Shard Technical White Paper: The Architecture of Decentralized Hybrid Inference

1. Introduction: The Inference Horizon

The contemporary landscape of Artificial Intelligence is defined by a paradox of scale. While the capabilities of Large Language Models (LLMs) scale predictably with parameter count and training tokens, the cost of deploying these models scales prohibitively for ubiquitous access. The centralized paradigm, where massive GPU clusters serve inference requests to thin clients, is approaching an economic and thermodynamic asymptote. The energy required to cool centralized data centers, combined with the quadratic scaling of autoregressive generation costs, necessitates a fundamental architectural rethinking of how intelligence is delivered. Shard (v0.4.0) represents this shift: a transition from monolithic, centralized server farms to a "Hive-Mind" protocol—a decentralized, peer-to-peer (P2P) mesh that leverages the latent compute of the edge to power server-grade intelligence.[1]

This white paper provides an exhaustive technical analysis of the Shard architecture. It details the convergence of three transformative technologies: **1.58-bit Extreme Quantization (BitNet)**, **Hybrid Speculative Decoding**, and **libp2p-based Mesh Networking**. By synthesizing these into a coherent protocol, Shard addresses the "Inference Trilemma"—the difficulty of simultaneously achieving low latency, high quality, and low cost. The system's core philosophy, "Compute is Currency," establishes a resource-backed economy where users pay for intelligence not with fiat currency, but by contributing their idle hardware to the network.[1]

The urgency for such a system is underscored by the physical limitations of current silicon. As Moore's Law slows, the "Memory Wall"—the bottleneck between processing units and memory bandwidth—has become the primary constraint in LLM inference. Centralized providers attempt to mitigate this through massive batching, which introduces queuing latency and degrades the user experience. Shard circumvents the memory wall by distributing the workload. It moves the memory-bandwidth-intensive task of token verification to high-end "Oracle" nodes while offloading the latency-sensitive task of token generation to "Scout" nodes at the edge.[2] This distribution of labor is not merely an optimization; it is a prerequisite for the next order of magnitude in AI scalability.

1.1 The Inefficiency of the Status Quo

Current centralized inference infrastructures suffer from immense redundancy. When a user requests a generation from a 70B parameter model, the server must load 140GB of weights (in FP16) and perform matrix multiplications for every single token, regardless of the token's complexity. Predicting a complex concept like "quantum entanglement" requires the same computational expenditure as predicting the word "the." This monolithic approach wastes

vast amounts of energy on trivial predictions. Furthermore, the latency penalty of round-trip network requests to a centralized data center creates a "laggy" experience for end-users, breaking the illusion of real-time conversation.

Shard addresses these inefficiencies through a heterogeneous architecture. It acknowledges that not all compute is created equal. Consumer devices—laptops, gaming PCs, and modern smartphones—possess powerful GPUs and Neural Processing Units (NPUs) that sit idle for the vast majority of their operational life.[2] By harnessing this "dark matter" of computational power, Shard creates a surplus of inference capacity. The challenge, however, lies in trust. In a decentralized network, no single node can be trusted to return correct data. Shard solves this via a "Verify, Don't Trust" protocol, where lightweight speculative drafts from untrusted edge nodes are rigorous verified by trusted authority nodes.[1]

1.2 Architectural Principles

The Shard codebase and protocol design are guided by three immutable principles codified in the system's agent documentation [1]:

1. **Compute is Currency:** The economic model is resource-based. Participation is incentivized by prioritizing traffic for those who contribute hardware. This creates a positive feedback loop: as demand for inference grows, the incentive to provide compute increases, naturally scaling the network's capacity.
2. **Verify, Don't Trust:** Security is handled at the protocol level. Every token generated by an edge node (Scout) is treated as a "guess" until it is cryptographically verified by a heavier node (Oracle). This effectively neutralizes the threat of malicious nodes injecting noise into the generation stream.
3. **Heavier is Truth:** In the event of algorithmic disagreement—where a Scout's draft differs from the Oracle's verification—the Oracle's output is axiomatically accepted as the ground truth. This ensures that the decentralized nature of the network never compromises the quality of the output; the user always receives 70B-parameter quality, even if the draft was generated by a 1B-parameter model.[1]

2. The 1.58-bit Paradigm Shift

The cornerstone of the Shard Oracle node's efficiency is the adoption of the BitNet b1.58 architecture. This is not a standard compression technique like INT4 or GPTQ; it is a fundamental restructuring of the neural network's mathematical representation. Standard Large Language Models utilize 16-bit floating-point numbers (FP16) for their weights, allowing for 65,536 possible values per parameter. BitNet b1.58 reduces this to a ternary system: every weight must be one of three values: $\{-1, 0, 1\}$.

