`namaster-proof`: Content-bound execution receipts as a shortcut detector for pseudo-C_ℓ computations

Houston Golden
Independent researcher
ORCID: 0009-0008-5616-5994

September 5, 2026 — manuscript revision v2B.0.23(2026-09-05 12:00 PT)

Abstract

`namaster-proof` is a focused Python verification layer for two error-prone steps in cut-sky spin-2 analyses. First, it evaluates a uniformly rotated EE, EB, BE, BB spectrum through the complete bandpower-window operator of a caller-supplied spin-2 NaMaster workspace tensor, avoiding replacement of the operator by bin-centre or effective-multipole templates; the layer has been exercised against PyMaster 2.6 in a retained production artifact (§8) and in the integration example (§9). Second, it writes JSON results and content-bound sidecar receipts with atomic per-file replacement and fails closed when result bytes or caller-asserted execution metadata change. We extend the receipt idea into an execution-trace contract and test it under a pre-registered, sealed blind protocol, run in four batches (an 18-run pilot; a 35-run primary sweep; a 48-run value-level extension; a 54-run post-commitment challenge) under an unmodified-instrumented-harness threat model — the analyst may alter the computation but not the harness that records it; the blind test's own instrumented estimator is this repository's spin-0 MASTER implementation, because PyMaster is not installed in the test environment and exposes no Wigner-3j counter. A structural rule set (R1–R6) flags every replicate of operator-skipping, operator-truncating, grid-reducing, and cache-substituting shortcuts and none of the honest runs; a value-level rule (R7) additionally catches an effective-multipole shortcut 6/6 by recomputing receipt-selected rows of the coupling operator, but R7's row draw is prover-predictable rather than truly random, so a rule-aware runner can defeat it at zero cost, and R7 fails open if the checked intermediate is simply omitted. A post-commitment verifier challenge (R8) fixes both gaps — the challenge randomness is committed before any receipt is published and revealed only after every receipt is bound — and catches both the rule-aware evasion and the omission 6/6 with zero honest false positives, at the cost of the committing verifier's honesty rather than a public randomness beacon. A metadata-forgery class that fabricates the declared intermediate wholesale still escapes every rule, unchanged across all four batches, and remains the primitive's stated limit. All detection claims are reported as class-level counts, never as run-level probability intervals (§6). Separately, the in-house spin-0 MASTER estimator was validated against the public NaMaster library (PyMaster 3.0) on an identical map, mask, and ℓ-range: the mode-coupling matrix and decoupled bandpowers agree with NaMaster to round-off ($\lesssim 5 \times 10^{-13}$ and $\lesssim 2 \times 10^{-12}$ relative difference), confirming the estimator's correctness independent of the blind test. The result establishes the primitive as a detector of *structural shortcuts in instrumented steps*, not of forged metadata or of value-level shortcuts taken downstream of a declared intermediate: we report a pre-registered, sealed blind map of what a receipt-based check does and does not catch, with the negative classes reported as escapes. The package also supplies explicit multipole-support contracts, fixed-grid rotation-angle recovery, command-line receipt verification, and compatibility tests against the production helpers from which it was extracted. The software is intended for method validation and reproducibility checks; it is not a sky-analysis pipeline, foreground model, or cosmological inference engine.

Keywords. Python; cosmology; pseudo-C_ℓ; NaMaster; reproducibility; provenance.

1 Introduction

Pseudo-C_ℓ estimators make masked spin-2 analyses computationally practical by relating full-sky spectra to coupled and decoupled bandpowers [1, 2]. Reproducible use of these estimators requires more than recording a best-fit number: the same window operator must be used in simulation and inference, harmonic limits must agree across fields and bins, and intermediate results must remain bound to their execution metadata.

`namaster-proof` packages these checks behind a small public API. Its numerical module constructs exact windowed responses for uniform polarization rotation, evaluates them at one or many angles, recovers an angle on a declared grid, and compares the window contraction with NaMaster's couple–decouple operator. Its provenance module publishes a JSON result and a SHA-256-bound receipt using atomic replacement for each file, then verifies the pair and caller-asserted metadata. A third module defines deterministic multipole and binning contracts. NumPy is the only required dependency; a NaMaster workspace is supplied by the user when exercising the physical window operator.

2 Statement of Need

For two spin-2 fields, NaMaster exposes a bandpower-window tensor with spectrum ordering $[EE, EB, BE, BB]$. A common shortcut is to evaluate a theory spectrum at a representative multipole for each bin. That shortcut is not generally identical to applying the complete mode-coupling and binning operator, and a mismatch between the simulation and fitting operators can appear as a parameter-recovery bias. On the synthetic $N_{\rm side} = 64$, $\ell_{\rm max} = 64$ configuration used for the blind test below (Sec. 6), replacing the full band evaluation with a single effective-multipole transfer factor per 8-ℓ band produces order-unity fractional deviations in several bands of the decoupled bandpowers relative to the complete-operator result — large enough to bias a fit. This is exactly the shortcut exercised as class S6 below; a per-band deviation figure traceable to a committed script and output is discussed in §6 below.

A separate reproducibility risk arises in long or sharded calculations. Ordinary filenames do not prove that a summary corresponds to the intended seeds, realization count, configuration, or result bytes. A process can also terminate between writing a result and its metadata, leaving a plausible but incomplete artifact. These problems are usually addressed with analysis-local helpers. `namaster-proof` provides a reusable, tested implementation with explicit failure behavior, while remaining narrow enough to audit.

3 Implementation and Architecture

The package is divided into three modules:

`windows`	Rotation of EE, EB, BE, BB spectra, exact contraction with a `get_bandpower_windows()` tensor, fixed-grid angle recovery, and an equivalence check against `couple_cell()` followed by `decouple_cell()`.
Multipoles	Validation of a common harmonic cutoff and construction of integer bandpower edges whose final exclusive edge is $\ell_{\rm max} + 1$.
`receipts`	Atomic JSON publication, streaming SHA-256 calculation, sidecar-receipt construction, and fail-closed content and metadata validation.

The functions accept array-like inputs and return NumPy arrays or plain Python dictionaries. NaMaster is deliberately not an installation dependency: window functions require a workspace exposing `get_bandpower_windows()`, `couple_cell(cls)`, and `decouple_cell(coupled)`. The first must return a finite $[4, n_b, 4, n_\ell]$ tensor and the couple–decouple route must return a finite $[4, n_b]$ array. This small protocol makes the algebra testable with synthetic linear operators and permits users to pin PyMaster independently.

