators reconstruct the Transformer and target vector using the sprint seed (derived from the blockchain state at sprint time) and derive the prover's node seed from their public key to verify that each submitted nonce indeed produces an Appropriate Vector. Voting weight is allocated proportionally to the number of valid proofs.

**Security Focus.** The primary security challenge in this system is preventing *algorithmic shortcut*—methods where an attacker could gain voting power without performing the proportional amount of computational work. The security of our approach is based on the computational infeasibility of finding inputs satisfying the target condition significantly faster than through direct inference.

This paper presents a formalization of the procedure, analyze the theoretical foundations of its security, and demonstrate why the system provides strong protection against potential shortcuts. The security analysis section (Section 2.1) focuses exclusively on the algorithmic security aspects of the proposed PoW mechanism. The key security pillars include:

- The inherent complexity of large, randomly initialized Transformer networks
- Sprint-specific randomization preventing amortized analysis
- Nonce-dependent permutation disrupting Lipschitz continuity for search-based attacks
- High-dimensional geometric properties creating natural search barriers

Through this multilayered approach, we establish a robust verification system where voting power is determined primarily by computational throughput rather than exploitable biases or shortcuts.

## 1.2 System Parameters & Definitions

We define the key parameters and functions used throughout our Transformer-based Proof-of-Work system.

- $H$: Blockchain state hash function (e.g., SHA-256).
- $PRNG$: A cryptographically secure pseudo-random number generator.
- $T$: A function implementing a Transformer. For the specific Transformer model architecture see Appendix A.
- `GetParams`($\text{seed}_S$): Function that generates the parameters for $T$ deterministically from the sprint seed.

- `GetTargetVector`$(\text{seed}_S, \text{seed}_N)$: Function that generates the $d_{\text{emb}}$-dimensional target vector $V_T$ deterministically from the sprint and node seeds.
- `GetPermutation`$(\text{seed}_S, \text{seed}_N, \text{nonce})$: Function that deterministically generates a permutation $P$ of $d_{\text{emb}}$ elements based on the sprint seed, node seed, and nonce.
- `GenerateInputSequence`$(\text{seed}_S, \text{seed}_N, \text{nonce})$: Function that combines the seeds and nonce to produce a sequence of $L_{\text{seq}}$ input tokens (indices from 0 to $V_{\text{size}} - 1$).
- `GetNodeSeed`(PubKey): Function to derive a unique node seed ($\text{seed}_N$) from the node's public key.
- $d_{\text{emb}}$: Dimension of the embedding/output vectors (for the specific value see Appendix A).
- $\tau$: The constant Euclidean distance threshold defining an Appropriate Vector.
- $p_{\text{hit}}$: Target probability of a single random nonce yielding an Appropriate Vector (for the specific value see Appendix A), achieved by setting $\tau$.
- $p_m$: Expected probability of legitimate computational mismatches during validation due to hardware non-determinism in GPU floating-point operations. It is used as the null hypothesis parameter in the binomial test that validates proof authenticity. $p_m$ is a small constant (e.g., $10^{-5}, 10^{-6}$) that is derived empirically.
- $N_{\text{max}}$: Maximum number of nonces allowed in a single submitted proof.
- $\alpha_{\text{sig}}$: Significance level for the statistical hypothesis test used during validation. If the observed mismatch rate is statistically significantly higher than $p_m$ (p-value $< \alpha_{\text{sig}}$), the proof is rejected as potentially fraudulent. For specific values, see Appendix A.

## 1.3 Protocol Specification

**Sprint Setup Phase.** *Performed by all nodes at the start of each sprint $r$.*

At the start of each sprint $r$, all participating nodes derive a common sprint seed ($\text{seed}_S$) from the blockchain state $S_r$ using $PRNG(H(S_r))$. This $\text{seed}_S$ is used to instantiate a unique, shared Transformer model $T$ with specific parameters (via `GetParams`). Sprint seed is also used to generate a common target vector $V_T$ (via `GetTargetVector`).

The key property of this phase is that all nodes independently arrive at the identical model parameters and target vector, creating the same computational challenge for each participant. This ensures a level playing field where the only differentiating factor is computational capacity.

