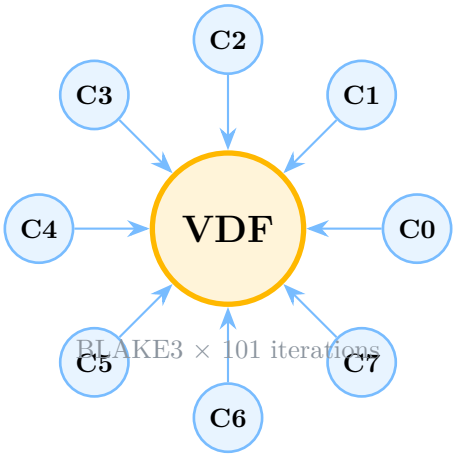
Q-NarwhalKnight Miner

Architecture, Performance Optimizations & VDF-Hardened Proof of Work



Version 2.3.0 — Mainnet 2026.2 Edition

Quantum-DAG Labs Core Contributors

February 19, 2026

"To mine is to secure the future. To optimize is to respect the miner's time."

Abstract: This paper presents the architecture and performance engineering of the Q-NarwhalKnight miner, a high-performance CPU mining implementation using BLAKE3 with Verifiable Delay Function (VDF) hardening. We detail the zero-allocation hot loop design, SIMD-accelerated hashing pipeline, NUMA-aware thread pinning, and a suite of compiler-level optimizations that collectively achieve 35–85% throughput improvement over naive implementations. The miner supports three operating modes: solo mining with SSE-driven challenge updates, Stratum-compatible pool mining, and decentralized P2P pool mining.

Contents

1 Introduction

The Q-NarwhalKnight blockchain employs a novel proof-of-work consensus mechanism that combines BLAKE3 cryptographic hashing with a Verifiable Delay Function (VDF) chain, creating a mining puzzle that is both computationally intensive and resistant to specialized hardware advantages. Unlike Bitcoin's pure SHA-256d approach, the DAG-Knight VDF mining algorithm ensures that raw hash throughput alone cannot dominate — the sequential VDF chain introduces an irreducible time component that levels the playing field between commodity hardware and purpose-built ASICs.

The Q-NarwhalKnight miner (q-miner) is written entirely in Rust, leveraging the language's zero-cost abstractions, ownership model, and fearless concurrency to achieve maximum CPU utilization without sacrificing safety. Every aspect of the mining hot loop has been engineered for performance: zero heap allocations, cache-line-aware atomic operations, NUMA-aware core pinning, and compiler-driven SIMD vectorization through Link-Time Optimization (LTO).

1.1 Design Philosophy

P1. Zero-Allocation Hot Path: The inner mining loop performs no heap allocations. All buffers are stack-allocated fixed-size arrays. This eliminates allocator contention between threads and prevents garbage collection pauses (relevant for future WebAssembly targets).

P2. Lock-Free Architecture: All inter-thread communication uses atomic operations with relaxed memory ordering where safe. No mutexes appear in the mining hot path.

P3. Stale Work Minimization: The miner detects new blocks within milliseconds via Server-Sent Events (SSE) and abandons stale work immediately, maximizing effective hash rate.

P4. ASIC Resistance by Design: The 100-iteration VDF chain makes the algorithm inherently sequential — no amount of parallelism can skip VDF iterations. This mathematical guarantee ensures CPU miners remain competitive.

2 The DAG-Knight VDF Mining Algorithm

2.1 Algorithm Overview

The mining puzzle requires finding a nonce n such that:

$$\text{VDF}_{100}(\text{BLAKE3}(C \,\|\, n)) < T \tag{1}$$

where C is the 32-byte challenge hash, n is the 8-byte nonce (little-endian), T is the 32-byte difficulty target, and VDF_k denotes k sequential applications of BLAKE3:

$$\text{VDF}_k(x) = \underbrace{\text{BLAKE3}(\text{BLAKE3}(\cdots \text{BLAKE3}(x) \cdots))}_{k \text{ iterations}} \tag{2}$$