2.1 Theoretical Underpinnings of Ternary Weights

The nomenclature "1.58-bit" is derived from information theory. The information content of a single parameter that can take on three distinct states is calculated as the base-2 logarithm of 3:

$$\text{Bits per Parameter} = \log_2(3) \approx 1.58496$$
This extreme quantization offers thermodynamic and computational advantages that are critical for distributed inference. In a standard matrix multiplication (MatMul), the primary operation is the fused multiply-add (FMA) of floating-point numbers. This is energy-intensive and consumes significant silicon area. In the BitNet paradigm, because weights are restricted to $\{-1, 0, 1\}$, multiplication is eliminated entirely. The operations are reduced to simple additions (for $+1$) and subtractions (for -1), with 0 acting as a feature filter (no operation).[4]
This reduction in arithmetic complexity fundamentally changes the hardware requirements for the "Oracle" nodes in the Shard network. A 70 billion parameter model, which would traditionally require over 140GB of VRAM and high-end enterprise GPUs (e.g., A100 80GB) to run efficiently, can be compressed to roughly 15-20GB in 1.58-bit precision. This brings the capability to host "Target Models" within the reach of high-end consumer hardware, such as the NVIDIA RTX 3090 or 4090.[4]

2.2 Implementation in Shard Core

The Shard repository integrates this capability via a native C++ engine, shard_engine.dll, which exposes the BitNet functionality to the Python driver. The file desktop/python/bitnet/ctypes_bridge.py serves as the Foreign Function Interface (FFI) layer, managing the memory pointers and execution flow between the high-level Python API and the low-level quantization kernels.[1]
The bridge implementation reveals a meticulous approach to memory management and token handling. The BitNetRuntime class wraps the raw C-types, exposing methods like shard_init, shard_eval, and shard_get_logits. The generate_next_token function, modified in the add_debug.py script, demonstrates the critical interaction loop: logits are retrieved from the engine into a floating-point buffer, and the optimal token index is selected via an argmax operation (or sampling).[1]
Code analysis of ctypes_bridge.py highlights the system's reliance on a custom tokenizer (shard_tokenize) that is tightly coupled with the quantized model. This ensures that the token IDs utilized by the Scout nodes (running distilled models) map correctly to the embedding space of the Oracle nodes (running BitNet models). The eval_text function is particularly significant; it allows the Oracle to ingest a sequence of tokens (the context window) and update the model's internal KV-cache state without necessarily generating new tokens. This "pre-fill" or "evaluation" capability is the mechanism that enables the Oracle to verify a batch of draft tokens in a single parallel forward pass, rather than generating them sequentially.

2.3 Comparative Efficiency Metrics

To contextualize the advantage of the 1.58-bit architecture within the Shard ecosystem, we compare the resource requirements of standard FP16 models against the BitNet implementation used by Shard Oracles.

Metric	Llama-3-70B (FP16)	Llama-3-70B (BitNet	Improvement Factor

		b1.58)	
VRAM Requirement	~140 GB	~16-24 GB	~6-8x
Memory Bandwidth	High (Bottleneck)	Low	Significant
Arithmetic Ops	FP16 Mul-Add	INT8 Add/Sub	Energy Efficient
Hardware Tier	Enterprise (H100/A100)	Consumer High-End (RTX 4090)	Democratized Access
Energy per Token	High	Ultra-Low	~70% reduction [6]

This efficiency is not merely a cost-saving measure; it is an enabling technology. Without 1.58-bit quantization, the "Oracle" role would be restricted to data centers, re-centralizing the network. With BitNet, the network can recruit thousands of prosumer-grade GPUs to act as verification authorities, creating a robust and decentralized trust layer.[7]

3. Hybrid Speculative Decoding: The Cooperative Mesh

The core innovation of Shard's inference strategy is **Hybrid Speculative Decoding**. Traditional speculative decoding accelerates inference on a single machine by using a small "draft" model to guess tokens and a large "target" model to verify them. Shard explodes this concept across the network, decoupling the draft and verification steps spatially and geographically.