**Proving Phase (Work Generation).** *Performed by a Prover node $i$ during sprint $r$.*

Each prover node $i$ uses its unique node seed ($\text{seed}_N$, derived from its $\text{PubKey}_i$) and the common $\text{seed}_S$. The prover then systematically evaluates different nonce values to find those that produce Appropriate Vectors.

For a given nonce, an Appropriate Vector is found when:

$$d = \|P(V_O) - V_T\|_2 \leq \tau, \tag{1}$$

where

- $V_O$ is the last vector of the output sequence from $T$ applied to an input derived from $\text{seed}_S$, $\text{seed}_N$, and nonce, normalized to unit length
- $P$ is the nonce-dependent permutation of coordinates applied to $V_O$
- $\|\cdot\|_2$ denotes Euclidean distance
- $\tau$ is the distance threshold

The probability $p_{\text{hit}}$ that a random nonce produces an Appropriate Vector is set through the calibration of $\tau$. The expected number of valid nonces found by a prover is proportional to the number of nonces they evaluate, which directly reflects their computational throughput.

This creates a direct correlation between computational power and successful proof generation. The detailed algorithm for this phase is provided in Appendix A.

**Validation Phase.** *Performed by a* ***Verifier*** *node $v$ upon receiving a* `Proof` *from Prover node $i$ for sprint $r$.*

Verifiers receive a prover's Proof (list of $n$ nonces) and public key $\text{PubKey}_i$. The verifier then:

1. Reconstructs the identical sprint parameters ($T$, $V_T$) using $\text{seed}_S = PRNG(H(S_r))$
2. Derives the prover's $\text{seed}_N$ from $\text{PubKey}_i$
3. For each nonce in the Proof, verifies that it produces an Appropriate Vector
4. Applies a statistical test validating proof trustworthiness to account for potential hardware discrepancies due to GPU non-determinism

For validation to succeed, the observed mismatch rate must not be statistically significantly higher than the expected rate due to hardware non-determinism. Full reproduction is not feasible and a small number of computational mismatches are expected due to GPU floating-point non-determinism. However, if the proportion of mismatched proofs is significantly higher than the natural mismatch rate, this indicates the proof batch likely contains fraud.

To detect this, we apply a binomial test with the null hypothesis that the true mismatch rate equals $p_m$. We compute the p-value as the probability of observing $k$ or more mismatches out of $n$ nonces under this null hypothesis:

$$p_{\text{val}} = P(X \geq k | X \sim \text{Binomial}(n, p_m)), \tag{2}$$

where:

- $k$ is the number of mismatches observed
- $n$ is the total number of nonces in the proof
- $p_m$ is the allowed hardware mismatch probability
- $X \sim \text{Binomial}(n, p_m)$ is a binomial random variable
- $\alpha_{\text{sig}}$ is the significance level for rejecting the null hypothesis

This statistical approach balances security with practical resilience to non-deterministic hardware behavior. The full validation algorithm is detailed in Appendix A.

**Voting Power Assignment.** The voting power $W_i$ of node $i$ for decisions related to sprint $r$ is proportional to its validated ValidNonceCount:

$$W_i = \frac{\text{ValidNonceCount}_i}{\sum_j \text{ValidNonceCount}_j} \text{ (for all accepted proofs } j) \tag{3}$$

This proportional allocation ensures that nodes with greater computational capacity (as demonstrated by finding more Appropriate Vectors) receive greater influence in the network, directly aligning voting power with contributed resources.

## 2 Security Analysis

### 2.1 Core Security Challenge: Hardness Against Algorithmic Shortcuts

Cheating in this Proof-of-Work context means gaining voting power without performing the proportional amount of computational work (Transformer inferences). The primary security challenge is preventing **algorithmic shortcuts** where an attacker could find nonces that produce Appropriate Vectors (permuted output vectors satisfying $\|P(V_O) - V_T\|_2 \leq \tau$) faster than by executing the intended PoW algorithm.

The security of this system is based on the **computational infeasibility** of finding inputs satisfying the target condition significantly faster than direct inference. This hardness stems from a combination of theoretical properties of neural networks, high-dimensional geometry, and the nonce-dependent permutation that work together to create a robust verification mechanism.

