Eight Prime Divisors in Odd Distinct Covering Systems

Companion guide · Version 6

Michael Schroeder

6 September 2026

Contents

Purpose and reading routes

This companion accompanies the paper *Eight Prime Divisors in Odd Distinct Covering Systems*. It separates three questions: what the formal theorems say, what was checked, and how to reproduce those checks. The mathematical proof remains in the paper; this guide does not replace it with a build log.

The complete publication archive is the authoritative source distribution for this version. Paths below are relative to its extracted root. The online standalone guide is a reading copy: source-file links resolve after extraction, not necessarily next to the standalone PDF or Markdown download.

- **Mathematical reader:** read Theorem 1.1, Corollary 1.2 and Theorem 1.3, followed by the architecture in the introduction and the concise Section 12.
- **Semantic reviewer:** inspect the two statements below, then the covering semantic contract, covering theorem map and scalar theorem map.
- **Re-verifier:** follow the tiered commands below; read the verification report and provenance guide before interpreting logs.
- **Developer:** start at `formal/Paper.lean` or `scalar/lean/RankScaling/Ceiling.lean`. The covering API index is navigational, not a dependency proof.

The two claims, exactly

The integer-covering theorem

Let a finite family of residue classes cover every integer. Suppose its moduli are natural numbers greater than one, odd and pairwise distinct. If N is their least common multiple, then

$$\omega(N) \geq 8, \qquad N \geq 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 = 111{,}546{,}435.$$

The public Lean declaration in `formal/Paper.lean` is:

```
theorem eight_prime_support_of_integer_cover {L : ℕ}
    (residue : Fin L → ℤ) (modulus : Fin L → ℕ)
    (hCover : ∀ z : ℤ, ∃ k,
      z ≡ residue k [ZMOD (modulus k : ℤ)])
    (hNontrivial : ∀ k, 1 < modulus k)
    (hOdd : ∀ k, Odd (modulus k))
    (hDistinct : Function.Injective modulus) :
    8 ≤ (Finset.univ.lcm modulus).primeFactors.card
```

Its namespace is `Rank8Paper`. The displayed signature is reformatted only; the shipped source is authoritative. The numerical lcm corollary follows immediately in the paper; it is not an additional premise of this declaration.

There is no supplied residual model, exponent cutoff, squarefreeness assumption, certificate assumption or numerical oracle. Integer residues may be negative. The formal bridge proves the claim for ordinary coverage of all integers, not merely for an artificial comparison distribution. Zero-height padding handles every support size at most seven. Earlier rank-seven results remain in the shared import graph but are excluded from the final proof term's transitive dependencies by `AuditPaper.lean`.

The original compact upper certificate gives

$$1 \leq F_1(A) \leq \frac{124847}{125000} < 1, \qquad 1 - \frac{124847}{125000} = \frac{153}{125000}.$$

It has 54 rational inequalities and six affine identities. Exactly two inequalities, at $(j, K) = (1, 4)$ and $(1, 8)$, hold with equality. The least positive slack among the others is $1/56000000$. Use exact arithmetic: rounding the table can destroy its validity. The affine tails cover all loads and arbitrary finite prime-power exponents.

The scalar-method ceiling

In `scalar/lean/RankScaling/Ceiling.lean`, namespace `RankScaling`:

```
theorem eight_prime_scalar_ceiling {θ : ℕ → ℝ}
    (hθ : Admissible θ) {order : List ℕ}
    (ho : order.Perm basePrimes) :
    (1001 : ℝ) / 1000 ≤
      infiniteEvaluate (schedule θ order) 1
```

Here `basePrimes` is `[3,5,7,11,13,17,19,23]` and `Admissible θ` requires $0 \le \theta(p) \le p-3$ on that list. The value is the composition of the paper's infinite-depth operators, with zero terminal continuation, evaluated at load one. Leftmost is outermost. Convergence is proved through eventual affine tails, not inferred from a large finite truncation.

The four architectural features are one fixed threshold per prime, unit ending charge, whole prime blocks, and the saturated single-index envelope $K \mapsto K(d+1)$. The reference-prime list is also specified. The certificate is a lower bound for every real threshold vector and every block order. Its real-box subdivision is coverage of a continuum, not a sampled grid. Statewise minimization is used only as a lower relaxation; it is not a proposed physical or past-measurable policy.

Additional admissible blocks in any positions cannot restore a value below one. Further extensions cover specified wider thresholds, upper envelopes, load-shaped chains and terminal penalties. These are different declarations with different hypotheses: use the scalar theorem map. General infinite continuations require summability at every actual suffix and state. The eventual-affine versions prove this internally. No claim uses Lean's totalized sum of a divergent series as a genuine convergent value.

This does **not** prove that a rank-nine theorem is false, construct a cover on eight primes, or exclude methods retaining more residue/type information. Reference lists arising from a case split on small actual primes are outside the ceiling unless they contain the specified eight-prime list as a subsequence. Neither the paper nor the formalization identifies a uniquely binding relaxation at rank nine.