4 Exact-Window Inference

For an initially vanishing EB spectrum and a uniform rotation angle β, the rotated spectra can be decomposed into angle-independent, $\cos(4\beta)$, and $\sin(4\beta)$ components. In particular,

$$C_\ell^{EE}(\beta) = \tfrac{1}{2}(C_\ell^{EE} + C_\ell^{BB}) + \tfrac{1}{2}(C_\ell^{EE} - C_\ell^{BB})\cos(4\beta), \tag{1}$$

$$C_\ell^{BB}(\beta) = \tfrac{1}{2}(C_\ell^{EE} + C_\ell^{BB}) - \tfrac{1}{2}(C_\ell^{EE} - C_\ell^{BB})\cos(4\beta), \tag{2}$$

$$C_\ell^{EB}(\beta) = C_\ell^{BE}(\beta) = \tfrac{1}{2}(C_\ell^{EE} - C_\ell^{BB})\sin(4\beta). \tag{3}$$

Let $W_{b\ell}^{ij}$ be the workspace tensor mapping input spectrum j to output spectrum i in band b. The package precontracts each of the three components,

$$R_b^{i,k} = \sum_{j,\ell} W_{b\ell}^{ij} C_\ell^{j,k}, \qquad k \in \{0, c, s\}, \tag{4}$$

where k indexes the three angle-independent, $\cos(4\beta)$, and $\sin(4\beta)$ components defined above, and subsequently evaluates

$$C_b^i(\beta) = R_b^{i,0} + \cos(4\beta) R_b^{i,c} + \sin(4\beta) R_b^{i,s}. \tag{5}$$

This retains the full window tensor while making repeated grid evaluation inexpensive. The recovery function minimizes a declared weighted or unweighted squared residual over a user-provided grid; its default is 2001 points spanning -1° to $+1^\circ$.

5 Content Validation

`publish_json` serializes a mapping with sorted keys, derives the receipt digest and byte count from that immutable serialized snapshot, writes and flushes a process-specific temporary file, atomically replaces the result, and synchronizes the containing directory. It then separately publishes a sidecar receipt containing schema version, result filename, byte count, SHA-256 digest, and caller-supplied execution metadata. The content-binding fields are protected from metadata override.

`verify_json_receipt` requires both files, reads each through one file handle, parses those immutable byte snapshots as JSON objects, and recomputes every protected field from the same result bytes it returns. `validate_json_receipt` additionally checks caller-asserted metadata such as a suite identifier, seed range, or realization count using recursive JSON-type-strict equality, so booleans, integers, and floating-point numbers are not interchangeable. Missing files, malformed objects, changed bytes, changes to protected receipt fields, or asserted-contract mismatches raise an exception. The two files are not one filesystem transaction, and coordinated replacement of both requires an external receipt anchor or trusted expected metadata to detect. The `namaster-proof verify` and `namaster-proof validate` commands expose the same behavior to shell workflows and return a nonzero status on failure.

What the receipt binds. Table 1 separates the fields a receipt *self-evidently* protects (recomputed from the result bytes at verify time; not overridable by caller metadata) from the fields it only *asserts* (caller-supplied, checked against a trusted expected contract). Shortcut detection, below, lives entirely in the asserted row: the package already gives content binding, and the blind test extends the asserted metadata into a measured execution trace.

Table 1: What a `namaster-proof` receipt binds, by trust level.

Field	Source	Trust
`result_sha256`, `result_bytes`	recomputed at verify time	self-evident
`result_file`, `schema_version`	protected	self-evident
caller metadata (suite, seeds, ...)	caller-asserted, contract-checked	asserted
`intermediates.pseudo_cl`	contract-checked *when present*	asserted, absent fails R7 open

6 Blind Shortcut-Detection Test

The Content Validation layer above gives content binding: the result bytes are the ones the receipt describes. It does not, by itself, establish that an expensive computation was actually performed rather than shortcut and then honestly reported. This section reports a pre-registered, sealed test of that narrower and falsifiable claim, under an unmodified-instrumented-harness threat model (the analyst may alter the computation but not the harness that records it): *a referee who cannot afford to re-run an expensive exact computation can, from receipts alone, decide whether the computation was actually performed or silently shortcut.*

Execution-trace extension. The blind test extends the caller-asserted metadata into an execution trace emitted by an instrumented harness rather than typed by the analyst: sha256 of the input map and mask, $N_{\rm side}$, $\ell_{\rm max}$; sha256 of the compute module actually imported; numpy/healpy versions and platform; sha256 and shape of the mode-coupling matrix $M_{\ell\ell'}$; the list of ℓ-values actually evaluated; an instrumented count of Wigner-3j symbol evaluations performed; the decoupled bandpowers (hash-bound, in the result file); and measured wall time.

Protocol, two batches. (1) A reference contract is published first, from one honest reference run: full ℓ-grid, M shape, exact 3j-evaluation count, code sha256, tolerance. (2) A sealing script draws a random key, assigns N runs to arms via a balanced, HMAC(key, run_id)-derived permutation of a fixed per-arm design, and commits only the sealed digest of the assignment before any run executes. (3) Each run executes its variant, unknown to the verifier, and publishes a result and receipt through the package's `publish_json`. (4) The verifier, implemented in `verify.py`, applies a frozen rule set (Table 2) to the run directories, the reference contract, and the input manifest. (5) The key is revealed and the confusion matrix computed. The protocol was run four times: a pilot, a primary structural sweep, a value-level extension, and a post-commitment challenge (§6–6). In batch 1 (18 runs, arms honest/S1–S5), two of the rules were changed *after* batch-1 outcomes were seen (§6); batch 1 is therefore relabelled a **pilot, rule-development round**, not a blind measurement of detector performance. In batch 2 (35 runs, arms honest/S1–S6), the rule set was frozen and committed alone, in its own commit, *before* the seal commitment existed in the repository, and **no rule was added, removed, or edited between that freeze and the reveal**; batch 2 is the primary result reported below. The pre-declared success criterion for batch 2 was: every

arm of S1–S4 detected 5/5, honest 0/5 flagged; S5 (metadata forgery) expected to escape 5/5 by construction; S6 (effective multipole, §6) had *no* pre-declared expected outcome.

Table 2: Frozen decision rules R0–R6 (batch 2). A run is called *shortcut* if one or more fires.

Rule	Fires when
R0	Receipt verification fails: result bytes do not match the content binding, or the receipt is malformed.
R1	`code.sha256` disagrees with the contract.
R2	The evaluated ℓ-grid disagrees with the contract (exact list equality).
R3	The instrumented Wigner-3j evaluation count disagrees with the contract.
R4	The coupling-matrix shape or support disagrees with the contract.
R5	The input mask sha256 disagrees with the contract.
R6	`sha256(bandpowers)` equals the contract's *reference-run* bandpower hash, **or** that digest already occurred on an earlier run with a different input-map hash (two disjuncts; both are load-bearing — §6).