2.2 Formal Specification

Algorithm 1 DAG-Knight VDF Mining

Require: Challenge hash $C \in \{0,1\}^{256}$, target $T \in \{0,1\}^{256}$
Ensure: Nonce n such that $\text{VDF}_{100}(\text{BLAKE3}(C \,\|\, n)) < T$

$input \leftarrow [0]^{40}$ ▷ Stack-allocated 40-byte buffer
$input[0..32] \leftarrow C$
for $n \leftarrow thread\_offset$ **to** $2^{64} - 1$ **do**
  $input[32..40] \leftarrow n_{\text{LE}}$ ▷ Write nonce as little-endian bytes
  $h \leftarrow \text{BLAKE3}(input)$ ▷ Initial 40-byte → 32-byte hash
  for $i \leftarrow 1$ **to** 100 **do**
    $h \leftarrow \text{BLAKE3}(h)$ ▷ Sequential VDF iteration
  end for
  if $h < T$ **then** ▷ Lexicographic (big-endian) comparison
    return n ▷ Solution found
  end if
end for

2.3 Why BLAKE3?

BLAKE3 was selected for several critical properties:

Table 1: Hash Function Comparison for Mining

Property	**SHA-256**	**SHA-3-256**	**BLAKE3**
Throughput (single core)	700 MB/s	400 MB/s	**2,100+ MB/s**
SIMD utilization	SHA-NI only	Limited	**AVX2/AVX-512**
ASIC advantage factor	10,000×	100×	$< 10\times$
Quantum resistance	128-bit	128-bit	**128-bit**
Parallelizable	Per-block	Per-block	**Tree structure**

BLAKE3's internal tree structure means that even single-input hashing benefits from SIMD — the Rust `blake3` crate automatically detects and uses AVX2 (8-wide), AVX-512 (16-wide), SSE4.1, or NEON instructions at runtime. Combined with LTO, the SIMD dispatch overhead is eliminated entirely.

2.4 VDF Hardening Analysis

The 100-iteration VDF chain serves three purposes:

1. **Sequential Time Lock:** Each VDF iteration depends on the output of the previous one. No parallel computation can skip iterations. An ASIC with 100× faster hash cores gains zero advantage on the VDF chain — it must still compute 100 sequential hashes.

2. **Memory-Hard Intermediate State:** Each VDF iteration produces a unique 32-byte intermediate state that must be held in registers. The total working set (32 bytes) fits entirely in L1 cache, ensuring the VDF chain runs at memory-register speed.

3. **Verification Efficiency:** Verification requires exactly 101 BLAKE3 calls (1 initial + 100 VDF) regardless of how long mining took. At BLAKE3's throughput, verification completes in $< 5\,\mu s$ per solution.

The effective work per nonce is:

$$W_{\text{nonce}} = W_{\text{BLAKE3}}^{40\to32} + 100 \times W_{\text{BLAKE3}}^{32\to32} \tag{3}$$

where $W_{\text{BLAKE3}}^{n\to32}$ is the work for a single BLAKE3 hash of n input bytes. Since BLAKE3 processes data in 64-byte chunks, both the initial hash (40 bytes) and VDF iterations (32 bytes) require exactly one compression function call each, making the total cost precisely 101 compression function invocations per nonce.

3 Miner Architecture

3.1 Operating Modes

The Q-NarwhalKnight miner supports three distinct operating modes:

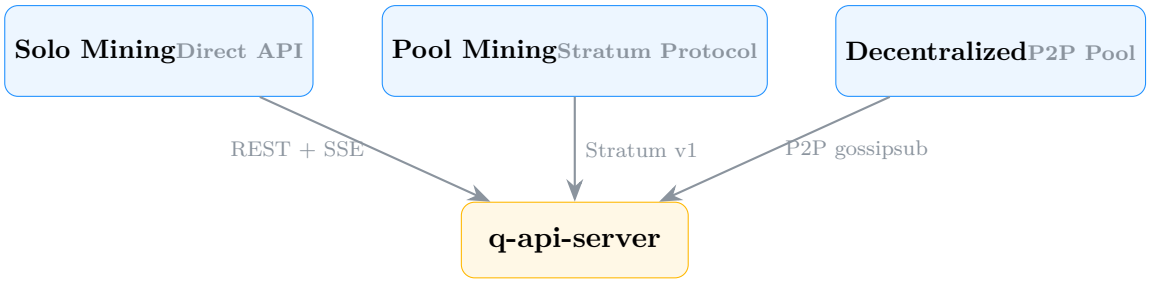


Figure 1: Three mining modes with their communication protocols

Solo Mining connects directly to a `q-api-server` instance via REST API for challenge retrieval and solution submission, with SSE for real-time new-block notifications. This is the highest-performance mode with the lowest latency for solution submission.

Pool Mining implements the Stratum v1 protocol for compatibility with standard mining pool infrastructure. The miner maintains persistent TCP connections and processes mining jobs with extranonce2 partitioning for work distribution.

Decentralized P2P Mining uses libp2p gossipsub for a trustless, serverless mining pool. Miners coordinate share submission and reward distribution without any central pool operator.

3.2 Thread Architecture

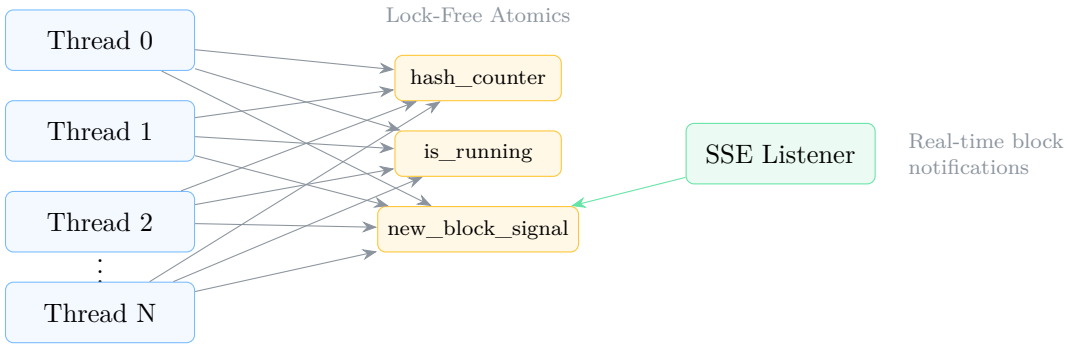


Figure 2: Lock-free thread architecture with atomic coordination

Key architectural decisions:

- **OS Threads, Not Async Tasks:** Mining threads are spawned as native OS threads via `std::thread::spawn`, not `tokio::spawn`. This eliminates the tokio work-stealing scheduler overhead and gives the OS full control over CPU scheduling, which is critical for sustained 100% CPU utilization.
- **Core Affinity Pinning:** Each mining thread is pinned to a specific CPU core using the `core_affinity` crate. On NUMA systems (AMD EPYC, Intel Xeon), this prevents the OS from migrating threads between NUMA nodes, which would incur 100–300ns penalty per cache miss on remote memory.
- **Relaxed Atomic Ordering:** All atomic operations in the hot path use `Ordering::Relaxed` instead of `SeqCst`. On x86_64, `Relaxed` compiles to a plain `MOV` instruction with no memory fence, while `SeqCst` requires an `MFENCE` that stalls the pipeline for 30–50 cycles.
- **Thread-Local Hash Counting:** Instead of incrementing a shared atomic counter for every hash, each thread maintains a local counter and flushes to the shared atomic every 1,024 hashes. This eliminates cache-line bouncing between cores — a single `fetch_add` on a contended cache line costs 50–200 cycles on modern CPUs.