3.1 The Scout-Oracle Handshake

In the Shard "Hive-Mind" protocol, the inference workload is split between two distinct node classes: the **Scout** and the **Oracle**.[1]

- **The Scout (Browser Node):** Operating within the constraints of a web browser (using WebGPU via WebLLM), the Scout holds a tiny, distilled model (e.g., Llama-3-1B-Int4). Upon receiving a user prompt, the Scout does not attempt to generate the final high-quality output. Instead, it "rapid-fires" a sequence of 5 to 10 speculative tokens—a "best guess" of what the larger model would say. This process is extremely fast due to the small model size and local execution.[1]
- **The Oracle (Titan Node):** The Oracle receives this batch of draft tokens via the P2P mesh. It does not generate from scratch. Instead, it feeds the user's prompt *and* the Scout's draft tokens into the massive 1.58-bit Target Model. Because the draft tokens are already known, the Oracle can process them in parallel (one single matrix operation for the whole batch) rather than sequentially. This is the "Verification Step."

3.2 Mathematical Verification Criteria

The integrity of the generation is guaranteed by a rigorous mathematical acceptance criterion. The Oracle ensures that the distribution of the final output is identical to what the Target Model would have produced independently. Shard employs a rejection sampling mechanism based on the ratio of probabilities between the Target and Draft models.
Let $p(x)$ be the probability distribution of the Target Model (Oracle) and $q(x)$ be the

distribution of the Draft Model (Scout). For a token x proposed by the Scout, the Oracle calculates an acceptance probability $\alpha(x)$:

$$\alpha(x) = \min\left(1, \frac{p(x)}{q(x)}\right)$$
If $p(x) \ge q(x)$ (the Target Model finds the token at least as likely as the Draft Model did), the token is **accepted**. If $p(x) < q(x)$, the token is accepted probabilistically. If a token is rejected at index i, the Oracle discards all subsequent draft tokens ($i+1, \dots, K$), samples the correct token x'_i from its own distribution $p(x)$, and returns the valid prefix plus the correction to the user.[8]
This mathematical guarantee means that users leveraging the Shard network via a browser-based Scout are receiving text that is **statistically indistinguishable** from the output of a massive server-hosted model, but with the latency profile of a local lightweight model.[10]

3.3 The Cooperative Inference Loop

The file desktop/python/inference.py (as referenced in the audit and context) orchestrates this complex dance. The cooperative_generate function serves as the main event loop. It manages the state of the generation, handling the asynchronous arrival of draft tokens and dispatching them to the local BitNet instance for verification.
The loop implements a "Double-Dip Prevention" mechanism, a critical safeguard for users running both node types on a single machine. Code analysis reveals that the system actively detects if a native .exe client (Oracle) is running on localhost:8080. If detected, the browser client (Scout) forces itself to disable its WebGPU inference engine. This prevents the disastrous scenario where both the heavy BitNet model and the WebGPU engine attempt to allocate the full VRAM of the GPU simultaneously, which would lead to system instability or crashes. Instead, the browser routes requests strictly to the local Oracle, creating a seamless "Localhost" mode.[1]

3.4 Latency and Throughput Dynamics

The Hybrid Speculative approach fundamentally alters the latency characteristics of AI inference.

Inference Step	Latency Source	Mitigation
Draft Generation	Local Compute (WebGPU)	Small model size (1B) ensures ultra-low latency.
Network Transmission	P2P Latency (WAN)	WebRTC direct connections minimize hops; small payload size (tokens are integers).
Verification	Oracle Compute (BitNet)	Parallel verification allows checking K tokens in the time of generating 1.
Correction	Oracle Compute	Occurs only on mismatch; heavily penalized by reputation

		system to discourage bad drafts.

If the Scout's acceptance rate is high (e.g., >80%), the effective throughput of the system multiplies. The Oracle spends most of its time verifying correct tokens in batches, freeing up compute cycles to serve more users. This creates a "force multiplier" effect where adding more edge devices (Scouts) actually increases the total capacity of the central nodes (Oracles) to verify work, solving the scalability crisis of centralized AI.[7]

4. Network Topology: The P2P Fabric

Shard is not merely a collection of isolated endpoints; it is a structured mesh network built on libp2p. This choice allows Shard to be transport-agnostic, resilient to censorship, and capable of traversing complex network topologies (NATs, firewalls).