Our approach relies on three key security mechanisms:

1. **Sprint-Specific Randomization:** The seed$_S$ generates entirely new, random weights for $T$ and a new target vector $V_T$ for each sprint. Any shortcut found for a particular neural network won't work for another one with different weights. Additionally, attackers have only the sprint duration to find shortcuts, as the weights are not known beforehand.

2. **Transformer Mapping:** The specified Transformer ($T$) is a deep and wide non-linear function with billions of parameters (for parameter values see Appendix A). This creates an inherently complex function with stochastic process-like behavior, protecting from shortcuts based on analytical reviersibility of the function (for details see subsection 2.2). The mapping from input space to the target region lacks exploitable structure, requiring exhaustive search through the high-dimensional input space.

3. **Nonce-Dependent Permutation:** The permutation function $P(\text{seed}_S, \text{seed}_N, \text{nonce})$ applied to the output vector $V_O$ creates a crucial defense layer that disrupts multiple potential attack vectors. This permutation changes with each nonce, preventing attackers from exploiting the smoothness and continuity properties of the underlying Transformer function. Even if nearby input vectors produce similar Transformer outputs, their permuted versions will be completely different, defeating gradient-based attacks, local search methods, and any approach that attempts to leverage the continuous nature of neural networks to guide the search process efficiently.

4. **Statistical Validation:** The protocol employs statistical analysis based on expected nonce distribution to detect fraudulent proof submissions and ensure computational authenticity.

These mechanisms collectively ensure that the probability of success $p_{\text{hit}}$ (achieving $d \leq \tau$) is primarily determined by the threshold $\tau$ and not exploitable biases, enforcing fairness by linking success directly to computational throughput. The subsequent sections will elaborate on the theoretical foundations and security assessment of this approach, demonstrating why our Transformer-based PoW provides strong resistance against algorithmic shortcuts.

### 2.2 Theoretical Foundations - Complexity of Randomly Initialized Transformers

The security of our approach relies on the fact that randomly initialized Transformers create functions so complex that their outputs appear essentially random, making it impossible to predict which inputs will produce desired outputs without direct computation. We demonstrate this complexity from three theoretical perspectives that work together to create a robust computational challenge.

**Analogy to Bitcoin PoW.** Classical proof-of-work systems (e.g. Bitcoin) rely on a cryptographic hash (SHA-256) that is *modeled* as a uniformly random oracle: given an input (nonce), its output (digest) is unpredictable and the cheapest known strategy to find a digest in a tiny target set is brute-force evaluation [11]. Our scheme aims to replicate that property with neural network inference rather than bit-wise hashing. A randomly initialized Transformer $T$, together with the nonce-dependent permutation $P$, behaves like a high-dimensional pseudo-random function:

- **Easy forward direction:** evaluating $T$ once costs $\sim\Theta(\text{params})$ flops, analogous to one SHA-256 call.
- **Hard inverse/pre-image search:** the combination of (i) GP–like stochastic behavior, (ii) chaotic sensitivity, (iii) sprint-specific weights, and (iv) nonce-dependent permutation destroys exploitable structure, so the best known way to hit the acceptance cap $\|P(V_O) - V_T\|_2 \leq \tau$ is exhaustive search over nonces.

Thus, just as Bitcoin miners repeatedly hash new nonces, participants in our PoW repeatedly *infer* new inputs; success probability scales purely with raw compute, not clever algebraic shortcuts.

**Neural Networks as Gaussian Processes.** At initialization, wide neural networks have been shown to converge to Gaussian Processes in the infinite-width limit. The work [12] first established this connection for single-layer networks, and [9] extended it to deeper architectures. The paper

[7] demonstrated that randomly initialized neural networks with infinite width behave as Gaussian Processes with specific kernels determined by the network architecture.

For Transformers specifically, [5] proved that randomly initialized attention networks also exhibit this Gaussian Process behavior. This means that predicting the output for specific inputs requires direct computation rather than analytical shortcuts, as the input-output relationship follows a complex stochastic pattern rather than simple deterministic rules.

**Function Complexity and Chaotic Behavior.** Randomly initialized deep neural networks can approximate a vast class of complex functions, as studied in [10] and [16].