Two rules considered during rule development are **not** part of the decision set: a wall-clock floor (advisory only, recorded per run but never contributing to a call) and an M-hash-collision rule (withdrawn, not evidential, since M depends only on the mask and ℓ-grid so every honest run legitimately shares one M hash). Both are discussed under the pilot round below.

Result (batch 2, primary). 35 runs were sealed and executed under the frozen rules (7 arms × 5 replicates; $N_{\rm side} = 64$, $\ell_{\rm max} = 64$, local CPU, ∼11 s for 36 runs including the reference, $0 marginal cost). The commit-ordered audit trail is: rules frozen alone (`4451b135`) → seal commitment with no run output in the tree (`28efa21c`) → 35 run outputs and verdicts (`27300504`) → key reveal and scorecard (`974e2859`) → manifest (`b3347c53`). Sealed assignment digest `c96b5bf1...`, key-file digest `bbf6373b...`, `seal_verified: true`: the assignment re-derived from the revealed key reproduces the digest committed before any batch-2 run output existed, and the sealed material was held outside the working tree for the entire sealed window. Table 3 gives the per-arm confusion counts.

Table 3: Blind shortcut-detection confusion table, batch 2 (35 sealed runs, 7 arms × 5, frozen rules).

Arm	n	Flagged	Rule(s) fired
Honest	5	0	—
S1 operator skipped	5	5	R3, R4
S2 operator truncated	5	5	R3, R4
S3 grid reduced + interpolated	5	5	R2, R3, R4
S4 cache substituted	5	5	R6 (reference disjunct 5/5, cross-run disjunct 0/5)
S5 metadata forgery	5	0	— (escapes, pre-declared)
S6 effective multipole	5	0	— (escapes; *no* outcome pre-declared)

All four structural classes were flagged in every replicate: 4/4 structural classes detected, 20/20 runs (one-sided 95% Clopper–Pearson lower bound **0.473** on the class-level detection rate, the only i.i.d. unit available — see the independence caveat below); honest false positives are 0/5. A pooled figure across all six shortcut arms is not reported: its value is fixed by the arbitrary choice of how many escape classes are included at equal replication and carries no external interpretation. **Independence caveat (pre-declared):** rules R1–R5 compare seed-independent quantities against a fixed contract, so the five replicates within an arm are seed-varied executions of one deterministic

variant, not five independent draws — effective independent n per class is 1 for those rules, and the replicates measure firing *determinism*, not sampling variance; no run-level detection-probability interval is claimed. R6 (result reuse) does vary with the seed-dependent input-map hash and is not subject to this caveat. S1–S3 are all caught by the same trace-mismatch mechanism (rules R3 and R4); S4 is caught by R6. The evidence is therefore two deterministic mechanisms across four classes, not four independent mechanisms. S4's five detections are all the *reference* disjunct of R6 (`r6_reference_disjunct = 5`); the *cross-run* disjunct fired 0/5 and is therefore untested in batch 2 (see §11). The frozen verifier's own sha256 is quoted from `public2/frozen_rules_digest.json`: `6a9acd70...`.

How to check this yourself. The freeze commit `4451b135` contains the rules *specification* (`RULES_v2_FROZEN.md`); `verify.py` was edited once between freeze and seal (`28efa21c`) for non-rule plumbing (a one-line output-directory argument; the `judge()` function and the R6 block are unchanged), and its post-seal sha256 `6a9acd70...` is the digest pinned in `public2/frozen_rules_digest.json`. The sealed digest `c96b5bf1...` is sha256 of the canonical JSON form of the arm assignment (`json.dumps(assignment,sort_keys=True)`), not the on-disk file bytes, which hash to a different value (`c8566d0e...`); recomputing the canonical-form digest from the revealed key reproduces `c96b5bf1...`. The pre-declared success criterion is recorded in `BATCH2_PREREGISTRATION.md` in the same seal commit. An OpenTimestamps proof of the sealed digest was submitted 2026-09-04 19:31 (`public2/sealed_digest.json.ots`); `ots upgrade` then `ots verify` against that file establishes that the digest existed before a given Bitcoin block time, not an external witness to the batch-2 execution itself. The R6 rule's stated behavior was corrected in `RULES_v2_FROZEN.md` as a documentation-only change (no code in `verify.py` changed); this is disclosed here as a third post-hoc change alongside the two batch-1 rule changes below, because S4's 5/5 result rests on the reference disjunct that description clarifies.

Batch 1: pilot, rule-development round, disclosed. An earlier 18-run batch (3 per arm, honest/S1–S5) is what motivated the frozen rules above; it is reported here as a pilot, not as a second blind measurement, because two of its rules were changed *after* its outcomes were seen. First, wall clock was found not to be a usable rule: the shortcut arms ran 3–15× faster than honest ones, but a cold-cache honest reference run took 4.29 s against 0.08 s warm, so the then-proposed wall-time floor (25% of the reference) would have fired on 3 of 3 honest runs; batch 1 excluded it from the decision set only after seeing this failure, and it survives in batch 2 as the advisory-only, non-deciding `wall_rule_would_fire` field (still 5/5 on honest runs in batch 2, confirming it remains unusable as a decision rule; §6 gives its batch-2 count). Second, a coupling-matrix hash collision across runs was initially treated as cache-substitution evidence, then withdrawn once it was recognized that M depends only on the mask and the ℓ-grid, so every honest run legitimately shares one M hash for a fixed survey mask; batch 1 removed this rule post hoc rather than having excluded it in advance. With these two post-hoc changes disclosed, batch 1's own confusion counts (12/12 on S1–S4, one-sided 95% lower bound 0.779; 0/3 honest false positives, upper bound 0.632) are rule-fitting numbers, not a blind-test result, and are not headlined; the primary batch-2 result below (paragraph "Result (batch 2, primary)") is the reported measurement.

S6: the effective-multipole class escaped, and that is the finding. S6 builds the full mode-coupling operator genuinely — the real ℓ-grid, the real $3j$-evaluation count, the real operator shape and support all match the contract — and takes the shortcut *downstream*, in the band evaluation, dividing each band by the coupling evaluated at one effective multipole (the shortcut quantified in

Sec. 2). Nothing in R0–R6 inspects result *values* against the contract, so S6 is structurally invisible to this verifier and escaped detection 5/5, with wall-clock-would-fire also 5/5 (again unusable as a decision signal). **No rule was added after seeing this outcome.** A value-level rule — a declared tolerance band on bandpowers against an independently-known reference, or a cross-checked second estimator — is proposed as a pre-registered batch 3, future work and not claimed here.