4.1 Transport Protocols and Multiaddresses

The Shard API documentation (API.md) details the specific transport mechanisms used to bind the heterogeneous node classes together.[1] The system exposes a /v1/system/topology endpoint that returns connection information in multiaddr format—a self-describing network address standard used by IPFS and libp2p.

- **WebRTC (Direct):** This is the critical transport for browser-based Scouts. Browsers cannot open raw TCP sockets. To allow a Scout to connect directly to an Oracle (without a relay server), Shard implements webrtc-direct. The Oracle advertises a multiaddress like /ip4/1.2.3.4/udp/9090/webrtc-direct/p2p/Qm.... This enables low-latency, encrypted, peer-to-peer data channels directly from the browser context to the Rust sidecar on the Oracle.[1]
- **WebSockets (WS):** As a fallback and for compatibility with restrictive corporate firewalls, Oracles also expose WebSocket endpoints (/tcp/4101/ws). This ensures connectivity even when UDP (required for WebRTC) is blocked.
- **TCP:** For Oracle-to-Oracle communication (e.g., gossip propagation, DHT maintenance), raw TCP provides the most stable and performant channel.

4.2 Gossipsub and Discovery

Shard utilizes Gossipsub, a scalable pub/sub protocol, for message dissemination. This is particularly relevant for the "Work Distribution" mechanism. When a user request enters the system (via a Leech or a busy Oracle), it can be broadcast to the shard-work topic. Idle Scouts subscribed to this topic can pick up the work, generate drafts, and submit results back to the shard-work-result topic.[1]

The network topology is maintained via a Kademlia DHT (Distributed Hash Table), allowing nodes to discover peers without a central directory. The rust-sidecar (the daemon running alongside the Python API) manages this DHT, creating a "Peer Store" of active nodes. The API endpoint /v1/system/peers allows the frontend visualizer to query this store and display the

real-time mesh.[1]

4.3 Network Visualizer and Observability

The file web/src/components/NetworkVisualizer.tsx provides a window into this complex topology. It implements a force-directed graph rendering of the local peer mesh. Nodes are color-coded (Green for Local Oracle, Blue for Connected Scouts), and edges represent active libp2p connections.
This visualizer is not just eye candy; it is a debugging tool for the "Pitch Mode" feature. The run_pitch_demo.bat script enables a simulation mode where "Bot" nodes can be spawned and killed to demonstrate the network's resilience. When a node is killed, the visualizer updates instantly, and the inference.py backend logs rerouting events ("Rerouting to Node..."). This demonstrates the network's self-healing capability: if a Scout goes offline mid-generation, the Oracle seamlessly switches to another peer or falls back to local generation, ensuring zero downtime for the user.[1]

5. Security and Incentive Mechanics

In any decentralized resource network, the primary threat is the Sybil attack: a malicious actor creating thousands of fake identities to flood the network with bad data or claim unearned rewards. Shard counters this with a rigorous "Proof of Inference" (PoI) protocol.

5.1 The Golden Ticket Protocol

The "Golden Ticket" is Shard's mechanism for auditing the honesty of untrusted nodes. An Oracle will periodically—and randomly—inject a "Golden Ticket" into the work queue sent to a Scout. A Golden Ticket is a prompt for which the Oracle **already knows the answer** (pre-solved).[1]

- **The Trap:** The Scout receives the Golden Ticket indistinguishable from a normal user prompt. It processes it and returns the draft tokens.
- **The Audit:** The Oracle compares the Scout's draft against the known correct answer.
- **The Consequence:**
    - If the draft matches the expected output (within the bounds of the model's temperature/stochasticity), the Scout is verified as "Honest."
    - If the draft is random noise or incorrect (indicating the Scout is faking work to save power), the Scout fails the audit.
- **The Penalty:** Failure of a Golden Ticket results in an immediate ban. This raises the cost of attack significantly: to pass the Golden Ticket checks, a Sybil node must actually perform the inference work (load the model, compute the tokens). Since they must do the work to avoid being banned, the economic incentive to cheat is eliminated.[1][2]

5.2 Reputation and Slashing

Beyond binary banning, Shard implements a granular reputation system. Every valid generation contributes to a Scout's score. The Oracle tracks the acceptance rate (α) of the drafts provided by each peer.