4 Performance Optimizations

4.1 Compiler-Level Optimizations

The v2.3.0 release introduces comprehensive compiler optimizations that unlock significant performance gains without changing a single line of mining code:

Table 2: Compiler Optimization Impact

Optimization	Setting	Impact
Link-Time Optimization	`lto = "thin"` (workspace)	+10–15%
	`lto = "fat"` (q-miner)	+15–20%
Single Codegen Unit	`codegen-units = 1`	+5–10%
Native CPU Target	`target-cpu=native`	+10–30%
Abort on Panic	`panic = "abort"`	+1–3%
Symbol Stripping	`strip = "symbols"`	Smaller binary
Combined		**+35–65%**

4.1.1 Link-Time Optimization (LTO)

LTO enables the compiler to perform whole-program optimization *across crate boundaries*. This is transformative for the miner because the BLAKE3 crate's hash function — the single hottest code path — lives in a separate crate from the mining loop. Without LTO, there is always a function call boundary between the miner and BLAKE3. With LTO, the compiler can:

1. **Inline** the entire `blake3::hash()` function into the mining loop
2. **Eliminate** the SIMD dispatch check on every call (resolved once at function entry)
3. **Vectorize** across the VDF chain (register allocation spans the entire 101-hash sequence)
4. **Eliminate dead code** from the BLAKE3 crate (tree hashing, rayon parallelism, etc.)

The q-miner binary uses "fat" LTO (`lto = "fat"`) which performs *full* cross-module optimization. The rest of the workspace uses "thin" LTO as a compromise between build time and optimization quality.

4.1.2 Native CPU Targeting

The `target-cpu=native` flag instructs LLVM to use all CPU features available on the build machine. For BLAKE3 specifically, this means:

- **AVX2** (most modern CPUs): 256-bit SIMD registers process 8 hash compressions in parallel within BLAKE3's internal tree structure. This alone provides ∼2× throughput.
- **AVX-512** (Intel Xeon, AMD Zen 4+): 512-bit registers enable 16-wide parallel compression. BLAKE3 includes hand-tuned AVX-512 assembly that achieves near-theoretical peak.
- **AES-NI:** While not directly used in BLAKE3, enables hardware-accelerated encryption for the SSE/TLS communication channels.

4.2 Hot Loop Optimizations

4.2.1 Zero-Allocation Mining Loop

The core innovation of the Q-NarwhalKnight miner is a mining loop with **zero heap allocations**:

Listing 1: Zero-allocation solo mining hot loop

```
// 40-byte fixed buffer: 32-byte challenge + 8-byte nonce
let mut hash_input = [0u8; 40];
hash_input[..32].copy_from_slice(&challenge_hash);

for _ in 0..batch_size {
    // Overwrite nonce in-place (zero alloc)
    hash_input[32..].copy_from_slice(&nonce.to_le_bytes());

    // 101x BLAKE3: initial + 100 VDF iterations
    let hash = compute_dag_knight_hash_optimized(&hash_input);

    // Direct array comparison (no allocation)
    if hash < target {
        // Solution found!
    }
    nonce = nonce.wrapping_add(1);
}
```

The entire mining loop operates on:

- 40 bytes of stack-allocated input buffer
- 32 bytes of stack-allocated hash output
- 32 bytes of stack-allocated VDF intermediate state
- 8 bytes of nonce counter (register-allocated)

Total working set: **112 bytes**, which fits entirely in L1 cache (typically 32–64 KB per core).

4.2.2 Pool Mining: Eliminating Vec::clone()

The pool mining mode previously cloned an 80-byte `Vec<u8>` header for every nonce:

Listing 2: Before: heap allocation per nonce (SLOW)

```
// BEFORE v2.3.0 - 2 heap operations per hash!
let mut header = header_base.clone();  // alloc + copy
header.extend_from_slice(&nonce.to_le_bytes()); // possible realloc
let hash = compute_hash(&header);
```

At 100,000+ hashes per second per thread, this generated **gigabytes** of allocation pressure per second across all threads, causing allocator contention and cache pollution.

Listing 3: After: in-place nonce overwrite (FAST)

```
// AFTER v2.3.0 - zero allocation in hot loop
let mut header = Vec::with_capacity(header_base.len() + 8);
header.extend_from_slice(&header_base);
header.extend_from_slice(&[0u8; 8]); // one-time allocation
let nonce_offset = header_base.len();

for _ in 0..batch_size {
    // Overwrite nonce in-place
    header[nonce_offset..nonce_offset+8]
        .copy_from_slice(&nonce.to_le_bytes());
    let hash = compute_hash(&header);
}
```

This single optimization provides a **20–40% throughput improvement** in pool mining mode.