Batch 3: a value-level rule (R7), pre-registered against S6. Batch 3 adds one instrumented intermediate, `intermediates.pseudo_cl` (the pseudo-spectrum every MASTER pipeline computes as step one), and one rule, R7: rebuild the mask and refuse unless its hash matches the contract; derive $K = 6$ spot rows of the coupling matrix from sha256(mask hash‖sha256(bandpowers)), a public function of two values the runner holds before it publishes anything: the draw is receipt-bound but *prover-predictable*, not unpredictable, and a runner that knows R7 can compute those six rows before publishing and make exactly them exact at zero marginal cost, since the effective-multipole shortcut already builds the full operator honestly. R7's 6/6 below is therefore a measurement against a rule-*unaware* adversary; §6 presents R8, the rule-*aware* value-level rule that restores the soundness R7 lacks; recompute only those six rows (∼9% of a full build); fire iff $|\sum_{\ell_2 \geq 2} M_{\ell_1 \ell_2} C_{\ell_2} - p_{\ell_1}| > 10^{-6} \|p\|_2$ on any spot row. TOL $= 10^{-6}$ was fixed from theory plus non-blind calibration before the seal (honest relative residuals 8.4×10^{-13}–1.4×10^{-12}; S6 under the same seeds, 7.5×10^{-2}–1.5×10^{-1}): an eleven-order gap, not tuned after unblinding. 48 runs were sealed (8 arms × 6; seal `abfe2793...`, `seal_verified: true`). Table 4 gives the result: honest 0/6 flagged; S1–S4 and the new S4b (cross-run cache variant) 6/6 each; **S6 6/6 by R7 alone**; S5 (metadata forgery, extended to forge $p := MC$) still escapes 6/6. Structural arms S1–S4 plus S4b are **five** arms × 6 = **30/30**, not the 24/24 an earlier draft misreported. S4b is the first replicate set to exercise R6's cross-run disjunct (§6): 4/6 fired it, and the other 2 collapsed to the reference disjunct because the prior run they substituted was itself cache-substituting, with the source run for each replicate named in `sealed3/crossrun_sources.json`. As in batch 2, R7's replicates are seed-dependent executions of one deterministic variant, so the inferential unit is the class, not the run: batches 1–4 uniformly report class-level detection counts and never a run-level Clopper–Pearson interval.

Batch 4: a post-commitment verifier challenge (R8), pre-registered against a rule-aware adversary. R7's two design gaps are real and are answered by building the exact adversaries they predict rather than by softening the prose above. First, R7's spot rows are a public function of values the runner holds before publishing, so arm **S7 (R7-aware effective multipole)** computes those six rows in advance and writes a declared pseudo-spectrum exact on exactly them, at zero marginal cost. Second, R7 fails *open* when `intermediates.pseudo_cl` is absent, so arm **S8 (omit pseudo_cl)** is an S6 run whose receipt simply omits the field. Rule R8 (`RULES_v4_FROZEN.md`, sha256 `a59caaf8...`) adds a post-commitment challenge: before the batch seal and before any run, the verifier draws 32 random bytes σ and publishes only sha256(σ); each run's receipt is then bound as $D =$ sha256(mask hash ‖ coupling hash ‖ band_hash(C) ‖ band_hash(p)); only after every receipt is published and every D committed is σ revealed, and the challenged rows are drawn as sha256(σ‖run id‖D); the verifier rebuilds the mask, recomputes those six challenged rows, and fires the same TOL $= 10^{-6}$ test. R8 fails **closed**: a missing, malformed, non-finite or wrong-length `pseudo_cl`, a missing operator hash, or a mask that fails to rebuild to the contract hash all fire, tagged `SHORTCUT-SUSPECT`. An honest run's residual is $-10^{-12}C$ on every row at once, so it passes any challenge set without knowing σ; a shortcut with $p \neq MC$ on generic rows escapes only if all K post-commitment rows land in its agreeing set. Nine arms × 6 replicates (54 runs, seal `dbe6a713...`) gave: honest 0/6; S1–S4, S6 6/6 on both R7 and R8; S5 (metadata forgery) still escapes 0/6; **S7**

6/6 flagged, R7 fired 0/6, R8 fired 6/6 — R7 evaded exactly as predicted, at zero residual, and R8 catches it; **S8 6/6 flagged, R7 fired 0/6, R8 fired 6/6** — the fail-open confirmed, and R8's fail-closed branch catches it. False positives on honest runs: 0/6. Reporting is class-level counts only, per the rule above; no per-run detection probability is computed for batch 4. This is, to our knowledge, the first post-commitment challenge protocol for pseudo-C_ℓ execution receipts: a row-sampled Freivalds test (§7) made sound by a Fiat–Shamir-correct post-commitment challenge, applied to this receipt. Detection is established at class level, not per run; a runner that fabricates the declared intermediate wholesale (S5) still escapes, unchanged across all four batches — receipts of this kind detect shortcuts taken by an analyst who does not forge, not forgery itself.

Table 4: Blind shortcut-detection confusion table, batch 3 (48 sealed runs, 8 arms × 6, R0–R7).

Arm	n	Flagged	Rule(s) fired
Honest	6	0	—
S1 operator skipped	6	6	R3, R4, R7
S2 operator truncated	6	6	R3, R4, R7
S3 grid reduced + interpolated	6	6	R2, R3, R4, R7
S4 cache substituted	6	6	R6 (reference), R7
S4b cache substituted, cross-run	6	6	R6 (cross-run 4/6, reference 2/6), R7
S5 metadata forgery (extended)	6	0	— (escapes; forges $p := MC$)
S6 effective multipole	6	6	R7 alone

R7 catches S6, bounded three ways. (1) R7 was written against S6's own batch-2 definition: a pre-registered rule targeting a known class, not a verifier discovering an unanticipated shortcut. (2) 0/6 honest false positives bounds the false-positive rate above by 0.393; six runs cannot establish a small rate. (3) R7 is bound to a *declared* intermediate, so an informed forger escapes by fabricating $p := MC$ — one matrix-vector product — which is exactly what the extended S5 arm does, 6/6, at residuals $\sim 10^{-17}$. R7 therefore moves the effective-multipole shortcut from "escapes by omission" into the S5 class, "requires deliberate fabrication of a recorded intermediate"; that is the whole of the improvement, and forgery remains the primitive's stated limit.

