

ROLV Primitive©

Validation Kit

Independent Verification Guide · March 2026

1. What This Document Is

Every ROLV benchmark publishes four SHA-256 hashes per test case. This document explains what can be verified independently, what cannot, and why the combination still provides strong proof of honest results.

Hash	What it covers	Who can verify it
hash_A	The weight matrix	Anyone — deterministic from published seed
hash_V	The input activation vector	Anyone — deterministic from published seed
hash_D	The dense baseline output (cuBLAS / rocBLAS / MKL)	Anyone — standard dense matmul, no ROLV code needed
hash_R	The ROLV output	ROLV only — proprietary operator

2. What You Can Verify Independently

hash_A and hash_V

Regenerate the weight matrix and input vector from the published seed. Compute SHA-256. If it matches our published hash, the inputs are confirmed identical to ours.

hash_D — the key verification step

Run a standard dense matmul on those inputs using any library (numpy, cuBLAS, rocBLAS, PyTorch). Compute SHA-256. If it matches hash_D then: our baseline timing is confirmed on your hardware, our baseline output is correct, and the comparison point for all speedup claims is verified. All ROLV speedup claims are ratios against hash_D.

3. What Only ROLV Can Produce

hash_R is the output of ROLV Primitive©, a proprietary operator. It cannot be reproduced without our code. However, hash_R is still meaningful for four reasons:

1. Correctness is bounded

hash_R must satisfy ATOL=0.05 on column-normalised outputs vs hash_D. Max observed error across all 1,684 cases is 9.87×10^{-7} — five orders of magnitude below threshold.

2. The perturbation test proves live computation

Every case modifies one non-zero weight by $1e-4$ and confirms hash_R changes. Pre-computed results cannot pass this. It passes in all 1,684 cases.

3. hash_D = hash_R at 0% sparsity

At zero sparsity ROLV and dense produce identical outputs. If $\text{hash}_R \neq \text{hash}_D$ at 0% sparsity, the result is invalid.

4. The anti-fabrication argument

V is not published in full — only hash_V is. Fabricating a plausible hash_R requires actually running the computation.

4. The Verification Chain

Step	Action	Who can verify
1	Confirm hash_A matches weight matrix from published seed	Anyone
2	Confirm hash_V matches input vector from published seed	Anyone
3	Run dense matmul yourself, confirm hash_D matches — this is the key step	Anyone
4	Accept hash_R as ROLV output on same inputs, proven live by perturbation test	ROLV LLC

The speedup claim $t_{\text{vendor}} / t_{\text{rolv}}$ is grounded: the vendor timing is measured against a confirmed baseline (hash_D), and the ROLV timing is proven to be a live computation producing a correct result within tolerance.

5. How to Verify a Specific Result

Step 1 — Reproduce the weight matrix

Each harness specifies dimensions ($M \times K$), sparsity level, and seed.

```
import numpy as np, hashlib

def make_matrix(M, K, sparsity, seed):
    rng = np.random.default_rng(seed)
    W = rng.standard_normal((M, K)).astype(np.float32)
    mask = rng.random((M, K)) < sparsity
    W[mask] = 0.0
    return W

W = make_matrix(M=128256, K=16384, sparsity=0.95, seed=42)
h = hashlib.sha256(W.tobytes()).hexdigest()[:8]
print(h) # Compare to hash_A in rolv_benchmarks.pdf
```

Step 2 — Run the baseline yourself

Use any library — numpy, PyTorch, cuBLAS, rocBLAS. No ROLV code needed.

```
import numpy as np, hashlib

V = np.random.default_rng(0).standard_normal((K, batch)).astype(np.float32)
out_dense = (W @ V).astype(np.float32)
h_dense = hashlib.sha256(out_dense.tobytes()).hexdigest()[:8]
print(h_dense) # If it matches hash_D, our baseline is confirmed
```

Step 3 — Compare to published hashes

Full hash tables for all 1,684 cases are in rolv_benchmarks.pdf. If hash_A and hash_D match your computed values, the baseline is independently confirmed.

6. Correctness Standard

Parameter	Value
Tolerance (ATOL)	0.05 on column-normalised outputs
Normalisation	Each output column divided by its L2 norm
Max error observed	9.87 x 10(-7) across all 1,684 cases
Threshold / actual ratio	5 orders of magnitude below threshold
BF16 error (reference)	10(-2) to 10(-3) — 3–4 orders worse than ROLV FP32

ROLV introduces zero approximation. The only numerical difference from dense is floating-point reordering — ROLV computes on a compressed submatrix, changing the order of FP32 additions. Differences are at FP32 epsilon level, not approximation error.

7. Perturbation Test

Every case modifies one non-zero weight by 1e-4 and confirms hash_R changes, proving the operator is computing live (not cached) and is sensitive to actual weight values. All 1,684 perturbation tests pass.

8. Platform and Hardware Details

Platform	Matrix	Batch	Baseline
NVIDIA H200	10k x 10k	2,500	cuBLAS / cuSPARSE
NVIDIA B200	10k x 10k	2,500	cuBLAS / cuSPARSE
AMD MI300X (synthetic)	10k x 10k	2,500	rocBLAS / rocSPARSE
AMD MI300X (models)	Prod dims	512	rocBLAS / rocSPARSE
Intel CPU	2k x 2k	500	MKL / CPU-CSR
AMD EPYC 7B13	2k x 2k	500	rocBLAS / CPU-CSR
Google Axion ARM	3k x 3k	1,000	OpenBLAS / CPU-CSR
Mistral-7B (Intel i7)	Prod dims	512	MKL / CPU-CSR

Baseline: below 70% sparsity → dense vendor. At 70%+ → sparse vendor. Both always measured and published.

9. Published Hashes — Synthetic Benchmarks

Platform	hash_A	hash_V	PASS
NVIDIA H200 — 10k x 10k	b2687223	f8b47533	22/22
NVIDIA B200 — 10k x 10k	6764dac0	eabab8fa	22/22
AMD Instinct MI300X — 10k x 10k	76252923	7f9f717a	22/22
Mistral-7B embed_tokens — 32000x4096	0d9b4a63	694c0a7f	7/7

Full per-case hash_D and hash_R for all 1,684 cases are in rolv_benchmarks.pdf.