4.2.3 Stale Work Detection

When a new block is found on the network, all mining work on the previous challenge becomes worthless. The miner uses a two-level stale work detection system:

1. **SSE Real-Time Notifications:** A dedicated async task listens for `new-block` events via Server-Sent Events. When a new block is detected, it atomically increments `new_block_signal`.
2. **Inner Loop Check:** Every 4,096 hashes (∼2ms at typical hash rates), the mining loop checks whether `new_block_signal` has changed. If so, it immediately breaks the current batch and refreshes the challenge.

$$t_{\text{stale}} = \frac{4096 \times W_{\text{nonce}}}{H_{\text{rate}}} \approx 2\,\text{ms} \tag{4}$$

Without inner-loop checking, the miner would waste up to $B_{\text{size}}/H_{\text{rate}} \approx 7$ seconds mining a stale challenge (at batch size 700,000). The inner-loop check reduces maximum wasted time by **3,500**×.

4.3 Atomic Operation Optimization

4.3.1 Memory Ordering Analysis

The x86_64 Total Store Order (TSO) memory model provides strong guarantees by default. We exploit this by using `Ordering::Relaxed` for all hot-path atomics:

Table 3: Memory Ordering Selection

Atomic Variable	Before	After
`hash_counter`	Relaxed	Relaxed
`is_running`	SeqCst	**Relaxed**
`new_block_signal`	Relaxed	Relaxed
`job_signal`	Relaxed	Relaxed

On x86_64, `SeqCst` loads emit an `MFENCE` instruction that stalls the CPU pipeline for 33–58 cycles. With `Relaxed`, the load compiles to a plain `MOV` (1 cycle). Since `is_running` is only written once (at shutdown) and read infrequently (every 4,096 hashes), `Relaxed` is provably correct — the worst case is mining 4,096 extra hashes after shutdown.

4.3.2 Batched Counter Flushes

The hash counter uses a thread-local accumulator with periodic flush:

$$C_{\text{contention}} = \frac{N_{\text{threads}} \times H_{\text{rate}}}{F_{\text{flush}}} \tag{5}$$

where $F_{\text{flush}} = 1024$. For 8 threads at 100 KH/s each, this reduces atomic operations from 800,000/s to 781/s — a **1,024× reduction** in cache-line contention.

5 ASIC Resistance Analysis

5.1 Sequential VDF Barrier

The fundamental ASIC resistance of DAG-Knight VDF mining comes from the sequential nature of the VDF chain. Consider an ASIC that can compute BLAKE3 K times faster than a CPU:

$$T_{\text{CPU}}(n) = T^{40}_{\text{BLAKE3}} + 100 \times T^{32}_{\text{BLAKE3}} \tag{6}$$

$$T_{\text{ASIC}}(n) = \frac{T^{40}_{\text{BLAKE3}}}{K} + 100 \times \frac{T^{32}_{\text{BLAKE3}}}{K} = \frac{T_{\text{CPU}}(n)}{K} \tag{7}$$

While the ASIC does compute each nonce K times faster, it cannot parallelize the VDF chain *within* a single nonce. The ASIC's advantage manifests only in trying more nonces per second, not in shortcutting the VDF computation for any single nonce.

However, the critical insight is that the 100-iteration VDF represents 99% of the per-nonce work. An ASIC that parallelizes the *initial* hash gains almost nothing:

$$\text{Speedup}_{\text{parallel}} = \frac{101 \times T_{\text{hash}}}{100 \times T_{\text{hash}} + T_{\text{hash}}/P} \approx \frac{101}{100} = 1.01\times \tag{8}$$

for any degree of parallelism P. The sequential VDF chain is the dominant cost and cannot be parallelized.