Attempt 1 was aborted before unsealing, and is preserved. The first batch-3 seal (`8b4fa840...`, commit `b19b72fc`) was executed and blind-judged, then abandoned *without ever being unsealed*: `pcl.make_map` ignored its `seed` argument (healpy `synfast` draws from the global NumPy RNG), so the new `pseudo_cl` instrumentation described a different random map than the run analysed, and R7 was measuring RNG divergence rather than operator consistency. The defect was diagnosed from the public verdict file and source with the assignment still sealed; all 48 invalid runs, receipts, and verdicts are preserved under `blind_test/public3_aborted/BATCH3_ABORT_NOTE.md`; the rules file was untouched; the harness fix (`variants3.seed_rng`) was committed *before* the new key was drawn. This also corrects a batch-1/2 statement: the sealed `map_seed` never actually selected those batches' maps either, so their replicates were fresh random draws rather than seed-varied executions of one map — a wording correction only, since no batch-1/2 rule depended on reproducing a map. The commit-ordered trail for all of batch 3 is `dcf96696` (rules frozen) → `d03fe376` (pre-registration + scripts) → `b19b72fc` (attempt-1 seal) → `60917635` (attempt-1 aborted) → `cd9ab366` (attempt-1 artifacts preserved) → `56ef3fd2` (harness fix) → `c7fb5e38` (fresh-key reseal) → `4a7f9f82` (runs published, blind-judged) → `5b643fc2` (reveal + scorecard) → `bf7d26e3` (OpenTimestamps anchor). No abort criterion was pre-registered before attempt 1 was sealed; the decision to abort

was made ad hoc, on discovering the seeding defect from the public verdict file, not against a pre-specified stopping rule, and is disclosed here as such rather than implied to have been anticipated. The attempt-1 assignment itself was never published: only its commitment (`assignment_sha256 8b4fa8404d4ee0d7bb1276d376353ac045dbf16b05c8a8622141d46d4d6ed554`) and its sealed key are retained under `blind_test/public3_aborted/`, so the abort decision cannot have been informed by which arm any run actually was. The attempt-2 (reporting) seal that followed the harness fix drew a fresh key over a freshly constructed assignment before any attempt-2 run existed, so no information from the unsealed, un-inspected attempt-1 assignment could have propagated into it; the two assignments share no key material.

Scope limits. The estimator exercised in both batches is this repository's own spin-0 MASTER implementation, not NaMaster itself: PyMaster is not installed in this environment, and the instrumented Wigner-3j counter is what makes the trace a measured consequence of the code path taken, so an equivalent hook would need to be added inside NaMaster to carry the claim over verbatim (§11). The in-house estimator was itself checked against `sympy.physics.wigner` on the $m = 0$ kernel for the six cases $(\ell_1, \ell_2, \ell_3) = (1,1,0), (2,2,0), (2,2,2), (4,4,2), (10,10,4), (37,41,6)$ and agrees to 6×10^{-16} on all six. The seal establishes *ordering*, not an external witness: the commit-ordered trail (rules, then seal commitment, then outputs, then key, each in its own commit, sealed material held outside the tree until reveal) is checkable by any reader from git history alone, but the reveal remains self-run — the same party sealed, ran, and unsealed. An external anchor is in progress: the sealed digest of each batch has been submitted to the OpenTimestamps Bitcoin calendar (`blind_test/public{,2}/sealed_digest.json.ots`); as of this revision `ots upgrade` returns *Timestamp complete* with a Bitcoin block-header attestation for the batch-1 and batch-2 stamps; the batch-3 stamp (§6) is submitted and still pending. `ots verify` requires a Bitcoin node, which this machine does not run, so the attested heights are left for a reader with a node, or a public verifier, to confirm. This is a genuine third-party *time* witness; it still does not witness *execution*, and the reveal remains self-run. The S5 metadata-forgery escape is structural, not a bug: closing it needs the same external anchor, or a witnessed execution environment, signed build/execution logs, or a third-party rerun. Under an unmodified-instrumented-harness threat model — the analyst may alter the computation but runs an unmodified, instrumented harness; an adversary who also alters the harness and writes a fabricated honest trace by hand is, by construction, invisible to a receipt-only check — the tested claim is: receipts of this kind are a detector of *structural* shortcuts in instrumented steps, not of forged metadata and not of value-level shortcuts taken downstream of a declared intermediate.

PyMaster cross-check: the estimator itself, not the blind test. Independent of the blind protocol above, the in-house spin-0 MASTER estimator exercised throughout this paper was checked against the public NaMaster library (`pymaster` 3.0, installed via a throwaway conda-forge environment) on an identical map, mask, and ℓ-range ($N_{\rm side} = 64$, $\ell_{\rm max} = 95$). Table 5 gives the result: the mode-coupling matrix and decoupled bandpowers agree with NaMaster to floating-point round-off, confirming the in-house implementation of the MASTER formalism NaMaster also implements. This validates the pointwise formalism itself, not NaMaster's binned-bandpower or single-field convenience APIs, which this check does not exercise; the S6 effective-multipole shortcut remains an in-house-only diagnostic, quantified against the NaMaster-verified exact decoupling as O(10–100%) per-multipole relative error, worst at low ℓ (0.23–1.18 across the 8-wide bands used in the blind test), never approaching the honest path's round-off agreement.

Table 5: In-house spin-0 MASTER estimator vs. NaMaster (`pymaster` 3.0), identical map/mask/ℓ-range.

Quantity	Max relative diff	Median relative diff
Coupling matrix $M_{\ell\ell'}$ (94×94)	4.25×10^{-13}	6.43×10^{-14}
Decoupled bandpowers	1.54×10^{-12}	1.25×10^{-12}

7 Relation to Provenance and Attestation Tooling

The receipt layer sits beside an established body of work on binding artifacts to the process that produced them. Supply-chain attestation frameworks — in-toto [5] and the SLSA levels built on it [6] — record signed statements that a declared build or analysis step ran on declared inputs and emitted declared outputs, and Sigstore [7] with its Rekor transparency log supplies exactly the externally witnessed anchor identified above as the missing ingredient for the metadata-forgery class. Execution-capture tools such as ReproZip [10] package the environment and file accesses of a run so it can be replayed elsewhere; workflow engines including Snakemake [11] and Nextflow [12] emit provenance reports over their task graphs, increasingly serialized as RO-Crate research objects [13]; and experiment trackers such as MLflow [14] bind parameters, code versions, and output artifacts into a queryable run record.