Certificate and source map

Object	Where to inspect	Role
Original upper certificate	Paper Table 1 and Appendix A; `checks/verify_rank8_compact_certificate.py`	Proves the covering contradiction; 54 inequalities and six tails.
Full-label and arithmetic bridge	`formal/Paper.lean`, `formal/AuditPadding.lean`, `formal/docs/THEOREM_MAP.md`	Connects the universal covering statement to the certificate.
Sharper seven-block scalar witness	Paper Proposition 9.2 and Appendix B; `scalar/improved_rank8_certificate.json`; `SevenWitness.lean`	Exact value below $998419/10^6$; does not replace the original covering certificate.
Eight-block lower certificate	Paper Sections 9–11; `scalar/barrier_certificate.json`; `scalar/lean_candidates/`	Supports the method ceiling, with every premise discharged in Lean.
Scalar soundness and closed conclusion	`Soundness.lean`, `Coverage.lean`, `Ceiling.lean` under `scalar/lean/RankScaling/`	Integer checks imply all-state real edges and continuous box coverage.

The large certificate uses scale 10^{12}, 24 small-state values and an affine tail from load 25. That height is a representation device, not a bound on covering-system exponents. Lean's completion index `n = 23` means 24 explicit increments. The certificate has 367,366 numerical cells, 1,441 numerical modules, 15,951 coverage nodes, 7,976 leaves and 125 coverage modules. There are 1,568 generated Lean files and 39 nongenerated files, for 1,607 local scalar sources in the expanded archive.

The compact `scalar/` tree includes the exact inputs and deterministic generators. The full archive `companion/Rank8_Method_Ceiling_Lean_2026-09-06.zip` additionally includes every expanded source and the original replay logs. Both forms represent the same proof inputs. Generating files is not a proof check; Lean must check the generated mathematical terms.

Verification evidence and trust

What the records establish

Procedure	Covering project	Scalar project
Current release source integrity	47 Lean files compared with verified baseline	Full ZIP manifest, 1,607 source hashes and compact inputs rechecked
Current source build	Fresh extracted-project build from the final source ZIP; dependency caches only reused	Cached build of unchanged expanded sources
Current public statement/axiom audit	`AuditPaper.lean`	`RankScaling/PrintAxioms.lean`, 15 guarded lists
Fresh imported-proof replay	`leanchecker --fresh -v Paper`, from extracted project	Not newly performed or claimed
Large cold numerical replay	Not applicable: compact covering certificate	Original successful 184 batches and 34 final stages preserved; not repeated for this editorial release
Python checks	Exact certificate, mutations, physical-state regressions and cross-format checks	Exact seven-block witness and integrity checks; not an oracle for either Lean theorem

The current covering sequence has eight stages: the compact checker, local Bellman/fibre regressions, full-chain regressions, paper consistency, source audit, project build, public statement/axiom audit and fresh kernel replay. The machine-readable `verification/archive_replay.json` binds the full run to the final compact source ZIP's SHA-256 and records successful independent TeX compilation and PDF-text/metadata agreement after extraction. The extracted project begins without its own compiled objects; pinned dependency caches are reused. This is a second local extraction, **not a second machine**.

The preserved scalar cold replay checked all numerical cells and coverage nodes; its final build reported 4,581 successful jobs. Regeneration reproduced all 1,568 generated source files byte for byte. Fresh current integrity and public-audit checks are explicitly distinguished from that preserved run. The original records are retained within the immutable scalar archive and selected compact records. Older covering logs are isolated under `verification/history/` and are not relabeled as current execution.

Trust boundary

Both public theorem audits report exactly:

```
propext
Classical.choice
Quot.sound
```

These are the standard foundations used by the developments, not additional covering-system or certificate axioms. Source scans and proof-term audits exclude admitted goals, custom mathematical axioms and native-computation proof shortcuts. The large arithmetic uses `decide +kernel`, not `native_decide`. Source scans complement, but do not replace, the actual transitive axiom and dependency audits.

`leanchecker --fresh` reruns Lean's own kernel; it is not a separately implemented checker. Invoking `lean --trust=0` on an importing file alone would not demonstrate that all imported terms were freshly replayed. Lean's implementation, runtime/compiler and hardware remain within the usual trust boundary. Semantic agreement between the intended mathematics and the formal statement still requires human inspection; that is why the signatures and theorem maps are supplied.

Finite regressions do not prove universal quantifiers. Hashes identify bytes, not correctness, authorship or independent replication. This release claims neither peer review nor second-platform verification.

Pinned environment and integrity identifiers

Both Lean projects use the same pins. Do not run `lake update` or substitute the latest release when reproducing this version.

```
Lean toolchain: v4.32.0-rc1
Lean compiler commit:
b4812ae53eea93439ad5dce5a5c26591c31cb697
mathlib commit:
360da6fa66c1273b76b6b2d8c5666fd5ac2e3b56
```

Every transitive dependency revision is specified in each project's `lake-manifest.json`; the installed dependency check records revision and tracked-source cleanliness, not binary provenance. The recorded environment is macOS arm64. Other platforms are instructed how to rebuild, not claimed to have been tested.