- **High Reputation:** Scouts with high acceptance rates are prioritized for work allocation, maximizing their potential to earn "Free Tier" credits.
- **Slashing:** When an Oracle rejects a batch of tokens due to poor quality (even if not a malicious Golden Ticket failure), the Scout's reputation is "slashed." Repeated poor performance leads to deprioritization and eventual soft-banning. This ensures that only capable hardware and well-configured nodes remain in the active pool.[1]

5.3 The Leech Economy

The "Leech" class (Class C) represents the demand side of the economy. These users contribute nothing and are thus subject to strict traffic shaping. The API documentation notes that Leeches are "Only served when Swarm Load < 60%".[1] This scarcity creates a powerful funnel: to get reliable, fast access, a user is incentivized to "Turn on Scout Mode," activating their own hardware and contributing to the supply side. This mechanic ensures the network naturally balances supply and demand; high demand creates congestion for Leeches, which drives more users to become Scouts, which increases supply.[13]

6. Implementation and Operational Readiness

The Shard v0.4.0 codebase demonstrates a system transitioning from prototype to production. The architecture is modular, separating the high-level application logic from the low-level networking and inference engines.

6.1 The Application Stack

- **Frontend (Web):** Built with Next.js 14, the frontend is more than a UI; it is an active participant in the mesh. It manages the WebLLM runtime, handles the js-libp2p connection, and creates the visual feedback loop for the user. The ChatPanel.tsx component handles the streaming response, parsing Server-Sent Events (SSE) from the Oracle API.[1]
- **Oracle API (Python):** The oracle_api.py is a FastAPI application that serves as the driver. It adheres strictly to the OpenAI API specification (/v1/chat/completions), making Shard a drop-in replacement for existing tools like LangChain or AutoGPT. It orchestrates the inference.py loop and communicates with the Rust sidecar via local HTTP calls.
- **Rust Sidecar (Daemon):** The shard-daemon (compiled from desktop/rust) handles the heavy lifting of P2P networking. It manages the DHT, maintains persistent connections via TCP/WS, and handles the cryptographic handshakes required by libp2p. Decoupling this from Python ensures that the node remains part of the mesh even if the Python inference thread is blocking or crashes.[1]

6.2 Deployment and CI/CD

The project emphasizes reproducibility and ease of deployment. The .github/workflows/global_release.yml file defines a matrix build strategy for Windows, Linux, and macOS.

- **Windows:** Builds a single .exe installer using PyInstaller.
- **Linux:** Generates an AppImage for universal distribution.
- **macOS:** Creates a .dmg artifact. This automated pipeline ensures that the "Oracle" software is accessible to a wide range of users, maximizing the potential node count. Furthermore, the crash_handler.py module includes an auto-updater that polls GitHub releases, allowing the network to push protocol upgrades and security patches to the decentralized swarm rapidly.[1]

6.3 Audit and Limitations

An internal audit (AUDIT_REPORT.md) acknowledges that while the architecture is sound, the current release (v0.4.0) has specific limitations:

- **Scaffolding:** Some inference paths in oracle_api.py still use scaffolded tokens for testing rather than full BitNet generation in all edge cases.
- **Reproducibility:** There are known issues with npm ci due to lockfile mismatches, and the build system requires internet access (fragile behind proxies).
- **Hardening:** The verifier logic is currently permissive. Future iterations must tighten the acceptance thresholds for Golden Tickets to ensure robust security against sophisticated adversarial attacks.[1]

7. Strategic Implications and Roadmap

Shard represents a disruptive force in the AI market. By shifting the cost structure from OPEX (paying AWS/Azure for GPU hours) to CAPEX (users buying their own hardware), it drastically lowers the barrier to entry for AI deployment. It creates a "Resource-as-a-Service" model that competes directly with the SaaS models of OpenAI or Anthropic.

7.1 Comparison with Centralized Cloud

Feature	Centralized Cloud (SaaS)	Shard Network (Decentralized)
Cost Model	Pay-per-token (Fiat)	Compute-for-access (Barter)
Scalability	Linear (buy more GPUs)	Organic (more users = more GPUs)
Privacy	Trusted Third Party	Trustless / Localhost Routing
Latency	Network RTT + Queue	Local Generation + Verification
Resilience	Single Point of Failure	Mesh / Self-Healing

7.2 Future Roadmap

Based on the codebase analysis and architectural documents, the roadmap for Shard focuses on hardening the protocol for mass adoption:

1. **Reputation Persistence:** Moving reputation scores on-chain or to a distributed ledger to make them portable and immutable.
2. **Model Diversity:** Expanding the v1/models endpoint to support architectures beyond

shard-hybrid (e.g., Mistral, Phi).