This exponential expressivity leads to chaotic dynamics, as demonstrated in [15] which shows that the complex function representations created by deep random networks operate at the "edge of chaos" where small input perturbations lead to large, unpredictable output variations. For two inputs $x$ and $x'$ in $\mathbb{R}^{d_{input}}$, the correlation between their outputs $f_\theta(x), f_\theta(x') \in \mathbb{R}^{d_{output}}$ decreases exponentially with network depth $d$:

$$\mathbb{E}_\theta[f_\theta(x)^\top f_\theta(x')] \approx e^{-\alpha d \|x-x'\|}, \tag{4}$$

where $\theta$ represents the random network parameters and $\alpha > 0$ depends on activation functions and weight initialization. Our Transformer model exhibits this chaotic behavior, making the output landscape highly sensitive to input variations and creating a natural computational barrier against shortcuts exploiting analytic reversibility of $T$ , independent of our permutation defense protecting against exploitation of local reversibility.

**High-Dimensional Geometry and Output Distribution.** Random Matrix Theory (RMT) provides insights into the behavior of randomly initialized neural networks. The works [13, 14] applied RMT to analyze the singular-value spectra of input-output Jacobians in randomly initialized deep networks, showing how network depth and initialization schemes affect gradient flow and dynamical isometry properties.

From a geometric perspective, for a randomly initialized Transformer, the final output vectors $V_O$ tend to distribute across the unit hypersphere in high-dimensional space. While this distribution is not perfectly uniform, it lacks exploitable structure that would allow predicting regions with higher concentration near an arbitrary target vector $V_T$. This property is crucial because it prevents attackers from easily identifying input patterns that are more likely to produce outputs in the target region.

The attention mechanisms in Transformers, even at initialization, further contribute to this complex output distribution. The paper [19] analyzed the behavior of attention heads and found that in randomly initialized Transformers, attention creates complex transformations of the input data.

In high-dimensional spaces, measure concentration phenomena dictate that random vectors are approximately orthogonal with high probability, making it difficult to find vectors in specific directions without exhaustive search. For our acceptance criterion $\|V_P - V_T\|_2 \leq \tau$, the target region forms a small hyperspherical cap on the unit hypersphere. With $\tau$ calibrated for $p_{\text{hit}} \approx 1/900$, finding inputs that map to this small target region through the complex Transformer function requires extensive computational search. Our PoW design ensures that the combination of Transformer complexity and nonce-dependent permutation eliminates exploitable structure that could accelerate this search.

## 2.3 Protection Against Search-Based Attacks

An attacker might attempt to use various optimization and search techniques to efficiently find inputs that produce Appropriate Vectors. Our analysis addresses this threat vector directly by examining both the potential vulnerability and our specific defense mechanism.

Since Transformers are differentiable, an attacker could theoretically compute the gradient of the distance function with respect to the input and perform gradient descent to find regions likely to produce outputs close to the target. This attack bears similarity to techniques used in adversarial machine learning, particularly Projected Gradient Descent (PGD) methods introduced by [8], which efficiently find inputs that produce specific model outputs.

Beyond gradient-based approaches, attackers could employ local search methods that exploit the continuous nature of neural network functions. For example, they might use nearby nonces that produce similar outputs to guide their search, or employ heuristic optimization techniques that leverage the smoothness properties of the underlying Transformer function.

Without additional protection, this optimization approaches could potentially provide a shortcut to finding appropriate vectors. The attacker could implement a differentiable version of the entire computation pipeline and use automatic differentiation to compute:

$$\nabla_{\text{nonce}} \|V_O(\text{nonce}) - V_T\|_2 \tag{5}$$

Then use this gradient information, along with local search strategies, to guide a search process that identifies regions in the input space more likely to produce outputs close to the target, rather than relying on random sampling. The attacker would then focus their nonce search on generating input sequences that fall within these promising regions.

The critical defense against these search-based attacks is our nonce-dependent permutation $P$. Even if nearby nonces produce similar Transformer outputs, their permuted versions will be completely different, defeating gradient-based attacks, local search methods, and any approach that attempts to leverage the differentiable nature of neural networks to guide the search process efficiently. Each nonce creates a fundamentally different permutation, making each evaluation an independent optimization problem and preventing the accumulation of useful gradient information across nonce evaluations.