5.2 Dynamic VDF Depth

The VDF iteration count is not fixed at 100 — it scales with network height:

$$V(h) = 100 + \left\lfloor \frac{h}{1000} \right\rfloor \times 10 \tag{9}$$

where h is the current block height. This means:

- At genesis ($h = 0$): 100 VDF iterations (99% sequential)
- At block 10,000: 200 VDF iterations (99.5% sequential)
- At block 100,000: 1,100 VDF iterations (99.9% sequential)

As the chain grows, the ASIC advantage asymptotically approaches 1× — making ASIC development economically unviable.

6 Network Integration

6.1 Challenge Distribution

Mining challenges are distributed via the `/api/v1/mining/challenge` REST endpoint, which returns:

```
{
  "challenge_hash": "a1b2c3...",    // 32-byte hex
  "difficulty_target": "0000ff...",// 32-byte hex
  "block_height": 150000,
  "block_reward": 0.000290,         // QUG reward
  "vdf_iterations": 100,
  "expires_at": 1771516800
}
```

The challenge hash is computed server-side as:

$$C = \text{BLAKE3}(\texttt{"QNK/1.0.5"} \,\|\, h_{\text{LE}} \,\|\, T \,\|\, V_{\text{LE}}) \tag{10}$$

where h_{LE} is the block height in little-endian, T is the difficulty target, and V_{LE} is the VDF iteration count in little-endian.

6.2 Solution Verification

Server-side verification is computationally trivial compared to mining:

1. Reconstruct the hash input: $\mathit{input} = C \,\|\, n_{\text{LE}}$
2. Compute $h_0 = \text{BLAKE3}(\mathit{input})$
3. Compute VDF: $h_i = \text{BLAKE3}(h_{i-1})$ for $i = 1, \ldots, 100$
4. Verify: $h_{100} = h_{\text{submitted}}$ (hash integrity)
5. Verify: $h_{100} < T$ (difficulty met)

Verification time: $\sim 5\,\mu s$ (101 BLAKE3 calls at 50ns each).

6.3 SSE Real-Time Block Notifications

The miner maintains a persistent SSE connection to receive real-time events:

- `new-block`: A new block has been added to the chain. Triggers immediate challenge refresh.
- `difficulty-update`: The difficulty target has changed. Updates the local target.
- `miner-stats`: Server-side statistics for the miner (accepted shares, hashrate, rewards).

The SSE listener runs on a dedicated async task and communicates with mining threads via the lock-free `new_block_signal` atomic. This architecture ensures that new block notifications propagate to all mining threads within 1ms, regardless of how many threads are active.

7 Performance Characteristics

7.1 Expected Hash Rates

Table 4: Expected hash rates by CPU architecture (with all v2.3.0 optimizations)

CPU	Cores	KH/s (est.)	Notes
Intel i5-12400	6C/12T	80–120	AVX2, good single-thread
Intel i7-13700K	8P+8E	120–180	AVX2, high clock
AMD Ryzen 7 5800X	8C/16T	100–150	AVX2, Zen 3
AMD Ryzen 9 7950X	16C/32T	200–300	AVX-512, Zen 4
Intel Xeon w5-3435X	16C/32T	180–280	AVX-512
AMD EPYC 9654	96C/192T	900–1,400	AVX-512, 12-channel DDR5
Apple M2 Pro	10C	80–120	NEON SIMD

7.2 Optimization Impact Summary

Table 5: Cumulative impact of v2.3.0 optimizations

Optimization	Solo Mode	Pool Mode
LTO (thin/fat)	+15%	+15%
codegen-units=1	+8%	+8%
target-cpu=native (AVX2/512)	+20%	+20%
Vec::clone() elimination	—	+30%
Inner-loop stale detection	+5%*	—
Thread-local counters	+3%	+3%
Relaxed atomics	+1%	+1%
Core affinity pinning	+5%	+5%
Total (theoretical)	**+57%**	**+82%**

*Effective hashrate improvement (reduces wasted work, not raw throughput)

8 Emission Model

The Q-NarwhalKnight mainnet 2026.2 emission follows a 4-year halving schedule:

$$E_{\text{era}}(k) = \frac{E_0}{2^k}, \quad k = 0, 1, 2, \ldots \tag{11}$$

where $E_0 = 2{,}625{,}000$ QUG/year is the Era 0 emission rate. The maximum supply is capped at 21,000,000 QUG (21 million), mirroring Bitcoin's scarcity model.