The immutable expanded scalar archive has SHA-256:

```
a3c40165e9ce979926cb9d469a0034d8cda58c86f3db69f844132513208293c8
```

The canonical real-box subdivision has SHA-256:

```
c83b1be3c0e2ddcfcfb4ec6a80c9f94706fb4bd168e59c040c556da902318b47
```

These complete digests intentionally appear here, not in the mathematical narrative. Current paper, guide, source and delivery digests are generated in release manifests rather than manually repeated. `SHA256SUMS` covers every distributed member except itself. The website bundle also supplies `release.json` with exact public filenames, byte sizes and hashes. These are integrity records, not digital signatures.

Reproduce in increasing depth

1. Inspect and run the compact checks

Extract the complete publication archive into a new directory. Work on a copy if you wish to preserve the shipped logs: verification scripts write new run records. From the extracted root:

```
shasum -a 256 -c SHA256SUMS
python3 checks/run_verification.py --quick
python3 tools/check_revision_math.py
```

Use `sha256sum -c SHA256SUMS` on GNU/Linux. The proof/regression checks need only Python's standard library. Do not use `python -O`: assertions are verification conditions. The compact checker is also printed in Appendix A and reads no external certificate file. The auxiliary JSON in `checks/` is solely an unrounded regression reference, not a Lean proof input.

2. Build and freshly replay the covering theorem

Install the Lean toolchain manager `elan` separately. From `formal/`:

```
lake exe cache get
lake build
lake env lean AuditPaper.lean
lake env leanchecker --fresh -v Paper
```

The cache command obtains pinned dependencies and their compiled library cache and needs network access. From the publication root, the complete eight-stage runner is `python3 checks/run_verification.py`. A local fresh run takes several minutes; hardware and dependency availability determine actual time. It does not build the scalar project.

3. Regenerate and replay the scalar theorem

The compact source deliberately has no pre-expanded large modules. From `scalar/`, with Python 3.9 or later:

```
python3 export_lean_candidates.py
python3 emit_lean_numerics.py
python3 emit_lean_coverage.py
python3 emit_seven_witness.py
```

Alternatively extract the immutable expanded scalar ZIP into a separate directory; do not merge its root with `formal/`. After generation or extraction, obtain the pinned dependencies from that scalar project's `lean/` directory with `lake exe cache get`. Return to its parent and run:

```
python3 replay_lean_numerics.py --jobs 8
python3 replay_lean_final.py --jobs 4
```

These bounded-concurrency scripts check the numerical batches, symbolic root, continuous coverage and final public audit. Budget hours and substantial disk/memory. Reduce the positive job counts for a smaller machine; a large unbounded parallel build may exhaust memory. The scripts replace their local replay records, so preserve the shipped evidence first. A cached `lake build` is useful as an incremental check but is not a cold arithmetic replay. Read `scalar/FORMALIZATION.md` for the complete mathematical dictionary and the original scalar release's reproduction detail.

4. Rebuild the documents and verify the source archive

The paper uses standard LaTeX packages, no external figures, no shell escape and no separate bibliography process. The recorded compiler is Tectonic 0.17.0; XeLaTeX is an alternative requiring its own layout check. Markdown is generated with Pandoc 3.11 using actual compiled theorem/equation numbering. See BUILD_NOTES.md for exact commands and companion-PDF generation. PDF checks use Poppler and pypdf; both pypdf 6.10.0 and 6.17.0 are tested for this release. FontTools is needed by the latter environment.

`tools/replay_source_archive.py` extracts the compact source ZIP into a specified empty directory, verifies its manifest, compiles the paper and runs the full covering sequence. `--packages` may point to an existing pinned dependency directory; no project binaries are copied. This script does not repeat the scalar cold replay. Its current record is tied to the exact source ZIP, not just a similarly named working tree.

Release, citation and research context

The paper cites the author's *Seven Prime Divisors in Odd Distinct Covering Systems* at its public author-hosted location. The rank-eight proof does not assume that earlier theorem. The literature ledger records the primary sources consulted and retrieval limits, including the distinction between covering systems introduced in 1950 and the odd-covering question explicitly recorded in 1965. Erdős Problem 7 remains open.

`CITATION.bib` and `CITATION.cff` identify this version without inventing a DOI or arXiv record. The website-ready assets are prepared for an author upload; their preparation is not evidence that they are already online. No blanket license has been chosen for this rank-eight release; read `RIGHTS.md`. Upstream dependencies retain their own licenses.

Acknowledgment

This work benefited from research assistance by AI systems developed by OpenAI and Anthropic, including support for proof exploration, proof development, and exact computational checks among others.

This paragraph reproduces the rank-seven acknowledgment. AI output is not a proof oracle. Responsibility for the claims, final review and publication rests with the author.