3. **Mobile Scouts:** Optimizing the WebLLM stack for mobile browsers to tap into the massive unused compute of global smartphone networks.
4. **Privacy:** Implementing Zero-Knowledge Proofs (ZKPs) or Trusted Execution Environments (TEEs) to allow Scouts to process prompts without seeing the plaintext user data.[13]

8. Conclusion

Shard is not merely a piece of software; it is an economic and technical proof-of-concept for the post-cloud era of AI. By combining the extreme efficiency of **BitNet 1.58-bit models** with the latency-hiding properties of **Hybrid Speculative Decoding** and the resilience of **libp2p**, Shard solves the fundamental bottlenecks of centralized inference.

The system demonstrates that "server-grade" intelligence does not require a server; it requires a protocol. It proves that the billions of idle consumer GPUs scattered across the globe are not waste, but a dormant supercomputer waiting to be networked. As the demand for AI inference outstrips the physical capacity of data centers, Shard offers a scalable, sustainable, and democratized alternative.

For technical acquirers and investors, Shard presents a functioning implementation of "DePIN" (Decentralized Physical Infrastructure Networks) applied to the highest-value commodity of the 21st century: Intelligence. The v0.4.0 codebase is a stable foundation, with a clear path toward a global, permissionless inference market.

Report synthesized from the Shard v0.4.0 Source Code and Technical Documentation.

Works cited

1. shard_context_clean.txt
2. Client-side AI for Cost-Effective, Secure Web Apps - Grid Dynamics, accessed February 13, 2026, https://www.griddynamics.com/blog/client-side-ai
3. Why On-Device AI Is the Future of Inference - Silex Technology, accessed February 13, 2026, https://www.silextechnology.com/platform-and-som-knowledge-pool/why-on-device-ai-is-the-future-of-inference
4. Fine-tuning LLMs to 1.58bit: extreme quantization made easy - Hugging Face, accessed February 13, 2026, https://huggingface.co/blog/1_58_llm_extreme_quantization
5. They've open-sourced bitnet.cpp: a blazing-fast 1-bit LLM inference framework that runs directly on CPUs - Reddit, accessed February 13, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1g7fory/theyve_opensourced_bitnetcpp_a_blazingfast_1bit/
6. Bitnet.cpp: revolution in local LLMs with BitNet 1-bit, b1.58, and a4.8 | by Antonio Troise, accessed February 13, 2026, https://levysoft.medium.com/bitnet-cpp-revolution-in-local-llms-with-bitnet-1-bi

t-b1-58-and-a4-8-6db7cf57089e
7. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits - arXiv, accessed February 13, 2026, https://arxiv.org/html/2402.17764v1
8. Speculative Decoding Math: Algorithms & Speedup Limits - Michael Brenndoerfer, accessed February 13, 2026, https://mbrenndoerfer.com/writing/speculative-decoding-math-acceptance-criterion
9. The Disparate Impacts of Speculative Decoding - arXiv, accessed February 13, 2026, https://arxiv.org/html/2510.02128v1
10. NeurIPS Poster A Theoretical Perspective for Speculative Decoding Algorithm, accessed February 13, 2026, https://neurips.cc/virtual/2024/poster/93154
11. Human Challenge Oracle: Designing AI-Resistant, Identity-Bound, Time-Limited Tasks for Sybil-Resistant Consensus - arXiv, accessed February 13, 2026, https://arxiv.org/html/2601.03923v1
12. Sybil-resilient Publisher Selection Mechanism in Blockchain-based MCS Systems - IEEE Xplore, accessed February 13, 2026, https://ieeexplore.ieee.org/iel8/8782664/9024218/10979902.pdf
13. A Framework for Sybil Attack Prevention in Decentralized Renewable Energy Marketplace - Halmstad University, accessed February 13, 2026, https://hh.diva-portal.org/smash/get/diva2:1976472/FULLTEXT01.pdf