This effectively creates a discontinuous optimization landscape with respect to the nonce. With $d_{\text{emb}}$ dimensions, there are $d_{\text{emb}}!$ possible permutations, growing rapidly with dimension. Even if the permutation function samples from a smaller subset, the disruption to optimization information remains significant.

This defense approach is conceptually related to gradient masking [1] in adversarial defenses. However, our application is fundamentally different: we're deliberately creating a discontinuous optimization landscape to enforce computational work, rather than attempting to protect a model from adversarial examples.

The permutation ensures that even if an attacker finds promising search directions for one nonce, this information cannot be effectively transferred to guide the search for other nonces. Each nonce essentially creates an independent search problem with its own unique permutation, preventing the accumulation of useful gradient information across different nonce evaluations.

## 3 Conclusion

The Transformer-based Proof-of-Work system addresses a fundamental challenge in decentralized AI networks: aligning voting power with actual computational contribution while preventing algorithmic shortcuts that could undermine the system's fairness.

Our approach draws inspiration from Bitcoin's SHA-256 PoW, where finding inputs that hash to outputs with specific properties requires brute-force search. Similarly, our system requires participants to find inputs that, when processed through a randomly initialized Transformer and nonce-dependent permutation, produce outputs within a small target region. The key difference is that instead of cryptographic hashing, computational work is performed through neural network inference, directly aligning the proof-of-work with the actual workload of Large Language Model operations.

The security of our system rests on four complementary defense mechanisms that work synergistically to prevent shortcuts:

1. **Stochastic Complexity**: The Gaussian Process behavior and chaotic sensitivity of randomly initialized Transformers create unpredictable input-output mappings that resist analytical shortcuts.

2. **High-Dimensional Geometry**: The concentration of measure in high-dimensional spaces and the geometric properties of the target region make exhaustive search the most efficient approach.

3. **Sprint-Specific Randomization**: New random parameters for each sprint prevent amortization of attack strategies across different competitions.

4. **Nonce-Dependent Permutation**: The permutation function disrupts the smooth optimization landscape, preventing gradient-based and local search attacks from exploiting the underlying Transformer's differentiability.

At present, no successful cheating strategies against this combination of defenses are known. However, we acknowledge that the field of neural network analysis and optimization continues to advance rapidly. As new theoretical insights and computational techniques emerge, the security landscape may evolve. Our system is designed to be adaptable: parameters such as the Transformer architecture, permutation function, and threshold values can be adjusted to maintain security against newly discovered attack vectors.

The strength of this approach lies not in any single defense mechanism, but in the synergistic combination of multiple theoretical foundations that would need to be simultaneously overcome for a successful attack. This multilayered security model provides robust protection while maintaining the core objective of fairly distributing voting power based on computational contribution to the network.

## A Parameter Values

The specific Transformer model architecture $T$ is:

- Layers: 32
- Attention Heads: 32 per layer
- Embedding Dimension ($d_{\text{emb}}$): 1024
- Vocabulary Size ($V_{\text{size}}$): 8192
- Feed-Forward Hidden Dimension ($d_{\text{ff}}$): 81920 ($10.0 \times V_{\text{size}}$)
- Sequence Length ($L_{\text{seq}}$): 128
- Total Parameters: $\approx 5.5$ billion
- Significance level ($\alpha_{\text{sig}}$):

Dimension of the embedding/output vectors $d_{\text{emb}} = 512$.

Target probability of a single random nonce yielding an Appropriate Vector $p_{\text{hit}} \approx 1/900$, achieved by setting $\tau$.

## B Detailed Algorithms (Pseudocode Illustrations)

The following pseudocode illustrates the conceptual procedures.

### B.1 Sprint Setup Phase

*Performed by all nodes at the start of each sprint $r$.*

**Algorithm 1** Sprint Setup

```
   Input: Blockchain state $S_r$ at the beginning of sprint $r$.
1: procedure SPRINTSETUP($S_r$)
2:     $\text{seed}_S \leftarrow PRNG(H(S_r))$                         ▷ Calculate Sprint Seed
3:     Params ← GetParams($\text{seed}_S$)                        ▷ Generate Model Parameters
4:     $T$ ← InitializeTransformerWithParams(Params)             ▷ Instantiate Transformer
5:     $V_{T,\text{raw}}$ ← GetTargetVector($\text{seed}_N$)                  ▷ Generate Target Vector
6:     $V_T$ ← Normalize($V_{T,\text{raw}}$)                    ▷ Ensure target vector is unit length
7:     return ($\text{seed}_S, T, V_T$)                             ▷ Shared sprint parameters
8: end procedure
   Output: Shared $\text{seed}_S$, instantiated Transformer model $T$, and shared $V_T$ for the sprint.
```