What these share is that the attested quantity is administrative: which step ran, on which inputs, under which environment, producing which bytes — all reported by the harness, none of it a consequence of the numerical path taken inside the step. An analyst who replaces an exact mode-coupling inversion with an effective-multipole rescaling (Sec. 6) still produces a well-formed in-toto attestation, a valid ReproZip bundle, and a complete MLflow run record. The delta claimed here is narrower: the receipt carries a *semantic execution trace* — the number of Wigner-3j evaluations actually performed, the multipole grid actually evaluated, the shape and support of the operator actually built — whose values are a measured consequence of the numerical code path, checked against a published contract derived from one honest reference run. That is complementary rather than competing: the trace is naturally expressible as an in-toto predicate, and anchoring such a predicate in a transparency log would close the metadata-forgery gap using existing infrastructure. Remote-attestation mechanisms such as measured boot and Intel SGX/TDX enclaves [8] address a different part of the same threat model: they let a remote party verify *which* harness binary ran, whereas the trace claimed here verifies *what the harness observed* about the computation's internal path once running; the two are complementary and neither substitutes for the other under an adversary who controls the harness itself. Verifiable-computation schemes such as SNARKs [9] instead prove that a claimed output follows from a claimed program and input without re-execution, which would close the value-level gap (§11) at the cost of a much larger proving system than this narrow layer aims to be. R7 and R8 (§6–6) draw directly on two older results outside the provenance literature above: Freivalds' randomized verification of a matrix product by checking it on a random subset of rows rather than rebuilding the product (R. Freivalds, IFIP Congress 1977, pp. 839–842), of which R7/R8's spot-row check of $MC = p$ is a direct application; and the Fiat–Shamir transform, which makes a randomized challenge sound only when the challenge is drawn after the prover's commitment rather than chosen or predictable by the prover (A. Fiat and A. Shamir, CRYPTO '86, LNCS 263, pp. 186–194) — the condition R7 violates (§6) and R8 restores via its post-commitment verifier seed (§6). The blind protocol itself follows the discipline of blind analysis in cosmology, in which decision rules are frozen before results are seen precisely to prevent post-hoc tuning (J. R. Klein and A. Roodman, Ann. Rev. Nucl. Part. Sci. 55, 141 (2005)). The scope of the guarantee remains

what Sec. 6 measured: structural shortcuts in instrumented steps, under an unmodified-harness threat model.

8 Quality Control

`namaster-proof` version 0.1.7 contains 41 automated tests covering numerical behavior, invalid inputs, the CLI, receipt mutation, protected fields, and compatibility with the original production helpers. All 41 run in a monorepo checkout; in a standalone package install 39 run and the 2 replay-equivalence tests skip cleanly, because those two tests replay against production helper modules that reside elsewhere in the monorepo rather than inside the package (39/41 standalone-effective). CI is defined in `.github/workflows/namaster-proof.yml` (Linux, Python 3.10–3.13, and Windows, Python 3.12); no line- or branch-coverage tool is currently configured in that workflow, so a coverage percentage is not reported here rather than estimated. The window suite uses an exactly represented synthetic linear workspace to compare the precontracted response with the couple–decouple route and to recover an injected grid angle. Rotation tests check conservation of total $EE+BB$ auto-power. Receipt tests mutate result bytes and receipt digests independently and require rejection.

Compatibility tests load the pre-package production modules and compare windowed bandpowers at negative, zero, and positive angles with zero requested absolute or relative tolerance. They also compare algebraic spectrum rotation, bandpower edges, and harmonic keyword contracts. In the associated physical production artifact, a workspace of shape $[4, 20, 4, 1025]$ gave a maximum absolute difference of 1.41×10^{-18} between direct window contraction and the couple–decouple operator — zero to double-precision rounding, not a fractional-error figure, since the original bandpower magnitudes were not retained alongside this scalar. This recorded execution check used IEEE-754 double-precision arrays and the production driver's 10^{-10} acceptance threshold; the package's separate synthetic legacy-helper comparison uses zero requested absolute and relative tolerance. Because the original workspace tensor was not retained, the scalar is not a self-contained reproducibility claim or a universal error bound. The tensor is nonetheless deterministically regenerable from committed, RNG-free code—the mask is a pure function of N_{side} and fixed latitude cuts and the 10^{-10}-gated equivalence check is committed—so the `examples/rebuild_workspace_check.py` script rebuilds the $[4, 20, 4, 1025]$ workspace and re-verifies $\max|\Delta| < 10^{-10}$ whenever PyMaster is installed (only the last digits of the $\sim 10^{-18}$ value are platform-dependent). The full suite runs with `python -m pytest packages/namaster-proof/tests` from the repository root (or `pytest` from within the installed package's test extra).

9 Worked Examples

Minimal synthetic operator. The documentation constructs an identity-like workspace, builds a response from synthetic EE and BB arrays, evaluates a known rotation, and checks operator equivalence. This example has no dependency on a paper-specific dataset and is suitable for installation tests. The minimal call sequence is `response = build_rotation_response(workspace, ee, bb)` followed by `windowed_bandpowers(response, beta_rad=np.deg2rad(0.25))`; the complete executable example also calls `validate_window_equivalence`.

Real PyMaster integration. The optional `examples/pymaster_integration.py` example constructs a real PyMaster workspace, verifies the exact operator equivalence, recovers a declared grid injection, and records the result beside an effective-multipole shortcut comparison. It requires separately installed PyMaster and healpy.

Synthetic CMB recovery campaign. The production use case employed CAMB 1.6.6 raw EE/BB spectra, PyMaster 2.6, $N_{\rm side} = 512$, $\ell_{\rm max} = 1024$, and deterministic seeds. For each of two nonzero injected angles plus a null injection, a 500-realization, foreground-free synthetic run recovered the injection from the mean bandpowers on the declared 0.001° grid, with the per-realization standard deviation and the mean-of-means standard error (SEM = SD/$\sqrt{500}$) computed from all 500 realizations for every injected angle: $0.270^\circ \mapsto 0.2699^\circ \pm 0.0006^\circ$ (SD 0.0128°), $0.342^\circ \mapsto 0.3419^\circ \pm 0.0006^\circ$ (SD 0.0128°), and $0.000^\circ \mapsto -0.0001^\circ \pm 0.0006^\circ$ (SD 0.0127°); all $N = 500$. These are software-recovery checks under the stated simulation contract, not measurements, detection significances, or evidence for a physical birefringence model.

Sharded result validation. Production shards record fields such as suite, realization count, and seed range alongside the content digest. Aggregation can therefore reject a shard whose bytes were altered or whose declared range does not match the requested campaign, rather than silently combining it with valid results.

10 Author Contributions

Houston Golden: conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing—original draft, writing—review and editing, visualization, project administration, and funding acquisition. Correspondence metadata remain author-supplied submission metadata and are not inferred by the software release process.