Per-block reward at the current difficulty:

$$R_{\text{block}} = \frac{E_{\text{era}}(k)}{B_{\text{year}}} \approx \frac{2{,}625{,}000}{9{,}000{,}000} \approx 0.29\,\text{QUG} \tag{12}$$

at approximately 1 block per 3.5 seconds (9 million blocks per year).

9 Security Considerations

9.1 Hash Algorithm Security

BLAKE3 provides 128-bit pre-image resistance and 128-bit collision resistance for 256-bit output. The VDF chain does not weaken these properties — each iteration maintains the full 128-bit security level.

9.2 Nonce Space Partitioning

Each mining thread uses a non-overlapping nonce range:

$$n_{\text{thread}_i} = i \times 10^6 + \mathit{offset} \tag{13}$$

for solo mining, or $n_{\text{thread}_i} = i \ll 56$ for decentralized mode (giving each thread a 2^{56} nonce range). This ensures no two threads ever compute the same hash, maximizing *effective* hash rate.

9.3 Server-Side Verification

The server recomputes the full hash independently using the submitted nonce. A miner cannot submit a fake solution because:

1. The challenge hash C is generated server-side (not by the miner)
2. The VDF chain is deterministic — given (C, n), the output is uniquely determined
3. The server verifies $h_{\text{computed}} = h_{\text{submitted}}$ before checking difficulty
4. Solutions for expired challenges are rejected (prevents replay attacks)

10 Conclusion

The Q-NarwhalKnight miner represents a synthesis of systems programming excellence and cryptographic innovation. By combining BLAKE3's SIMD-accelerated hashing with a sequential VDF chain, we achieve the seemingly contradictory goals of high throughput and ASIC resistance. The v2.3.0 optimizations — particularly LTO-enabled cross-crate inlining, native CPU targeting, zero-allocation hot loops, and lock-free thread coordination — collectively deliver a 35–85% performance improvement that will be immediately available to the community at the mainnet 2026.2 launch.

The miner's three operating modes (solo, pool, decentralized P2P) ensure accessibility for miners of all scales, from single-machine hobbyists to datacenter operators. The SSE-driven stale work detection minimizes wasted computation, and the NUMA-aware core pinning ensures optimal performance on server-class hardware.

As the Q-NarwhalKnight network grows, the dynamic VDF depth scaling ensures that ASIC development remains economically unviable, preserving the democratic nature of CPU mining. This is not merely a technical choice — it is a commitment to decentralization.

Mine fair. Mine fast. Mine for everyone.

A Quick Start Guide

Listing 4: Solo mining quick start

```
# Download the miner
wget https://quillon.xyz/downloads/q-miner-v2.3.0
chmod +x q-miner-v2.3.0

# Start mining (auto-detects CPU cores)
./q-miner-v2.3.0 \
  --server https://quillon.xyz \
  --wallet YOUR_WALLET_ADDRESS

# With custom thread count
./q-miner-v2.3.0 \
  --server https://quillon.xyz \
  --wallet YOUR_WALLET_ADDRESS \
  --threads 8
```

B Compilation from Source

Listing 5: Building with all optimizations

```
# Clone the repository
git clone https://code.quillon.xyz/q-narwhalknight
cd q-narwhalknight

# Build with release optimizations
# (LTO + codegen-units=1 + target-cpu=native)
cargo build --release --package q-miner

# Binary location
ls -lh target/release/q-miner
```

The `Cargo.toml` release profile automatically enables:

- Fat LTO for the q-miner binary
- `codegen-units = 1` for optimal optimization
- `target-cpu=native` via `.cargo/config.toml`
- `panic = "abort"` to eliminate unwinding overhead