## B.2 Proving Phase (Work Generation)

*Performed by a **Prover** node $i$ during sprint $r$.*

**Algorithm 2** Proving Phase

```
   Input: $\text{seed}_S, T, V_T$ (from Sprint Setup); Prover's Node Public Key ($\text{PubKey}_i$).
1: procedure GenerateProof($\text{seed}_S, T, V_T, \text{PubKey}_i$)
2:     $\text{seed}_N \leftarrow \text{GetNodeSeed}(\text{PubKey}_i)$
3:     $\text{FoundNonces} \leftarrow []$
4:     $\text{nonce} \leftarrow 0$
5:     while sprint duration permits do
6:         $\text{InputSeq} \leftarrow \text{GenerateInputSequence}(\text{seed}_S, \text{seed}_N, \text{nonce})$
7:         $\text{OutputSequence} \leftarrow T(\text{InputSeq})$                        ▷ Perform Inference
8:         $V_{O,\text{raw}} \leftarrow \text{LastVector}(\text{OutputSequence})$
9:         $V_O \leftarrow \text{Normalize}(V_{O,\text{raw}})$          ▷ Normalize output vector to unit length
10:        $P_{\text{perm}} \leftarrow \text{GetPermutation}(\text{seed}_S, \text{seed}_N, \text{nonce})$          ▷ Get Permutation
11:        $V_P \leftarrow P_{\text{perm}}(V_O)$                                  ▷ Apply Permutation
12:        $d \leftarrow \text{EuclideanDistance}(V_P, V_T)$
13:        if $d \leq \tau$ then
14:            Append nonce to FoundNonces
15:        end if
16:        $\text{nonce} \leftarrow \text{nonce} + 1$
17:    end while
18:    return FoundNonces
19: end procedure
   Output: Proof: The list FoundNonces (submitted with $\text{PubKey}_i$ and reference to sprint $r$).
```

## B.3 Validation Phase

*Performed by a **Verifier** node $v$ upon receiving a `Proof` from Prover node $i$ for sprint $r$.*

**Algorithm 3** Validation Phase

```
    Input: Proof (list: $\{\text{nonce}_1, \ldots, \text{nonce}_n\}$); Prover's $\text{PubKey}_i$; Blockchain state $S_r$.
 1: procedure ValidateProof(Proof, $\text{PubKey}_i, S_r$)
 2:     $\text{seed}_S \leftarrow PRNG(H(S_r))$                              ▷ Reconstruct common Sprint Seed
 3:     $\text{Params} \leftarrow \text{GetParams}(\text{seed}_S)$                   ▷ Reconstruct Model Parameters
 4:     $T \leftarrow \text{InitializeTransformerWithParams}(\text{Params})$          ▷ Re-instantiate Transformer
 5:     $V_{T,\text{raw}} \leftarrow \text{GetTargetVector}(\text{seed}_S)$              ▷ Reconstruct Target Vector
 6:     $V_T \leftarrow \text{Normalize}(V_{T,\text{raw}})$                    ▷ Ensure target vector is unit length
    Note: Verifier must use deterministic computation settings where possible.
 7:     $\text{seed}_N \leftarrow \text{GetNodeSeed}(\text{PubKey}_i)$                   ▷ Derive Prover's Node Seed
 8:     $k \leftarrow 0$                ▷ Initialize counter for nonces failing validation (mismatches)
 9:     $n \leftarrow |\text{Proof}|$                                  ▷ Total nonces in the submitted proof
10:     if $n = 0$ OR $n > N_{\max}$ then
11:         return (Rejected, 0)                                          ▷ Invalid proof size
12:     end if
13:     for each nonce in Proof do
14:         $\text{InputSeq} \leftarrow \text{GenerateInputSequence}(\text{seed}_S, \text{seed}_N, \text{nonce})$
15:         $\text{OutputSequence} \leftarrow T(\text{InputSeq})$
16:         $V_{O,\text{raw}} \leftarrow \text{LastVector}(\text{OutputSequence})$
17:         $V_O \leftarrow \text{Normalize}(V_{O,\text{raw}})$                           ▷ Normalize output vector
18:         $P_{\text{perm}} \leftarrow \text{GetPermutation}(\text{seed}_S, \text{seed}_N, \text{nonce})$
19:         $V_P \leftarrow P_{\text{perm}}(V_O)$
20:         $d \leftarrow \text{EuclideanDistance}(V_P, V_T)$
21:         if $d > \tau$ then
22:             $k \leftarrow k + 1$
23:         end if
24:     end for
25:     $p_{\text{val}} \leftarrow \text{binomial\_test}(k, n, p_m, \text{alternative='greater'})$   ▷ Statistical test for overall proof validity
26:     if $p_{\text{val}} < \alpha_{\text{sig}}$ then
27:         $\text{ValidNonceCount} \leftarrow 0$
28:         return (Rejected, ValidNonceCount)
29:     else
30:         $\text{ValidNonceCount} \leftarrow n - k$
31:         return (Accepted, ValidNonceCount)
32:     end if
33: end procedure
    Output: Decision (Accepted/Rejected); ValidNonceCount.
```