11 Limitations

`namaster-proof` does not create masks or maps, estimate covariance matrices, separate cosmic rotation from instrumental polarization calibration, model foregrounds, choose a likelihood, or infer cosmological parameters. Uniform rotation and initially vanishing EB are assumptions of the optimized three-component response; the general algebraic rotation helper accepts all four input spectra, but spatially varying rotation requires another model. Fixed-grid recovery inherits the range and resolution selected by the caller. Receipt validation establishes file integrity and agreement with caller-asserted metadata, not the scientific correctness of that metadata. SHA-256 sidecars are content bindings, not digital signatures, an authorization system, or protection against coordinated replacement without an external anchor. Schema version 1 has no migration requirement yet; a future incompatible schema will require an explicit reader or conversion path. The blind shortcut-detection test (Sec. 6) establishes a detector of *structural* shortcuts in instrumented steps and, via R7/R8 (§6–6), of both rule-unaware and rule-aware *value-level* shortcuts and of omitted declared intermediates; it is not a fraud detector: it cannot in principle catch metadata forgery (an adversary who alters the harness itself, or who fabricates a declared intermediate such as $p := MC$ as the extended S5 arm does 6/6 in batch 3), a semantically wrong but equally expensive computation, or numerical drift below the declared tolerance. Batch 3's value-level rule R7 (§6) catches the effective-multipole class S6 6/6, but only because R7 was pre-registered against S6's own definition; a shortcut that reproduces a correct *declared* `pseudo_cl` by a different internal path, without ever computing it from the honest operator, is not exercised by R0–R7. All detection rates in this paper (R1–R7) are class-level, not per-run, probabilities (§6).

Remaining open items, numbered independently of the shortcut classes S1–S6 to avoid collision:

L1 (estimator).
The blind test exercises this repository's own spin-0 MASTER estimator rather than NaMaster/PyMaster itself; a separate, non-blind cross-check now validates that estimator

against `pymaster` 3.0 to floating-point round-off (§6), but an equivalent instrumented hook has not been added inside NaMaster itself, so the blind-test claim does not transfer to NaMaster verbatim (closed for the estimator; open for the hook).

L2 (external anchor).
Its seal establishes commit ordering, not an external witness. `ots upgrade` now returns *Timestamp complete* with a Bitcoin attestation for the batch-1 and batch-2 stamps; the batch-3 stamp is submitted and pending as of this revision (partially closed).

L3 (mechanism coverage).
S1–S3 in batch 2 are all detected by the same trace-mismatch mechanism (R3/R4); S4's five detections were all the *reference* disjunct of R6. Batch 3's S4b arm exercised the *cross-run* disjunct for the first time (4/6 fired it; the other 2 collapsed to the reference disjunct via a named source run, §6) (closed).

L4 (single run).
The batch-2 sequence was run once and not repeated. Git history proves the order of surviving commits, not the absence of discarded attempts; an OpenTimestamps anchor does not close that gap either. A third-party rerun or witnessed environment would.

12 Availability

Software version. This paper documents `namaster-proof` version 0.1.7. Every version number of the form 0.x.y in this paper refers to a release of the `namaster-proof` software package, and matches the released artifact: `pyproject.toml`, `codemeta.json`, `CITATION.cff`, and the package `__version__` all declare 0.1.7, as does the Zenodo software record cited under Archive below. The separate stamp on the title page (v2B.0.23) is the revision number of this manuscript as a document; it tracks editorial revisions of the metapaper and is deliberately independent of the software release line. The two numbers are not expected to agree.

Operating system. The package supports Linux and Windows where Python and NumPy are available; CI covers Linux with Python 3.10–3.13 and Windows with Python 3.12. Directory metadata are synchronized on POSIX systems after atomic replacement. The suite (including both blind-test batches, Sec. 6) is developed and exercised locally on macOS (arm64), but macOS is not covered by continuous integration, so it is exercised locally, not CI-verified.

Additional system requirements. The package has no special hardware requirements: it is pure Python with a single NumPy runtime dependency, requires no GPU, and its full test suite completes in under one second on a commodity CPU; no formal memory profiling was performed, but the largest retained arrays are the $[4, 20, 4, 1025]$-shaped bandpower-window tensors in §8 (order 10^5 double-precision entries, a few MB), so heap use is dominated by NumPy allocation overhead rather than by the package's own data. The optional PyMaster integration example ($N_{\rm side} = 16$) completes in a few seconds, whereas the retained physical validation campaign ($N_{\rm side} = 512$, $\ell_{\rm max} = 1024$; 500 realizations at each of three injected angles) recorded a wall time of order 7×10^2 s with eight parallel realization workers on a commodity multi-core CPU (correspondingly of order a couple of hours single-worker). These campaign timings are environment-dependent; both workloads depend on the user-supplied PyMaster and healpy stacks and correspondingly more memory, but lie outside the self-contained packaged code path.

Programming language and dependencies. `namaster-proof` version 0.1.7 requires Python 3.10 or later and NumPy 1.24 or later. PyMaster is optional and is supplied by the user for physical NaMaster workspaces; the retained physical validation used PyMaster 2.6 and healpy 1.19.0.

Code repository. The source, issue tracker, installation instructions, API documentation, tests, and examples are available at https://github.com/Hubify-Projects/bigbounce/tree/main/packages/namaster-proof. From a checkout of that repository the package installs locally with `python -m pip install ./packages/namaster-proof` (append `-e '.[test]'` from within the package directory for the test extra). `namaster-proof` is an independent downstream verification layer and is not an official release of, or affiliated with, the NaMaster project. Citation metadata are supplied in `CITATION.cff` and machine-readable metadata in `codemeta.json`. The package directory was first published to the repository on 2026-07-16.

License. The software is released under the MIT License.

Archive. `namaster-proof` version 0.1.7 is bound to an immutable archive: the commit-pinned source tree (`packages/namaster-proof` at repository commit `0a587b583f8e86c4ce1ee4a20526fcdcd8035fe6`), together with its license, citation metadata, and checksums, is published on Zenodo under doi:10.5281/zenodo.21481753 (deposited July 21, 2026). The manuscript itself, with its exact source, arXiv bundle, and provenance manifest, is separately archived under doi:10.5281/zenodo.21481842 (CC-BY-4.0, deposited July 21, 2026).

Validation artifacts. The corrected physical example is bound to the repository summary artifact (SHA-256 `745b0a2f060773ce69c005ea84b74b305ec26a85f6aaafe58f0b3244b7f39914`) and the repository bandpower artifact (SHA-256 `b00f850e338007caea6af76f4e9305ab6b54a68e6799efd450bc76f1c325f331`).