## References

[1] Athalye, A., Carlini, N., & Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *Proceedings of the 35th International Conference on Machine Learning (ICML).* (Cited on page 7)

[2] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303-314. (Not cited.)

[3] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572.* (Not cited.)

[4] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2), 251-257. (Not cited.)

[5] Hron, J., Bahri, Y., Sohl-Dickstein, J., & Poole, B. (2020). Infinite attention: NNGP and NTK for deep attention networks. *Proceedings of the 37th International Conference on Machine Learning (ICML).* (Cited on page 6)

[6] Jacot, A., Gabriel, F., & Hongler, C. (2018). Neural Tangent Kernel: Convergence and Generalization in Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 31. (Not cited.)

[7] Lee, J., Xiao, L., Schoenholz, S. S., Bahri, Y., Sohl-Dickstein, J., & Pennington, J. (2018). Deep Neural Networks as Gaussian Processes. *Proceedings of the 6th International Conference on Learning Representations (ICLR).* (Cited on page 6)

[8] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2017). Towards deep learning models resistant to adversarial attacks. *Proceedings of the 6th International Conference on Learning Representations (ICLR).* (Cited on page 6)

[9] Matthews, A. G. de G., Rowland, M., Hron, J., Turner, R. E., & Ghahramani, Z. (2018). Gaussian Process Behaviour in Wide Deep Neural Networks. *Proceedings of the 6th International Conference on Learning Representations (ICLR).* (Cited on page 5)

[10] Montúfar, G. F., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the number of linear regions of deep neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 27. (Cited on page 6)

[11] Nakamoto, S. (2008). A peer-to-peer electronic cash system." Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf 4.2: 15. (Cited on page 5)

[12] Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer Science & Business Media. (Lecture Notes in Statistics, Vol. 118). (Cited on page 5)

[13] Pennington, J., Schoenholz, S. S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in Neural Information Processing Systems (NeurIPS)*, 30. (Cited on page 6)

[14] Pennington, J., Schoenholz, S. S., & Ganguli, S. (2018). The Emergence of Spectral Universality in Deep Networks. *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS).* (Cited on page 6)

[15] Poole, B., Lahiri, S., Raghu, M., Sohl-Dickstein, J., & Ganguli, S. (2016). Exponential expressivity in deep neural networks through transient chaos. *Advances in Neural Information Processing Systems (NeurIPS)*, 29. (Cited on page 6)

[16] Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., & Sohl-Dickstein, J. (2017). On the expressive power of deep neural networks. *Proceedings of the 34th International Conference on Machine Learning (ICML).* (Cited on page 6)

[17] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2013). Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199.* (Not cited.)

[18] Vershynin, R. (2018). High-dimensional probability: An introduction with applications in data science (Vol. 47). *Cambridge university press.* (Not cited.)

[19] Voita, E., Talbot, D., Moiseev, F., Sennrich, R., & Titov, I. (2019). Analyzing multi-head self-attention: Specialized heads form task-specific graph structures. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL).* (Cited on page 6)

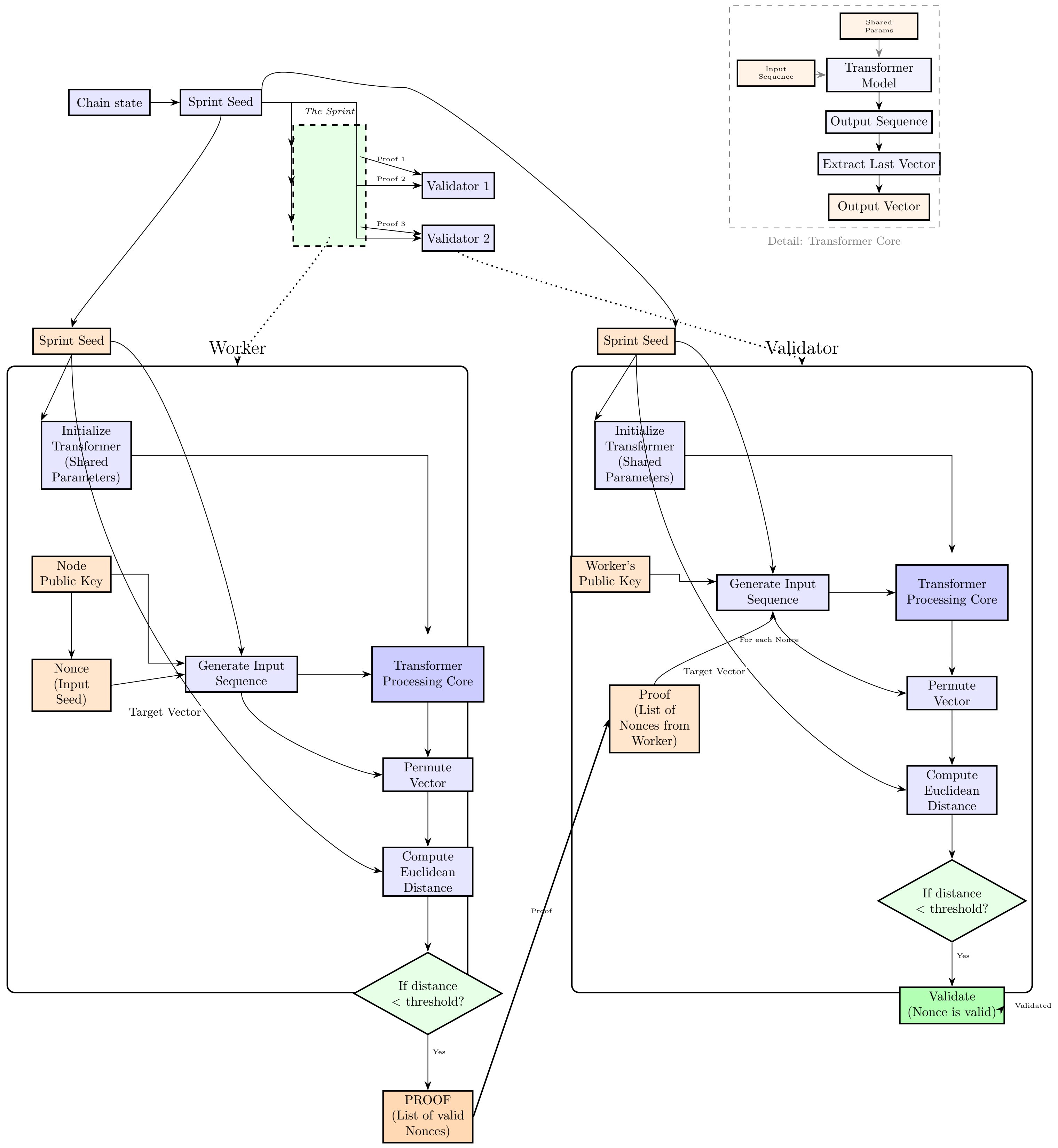

Figure 1: The LLM-based Proof-of-Work scheme illustrating the Sprint, Worker, and Validator processes. The Worker iterates through nonces to find input sequences that, when processed by a Transformer model, result in an output vector close to a target vector after permutation. The Validator verifies these nonces using the same procedure.