Reproducibility: blind shortcut-detection test. All four batches' manifests live in `reproducibility/manifests/experiments/`: the pilot (batch 1, Sec. 6) as `p1b-blind-shortcut-detection.json`; batch 2 (Sec. 6) as `p1b-blind-shortcut-detection-batch2.json`; batch 3 (Sec. 6) as `p1b-blind-shortcut-detection-batch3.json`; and batch 4 (§6) as `p1b-blind-shortcut-detection-batch4.json`. Scripts for all four live in `pipelines/namaster_proof/blind_test/`: batch 1 uses `seal.py`, `run_blind.py`, `verify.py`, and `reveal.py`; batch 2 adds `variants2.py` (the S6 class) and uses `seal2.py`, `run_blind2.py`, the same `verify.py`, and `reveal2.py`; batch 3 adds `variants3.py` (R7, S4b, and the extended S5) and uses `seal3.py`, `run_blind3.py`, the R7 extension to `verify.py`, and `reveal3.py`, with the aborted first attempt preserved under `blind_test/public3_aborted/`; batch 4 adds `variants4.py` (S7, S8) and uses `seal4.py`, `run_blind4.py`, `verify4.py` (R8), `reveal4.py`, and `verifier_seed4.py` for the commit–reveal challenge. The frozen rule-set files themselves, as committed and pinned in each batch's `frozen_rules_digest.json`, are sha256 `169b3690...` (`RULES_v2_FROZEN.md`, batch 2), sha256 `856f4c50...` (`RULES_v3_FROZEN.md`, batch 3), and sha256 `a59caaf8...` (`RULES_v4_FROZEN.md`, batch 4). Reproducing any committed run reuses its committed `sealed{,2,3,4}/key.txt` and `sealed{,2,3,4}/assignment.json` rather than re-running the seal script, which draws a fresh random key. All four batches cost $0 on any CPU-only machine; batch 1's 18 runs plus overhead take approximately one to two minutes, batch 2's 35 runs measured ∼11 s of core execution, and batch 3's 48 runs (plus the 48 aborted) complete in under two minutes total. The PyMaster cross-check (§6)

is `reproducibility/manifests/experiments/p1b-pymaster-crosscheck-2026-09-05.json`, script `pipelines/namaster_proof/blind_test/pymaster_crosscheck.py`, run in a throwaway conda-forge `pymaster` 3.0 environment ($0, under one minute).

13 Reuse Potential

Researchers can reuse the window layer to test uniform spin-2 rotation models against any workspace exposing the small NaMaster protocol, reuse the multipole contracts to prevent field/bin support drift, or use the receipt layer for sharded JSON calculations outside cosmology. The receipt primitive remains in the same package because it authenticates the exact validation artifacts produced by the window workflows; it is a small reusable module rather than a claim that the package is a general provenance framework. Extensions can add other angle-dependent spectrum bases, signed receipt anchors, or schema migrations. Questions, bug reports, and contributions are handled through the public repository issue tracker and pull-request workflow. With the post-commitment verifier challenge of §6, §6 is now a pre-registered, sealed measurement with a confusion matrix and an adversary model rather than a software usage note; the package itself remains a short reusable module. Accordingly this manuscript targets a venue for the measurement (e.g. ACM REP) with arXiv `astro-ph.IM` as primary category and `cs.SE` as secondary, alongside a short software-description submission for the package itself (e.g. JOSS or JORS) that cross-cites the measurement.

14 AI Usage Disclosure

AI coding and review agents assisted with implementation review, test generation, documentation, and manuscript editing. Their outputs were checked against committed source and retained numerical artifacts; executable claims were accepted only after local tests or direct recomputation.

Acknowledgements

The exact-window operations build on NaMaster [2] and the MASTER pseudo-C_ℓ method [1]. The production example used CAMB [3] and HEALPix [4].

Funding Statement. No external funding supported this software release.

Competing Interests. The author declares no competing interests.

Appendix: Batch-2 Per-Run Verdicts

Table 6 lists all 35 batch-2 runs from `blind_test/public2/verdicts.json` and `blind_test/sealed2/assignment.json` (run id, assigned arm, call, rule(s) fired), reproducing Table 3 at per-run granularity.

References

[1] E. Hivon, K. M. Górski, C. B. Netterfield, B. P. Crill, S. Prunet, and F. Hansen, "MASTER of the CMB anisotropy power spectrum: A fast method for statistical analysis of large and complex cosmic microwave background data sets," *Astrophysical Journal* 567, 2 (2002), doi:10.1086/338126.

[2] D. Alonso, J. Sanchez, and A. Slosar, "A unified pseudo-C_ℓ framework," *Monthly Notices of the Royal Astronomical Society* 484, 4127–4151 (2019), doi:10.1093/mnras/stz093.

[3] A. Lewis, A. Challinor, and A. Lasenby, "Efficient computation of cosmic microwave background anisotropies in closed Friedmann–Robertson–Walker models," *Astrophysical Journal* 538, 473–476 (2000), doi:10.1086/309179.

Table 6: Batch-2 per-run verdicts (run id → arm → call → rules fired).

#	Arm	Call	Rules	#	Arm	Call	Rules
0	S5	honest	—	18	S2	shortcut	R3,R4
1	S3	shortcut	R2,R3,R4	19	S2	shortcut	R3,R4
2	S6	honest	—	20	honest	honest	—
3	honest	honest	—	21	S1	shortcut	R3,R4
4	S1	shortcut	R3,R4	22	honest	honest	—
5	S4	shortcut	R6	23	S4	shortcut	R6
6	S3	shortcut	R2,R3,R4	24	S5	honest	—
7	S4	shortcut	R6	25	S6	honest	—
8	S4	shortcut	R6	26	S1	shortcut	R3,R4
9	S4	shortcut	R6	27	S3	shortcut	R2,R3,R4
10	honest	honest	—	28	S5	honest	—
11	S3	shortcut	R2,R3,R4	29	S1	shortcut	R3,R4
12	S5	honest	—	30	S2	shortcut	R3,R4
13	S2	shortcut	R3,R4	31	S5	honest	—
14	S6	honest	—	32	S1	shortcut	R3,R4
15	S3	shortcut	R2,R3,R4	33	S2	shortcut	R3,R4
16	S6	honest	—	34	S6	honest	—
17	honest	honest	—				

[4] K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann, "HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere," *Astrophysical Journal* 622, 759–771 (2005), doi:10.1086/427976.

[5] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium* (2019).

[6] OpenSSF, "Supply-chain Levels for Software Artifacts (SLSA)," slsa.dev.

[7] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2022).

[8] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive* 2016/086 (2016).

[9] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture," in *23rd USENIX Security Symposium* (2014).

[10] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational reproducibility with ease," in *ACM SIGMOD International Conference on Management of Data* (2016).

[11] F. Mölder, K. L. Jablonski, B. Letcher, et al., "Sustainable data analysis with Snakemake," *F1000Research* 10, 33 (2021).

[12] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology* 35, 316–319 (2017).

[13] S. Soiland-Reyes, P. Sefton, M. Crosas, et al., "Packaging research artefacts with RO-Crate," *Data Science* 5, 2 (2022).

[14] M. Zaharia, A. Chen, A. Davidson, et al., "Accelerating the machine learning lifecycle with MLflow," *IEEE Data Engineering Bulletin* 41, 4 (2018).